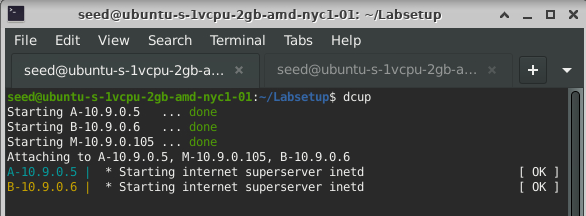
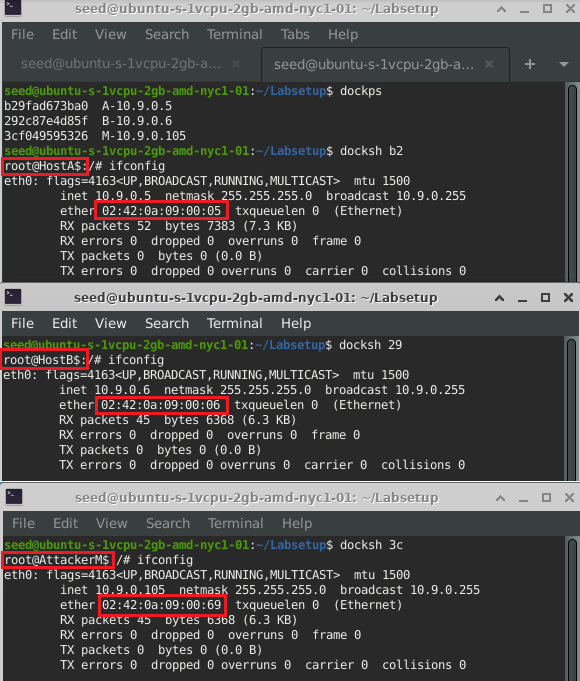
**LAB 2: ARP ATTACK**

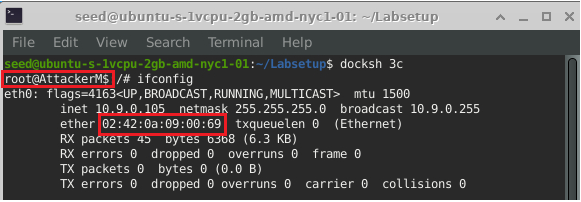
## TASK 0: Setting up SEED labs

The following is the environment setup for the lab:

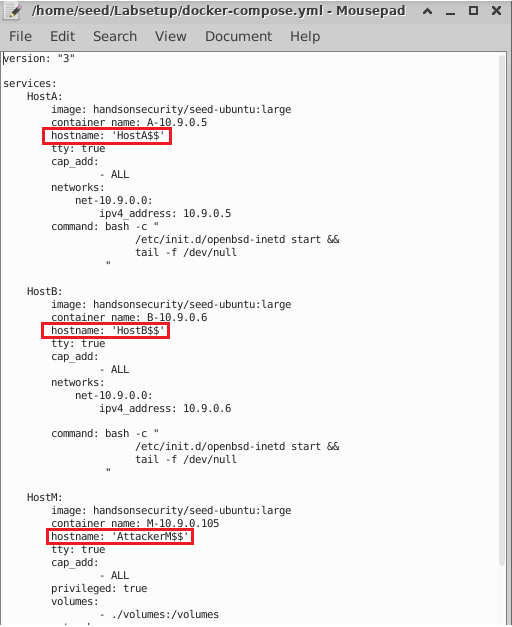
|  |  |  |
| --- | --- | --- |
| **SYSTEM** | **IP ADDRESS** | **MAC ADDRESS** |
| Host A | 10.9.0.5 | 02:42:0a:09:00:05 |
| Host B | 10.9.0.6 | 02:42:0a:09:00:06 |
| Attacker M | 10.9.0.105 | 02:42:0a:09:00:69 |







We have added the hostname field is docker-compose.yml file to easily identify the systems.

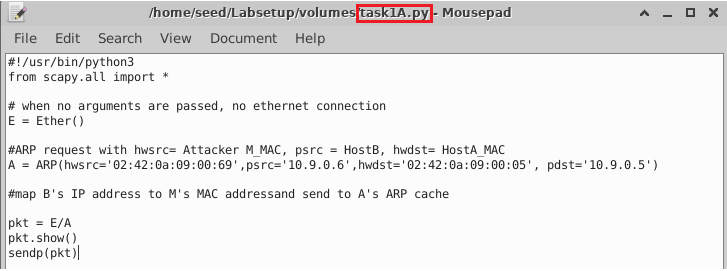


## TASK 1: ARP Cache Poisoning

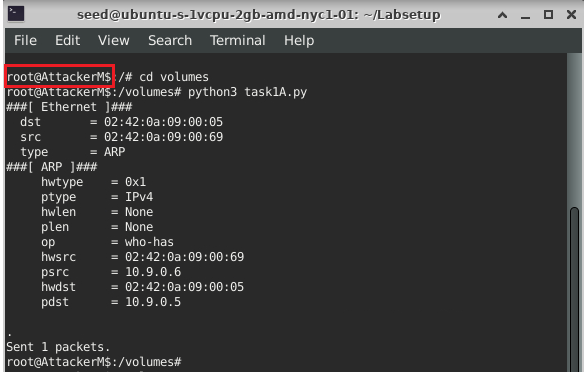
In this task, we attack Host A’s ARP cache by mapping B’s IP address to Attacker M’s MAC address in A’s ARP cache. We achieve this using 3 different methods as below.

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

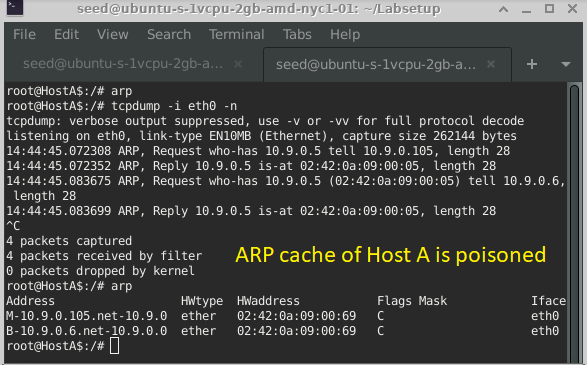
The following is the code snippet to perform ARP cache poisoning using spoofed ARP request from attacker to A. An ARP packet with host B’s IP address and attacker M’s MAC address as source and host A’s IP and MAC address as destination. The op parameter in ARP() function has default value 1 indicating it’s an ARP Request.



We run the above code from Attacker M’s machine and get output as seen beow. The op field ‘who-has’ indicates that it is an ARP request.



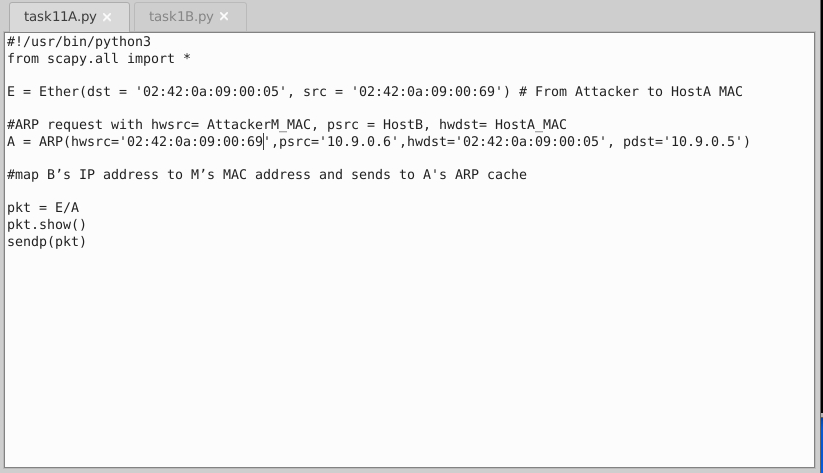
The following is the ARP cache for Host A and Host B:



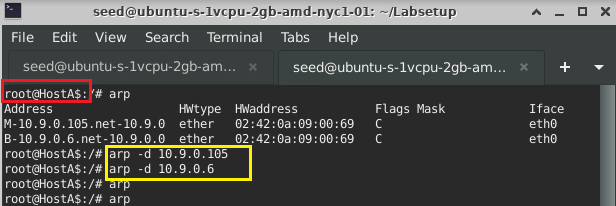


**Normally, ARP requests are broadcasted** over the LAN. But here our intention is to poison only A’s ARP cache, so we create a unicast message and send it to Host A and find that the attack is successful. However, the above code also creates an entry of Attacker M machine 10.9.0.105 in A’s ARP cache. This is because the Ethernet header’s fields are filled by the OS based on the packet received.

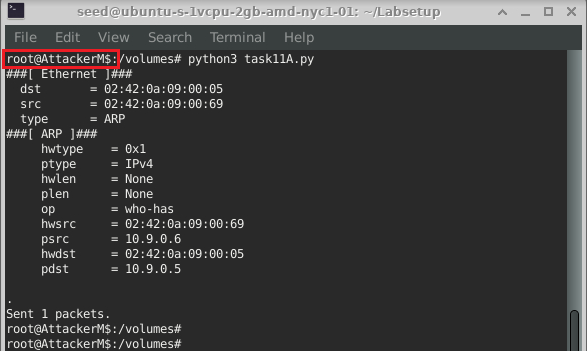
To avoid this, we modify the code by entering the Ethernet header’s fields:



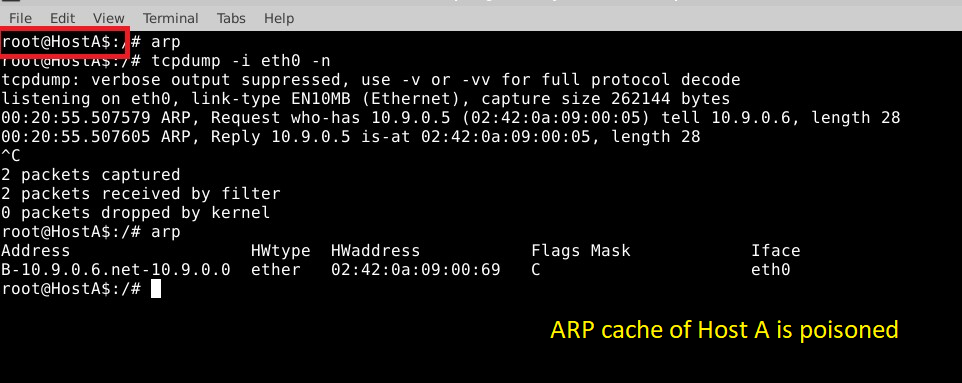
Before executing the code, we delete the ARP cache in machine A using the command “arp -d [IP]”.

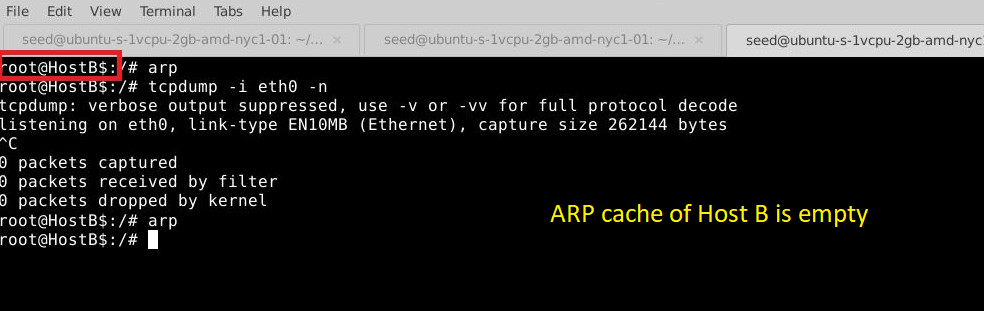


On running the code in Attacker M, we observe same result as before:



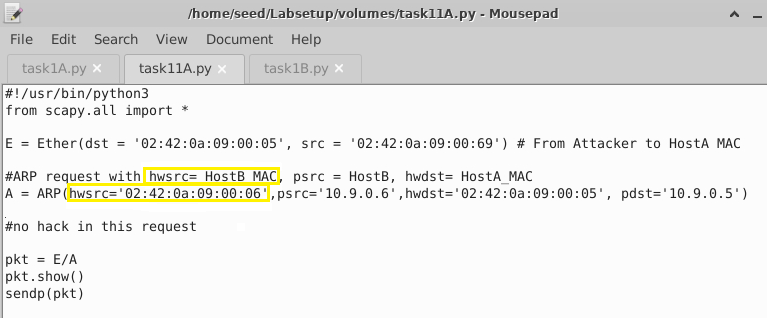
And we see the following result in Host A and B. We observe that Attacker M’s entry is not stored in the ARP Cache of A anymore.



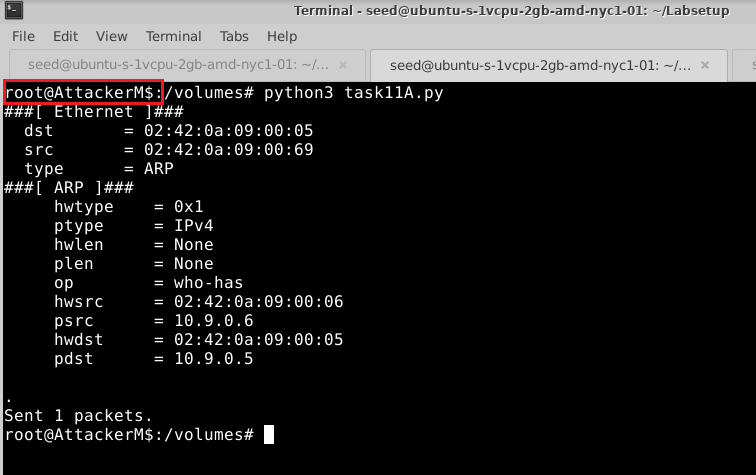


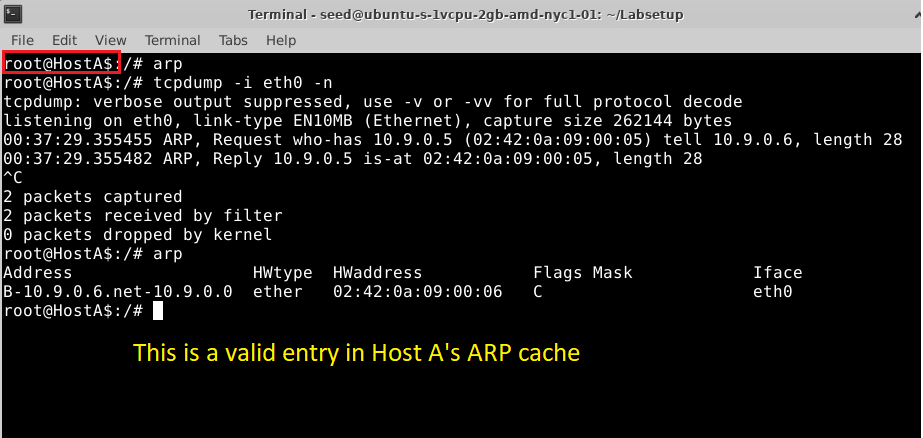
### Task 1B (using ARP reply):

#### Scenario 1: B’s IP is already in A’s cache

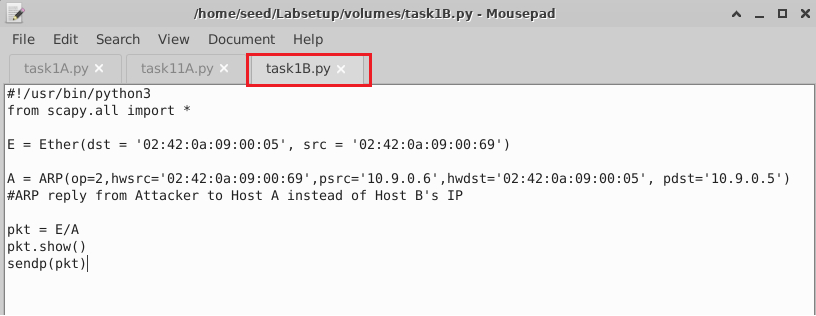
The following is the code to add valid address mapping of host B in ARP cache of machine A using ARP request

We run the above code and observe that ARP cache in machine A is filled with valid entry.

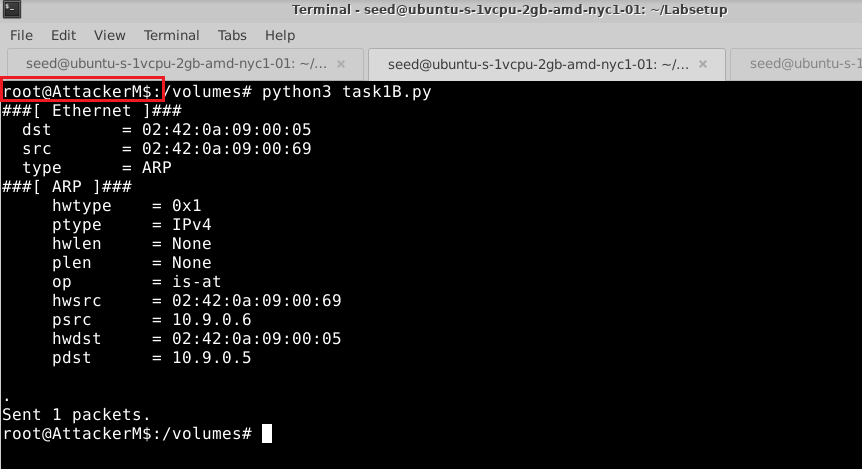




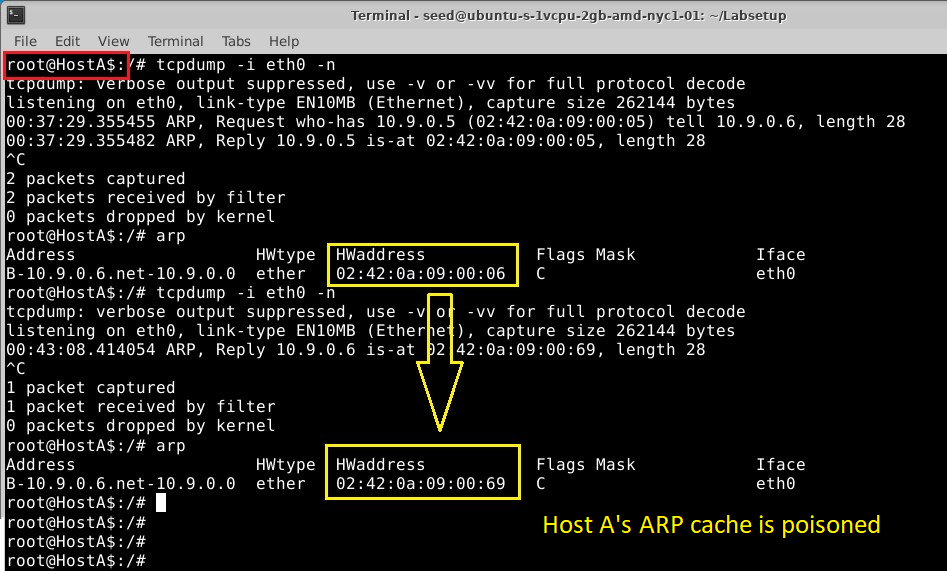
The following is the code to perform ARP Cache poisoning using spoofed ARP reply to A. The only difference we can observe here is that op field is set to 2 in ARP() function.

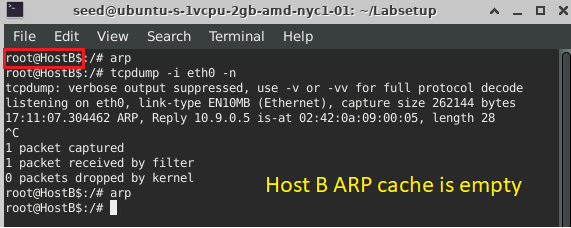


We run the above code from attacker M machine and see the following packet is sent out. The is-at string in op field indicates that it is an ARP reply.



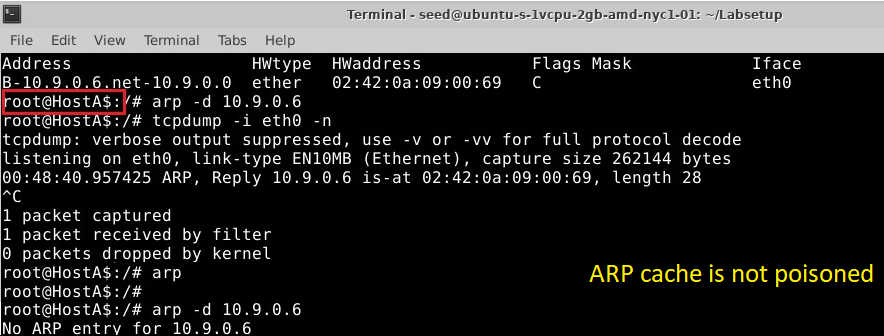
The ARP Cache entry in A has been poisoned.





#### Scenario 2: B’s IP is not in A’s cache.

First, we delete the ARP cache in machine A and re-run the above code “task1B.py” from attacker M’s machine and observe that ARP cache in machine A is not poisoned using ARP reply.



**OBSERVATION**

**An ARP reply is unicast unlike the broadcasted ARP request to destination**. Here, machine A does not have Attacker M’s MAC address in its ARP cache. We know ARP cache table is a collection of recently learned MAC addresses and corresponding IP addresses.

The ARP reply packet is received only by the host which transmitted the ARP request packet as it is unicast transmission. If that is the case, ARP module adds the Ethernet hardware address to IP address mapping present in the ARP reply packet to the ARP cache. If the request packet is not delivered to destination, source will receive ICMP unreachable message.

Here machine A didn’t send any request to attacker M but still receives an ARP reply packet. **Linux OS provides some level of security by ignoring unsolicited ARP replies**. It checks whether there was an ARP request packet sent from machine A to M or there already exists an entry in its ARP cache table. When both fails, the packet is discarded.

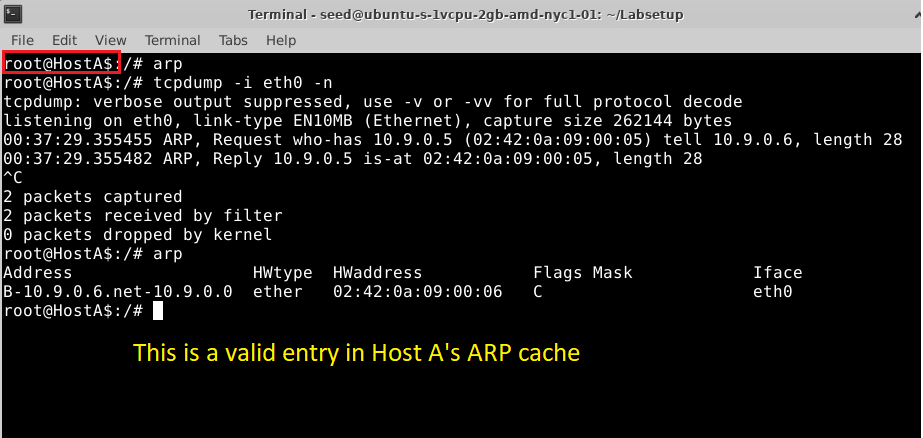
### Task 1.C (using ARP gratuitous message)

A gratuitous ARP is an ARP request sent when a host wants to resolve its own IP address. It is a broadcast message with destination MAC ff:ff:ff:ff:ff:ff:ff and source & destination IP address to be its own same IP address.

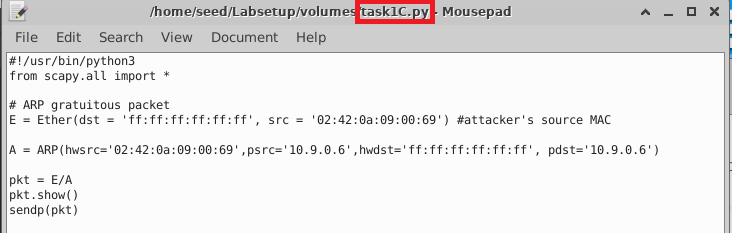
Devices will send gratuitous ARP when they boot up, which announces their presence to the rest of the network and when there is a change on its IP address or MAC address. Attacker makes use of this feature of gratuitous ARP and cause the devices in the network to update their ARP table with a wrong MAC address to IPv4 address mapping.

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

We perform same steps as done in Scenario 1 in Task 1.B to add valid entries in machine A’s ARP cache.

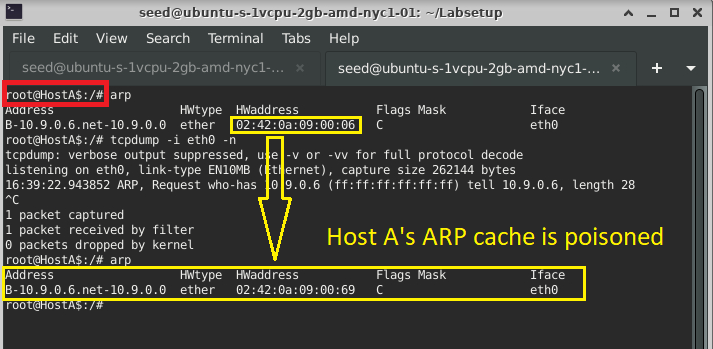


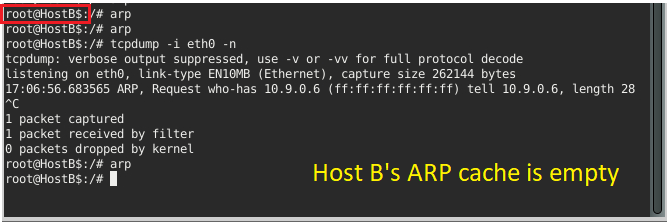
Below is the code snippet to send gratuitous ARP request.



On running the above program, we see that the desired packet is sent out:

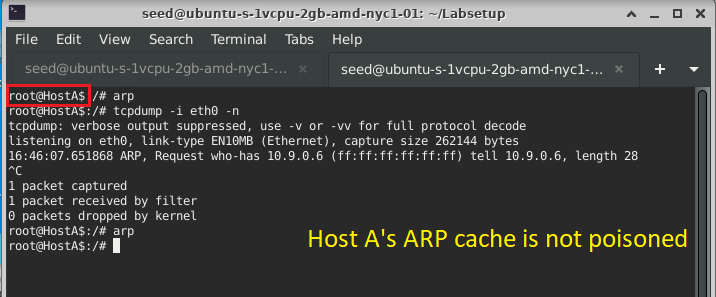


The following shows the ARP cache in machine A before and after running the program:



Scenario 2: B’s IP is not in A’s cache

After deleting the ARP cache, we re-run the above code “task1B.py” from attacker M’s machine and observe that ARP cache in machine A is not poisoned using gratuitous ARP request.



**OBSERVATION:**

In this case, ARP cache of machine A will not be poisoned with mapping of attacker M’s MAC address to B’s IP address because it does not have any entry consisting of B’s IP address for it to be updated nor will it be added to cache generating a reply packet as it is gratuitous ARP. **Linux OS provides some level of security by ignoring unsolicited ARP but, it uses responses to requests from other machines to update its cache and not add entry.**

Therefore, from Scenario 2 in Task 1.B and 1.C, we infer that ARP Poisoning can modify existing pairings in victim’s IP to MAC address table. It cannot add new mappings in ARP cache table.

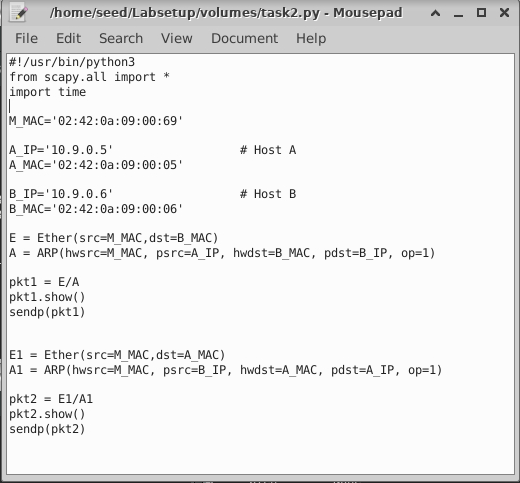
Also, in the above outputs, we see that only machine A’s ARP cache changes and even though machine B received the packet (as the packet is broadcasted), machine B’s ARP cache is empty. This is because the source IP address matches B’s IP address and hence B assumes that the packet was sent by it. The ARP cache consists only of those IP addresses that does not belong to the host itself.

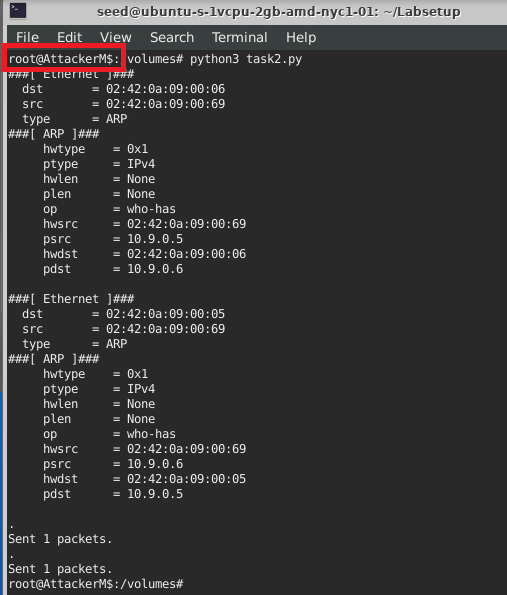
In these 3 ways, we can spoof an ARP packet and perform ARP Cache Poisoning.

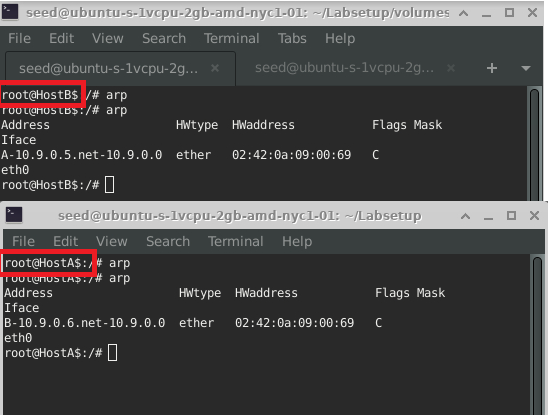
## Task 2: MITM Attack on Telnet using ARP Cache Poisoning

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

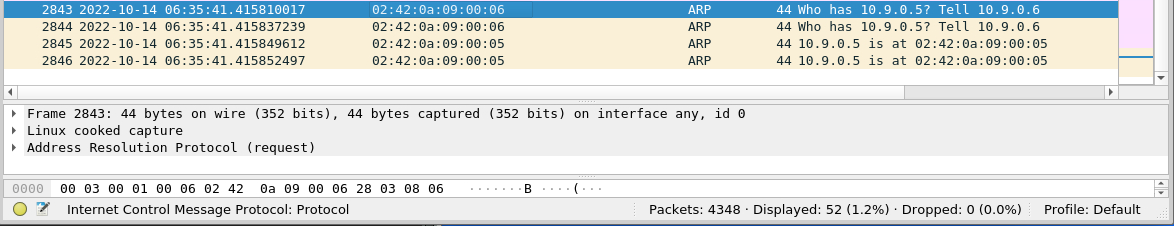
Below is the code to perform ARP Cache Poisoning on A and B, such that in A’s ARP cache, B’s IP address maps to M’s MAC address, and in B’s ARP cache, A’s IP address also maps to M’s MAC address. This code uses the ARP request method to perform ARP Cache Poisoning.





The ARP cache before and after running the code on machine A and B is as shown below:

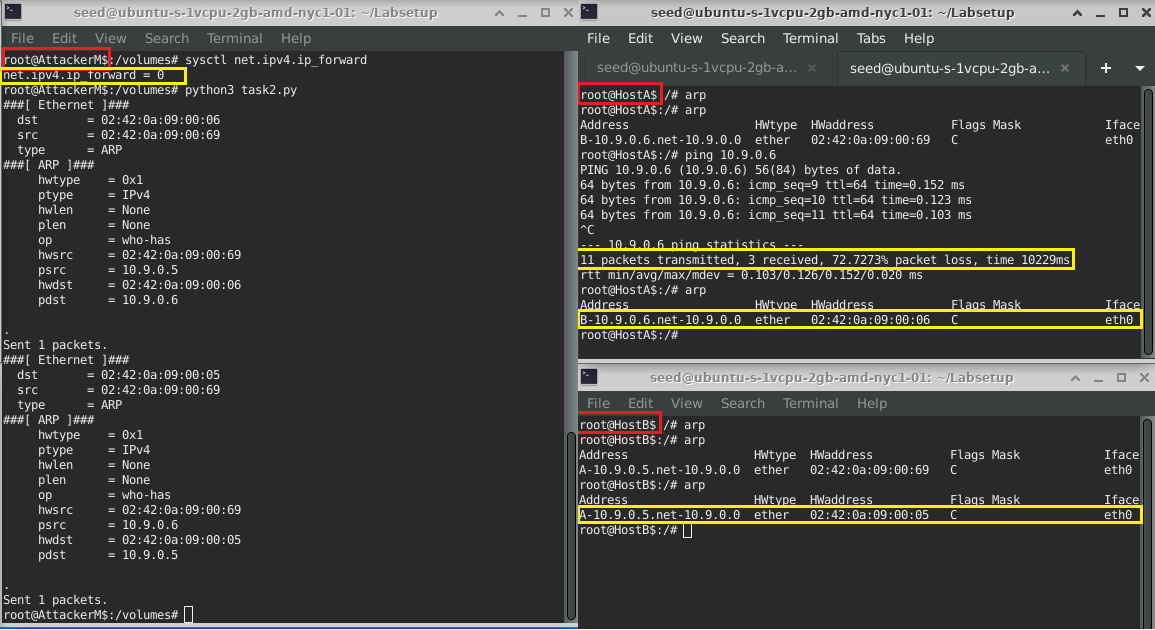
Wireshark capture below shows that the ARP request and replies are generated as follows:



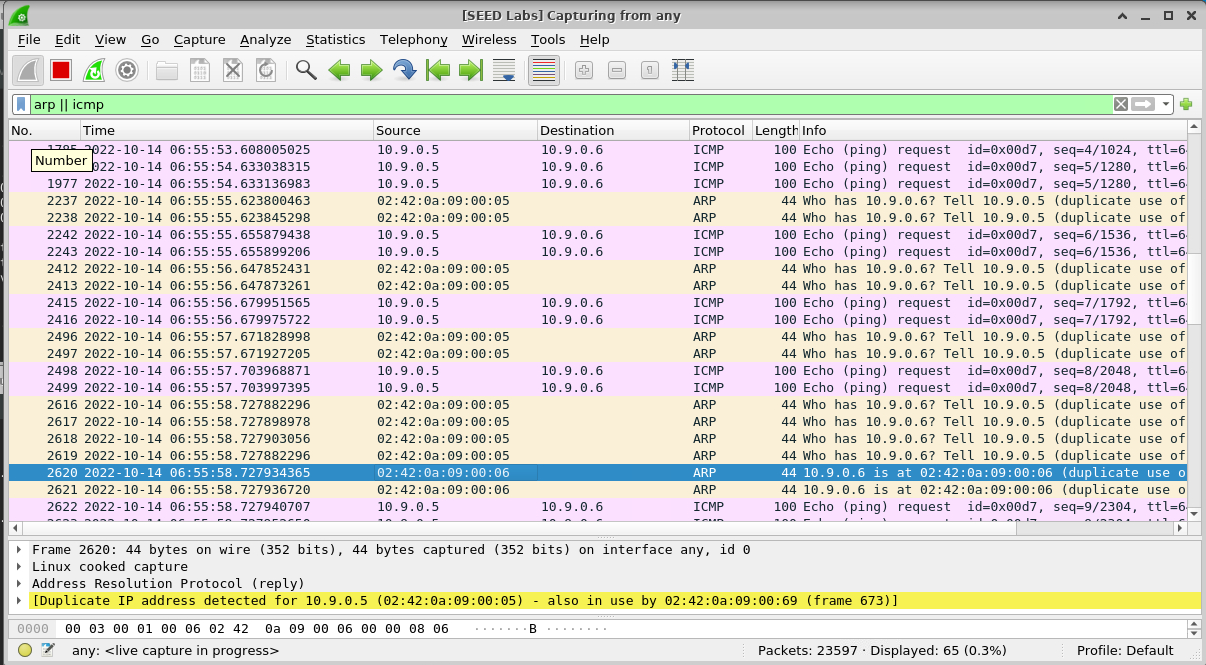
### Task 2.2 (Testing)

We turn off IP forwarding by setting sysctl net.ipv4.ip\_forward=0

After performing the ARP Cache poisoning, we ping from A to B and see the following result. We see that 11 packets are transmitted and only 3 are received.



The Wireshark capture is as follows:

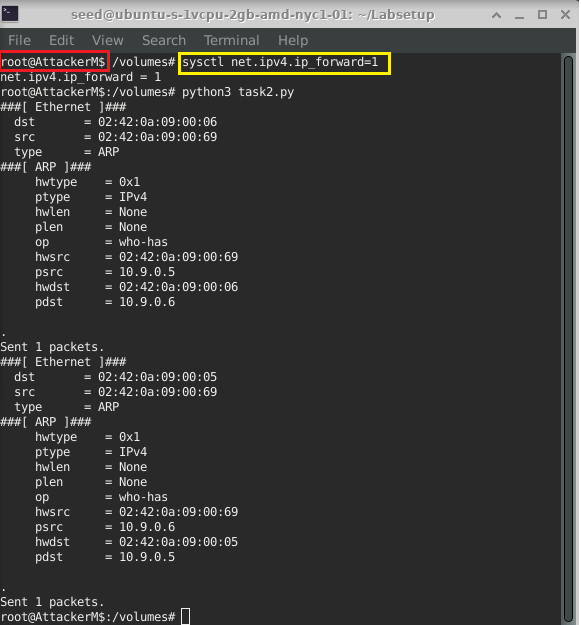


It is observed that initially the ping was unsuccessful as there was no echo reply captured. After some unsuccessful ping requests, an ARP request made from A for B’s MAC address. We see that there was no ARP response seen for some time, and A continuously broadcasted an ARP request for B’s MAC address. In frame number 2620, there was an ARP response from B and after that the Ping was successful.

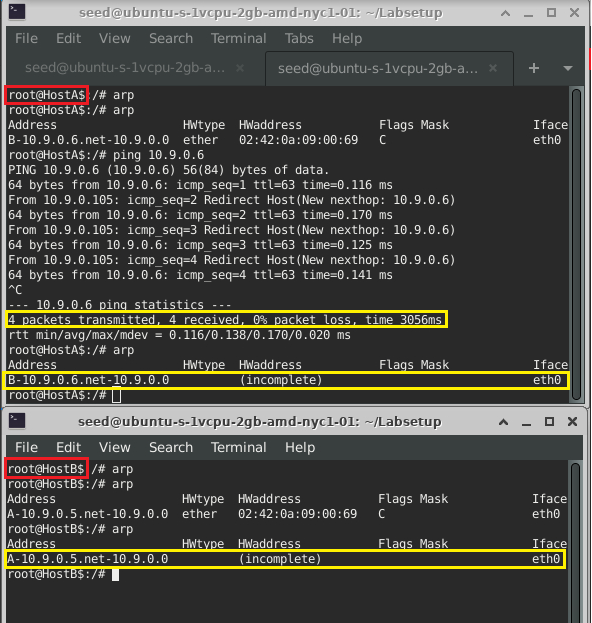
This was because A had M’s MAC address instead of B’s MAC address. This caused all the ping requests to go to M and on receiving these ping requests, the M’s NIC card accepted these packets since they had M’s MAC address on it. When the packet to the Kernel, it realized that the packet’s destination IP address doesn’t match the IP address of the host and hence packet is dropped. This caused the ping requests to be dropped and there was no ping reply from attacker M or machine B (because B never received the packet). After certain unsuccessful ping requests, machine A sent an ARP request and then B’s original MAC address was received, over-riding the effect of our attack of ARP Cache poisoning. After this the ping was successful.

### Task 2.3 (Turn on IP forwarding):

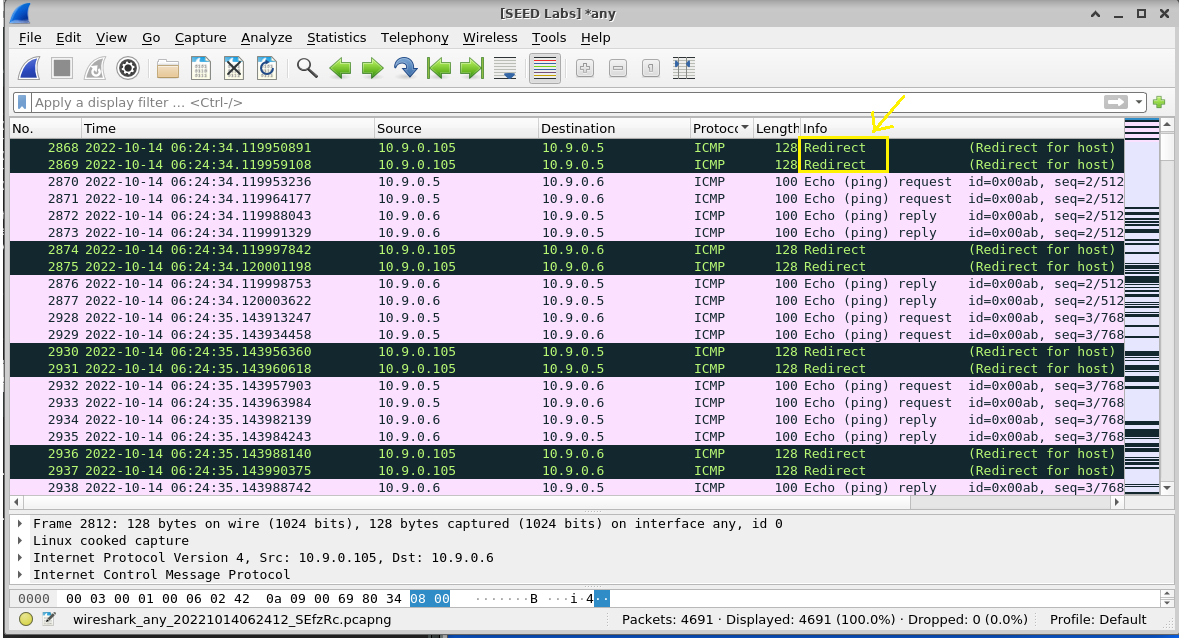
We turn on IP forwarding and perform the attack again.



We ping machine B from machine A and observe that ping is successful with no packet loss.



Below is the Wireshark capture of the ping:



**OBSERVATION:**

This shows that the ping request to machine B from machine A causes an ICMP redirect message from attacker M to machine A. That is, whenever A ping for machine B’s IP address, the packet is received by attacker M. M understands that the packet is not destined to it and forwards to B after sending an ICMP redirect message to A saying that it has redirected the packet. On receiving the packet, B then responds with an echo reply.

Since B’s ARP cache is also corrupted by M, M receives the packet and similarly M sends an ICMP redirect message to B and forwards the packet to A.

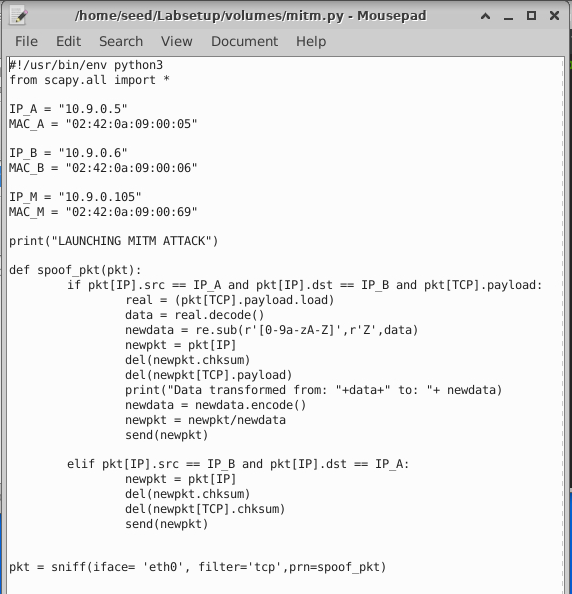
The IP forwarding option enables M to forward the packet instead of dropping the packet.

### Task 2.4 (Launch the MITM attack):

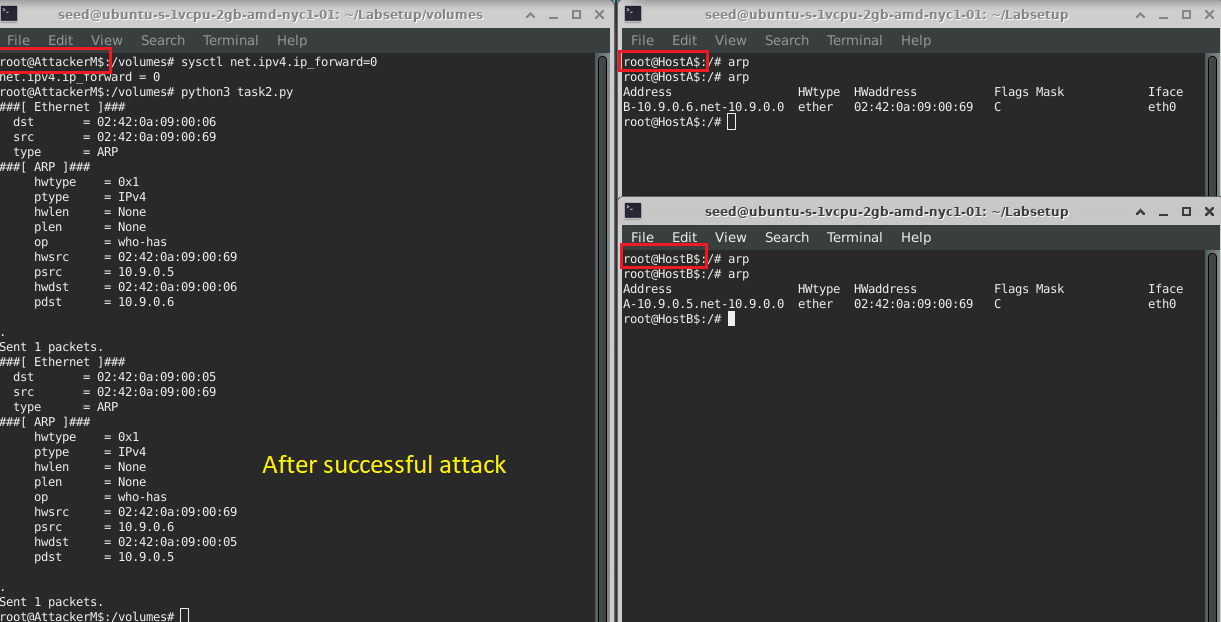
Below is the code to launch an MITM attack after ARP Cache Poisoning on Telnet session. Alphanumeric characters are replaced by Z when typed on machine A.

Sniff() function in python listens to filtered traffic (passed in filter parameter) in the specified interface for an infinite period of time until the user interrupts. We can also filter any packet based on source/destination IP address, port number, protocol etc. Parameter ‘prn’ allows you to pass a function that will be executed for each packet sniffed to do some custom actions on those sniffed packets.

Scapy ‘payload.load’ will provide the original payload data in byte string format. So, we perform decode() to convert to regular string. The ‘re.sub()’ function belongs to the Regular Expressions (re) module in Python that returns a string where all matching occurrences of the specified pattern are replaced by the replace string (here ‘Z’). The modified string is stored into newdata. This modified data is then forwarded with the packet to machine B after byte encoding.



We perform the ARP cache poisoning attack using the same code as used in Task 2.1.

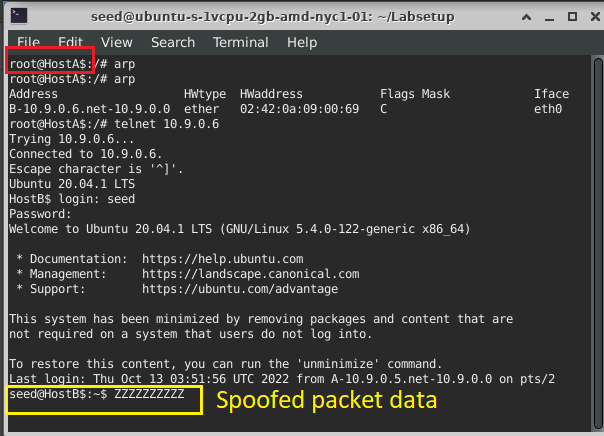


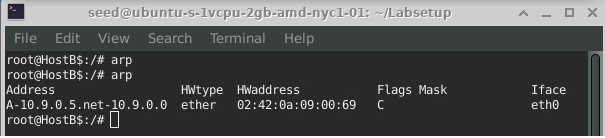
We keep the IP forwarding on in attacker machine M to create a Telnet connection between A to B across machine M. Once the connection is established, we turn off the IP forwarding so that we can manipulate the packet in machine M. In order to change the contents of the packet, we sniff and spoof the packets in the above python code ‘mitm.py’. We spoof a packet that are sent from A to B such that all the alphanumeric characters of the original packet are replaced by Z. For packets from machine B to A (Telnet response), we do not make any changes, so the packet is exactly the same as the original one.

The following is the steps performed in the Attacker M’s terminal:

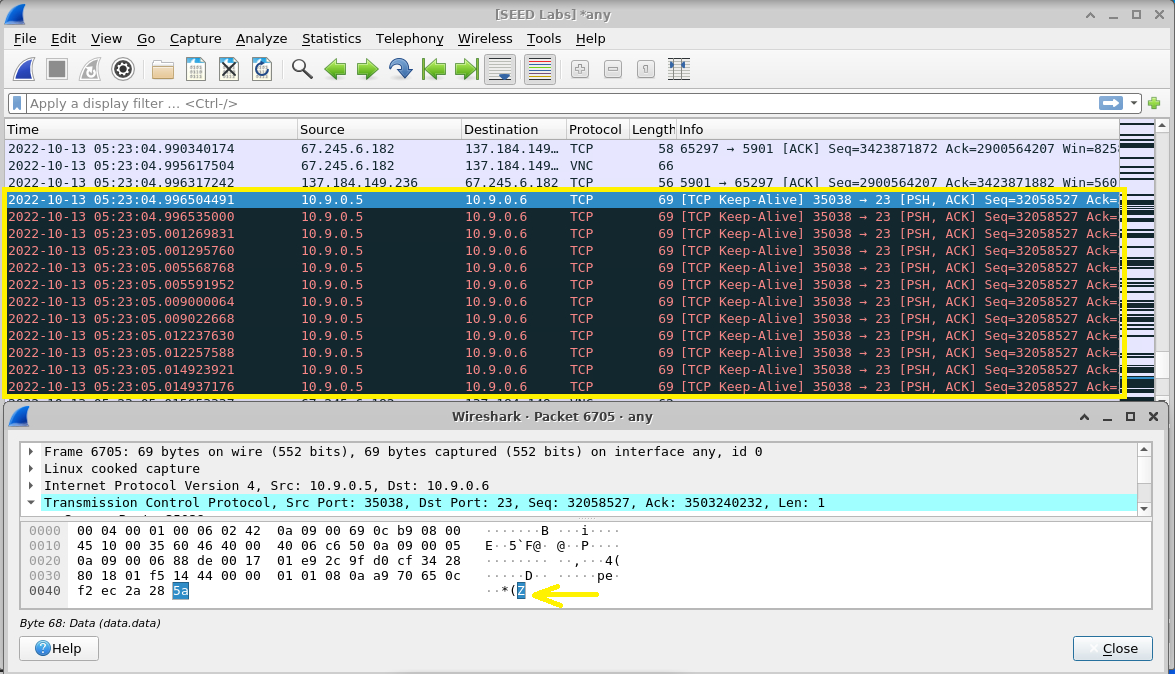


The following is the output on Machine A telnetting to Machine B:





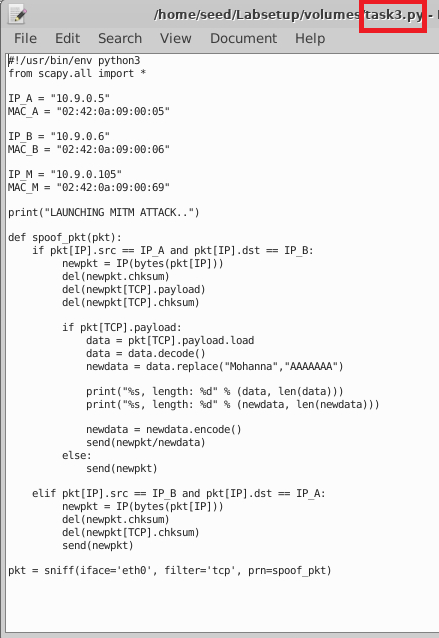
The Wireshark capture while typing in the characters on Terminal A is as following. We can see that TCP packet from machine A (source IP: 10.9.0.5) contains payload data as ‘Z’.



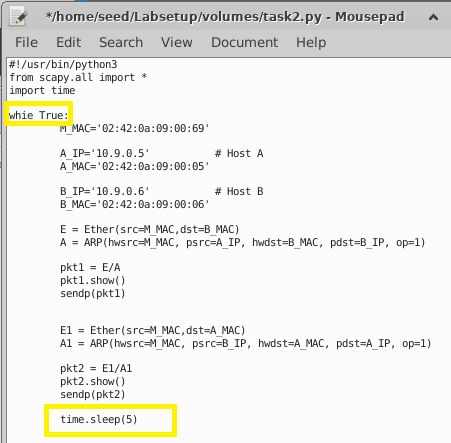
Hence, we are able to perform Man-in-the-middle attack on Telnet session using ARP cache poisoning.

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

The sequence of commands performed now are similar to that of Task 2.4 with only difference of communicating with netcat. In the code below, the attacker sniffs for TCP traffic and if the traffic is from A to B, it replaces the string “Mohanna” with “AAAAAAA”. If the data doesn’t contain “Mohanna”, then there is no change in the payload. This packet is then forwarded to the desired destination. The TCP traffic from B to A remains unchanged.



Also, we have modified the code in ‘task2.py’ to run every 5 seconds so that fake entries are not replaced with real ones.



Before running ‘task3.py’, we perform ARP Cache Poisoning on both machines and establish the netcat session. For this, we run the following commands on the Attacker M’s terminals:

*In terminal 1 of attacker M:*

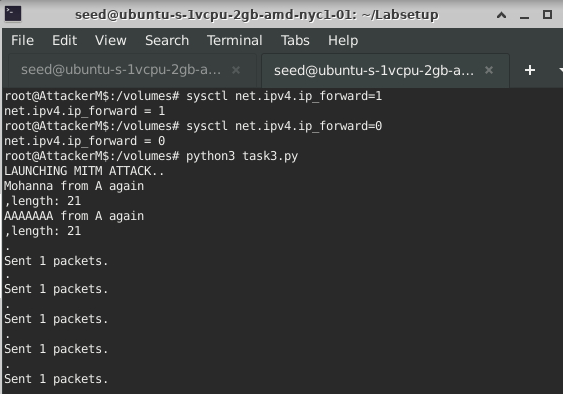
*$* python3 task2.py *(spoofs ARP table every 5 seconds)*

*In terminal 2 of attacker M:*

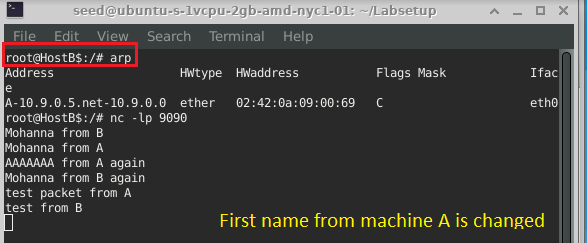
$sysctl net.ipv4.op\_forward=1 *{before establishing netcat session at A and B}*

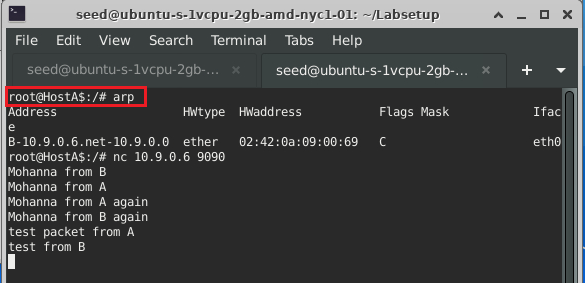
$ sysctl net.ipv4.op\_forward=0 *{after establishing netcat session}*

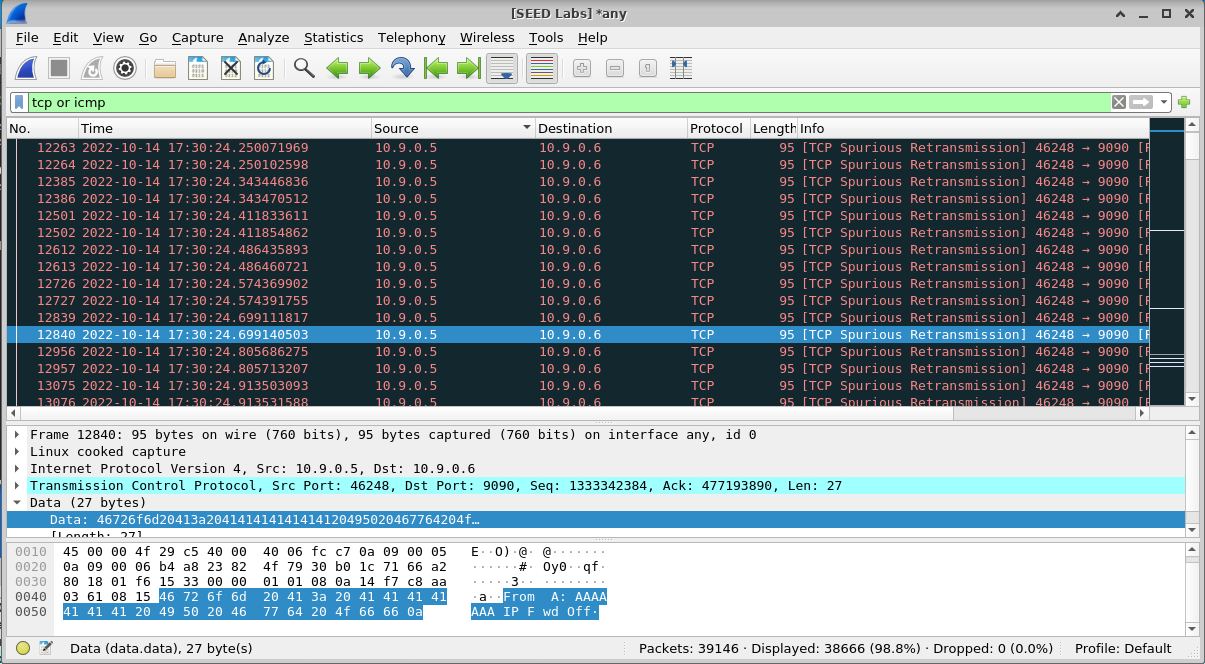
$ python3 task3.py



The following is the output on Terminal of Machine B and A before and after turning on IP forwarding respectively:







**OBSERVATION:**

We observe that both ARP cache is poisoned with M’s MAC address in B’s and A’s IP, respectively. Machine B acts as the server that listens on port 9090 and A acts as client.

When first 2 packets are sent from both machines with IP forwarding on, it is not manipulated because the packet will be forwarded to destined IP address instead of processing it in M (as discussed in task 2.3) and data is sent as it is. After turning off IP forwarding and running the program, we again send a similar string from machine A and see that the string ‘Mohanna’ at the client is replaced by ‘AAAAAAA’ by attacker M and is then forwarded to machine B. Wireshark capture shows one of the case where the TCP packet is

We then send a string containing Mohanna from B to A, and see that it is not changed.

## Task 4: Write-Up:

Address Resolution Protocol (ARP) maps IP address and MAC address in a table in memory called ARP cache. The entries in this table are dynamically added and removed based on the ARP requests and replies.

An ARP spoofing is aimed to associate attacker's MAC address with the IP address of another host, causing any traffic sent for that IP address to be directed to the attacker instead. There are 2 stages or types in ARP attack. ARP spoofing is sending fake ARP packets to link an attacker's MAC address with an IP of existing machine. Whereas, ARP poisoning is contagion spread of the false IP-MAC mapping once ARP spoofing is successful. Successful ARP attack may allow an attacker to intercept data frames, modify the traffic, or can even terminate all traffic. It requires attacker to have direct access to the local network segment (that use ARP) to be attacked. ARP spoofing and poisoning are closely related. Often ARP attack is used to kick-off other attacks including denial of service, man in the middle, or session hijacking.

In this lab, we have performed MITM attack after the ARP cache is poisoned and reported all the observations for each task respectively. We noted that there were some ARP poisoning attacks that failed.

There was one interesting thing that was noted in this lab. We know ARP is stateless protocol and hosts will automatically cache any ARP replies they receive, regardless of whether network hosts requested them. But Linux OS does not trust unsolicited ARP replies which avoids adding any spoofed entry into the ARP cache, but only update the existing entries. This provides security in the system to certain extent by the operating system.

We also noted that in some cases ARP cache poisoning won’t have a lasting impact (scenario where IP forwarding is not enabled). ARP cache entries are cached for a few seconds or minutes on end hosts. As soon as an attacker stops actively poisoning the tables, the corrupted entries will expire and proper flow of traffic will soon resume. Therefore, ARP Poisoning will not leave a permanent foothold on victims but can be used to start other attacks.

When IP forwarding is enabled, if a datagram is received which is not for the local system, the datagram will be forwarded to a different system. When IP forwarding is disabled, datagram will be dropped if it is not intended for the local system. It is recommended that unless the remote host is a router, we need to disable IP forwarding. Here we enable and disable IP forwarding in attacker’s machine after successful spoofing of ARP tables in A and B. This way any attacker can exploit this to route packets through the host and potentially bypass some firewalls, routers or NAC filtering.

While testing the python code for each task, it was quite a tedious task to delete each entry in ARP cache in all the systems one by one using the command ‘arp -d {IP}’. An alternate command ‘ip -s -s neigh flush all’ was helpful in deleting all the mappings from ARP cache table, i.e., which is similar to flushing the ARP table when it corrupted as that can be hectic and troublesome because it can stop a device from communicating on the network.

Even though ARP poisoning attacks can be prevented using Static ARP Tables, Dynamic ARP Inspection( DAI) switch security, use VPN, monitor using Wireshark, ARPWatch and so on, it is still a threat that network organizations and administrators need to address.