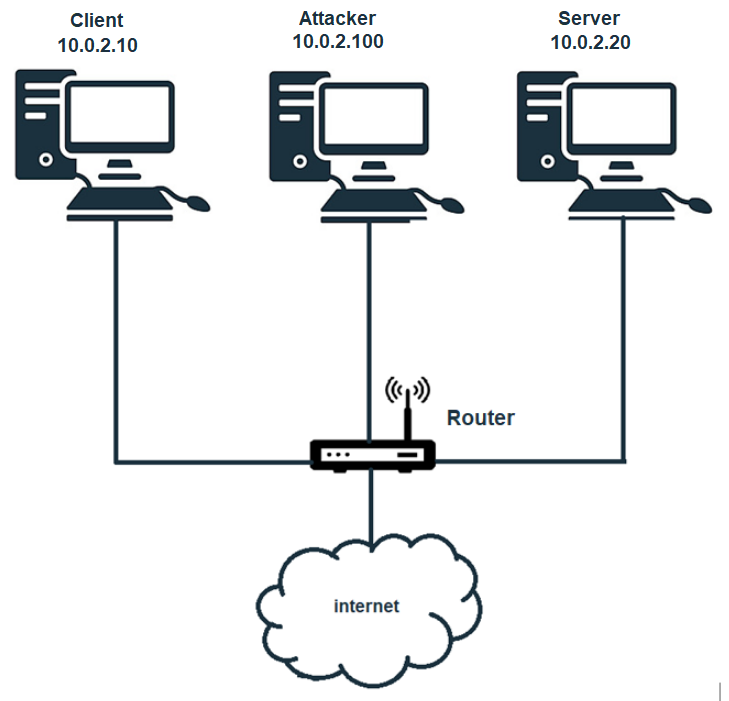
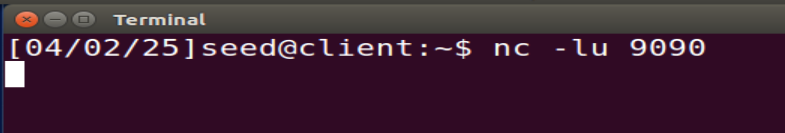
**Network Setup:**

* Server (10.0.2.20)
* Client (10.0.2.10)
* Attacker (10.0.2.100)

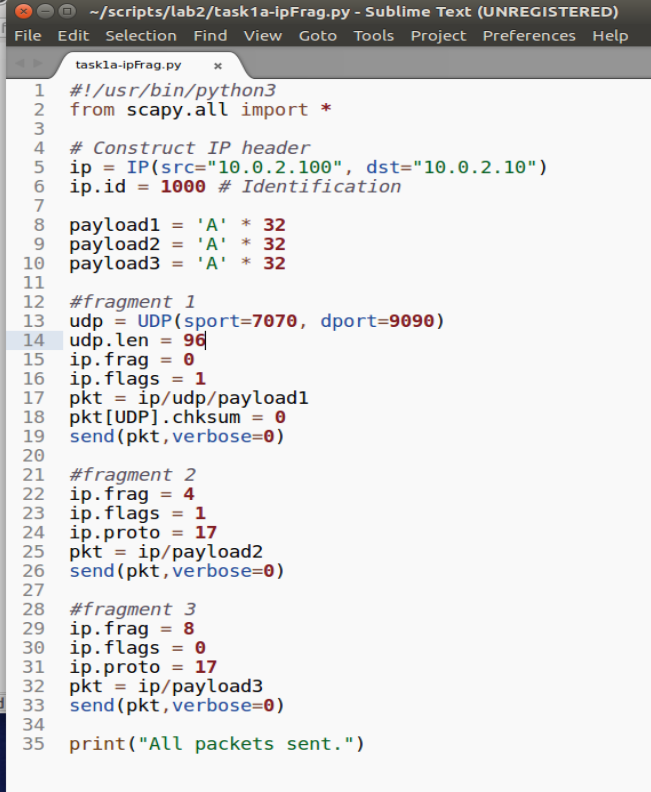
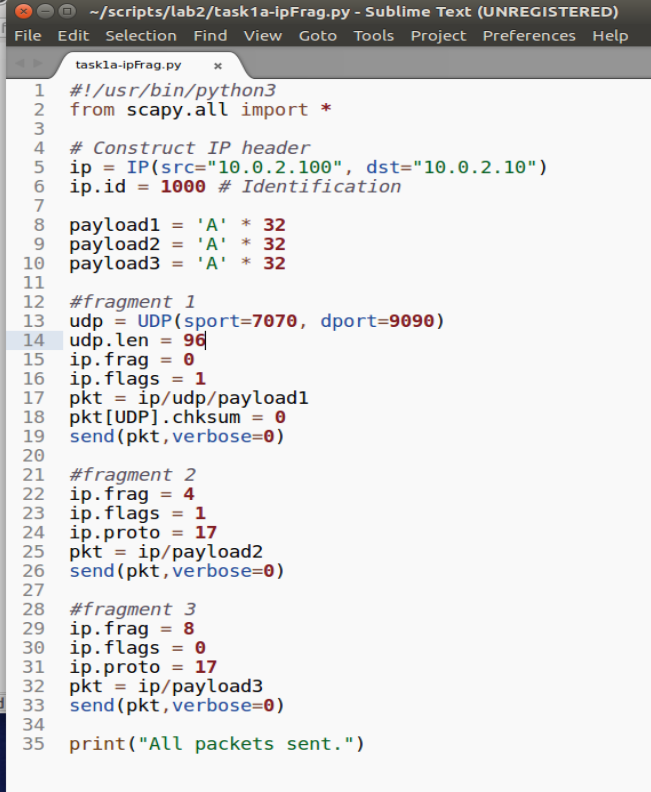
**Diagram** (Similar for tasks 1&2):



**TASK Execution:**

**Task 1.a: Conducting IP Fragmentation**

Netcat listener on port 9090

**Attacker:** Preparing the script:  


Len includes only data

(without UDP header)

**ERROR!**

17 - to ensure UDP protocol

we set ip.proto=17, as without it being stated, our packets were being sent via ipv6. this fixes that.

**Fixing the script:**

Len = 104 (A \* 32 times \* 3 packets + 8 bytes of UDP header = 104)

First packet include - offset = 0 include:

udp header = 8 byte

data

fragment 1 = 8 (UDP header) + 32 (data) = 40

1 = MF = More fragment to come.

New len (include UDP header)

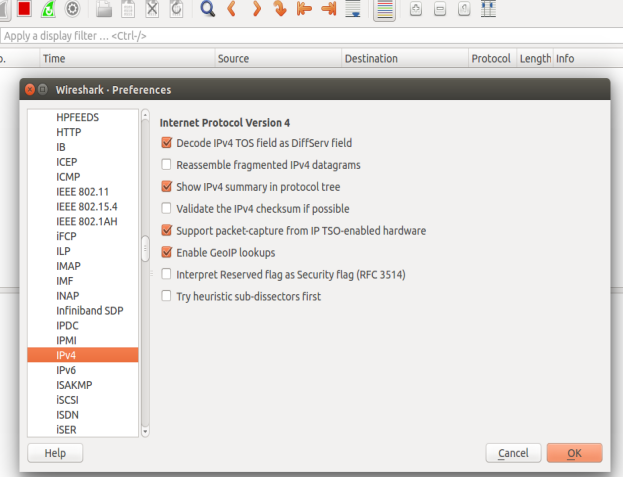
fragment 2 = start from 40 (5\*8)

fragment 2 = start from 40 (5\*8)

fragment 3 = start from 72 (9\*8)

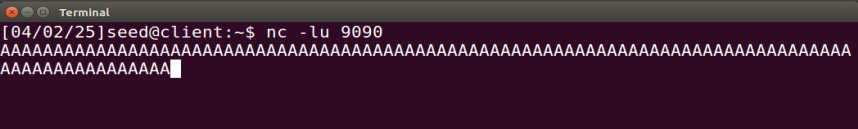
Last Fragment/Dont Fragment

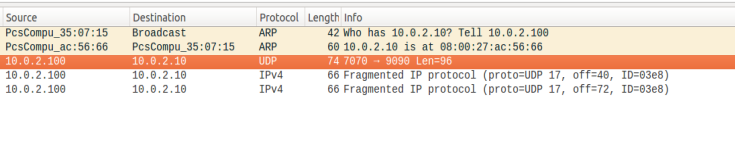
every fragment starts from the correct position in accordance with the last fragment size.

**Wireshark:** edit -> protocols -> IPv4 -> uncheck "Reassemble fragmented IPv4 datagrams"  


disable IP fragmentation reassembly

Executing the script with: sudo python3 ip\_frag.py





**ask 1.b: IP Fragments with Overlapping Contents**

1. **sending the first fragment first:**

1.b



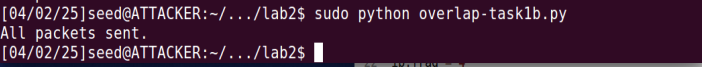
adjusting payload for better  
visual representations

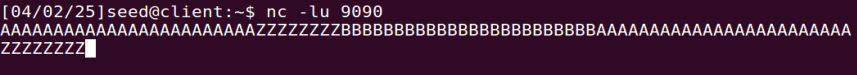
adjusting back len to 96 because of the overlap

fragment 2 = start from 32 (4\*8)

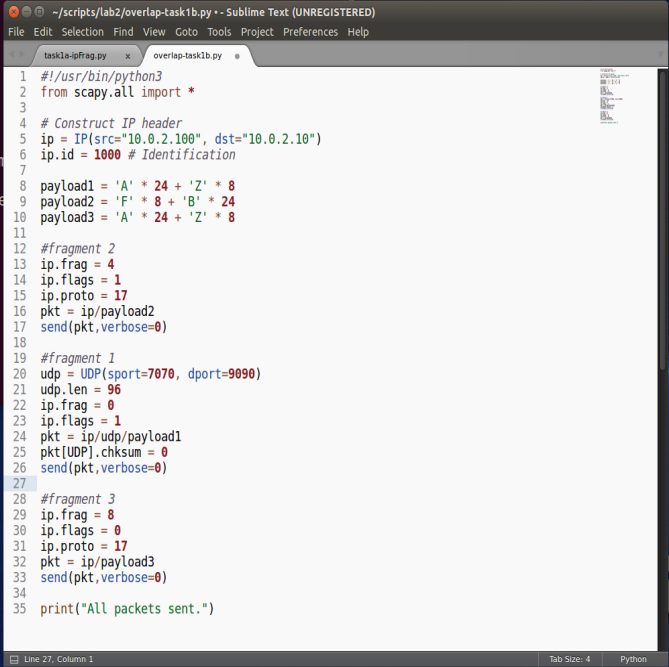
OVERLAP

fragment 3 = start from 64 (8\*8)

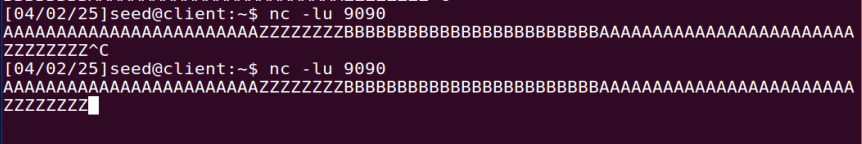




‘F’ is not showing due the overlap

**(2) sending the second fragment first**

sending fragment 2 first

**Client:**

‘F’ is not showing due the overlap

We can see the **same results**.

fragmentation offsets are dictating the order of reassembly.

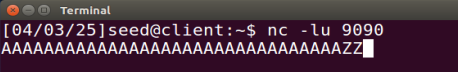
1.b - enclosed variation:



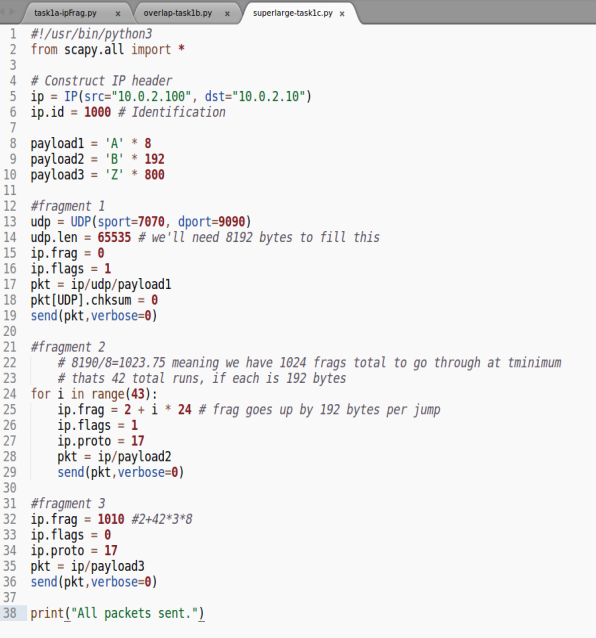
in this variation we completely enclose the 2nd fragment within the 1st fragment. adding the 3rd fragment in the end to signal last fragment with the ip.flags = 0.

you might guess some of the “F” characters would appear in the middle of the “A”.  
but this is not the case.

so we run the script:  
  
and get the result:

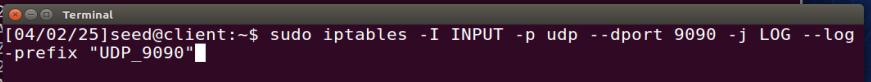


As you can see, the enclosed 2nd fragment is completely erased and overlapped by the first larger fragment, then continuing to the 3rd fragment without seeing the 2nd.

**Task 1.c: Sending a Super-Large Packet**

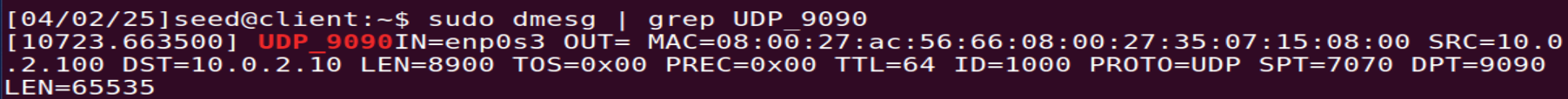
in order to send a super-large packed we changed the script as displayed above with a larger length and offsets which will try to support the size.

**Client:**



we were not able to see the traffic on netcat because its larger than the allowed size, so we added a rule to the iptables to log the traffic for this task.

after executing the script from the attacker machine we can see the log recorded on the client machine.   
thus concluding the client did in fact get the super large packet, as can be seen below in the logged data (but it was not displayed on NC because its too large):



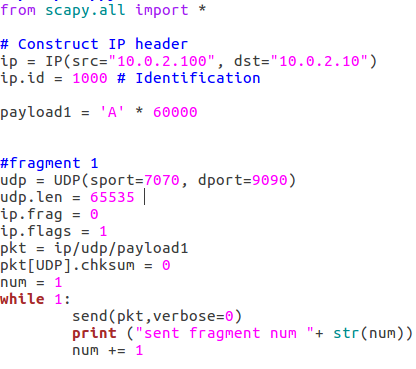
UDP\_9090 IN=enp0s3 OUT= MAC=08:00:27:ac:56:66:08:00:27:35:07:15:08:00 SRC=10.0.2.100 DST=10.0.2.10 **LEN=8900** TOS=0x00 PREC=0x00 TTL=64 ID=1000 PROTO=UDP SPT=7070 DPT=9090 LEN=65535  
  
looking at the marked LEN, and doing some quick math, we can come to the conclusion that we have successfully sent a packet larger than 65535  
we send 42 packets the size of 192 each, plus 16 from the first fragment, plus 800 from the last packet.

16+192\*42+800 = 8880  
but if we also add the size of the IP header, we get 8880+20=8900, which is exactly the LEN stated in the packet.

**Task 1.d: Sending Incomplete IP Packet**

**Attacker:**

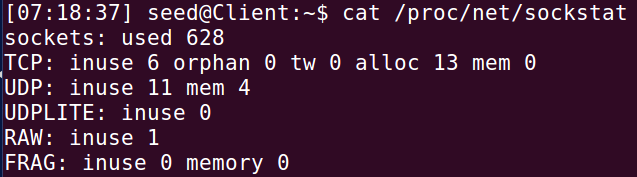
we set our script to send many large incomplete packets:



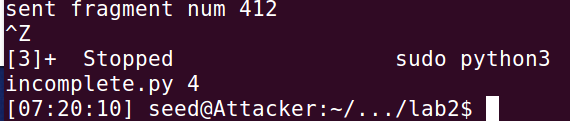
but once more we run into the issue of not knowing how to actually make sure we've succeeded

for this we decided to use two commands:  


shows the active memory usage on our machines network



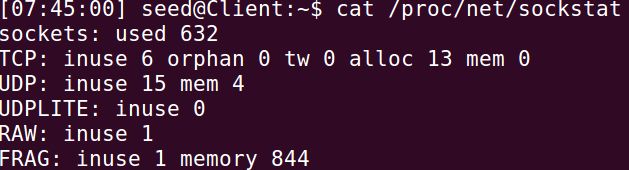
we then run the code again, letting it run for 412 loops

The max packet size that can actually be sent is 1516, so to make sure there isn’t a user error, we set our packet to be 1496 in length, and attempt again, wanting to make sure that by fragmenting the previous packets nothing was messed up.

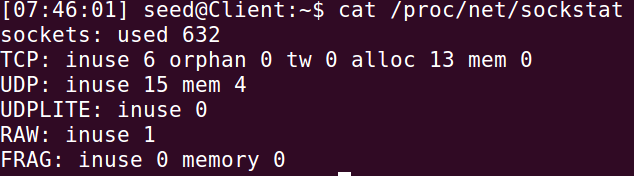


we can see that for frag, one is in use.

and until the TTL is exceeded it remains in use.



and once the TTL exceeded is sent it changes to 0



notably, no matter how many packets we sent the memory remind at 844

TO NOTE, we also attempted to change the ip.id and run a loop, in the hopes of using more memory, but this failed.

**Task summary**

**What did we discover?**

we learned how to manipulate ICPM header, fields and payloads, to be able to execute different types of attacks.  
while also seeing how different variations of fragmentations and payloads get treated by the network and machines.

and also we by curiosity we leared about the linux iptables and rules for logging and viewing a lot more traffic than what is usually presented.

**Did it match expectations/theory?**

Yes, the results matched the theoretical expectations:

* sending fragments in a different order would get rearranged correctly as the assembly takes place.
* if an offset does not match, some of the payloads or part of them would get erased.
* a payload larger than the allowed amount would get dropped.

**Problems and Solutions**

calculating the length of data is key, most of the problems would usually amount to miscalculations in the fragmentation offsets.

**Task 2: ICMP Redirect Attack**

**Introduction**

ICMP Redirect messages are error messages sent by routers to IP packet senders when they detect that packets are being routed inefficiently. The router sends a redirect message to inform the sender to use a different router for future packets to the same destination. This legitimate network optimization mechanism can be exploited by attackers through spoofing ICMP Redirect messages, allowing them to modify a victim's routing table and redirect traffic through the attacker's machine (enabling Man-in-the-Middle attacks

Expected Results:

If the attack is successful, the victim's routing table will be modified to route traffic for specific destinations through our attacking machine instead of the default router. This would allow us to intercept, analyze, and potentially modify traffic in a MITM attack.

Steps to be Performed:

* + Enable ICMP Redirect acceptance on the victim machine
  + Create a Python script using Scapy to craft spoofed ICMP Redirect packets
  + Send these spoofed packets to redirect traffic through our machine
  + Test different scenarios: redirecting to our machine, to a remote machine, and to a non-existent machine
  + Analyze the results using Wireshark and routing table inspection

**TASK Execution:**

**Server:**

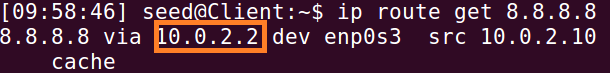
we first set our machine to accept any redirects:

תמונה שמכילה טקסט, גופן, צילום מסך

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

Enable redirect

In our case, we had to specifically name our web interface for this command to work, rather than use .all  
Next, we get a trace route for a baseline:



**Attacker:**

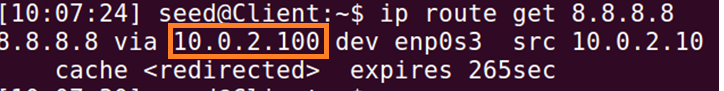
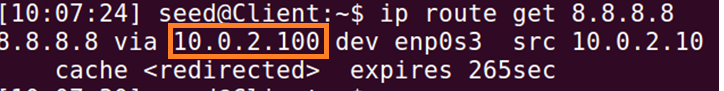
Next, we run our script:

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

And observe wireshark:



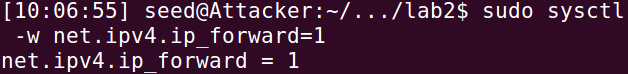
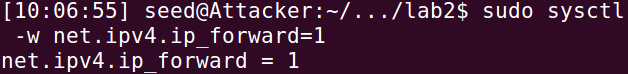
Where we can see the successful redirect packet, spoofed to be from 10.0.2.2  
we now get a trace route to 8.8.8.8 from 10.0.2.10  


Attacker is our new gateway; success!

**Client:**  attempt to ping 8.8.8.8 - failed!

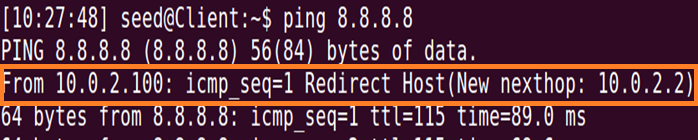


client cannot reach 8.8.8.8  
a way to fix this, is by turning on ping forwarding in our attacker:

**Attacker:**  


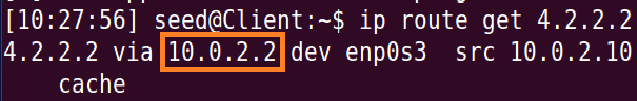
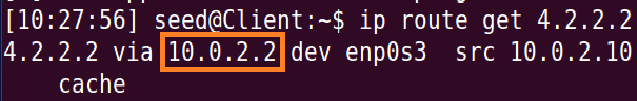
Enable forwarding

**Client:** pinging once more



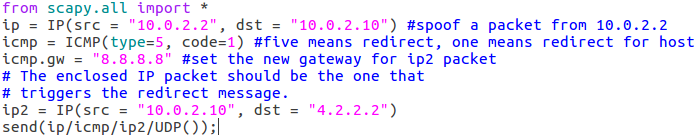
we can see that despite the ping being from the client, it goes through 10.0.2.100, which while it does give us away, we can use a MiTM script instead, as we did in a previous lab.

**Q1 - Can you use ICMP redirect attacks to redirect to a remote machine?**

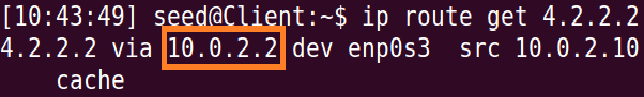
**Client:**  
we first take a baseline route to 4.2.2.2, as our plan is to set the gateway to 8.8.8.8  


we can see the base gateway is used, as we want.

**Attacker:**

so we use the following script:  
  
we can see the packet in wireshark:  

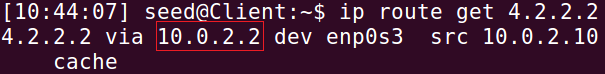
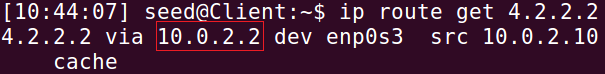

**Client:**

  
best we can gather, this is due to the fact that ICMP redirects to remote machines are typically not accepted, since redirects are intended for optimizing routes within the same network, and works on layer 2.  
**P.S**after successfully doing Q2, we know we can spoof an arp reply, so as to allow this to work.  
meaning, with proper execution, we can spoof the client to believe it is using 8.8.8.8 as a gateway, using ARP poisoning.  
but we lack a way to properly test this with a machine that is not on the network.

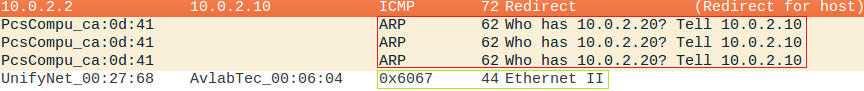
No change!

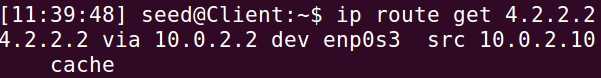
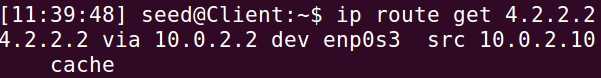
**Q2 - Can you use ICMP redirect attacks to redirect to a non-existing machine on the same network?**

**Client:**

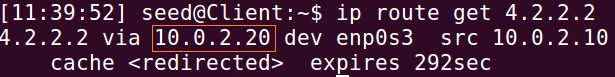
first, we change our code to set the icmp.gw as our server, 10.0.2.20, in the code.we then run it, and check the route:  


and we can see that it failed, as the gateway is 10.0.2.2, rather than what we set it to be.  
looking at wireshark:  
  
we can see that it does not know the mac address for 10.0.2.10  
but we have made, in a previous lab, code to spoof arp replies, so we run that after running the script again:



we can see in red the ARP request, and in green our reply, and look at the table

No change!

trying to run the script again, just in case:  
  
**Success!**

and see that now that the ARP is poisoned, because we have successfully set the gateway to be a machine that is offline.  
from here was can use a script to be a MiTM, while keeping ourselves hidden.

**Task summary**

**What did we discover?**

Despite successfully creating and sending ICMP Redirect packets correctly (as proven by Wireshark captures), we were unable to change the victim's routing table.

* Modern operating systems implement additional protection mechanisms beyond the simple accept\_redirects=1 setting
* ICMP Redirect attacks to remote or non-existing computers are systematically rejected
* Operating systems perform additional checks to verify the validity of ICMP Redirect messages, such as checking that the proposed gateway is on the same network segment

**Did it match expectations/theory?**

Partially. We were successful in creating and sending properly formatted ICMP Redirect packets (as confirmed by Wireshark captures), but we were not successful in changing the victim's routing table, which was the ultimate goal of the attack.

**Problems and Solutions**

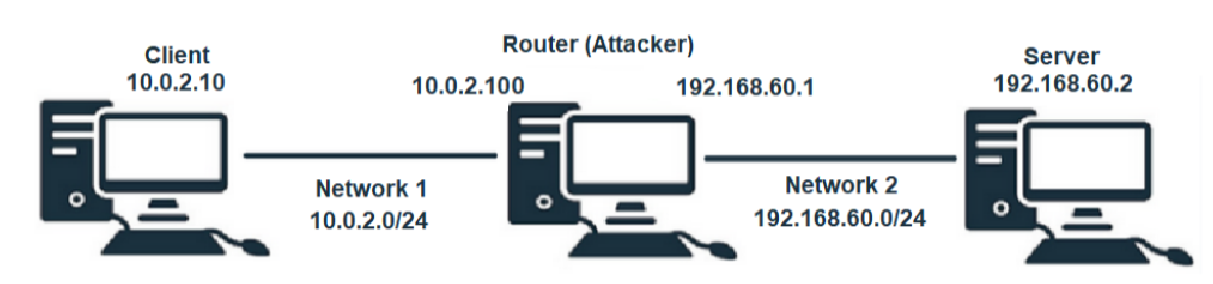
The main problem was that the attack didn't work as expected despite enabling accept\_redirects. We addressed this by conducting additional experiments with different configurations (such as using different IP addresses) to better understand the limitations of the attack and the existing protection mechanisms.

**Task 3: Routing and Reverse Path Filtering**

**Network Setup:**

* Server (192.168.60.2)
* Router(Attacker) (10.0.2.100/192.168.60.1)
  + 10.0.2.1 (enp0s3 NAT Network)
  + 192.168.60.1 (enp0s8 Internal Network)
* Client (10.0.2.10)

**Diagram:**

****

**Introduction**

Reverse Path Filtering is a security mechanism in Linux that prevents IP spoofing by ensuring symmetric routing. When a packet with source IP address X arrives through interface I, the kernel checks whether packets sent to X would use the same interface I. If not (asymmetric routing), the packet is considered suspicious and dropped. This effectively prevents attackers from spoofing packets with source addresses from networks that would normally be reached through different interfaces.

Expected Results:

* + Packets with spoofed source IPs from the attacker's own network will be forwarded
  + Packets with spoofed source IPs from the internal network will be dropped by reverse path filtering
  + Packets with spoofed external IPs will be forwarded

Steps to be Performed:

* + Set up a network with three machines across two different network segments
  + Configure routing to enable communication between all machines
* Test packet spoofing using three different source IP addresses:
  + An IP from the attacker's network (10.0.2.0/24)
  + An IP from the internal network (192.168.60.0/24)
  + An external IP from the Internet (1.2.3.4)
* Observe which packets are forwarded and which are blocked by reverse path filtering

**TASK Execution:**

**Task 3.a: Network Setup**

Attacker (router):

* Connected to both networks using two interfaces:
* First Interface - (enp0s3): 10.0.2.100 (NAT Network)
* Second Interface - (enp0s8): 192.168.60.1 (Internal Network)

Client (machine A):

* Connected to NAT Network (10.0.2.0/24)
* IP Address: 10.0.2.10

Server:

* Connected to Internal Network (192.168.60.0/24)
* IP Address: 192.168.60.2

**Task 3.b: Routing Setup**

We configured the routing to enable communication between all machines:

Client:  
Added route to reach internal network through Router

sudo ip route add 192.168.60.0/24 via 10.0.2.100 dev enp0s3

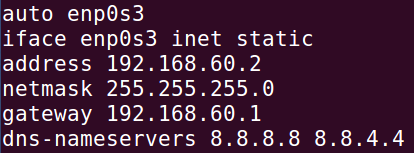
Router (Attacker):   
Enable IP forwarding to make it function as a router

sudo sysctl net.ipv4.ip\_forward=1

Server:

added default gateway via system files.

**Server:**

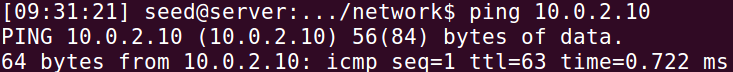
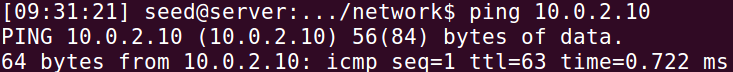


Setting new IP in server

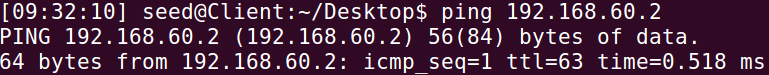
Setting new gateway in server

after setting the default gateway and IP in the file interfaces, we reboot the network

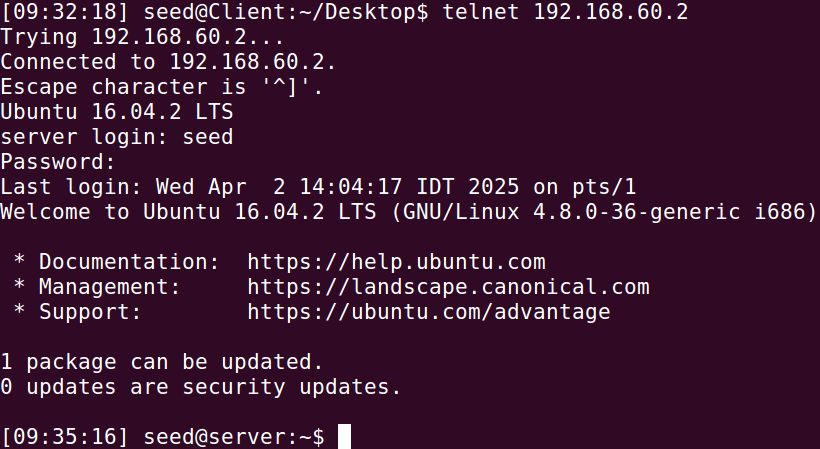
and attempt a ping:

**Client to Server:**

**Server to client:**

  
we can see we successfully ping.

**telnet: Client to Server**

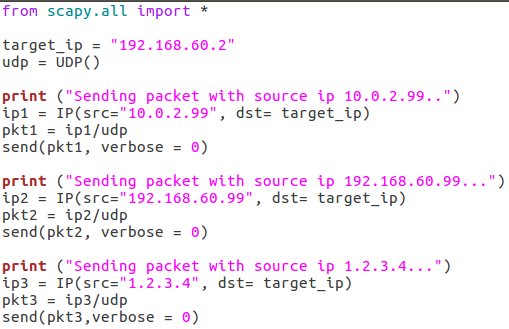
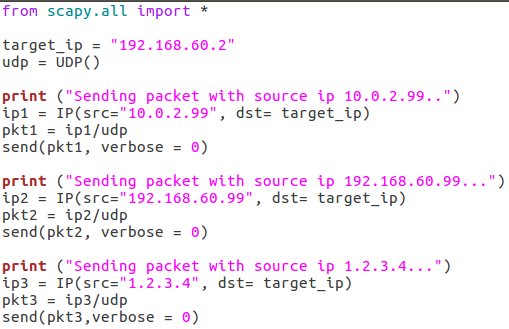


seeing as we have managed to ping both machines from each other, and to create a telnet connection, we **successfully set up our attacker as a router.**

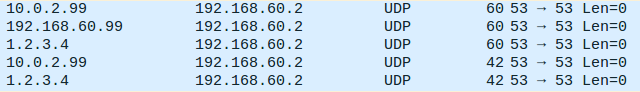
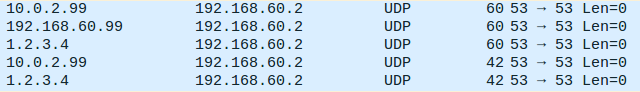
**Task 3.c: Reverse Path Filtering -  
  
we first enable reverse path filtering:**



Verify reverse path filtering

**then send 3 spoofed packets using the following sources:**

**and our expectation is for the packet from 192 to be dropped, as the router will forward it and detect it is from a different interface. (in our routers case, enp0s8 rather than enp0s3)**

**wireshark logs:  
ROUTER:**

#1

#2

Dropped

#1

#2

we can see by looking further into the packets that packets #1 and #2 have different mac addressed.  
from this we can confer that they are infact being forwarded to 60.2, and we can further see that 192.168.60.99 is being dropped.

to make sure, we look at our servers wireshark logs:

 **and as we can see, only the forwarded packets were recieved.  
thus, by the fact only packets that are *not* from 192.168.60.0/24 network are being forwarded, we can understand we have successfully implemented the reverse path filtering in our server, as required.**

Packet with source IP 10.0.2.99:

* Successfully forwarded to Server
* Visible in Server's Wireshark capture

Packet with source IP 192.168.60.99:

* NOT forwarded to Server (dropped by reverse path filtering)
* NOT visible in Server's Wireshark capture

Packet with source IP 1.2.3.4:

* Successfully forwarded to Server
* Visible in Server's Wireshark capture

The router dropped the packet with a spoofed internal network IP (192.168.60.99) while allowing the other spoofed packets through.

**Task summary**

**What did we discover?**

* + Reverse path filtering effectively blocks spoofed packets where the source IP belongs to a network that would be reached through a different interface
  + The filtering happens at the routing level before packets reach their destination
  + Not all spoofed packets are blocked - only those that violate the symmetric routing principle
  + External IP addresses (like 1.2.3.4) can still be spoofed as they don't trigger the reverse path filter

**Did it match expectations/theory?**

Yes, the results perfectly matched our theory.

We successfully demonstrated how reverse path filtering works by showing which spoofed packets get dropped and which get forwarded.

* + Packets with source IPs from networks reachable through the same interface were forwarded
  + Packets with source IPs from networks reachable through different interfaces were dropped
  + This demonstrates the exact filtering behavior described in the theory

**Problems and Solutions**

Setting up the multi-interface network configuration required careful planning and verification. We had to make sure the routing tables were correctly configured on all machines. We solved this by testing connectivity after each configuration step and using Wireshark to verify packet flows.

**Innovation**:

[Exploiting Cross-Layer Vulnerabilities: Off-Path Attacks on the TCP/IP Protocol Suite – Communications of the ACM](https://cacm.acm.org/research/exploiting-cross-layer-vulnerabilities-off-path-attacks-on-the-tcp-ip-protocol-suite/)

off-path attacks:

generating and exploiting fake ICMP error messages.

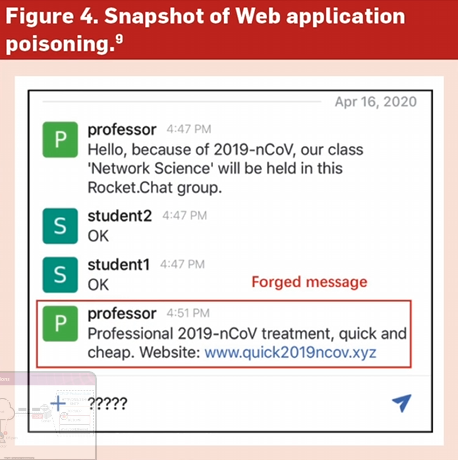
usually ICMP error messages are generated in response to network issues. and can influence the behavior of higher layers protocol such as TCP and UDP

this means ICMP error messages could induce cross layer interactions within the TCP\IP protocol.

these interactions are vulnerable to off-path attacks

by manipulating IP ID an off-path attack can establish a side channel to guess the sequence number of the TCP connection.

so the attacker inject fake TCP package in the connection



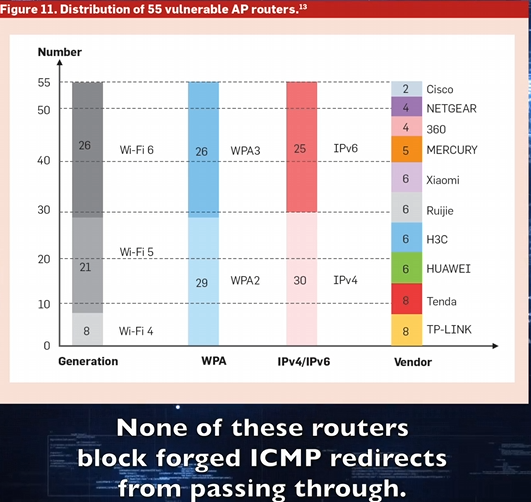
if the connection is secured with TLS - the attacker can reset the connection

desynchronization:

in order to avoid the packet fragmentation a mechanism has been developed (PM-TUD) to probe the minimal size of end-to-end path

off-path attack can impersonate the server with IP spoofing and send crafted IP fragments to the server which then **cause the legitimate fragments to be incorrectly reassembled with the malicious ones** and sent to the victim client. thus the attacker can inject fake fragments to poison the TCP connection.

Identity Deception:



the existing WIFI routers were vulnerable to ID deception.  
an attack can hijack all traffic in a wifi network by **generating a fake ICMP package.**

the root cause of this vulnerability is that the network processing unit (NPU) is unable to verify the authenticity of WIFI packets which allows the attacker to bypass the security mechanisms of link-layer WIFI networks