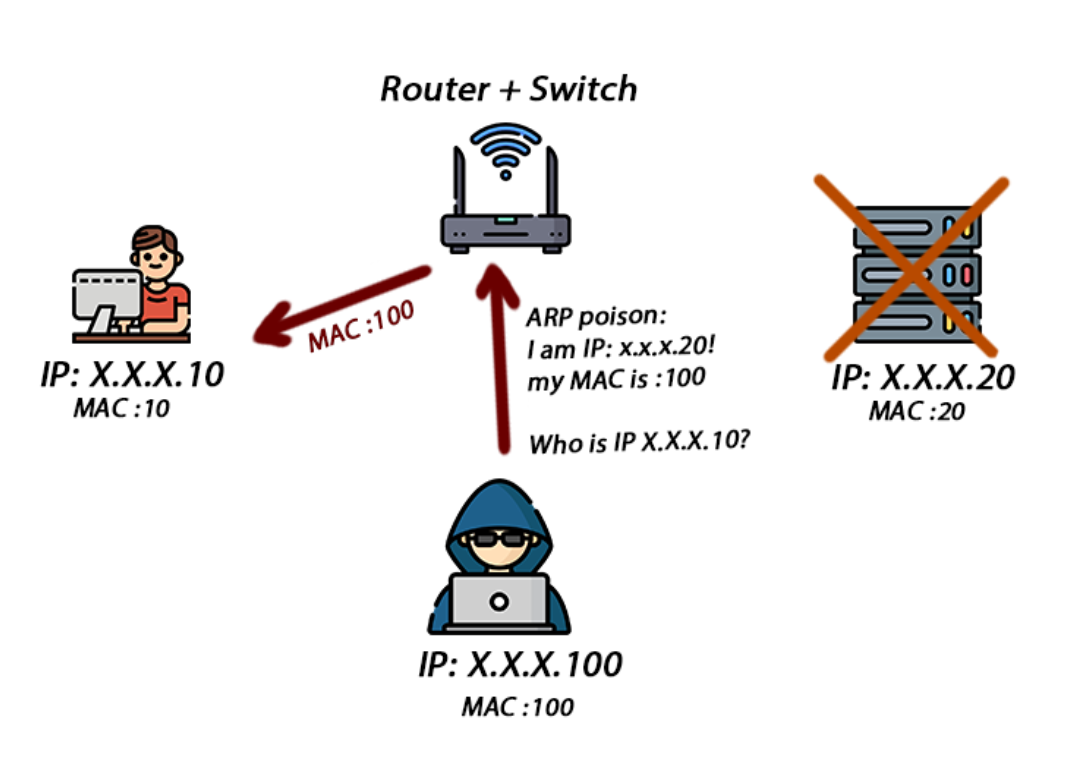
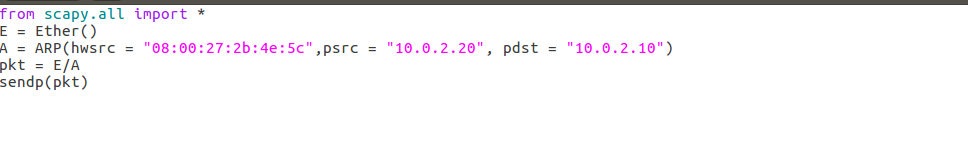
**Task 1: ARP Cache Poisoning**

In this task we will use “scapy” to send out arp packets from our attacking machine in order to poison our victim’s arp table.  
We want the victim to save our malicious MAC address in tandem with the IP address of the server. That way every time the victim client will search for the server, they will find our attack machine Instead.

**Task 1A: ARP REQUEST**

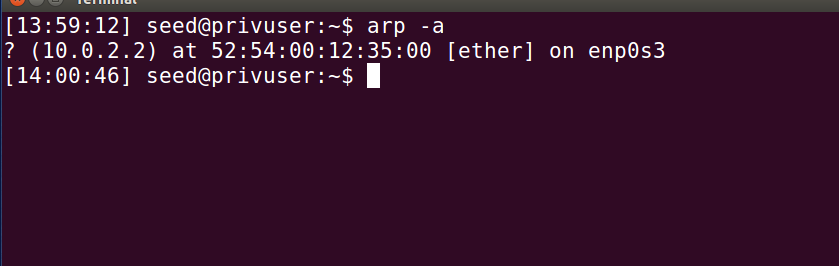


here is our first code

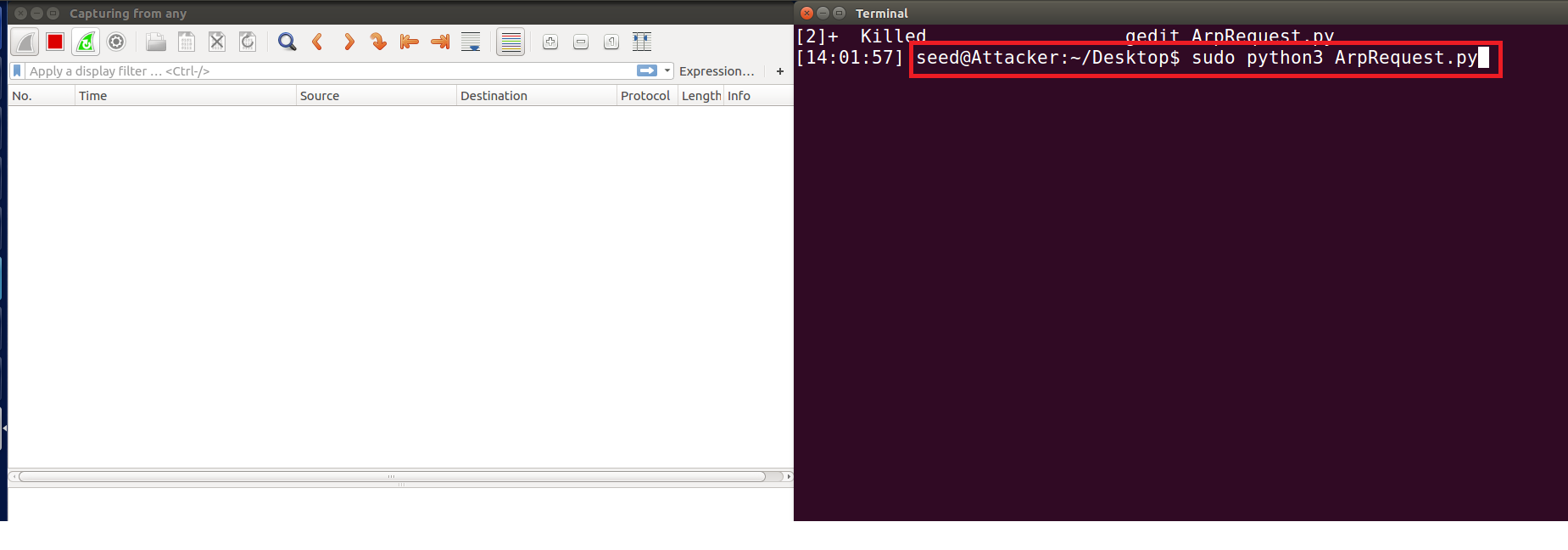


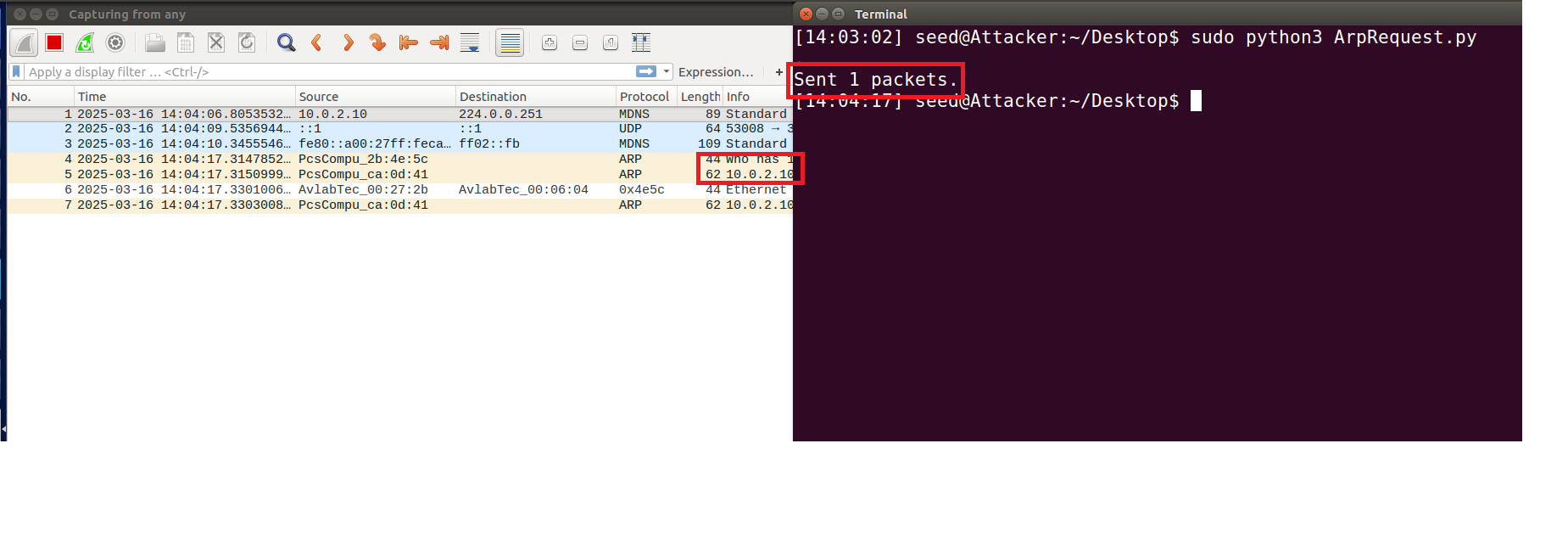
First, we set the source mac address, the ip source we want to poison, and the destination.  
Then, we combine our packet with the arp payload, and send them.

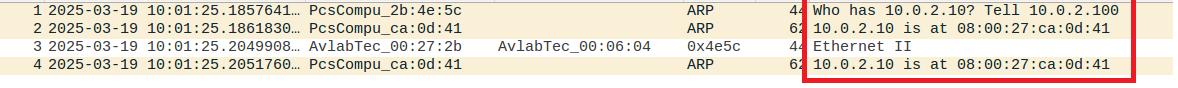
We use arp -a to check what is on our users arp table



as we can see, right now we only have 10.0.2.2, which is our gateway.  
we then run our script, while also keeping wireshark sniffing.







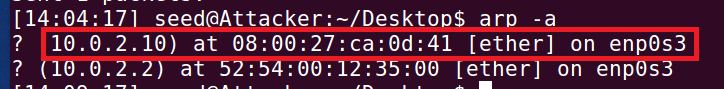
First, we can see that an arp request was sent, and a reply was sent to 10.0.2.100 but we can also see the attacker sent an arp request for .10, which might be less good for us.

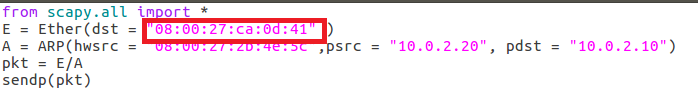
seemingly successful, we go to check the arp table in the user



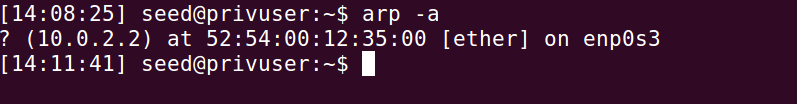
While we did manage to poison the arp table, we can see we have also given ourselves away.  
were going to attempt to fix this by setting the target mac via the Ether() part of the packet, so that we do not send an ARP request.

First, we check our own arp table to locate the targets mac

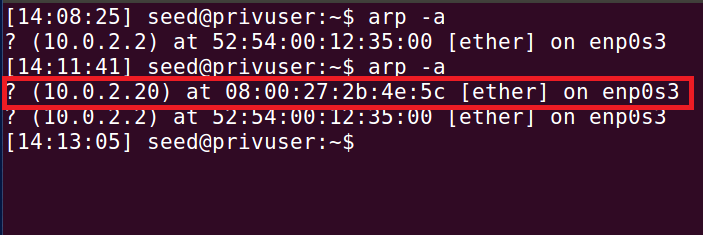


We now know .10 is at 08:00:27:ca:0d:41 so we edit our code appropriately  


We hope that by adding the dst, we do not send an arp request, thus not giving ourselves away.

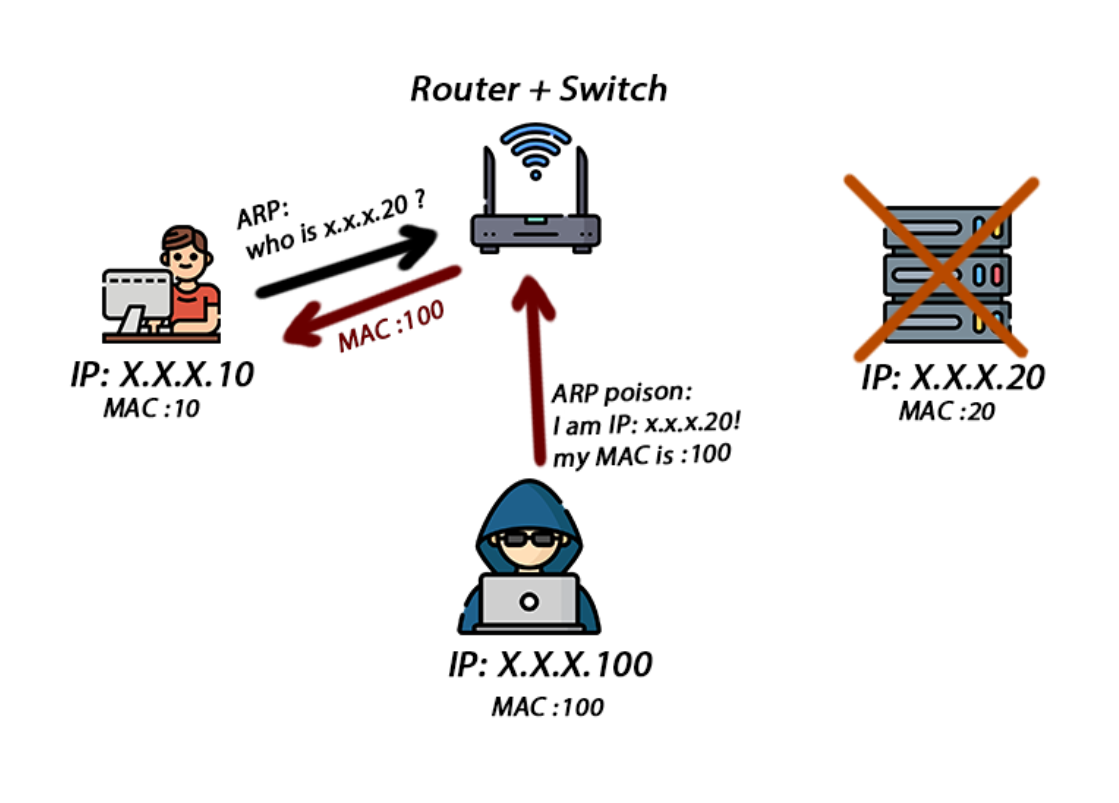
We now test it. after making sure our arp table in user has been reset  


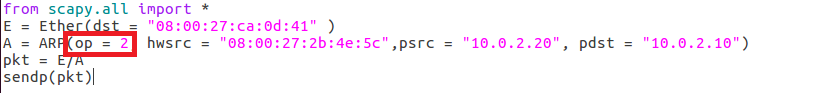
Here is proof we've reset our network, and reset the arp table.

We then send our our packet, and take a look at the arp table in our user  


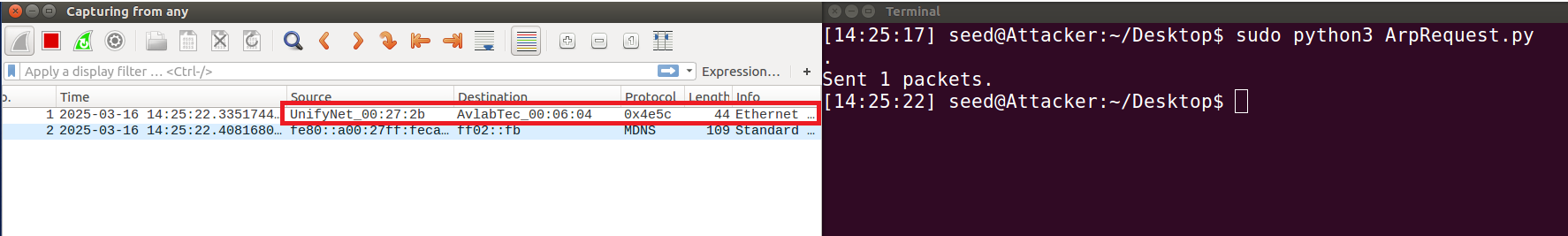
Thus, we can see that after editing our code so that our system does not send out an arp request, we successfully poison our target without giving ourselves away.

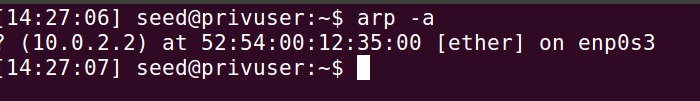
**Task 1B:ARP REPLY**



Taking into account what we learned, we attempt to do the same, but as an arp reply instead.   
here is our first code:

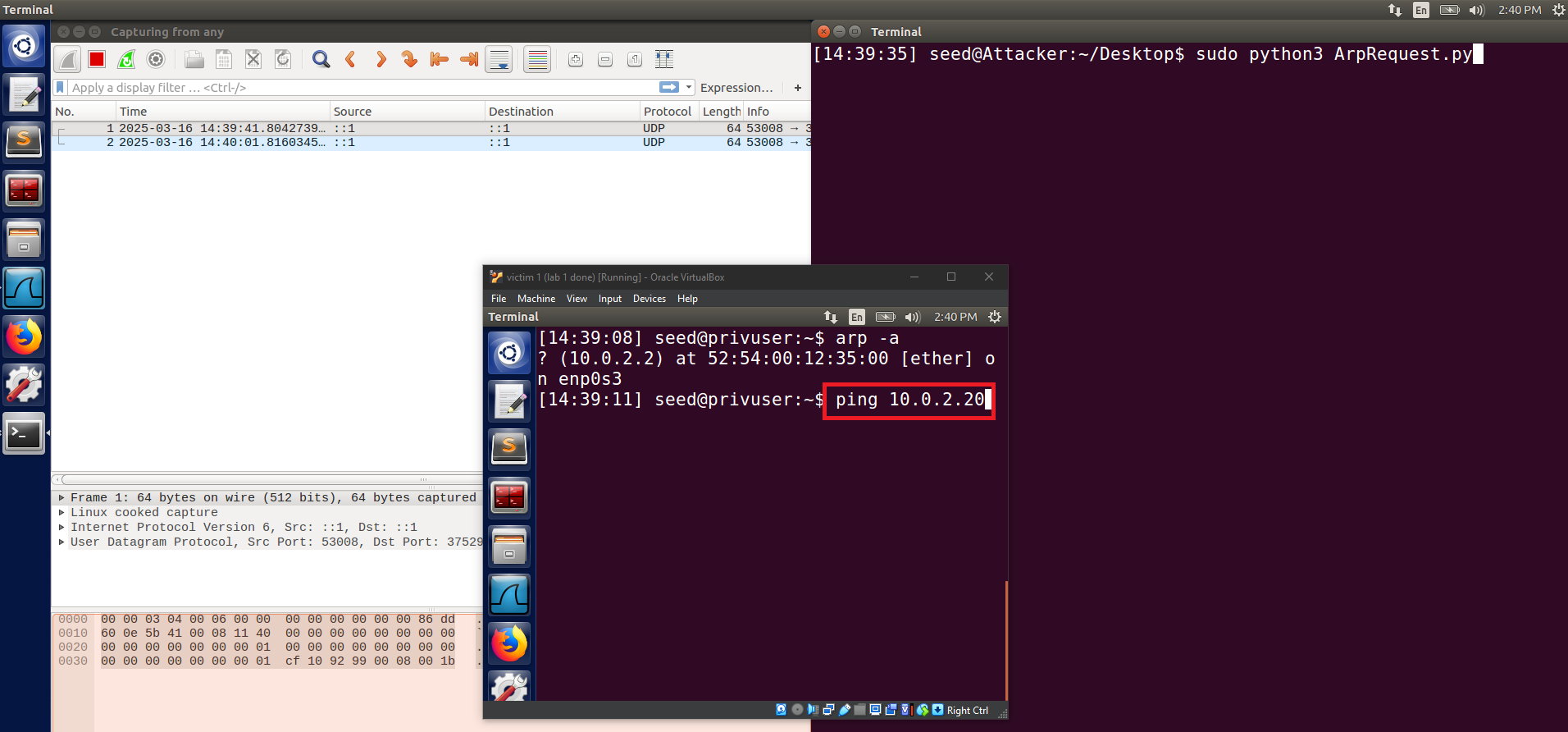
The only real change is setting the operation within our ARP packet to 2, meaning it's an ARP reply, rather than a request.

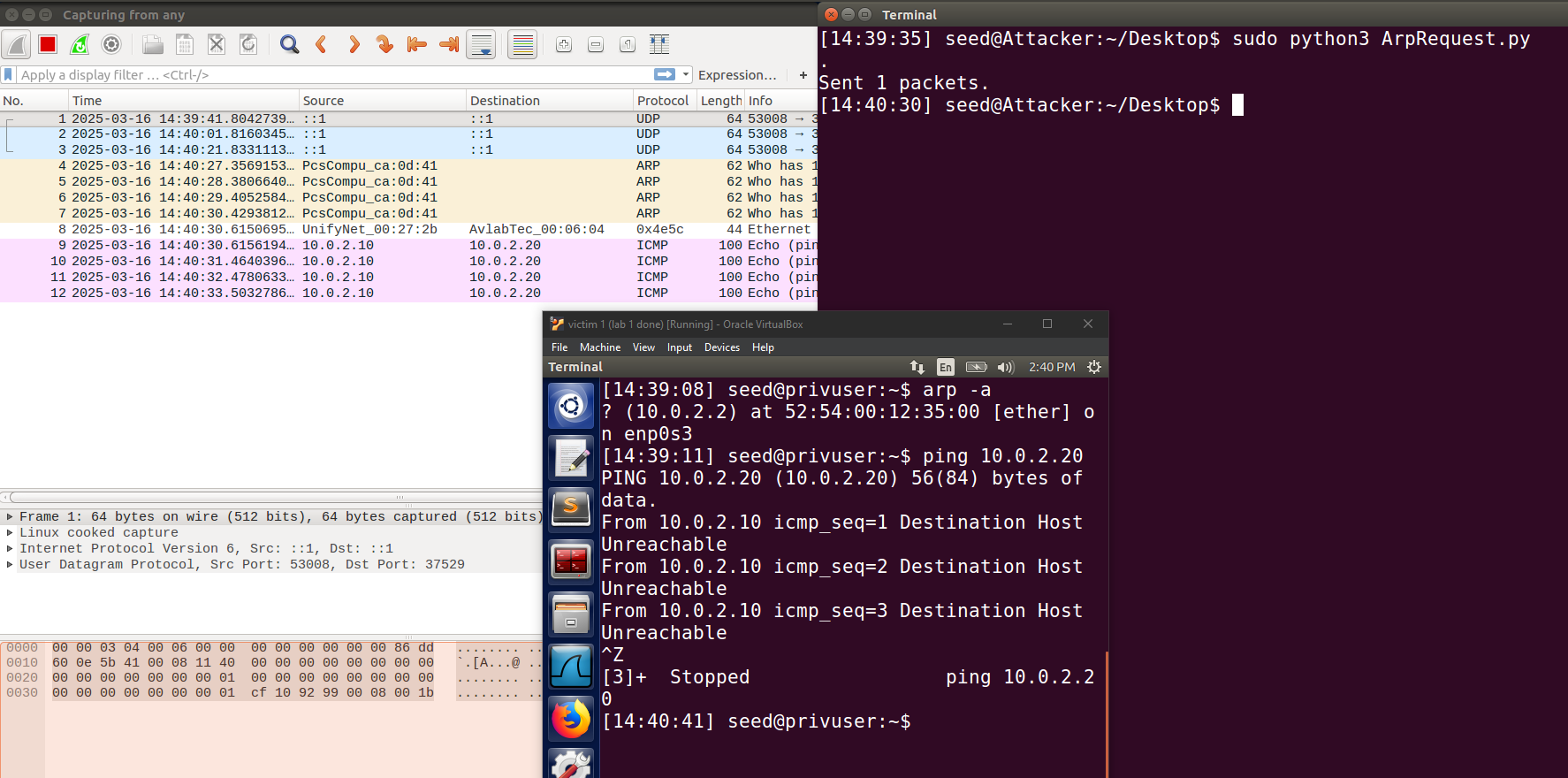
We then run our script, keeping wireshark open, after having reset our arp table in our user once more  
  
We can see a reply was sent, so we go and check the users arp table

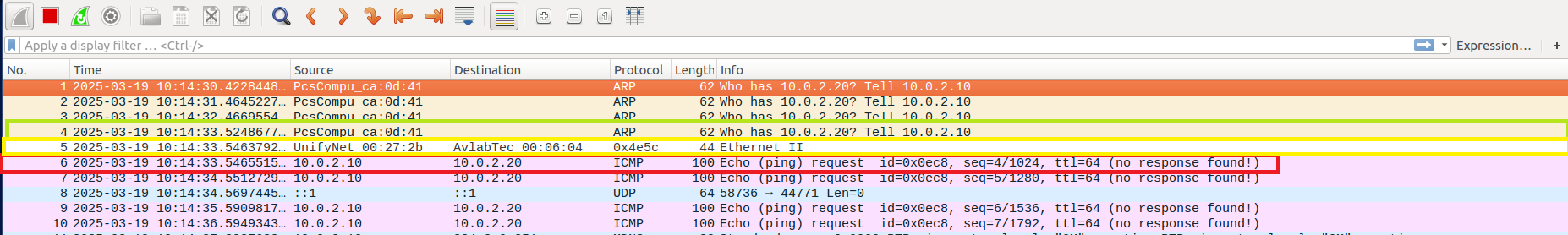


as we can see, we've failed to poison the user as we hoped to do. Our hypothesis is that the reason for this is the fact that no request was sent out, meaning the user simply ignored our reply, as no request was sent.

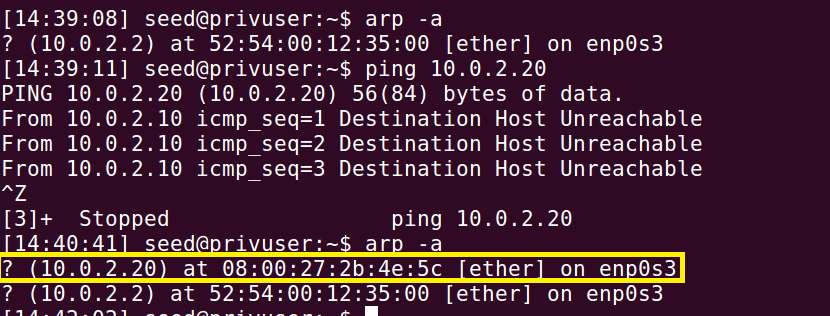
We test our theory by sending a ping to 10.0.2.20, the server whose mac was spoofed, from 10.0.2.10, our user, and then immediately sending out our arp reply, to see if that has any effect.





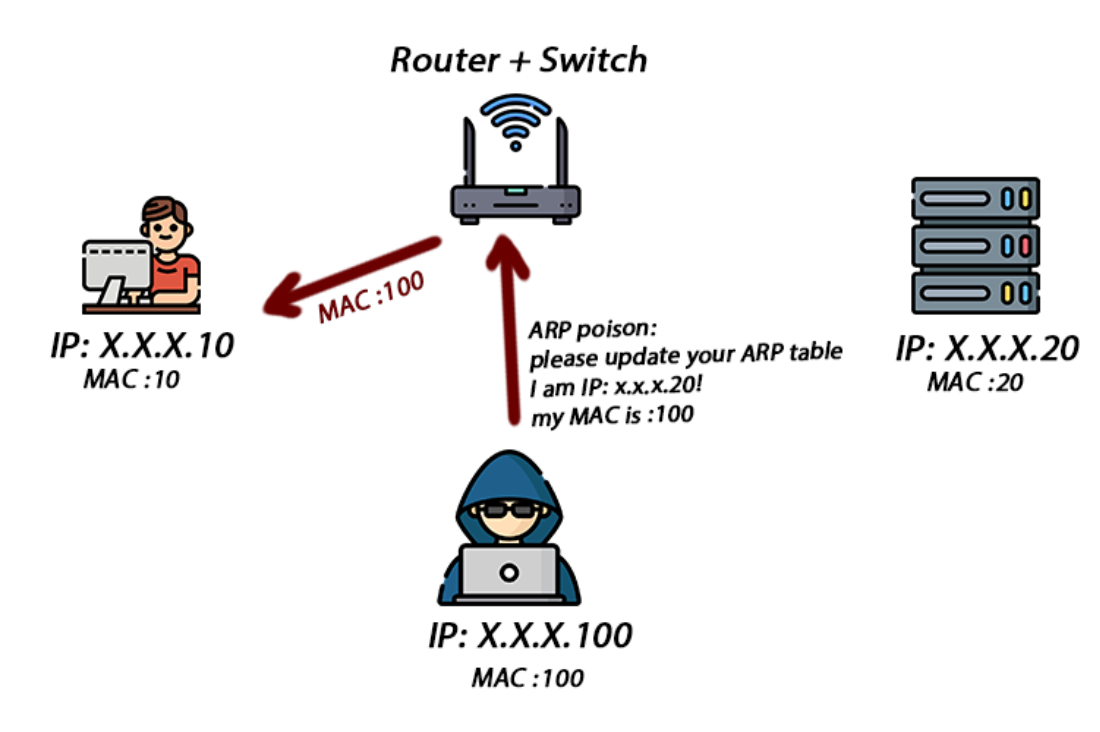


As we can see via wireshark, it seems that after attempting to ping .20 (colored green), .10 began sending arp requests, we then sent our arp-reply (colored yellow) and we can assume by the ICMP pings (colored red) that we have successfully poisoned .10 but well still look at its arp table, just to be sure.

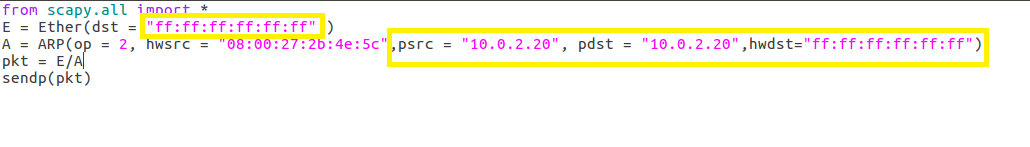


And so, we can see that we have successfully managed to now poison it.

**Task 1C: ARP GRATUITOUS MESSAGE**

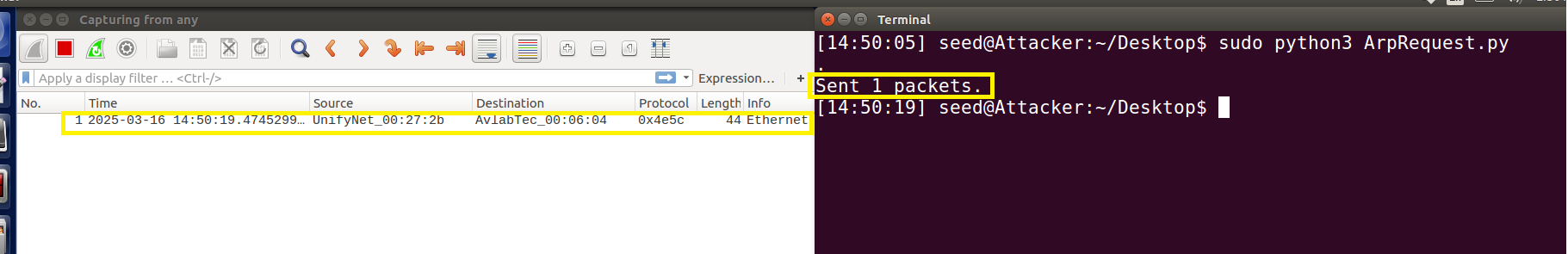


First, we edit our script so that it answers the requirements to be a gratuitous packet.

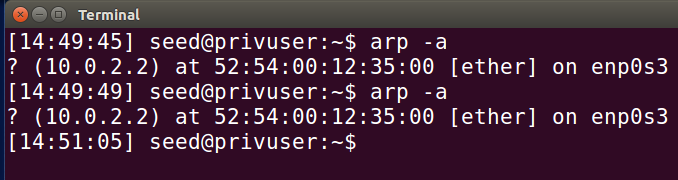


Changing the dst in ether to broadcast, setting psrc and pdst to the ip we want to spoof the content for, and set the destination in the ARP to broadcast, as is required.

after making sure our users arp table is empty, we send our packet with wireshark open:

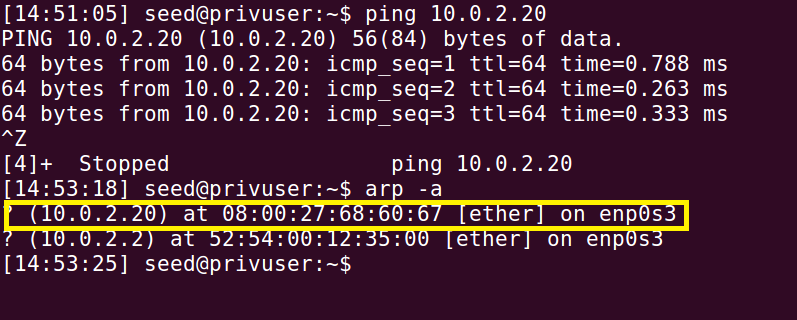


Everything seems to work as intended, as we know no reply is supposed to be sent from our user.



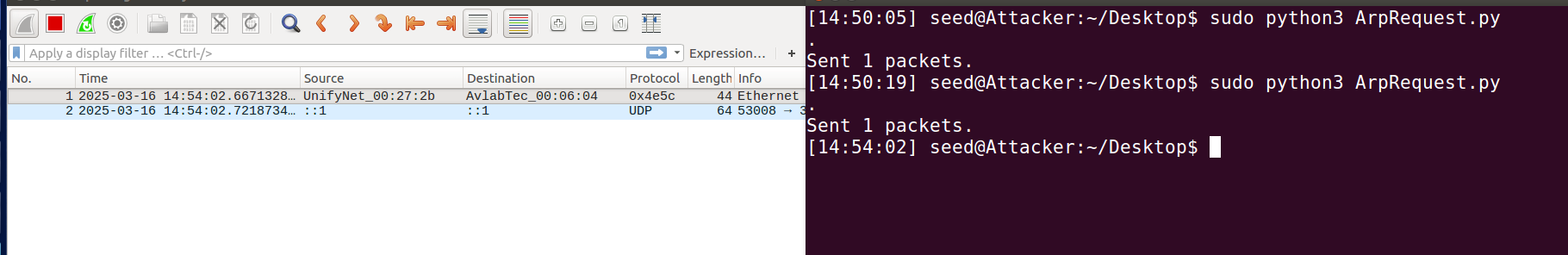
But as we can see, we again failed to poison it as intended.

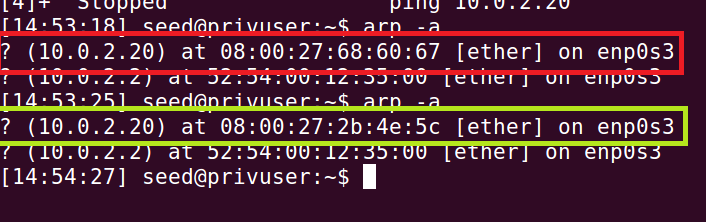
Our immediate thought is that for the broadcast to work, there needs to be an existing address for 10.0.2.20 on our user. so we quickly turn the server on, and ping it from our user



After a quick ping, we now attempt the broadcast again.

and then we check the arp table on our user





Marked in red: is the mac in the arp table before the message was sent, and marked in yellowish-green is the mac after the message was sent.

Thus we can see that after sending our broadcast, we have successfully managed to poison the table of our user, with there being no clear, immediate signs our untrained eyes can see on WireShark.

**Task summary:**

* We managed to successfully complete every part of the requirements.
* proof is presented in the screenshots
* We learned how to perform different vectors of ARP poisoning, without giving ourselves (the attacker) away.
* The task went right as we expected, with any hurdle we found being quickly solved with a quick brain-storm.

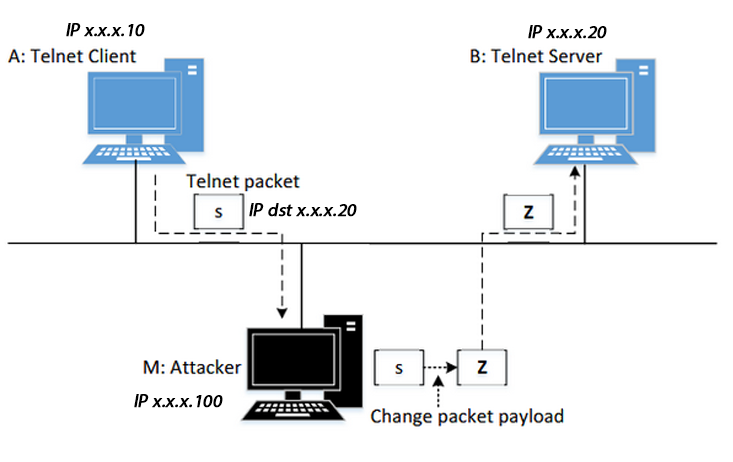
The only hurdle was us revealing our mac, thus giving ourselves away in case the user checked his arp table.

* task 1 main hurdle:

our machine kept sending arp requests to find the client at .10, which revealed our mischievous acts.  
We fixed this by adding the MAC address of .10 to our packet, so we don't need to search for it.

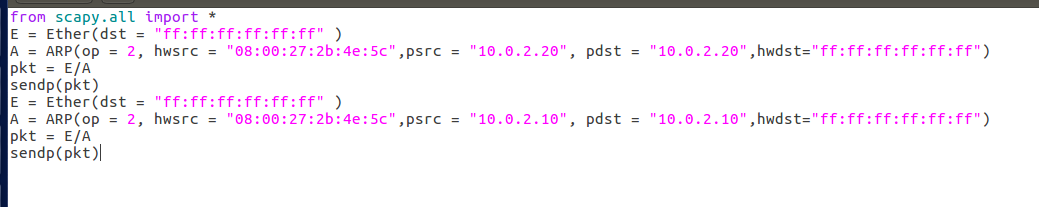
**Task 2: MITM Attack on Telnet using ARP Cache Poisoning**

In this task we will use our previous abilities from task 1, to arp poison both the client and the server machines. Then we will interfere with a Telnet session providing a proof of concept by changing packet information without any of the sides knowing about our attack over the session.



Step 1:

First, reusing our gratuitous packet script from task 1C, as we saw we can use it to poison existing addresses, we make a new script



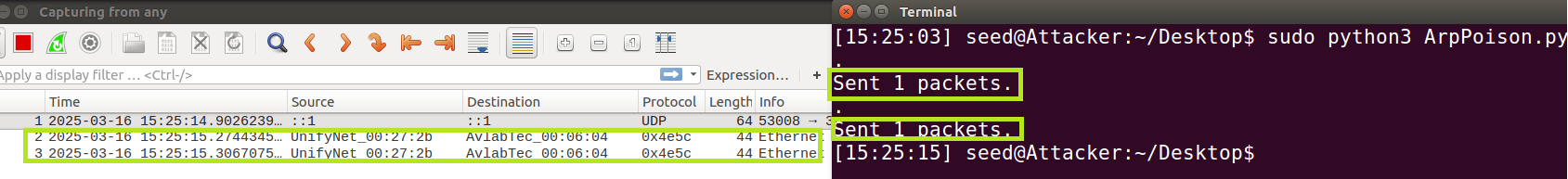
In this script, we send two packets, in the hopes of poisoning both machines.

We start a telnet connection between our user and our server



We can see we successfully did so!

Now, we run our script

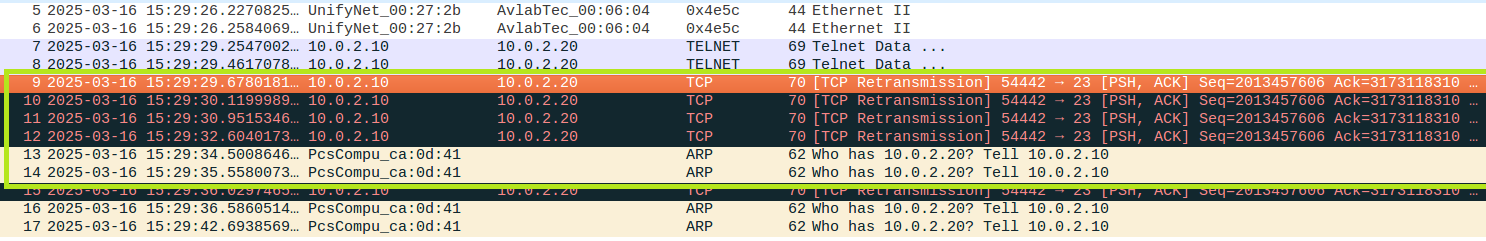


marked in green we can see the broadcast that we sent.

Step 2:

After running our script, we attempt to ping 10.0.2.20 and observe wireshark for the results.

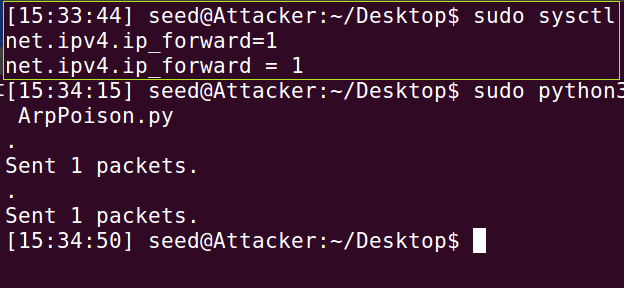
As we type in our user who is connected to the server, we can see that it attempts to transmit the data, fails, and after a few failures it sends a new arp request over the network (marked in green).



We assume the reason for this is the lack of communication between .10 and .20 as each letter is attempted to be sent, so our guess is that by turning on IP forwarding, we will manage to communicate between .10 and .20 without ruining the poisoned ARP.

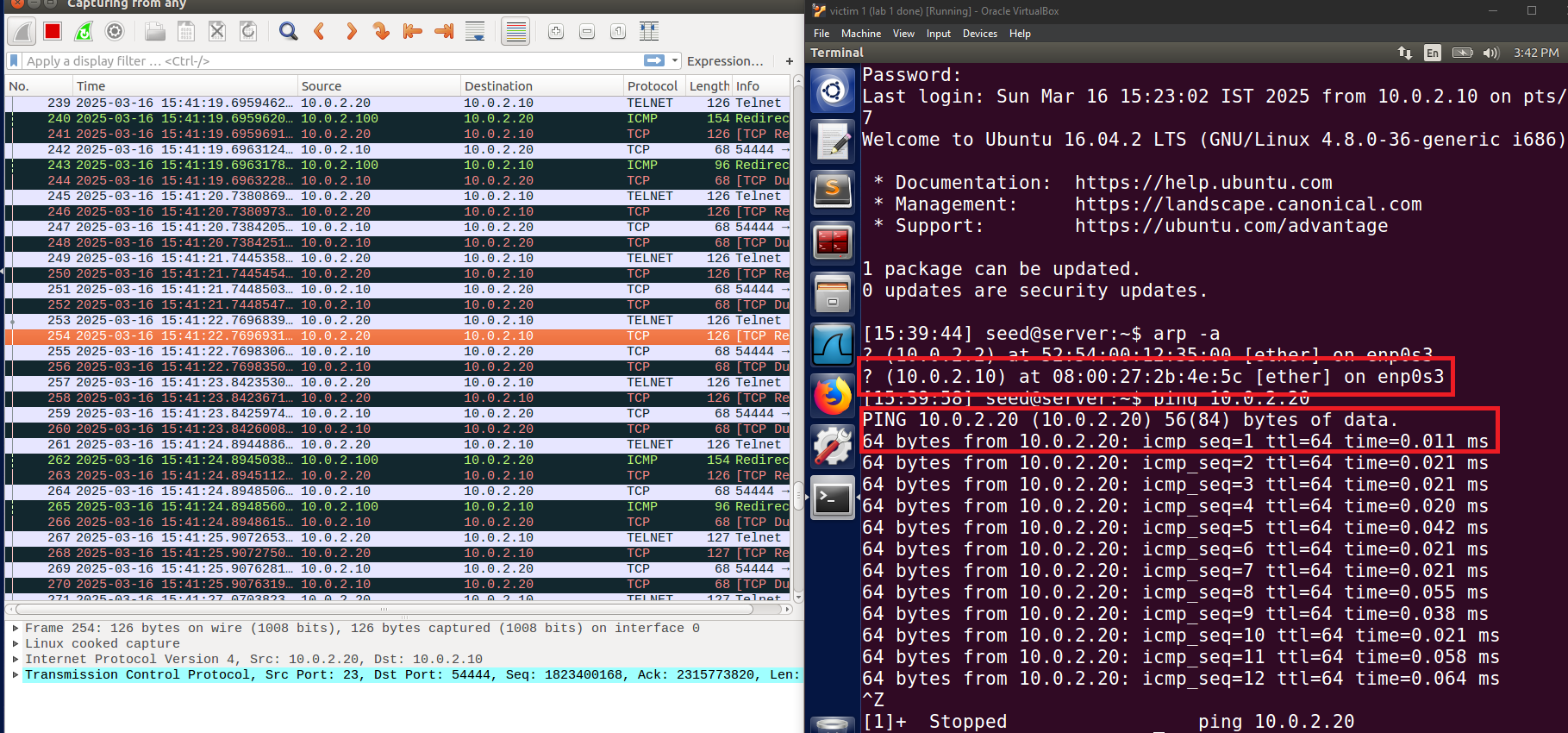
Step 3:

thus we set the system to forward the messages, and run our script again



where everything seems to work correctly.

we now attempt to use the telnet connection:

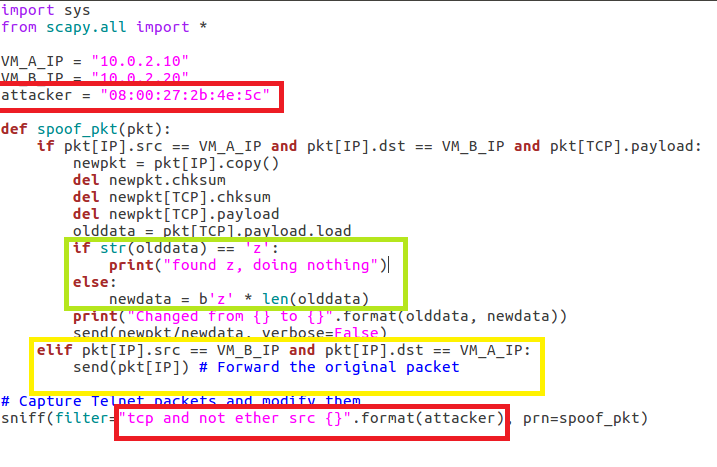


as we can see, everything is working as required, and the pings are being redirected.

Step 4, MITM attack

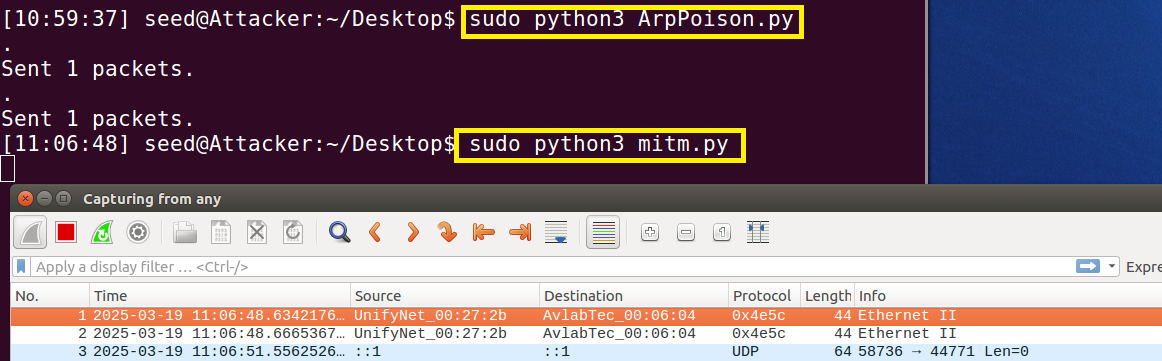
First, we make a telnet connection between our user and our server, this is before we poison the ARP  


as we can see in red, we successfully make a telnet connection, and marked in green is the original mac for .10.  
While the lab calls for us to once more observe the changes with port-forwarding on and off, we have done so in step 3, so we forgo this.  
we do note, in case the meaning was to run our program while the port forwarding is on, then in that case there would be, what is essentially a race between our program, and our OS, to forward the packet, with the first forwarded packet “winning” the race, and being the deciding factor. most of the time the OS will be faster.

we then edit the given sniff and spoof program to our needs, as follows.  


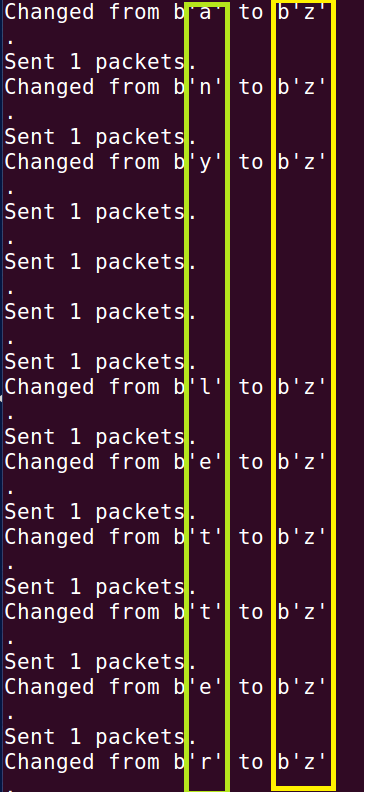
marked in red we can see our filter, set to ignore all packages sent from our attackers mac, thus stopping us from entering an infinite loop.  
marked in green we see our code for changing any input that isn't z to z, and letting us know if it already was z.  
marked in yellow is our code to simply forward any packet that is sent from b (the server) to a (our user).

we now run the code:



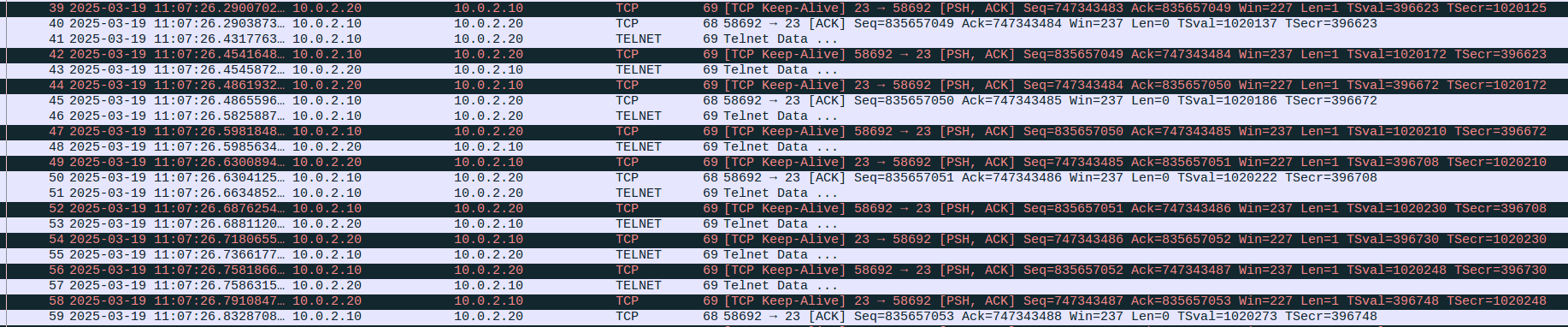
and we attempt to type in our user, looking at the result:





wדייs.

A quick look at wireshark also shows the data being forwarded.

in this very small screenshot we show that the packets are passing:

thus concludes task 2.

Task summary:

* We managed to complete the task and all of its requisites.
* proof is presented in the screenshots
* In this task we learned how to use IP forwarding on our machines, how to poison a telnet connection, and how to edit the data that is sent in the TCP packets of the telnet connection so that the output is whatever we desire.  
  We of course also learned how to write a sniff and spoff script.
* everything we went through fit what we expected, and our theory, with the only issue being our main hurdle, which was an infinite loop, which is expanded upon in the next point.
* task 2 main hurdle:

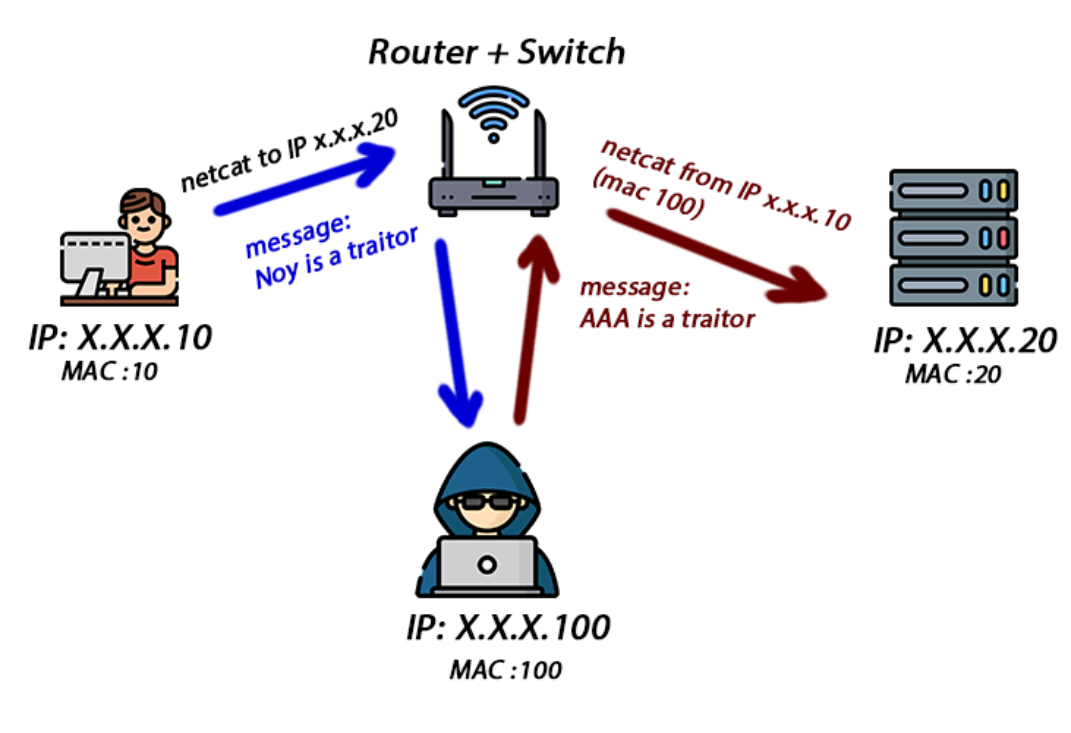
For a while, we couldn't figure out how to stop our machine from running in an infinite loop, as it kept working on already edited packets.

We fixed this by filtering out all packets with our attackers MAC address.

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

In this task we will try to do a very similar attack to task 2, but this time instead of being a MITM over a telnet session, we will be a MITM over a “netcat” session.

While every message from the client to the server should be shown as inputted by the user, we provide proof of being able to change the contents of the message and passing a different one. (censoring our name and changing it to a string of A’s..)



First, in our attempts to run the MITM attack, we found that our poisoning kept breaking due to, what we can only assume, is ARP timeout.  
Thus, we decided to run our poisoning script on loop.



Here we can see our poisoning gratuitous message script running on loop, with a 2 second sleep timer.

next, here is our code:Explanations in next page:

Prevents the processing packets sent by the attacker itself.

if: Source MAC address = Attacker: return;

without processing to avoid an infinite loop where the attacker would capture and process its own spoofed packets.

Check if the string “noy” appears in the packet data; and change it to “AAA” - same length to keep the same TCP sequence numbers.

Creating new spoofed packets with the modified content while maintaining all the original TCP header values but replacing the data (payload) to the modified data.

Handles traffic from server from user; create new identical packet to the original enduring the responses from the server reach the user unchanged.

Handled non-TCP/TCP without data; ensuring all traffic continues to flow

We want the sniffer to capture only packets that meet specific criteria:

* TCP packets
* port 9090
* store = 0; don't save packets in memory (process them immediately)

Launching the attack:

First, we turn on the ip forwarding in our attacker to run the arp\_poison script, and connect our user to the server using “netcat”



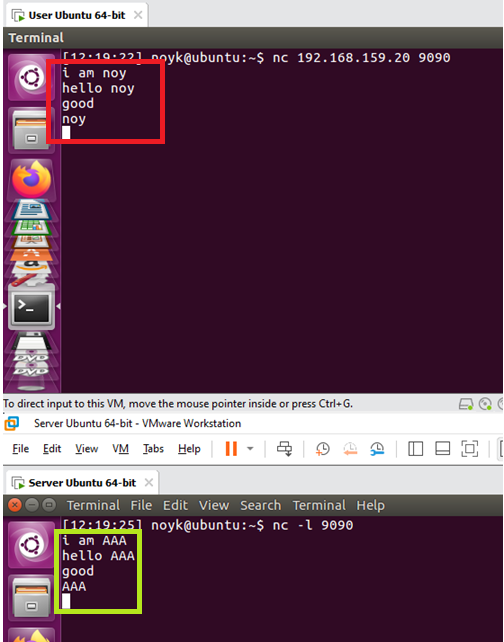


note - the time is later than our next images, as we had to retake this screenshot.

next, we turn off the ip forwarding, run our script, send some messages, and look at the results:

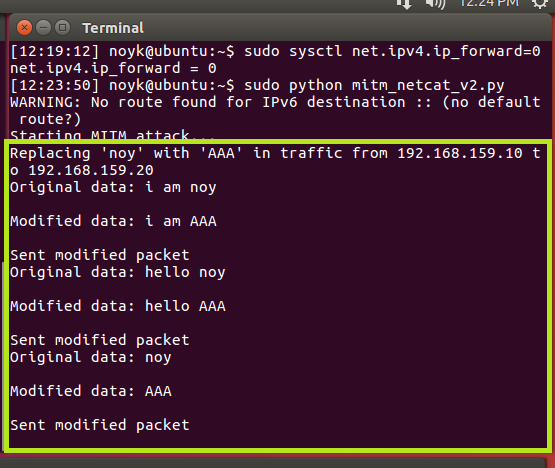
first, we turn off the ip forwarding  


next, we send a message using “netcat”, as our script is already running



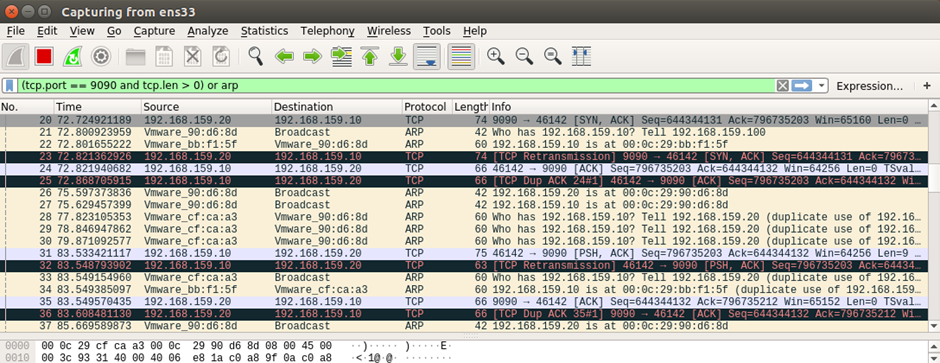
As we can see, marked in red is the original message, while marked in green is the edited messages.

taking a look at our attacker while this happens:



as we can see marked in green, we can see each message before and after the edit.

Finally, we look at wireshark



Filter - TCP traffic on port 9090 or ARP packets (to focus on the relevant communications

PSH/ACK packet - Actual data being transmitted between our user and server; our code intercept these packets and modify their data.

ARP packets - indicating out ARP poisoning attack running over and over

DUP ACK - we have a processing delay (because we are intercepting and modifying packets to send them to the real mac address of the target) and since wireshark sees all the traffic in the network, it considers them to be duplicates.

Task summary:

* we managed to complete the task and all its requirements.
* proof is presented in the screenshots
* We found and learned how to enact Man In The Middle attacks against “netcat”, and we also learned how to initiate “netcat” connections.
* This task does not truly fit our theory, as we assumed we will simply be able to take our script from task 2, and edit the condition for which the text is changed.  
  In reality, we got stumped for a while with our script working for a bit, only to then stop, and .10 and .20 sending out new arp requests, which is our main hurdle.
* Task 3 main hurdle:

After we gained the MITM position and changed a random amount of messages (sometimes it only worked for one message, other times for multiples) other messages would not be changed.  
We added a loop to keep updating the arp table of all the machines, thus maintaining the poisoned arp caches, and keeping our position as a MITM capable of changing messages.

Also we were needed to handle the packet’s that arrived at the attacker MAC address and avoid them so we won’t capture and process our own spoofed packet that can cause an infinite loop.

**Summary reflection and innovation:**

Once you have successfully performed ARP cache poisoning and established a Man-in-the-Middle attack over TCP, you gain control over network traffic between the victim machines.

This allows you to:   
Intercept Network Traffic – capturing sensitive data, which theoretically could be: usernames, passwords, session tokens

Tamper with files or messages sent over the network.  
Monitor unencrypted communication (like HTTP, Telnet, FTP)

and many more ways by just leveraging some creative thinking and just a little bit of python or other programming and networking knowledge.

By leveraging ARP poisoning with TCP MITM, attackers can control, manipulate, and exploit network traffic, posing a severe cybersecurity risk.  
Proper defenses, such as using static ARP entries, encrypted protocols, and intrusion detection systems (IDS), are crucial to mitigating these attacks.

One thing we would enjoy doing further testing upon is “netcat”.  
We would like to see if we can somehow edit the message after it is sent via our attackers console, and edit the message to whatever we want on the spot, while editing the packet's meta-data so that it doesn't break the TCP connection.

(Example next page)

For example:

In 2024, security researchers reported that a vulnerability enables hackers to launch an MITM attack to unlock and steal Tesla vehicles.

The attacker was using a “flipper zero” device to pretend to be a tesla network access point.  
the device looks like this:  


The car, key and application would talk to each other through the pretend network and all the information would pass through the attacker machine, becoming visible and available to hijacking.

So now after we learned how to do a basic attack, some students might be driving to campus in a stylish new vehicle (for legal reasons this is a joke).

<https://www.ibm.com/think/topics/man-in-the-middle>

<https://hothardware.com/news/steal-a-tesla-by-mitm-attack-with-flipper-zero-or-other-hardware>