**IP/ICMP Attacks SEED Lab**



**IP/ICMP Attacks SEED Lab**

**Main Introduction:**

spoofing fragmented IP packets is non-trivial. Constructing spoofed IP fragments is a good practice for students to gain their packet spoofing skills, which are essential in network security.   
I will use Scapy to conduct packet spoofing.   
This lab covers the following topics:

• The IP and ICMP protocols

• IP Fragmentation and the related attacks

• ICMP redirect attack

• Routing and reverse path filtering

**Task 1: IP Fragmentation**

**Task Description:**

I will need to construct a UDP packet and send it to a UDP server.

Create IP Fragments with Overlapping Contents.

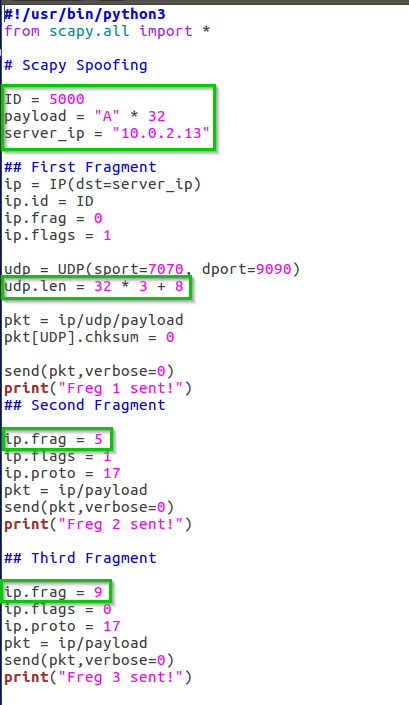
Sending a Super-Large Packet.

Sending Incomplete IP Packet – Leading to Denial-of-Service.

**Network physical topology:**

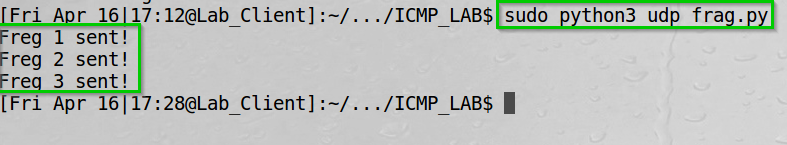


**Task 1A**

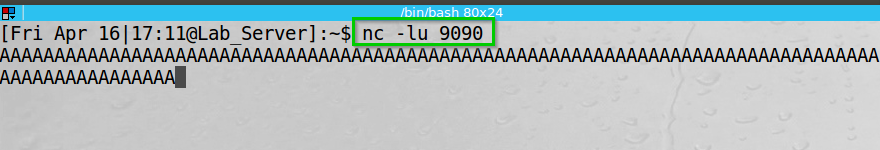
I wrote the following code using Scapy in python to split the message into 3 fragments:

I set the ID, Payload and the UDP length (32 \* 3 fragments + 8 for the first fragment).

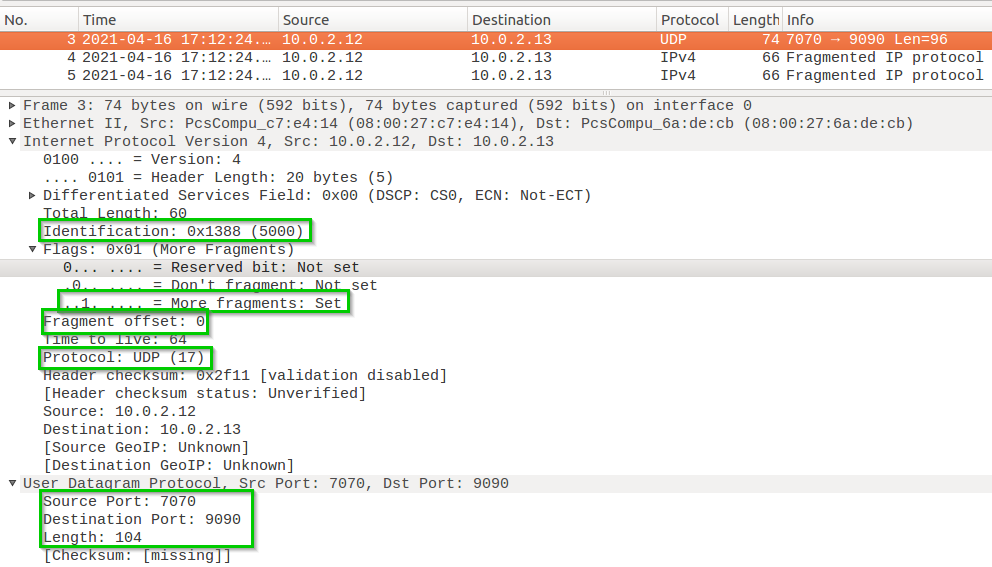
By executing the command “nc -lu 9090” on the Server, I opened a udp connection on port 9090.

On the Client I executed the python code.

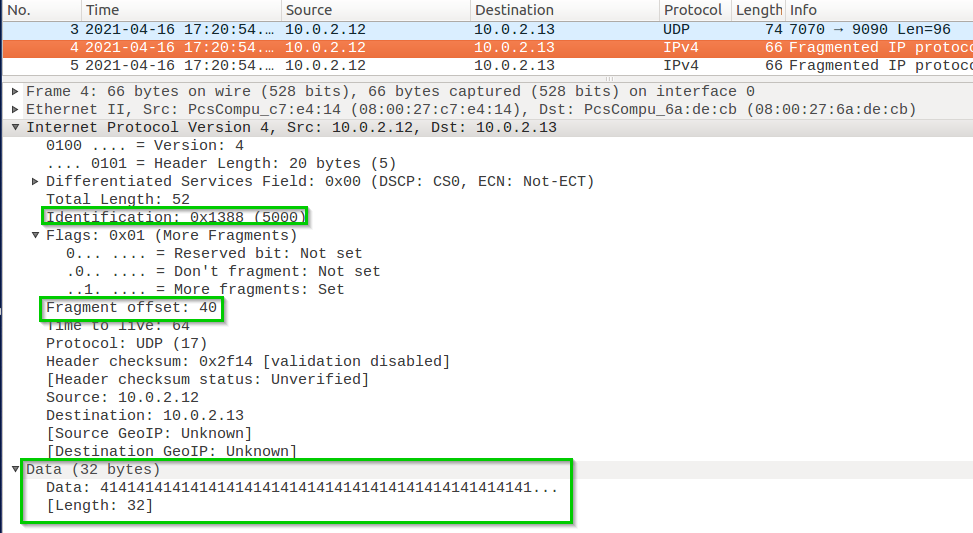
We can see the fragments sent.  
  
  
On the server we can see the 96 ‘A’s we sent.



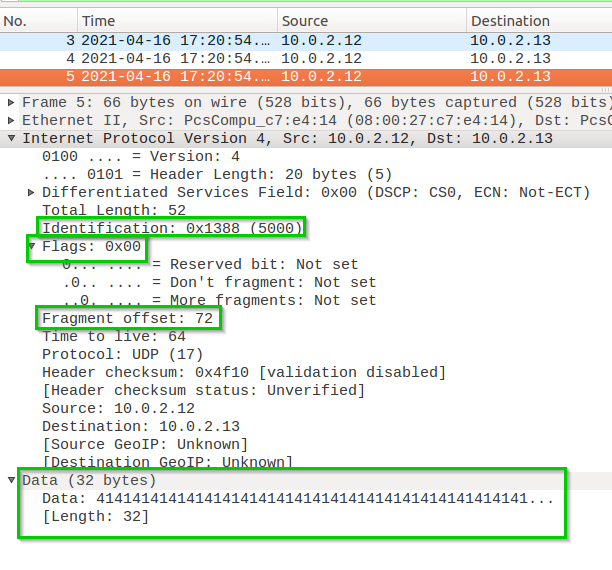
By exploring Wireshark on the Server, we can see the fragments:

On the first fragment:

The ID 5000, the More fragments is set, the Fragment offset of 0 since it is the first fragment the UDP protocol, the source port, destination port, and length of the packet. (32\*3+8 = 104).

  
On the second fragment:

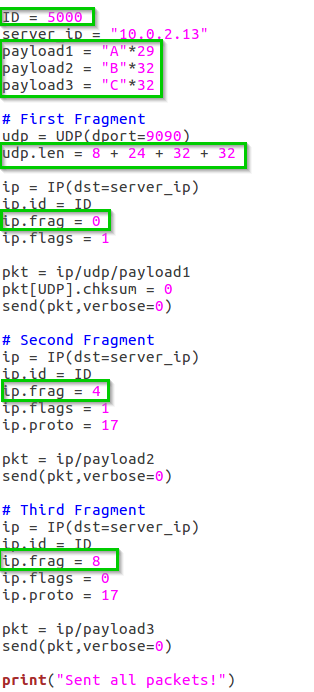
The ID 5000, the More fragments is set, the Fragment offset of 40 (32 + 8 from the first fragment), the source port, destination port, and the data – 32 bytes of ‘A’ – 41 in ASCII.

On the third fragment:

The ID 5000, the More fragments is not set, the Fragment offset of 72 (32 + 8 from the first fragment + 32 from the second fragment), the source port, destination port, and the data – 32 bytes of ‘A’ – 41 in ASCII.

**Task 1B-1**

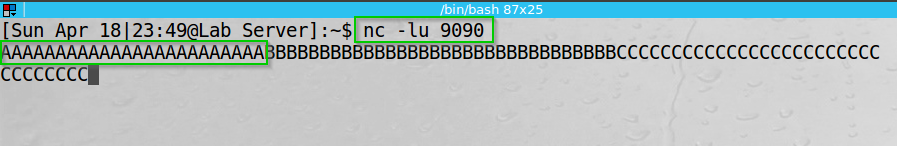
The end of the first fragment and the beginning of the second fragment overlap by 5 bytes.

  
I wrote the following code in python using Scapy:

The payloads are 29 A’s, 32 B’s, 32 C’s.  
I set the ID, Payload and the UDP length (8 + 24 (32) for the first fragment, 32 for the last two fragments).

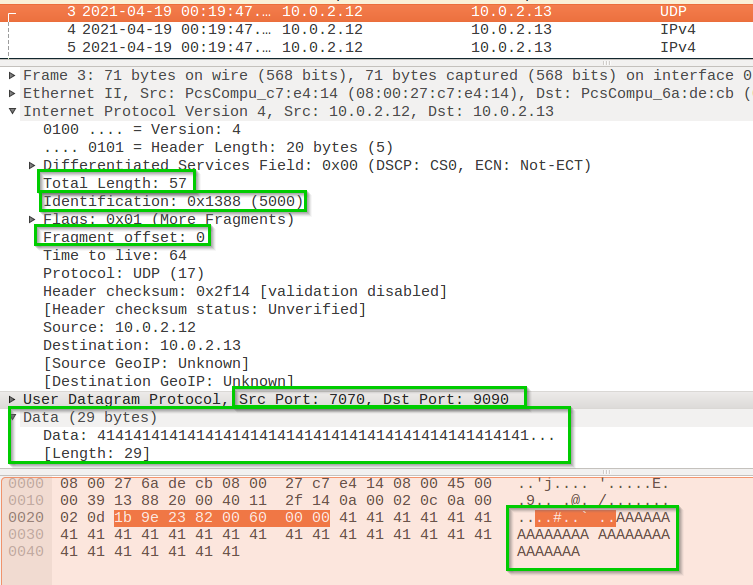
By executing the command “nc -lu 9090” on the Server, I opened a udp connection on port 9090.

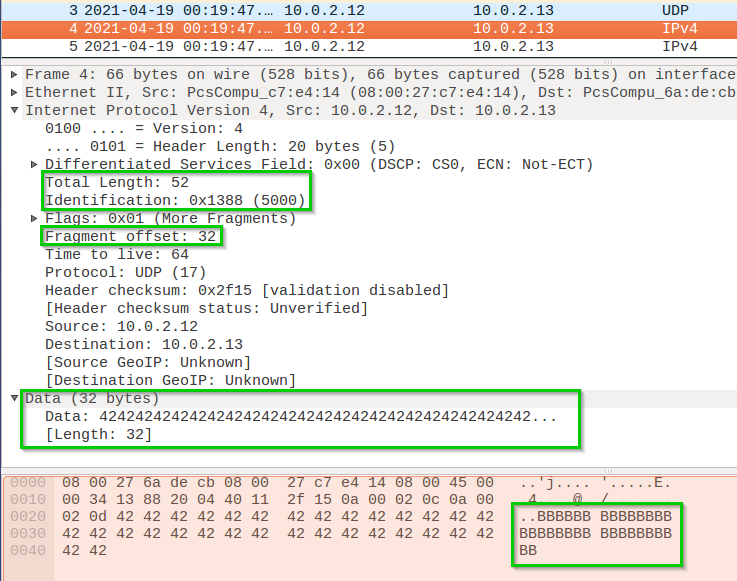
On the client we sent the packets:  
We can see the packets sent.

  
On the server we can see the output.

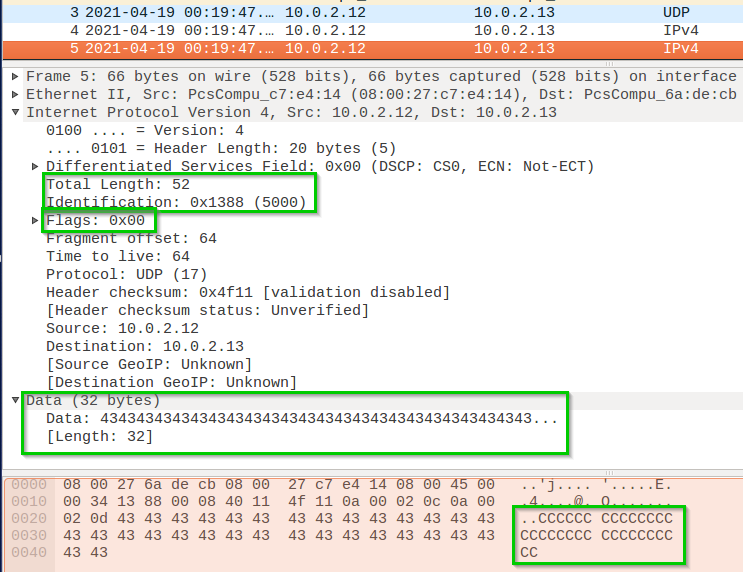
The first fragment ends at the 37 bytes of the packet, however, we start the second fragment at the 32 bytes (24+8) of the packet, by giving it an offset of 4 (4 \* 8 = 32). So, we can see the 24 A’s out of the 29 we sent, and the rest are as should be 32 B’s and A’s.

By exploring Wireshark on the Server, we can see the fragments:

On the first fragment:  
We can see the length (8+29+20 = 57), Id 5000, fragment offset 0, source and destination ports, and 29 bytes of data – ‘A’ – 41 in ASCII.

On the second fragment:

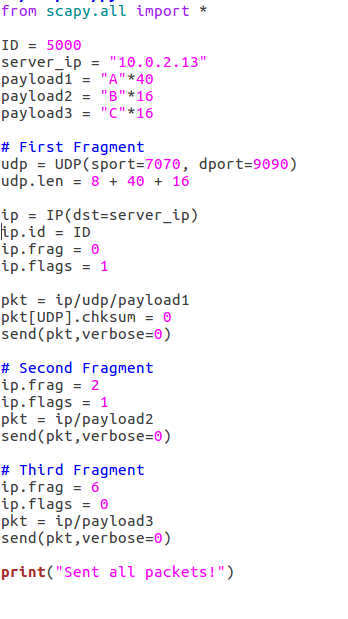
We can see the length (32+20 = 52), Id 5000, fragment offset 32 (4\*8), source and destination ports, and 32 bytes of data – ‘B’ – 42 in ASCII.

On the third fragment:

We can see the length (32+20 = 52), Id 5000, fragment offset 64 (8\*8), source and destination ports, and 32 bytes of data – ‘C’ – 43 in ASCII.

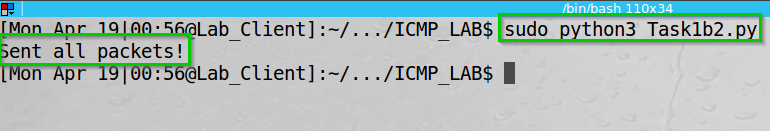
**Task 1B-2**

The second fragment is completely enclosed in the first fragment.

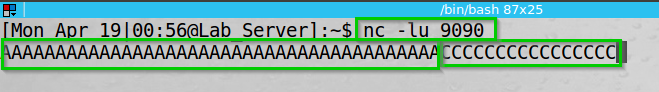
I wrote the following code in python using Scapy:

The payloads are 40 A’s, 16 B’s, 16 C’s.  
I set the ID, Payload and the UDP length (8 for the first fragment + 40 + 16).

By executing the command “nc -lu 9090” on the Server, I opened a udp connection on port 9090.

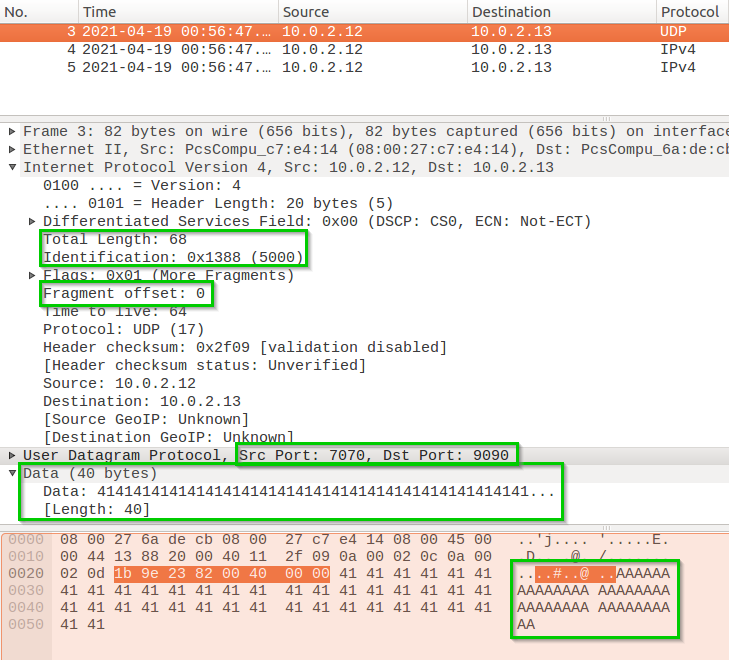
On the client we sent the packets:

We can see the packets sent.

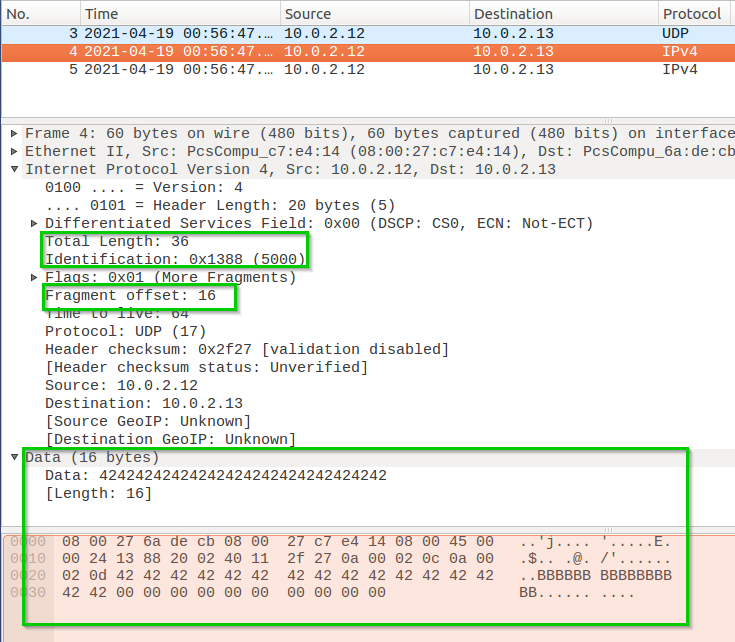
  
On the server we can see the output.

The first fragment data is larger than the second fragment. The second fragment offset is 2 – and completely inside fragment one length.

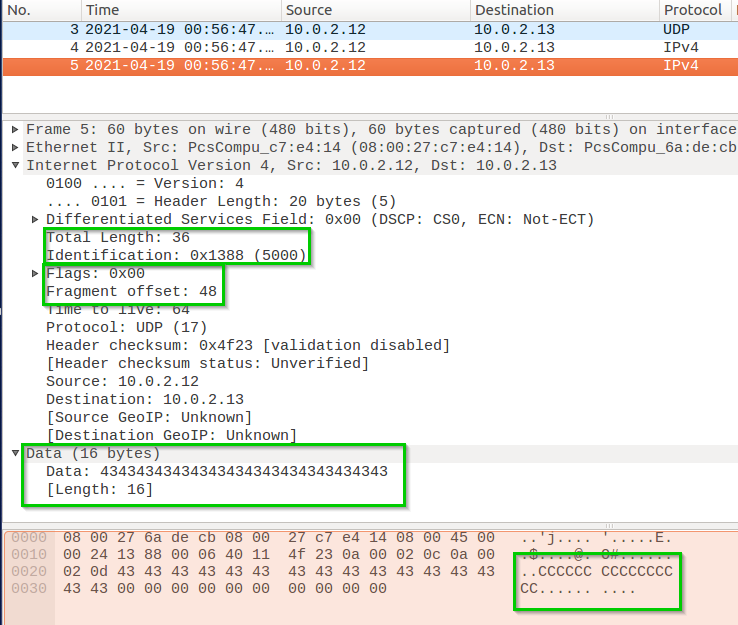
By exploring Wireshark on the Server, we can see the fragments:

On the first fragment:

We can see the length (8+40+20 = 68), Id 5000, fragment offset 0, source and destination ports, and 29 bytes of data – ‘A’ – 41 in ASCII.

On the second fragment:

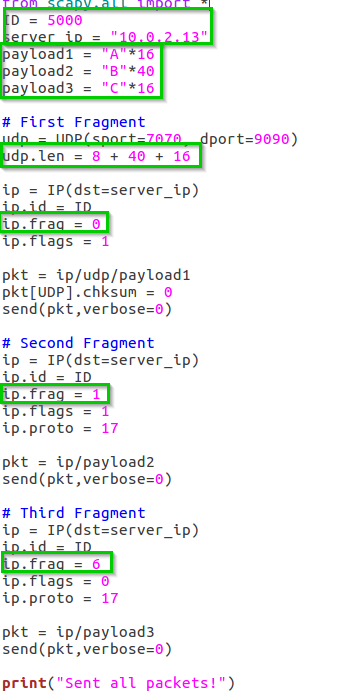
We can see the length (16+20 = 26), Id 5000, fragment offset 16 (2\*8), source and destination ports, and 16 bytes of data – ‘B’ – 42 in ASCII.

On the third fragment:

We can see the length (16+20 = 36), Id 5000, fragment offset 48 (6\*8), source and destination ports, and 32 bytes of data – ‘C’ – 43 in ASCII.

**Task 1B-3**

The first fragment is completely enclosed in the second fragment.

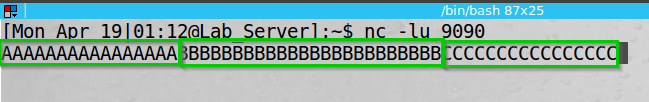
I wrote the following code in python using Scapy:

The payloads are 16 A’s, 40 B’s, 16 C’s.  
I set the ID, Payload and the UDP length (8 for the first fragment + 16 + 40).

By executing the command “nc -lu 9090” on the Server, we opened a udp connection on port 9090.

On the client we sent the packets:

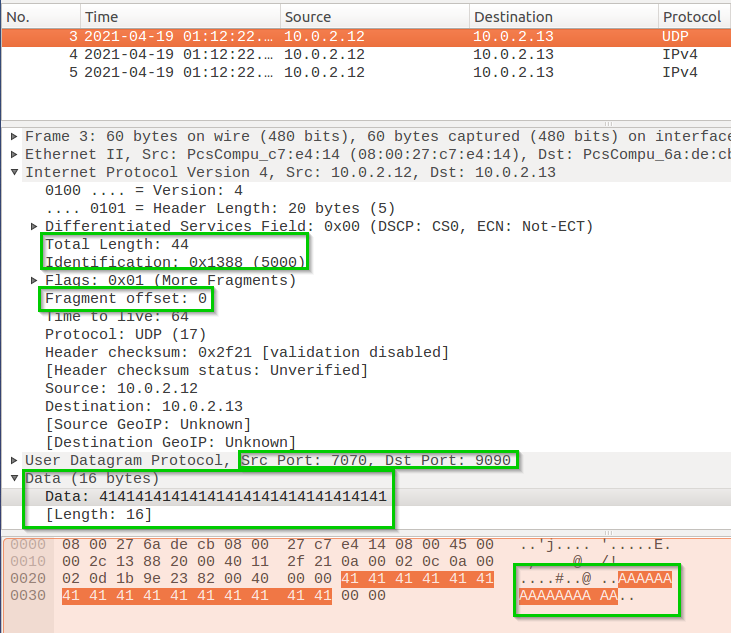
We can see the packets sent.

  
On the server we can see the output.

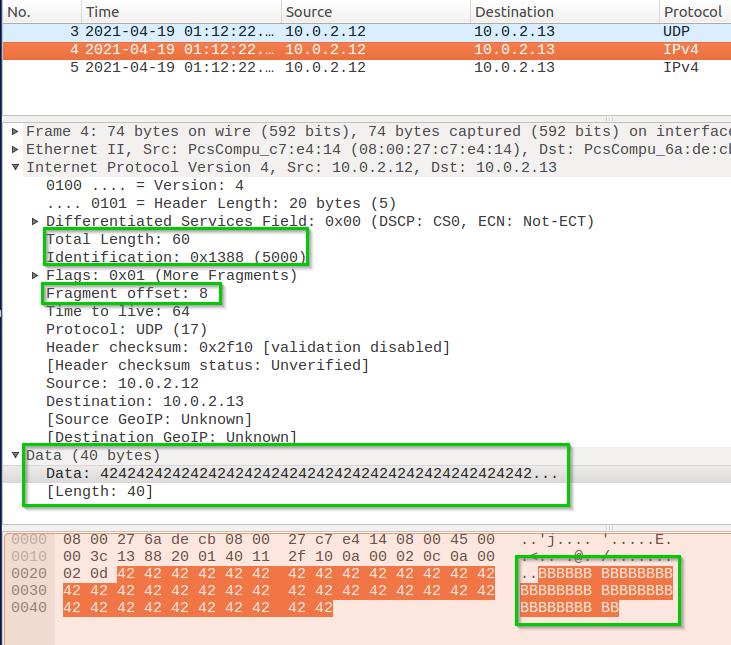
The first fragment data is larger than the second fragment. The second fragment offset is 2 – and completely inside fragment one length.

The observations in this task indicate that the fragments are written in sequence, so no matter how they are received. If the number of bytes to be overwritten are a multiple of 8, then the second fragment cannot overlap the first fragment.

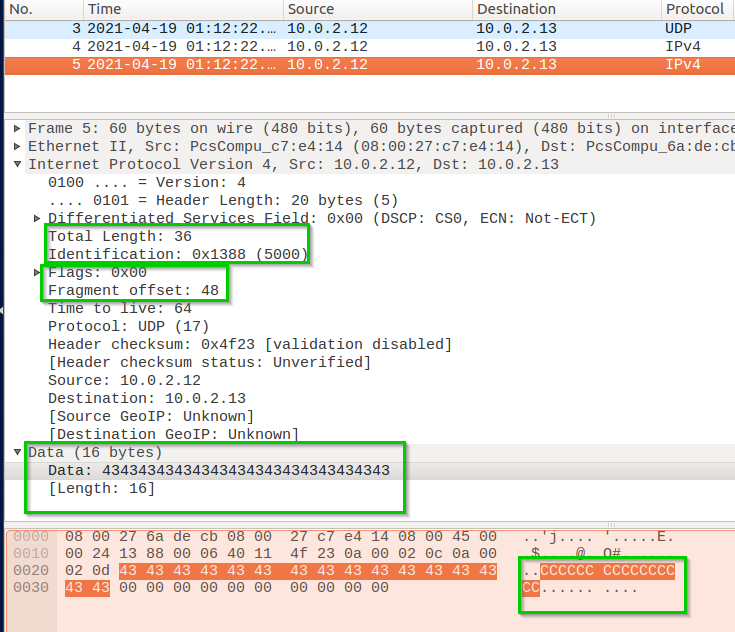
By exploring Wireshark on the Server, we can see the fragments:

On the first fragment:

We can see the length (8+16+20 = 44), Id 5000, fragment offset 0, source and destination ports, and 16 bytes of data – ‘A’ – 41 in ASCII.

On the second fragment:

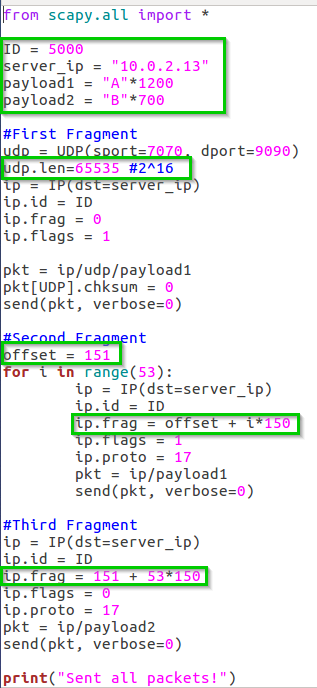
We can see the length (40+20 = 60), Id 5000, fragment offset 8 (1\*8), source and destination ports, and 40 bytes of data – ‘B’ – 42 in ASCII.

On the third fragment:

We can see the length (16+20 = 36), Id 5000, fragment offset 48 (6\*8), source and destination ports, and 32 bytes of data – ‘C’ – 43 in ASCII.

**Task 1C**

Sending a Super-Large Packet.

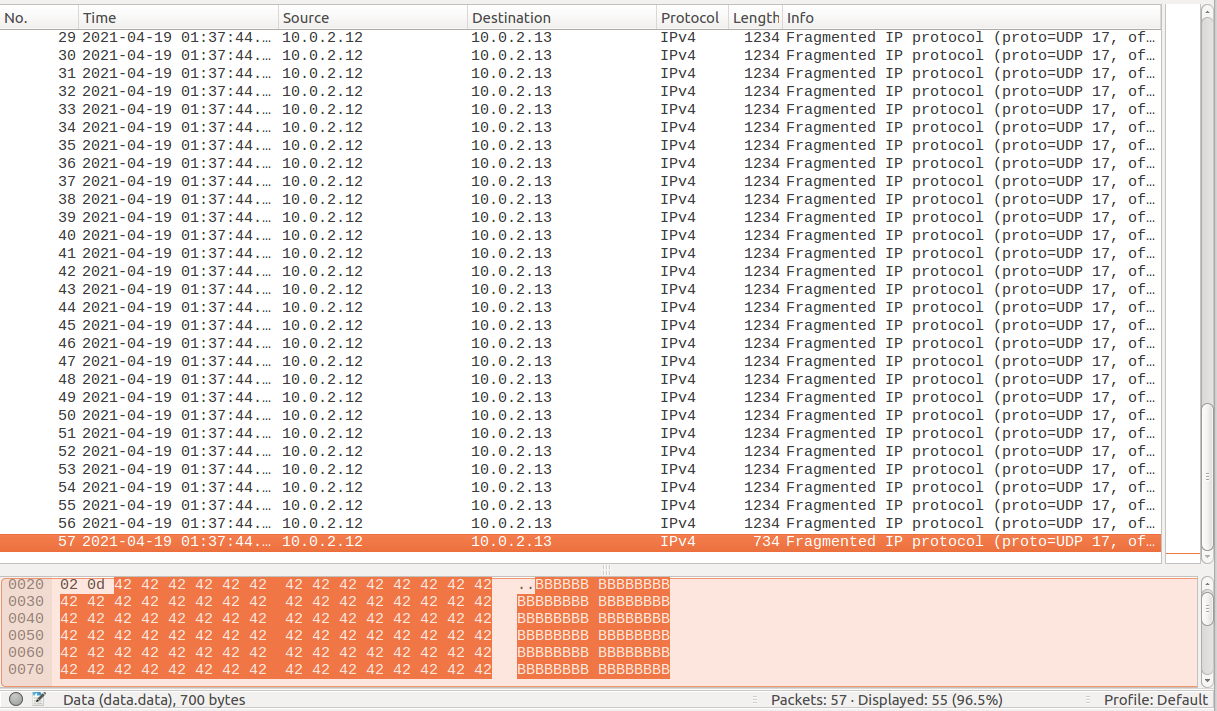
I wrote the following code in python using Scapy:

The payloads are 1200 A’s, 700 B’s.  
I set the ID, Payload and the UDP length (2^16 = 65,535).

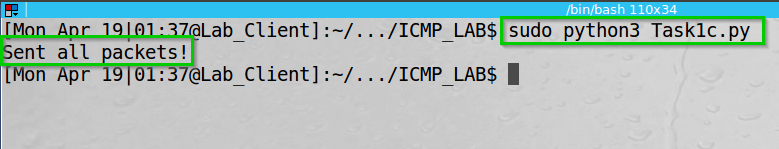
The code set udp length to 65535.   
(Anything above this value will crash the code because that is the max value the UDP length field can store).

I create multiple fragments of the same packet (ID – 5000) with 1200 bytes of data. With this amount of data in each fragment, we will need to send out 55 packets – 1st packet with the UDP header is sent out, second to 54th packet is sent out in a for loop with the changes in the fragment offset.

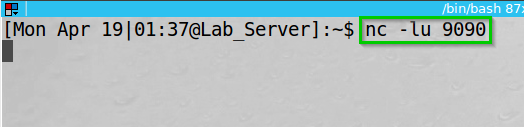
The last fragment will have the offset of 151 (after first packet) + 53 \* 150 (second to second last packet).   
To send a packet of max number of allowed bytes - 65535, we would need to send just 735 bytes, but we instead send 800 bytes, therefore - exceeding the maximum packet length.

By exploring Wireshark on the Server, we can see multiple packets are sent out:

By executing the command “nc -lu 9090” on the Server, I opened a udp connection on port 9090.

On the client we sent the packets:

We can see the packets sent.

  
On the server we can see the output is none.

As expected, the Netcat server does not display anything since the UDP length does not match the actual data length sent.   
This indicates that I have created a super-large packet, greater than the actual allowed packet length.

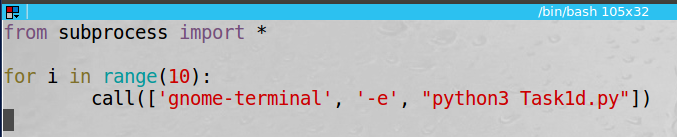
**Task 1D**

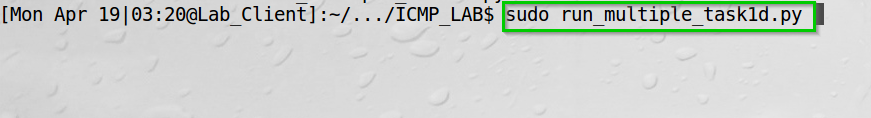
Sending Incomplete IP Packet.

I wrote the following code in python using Scapy:

I set the payload to “Hello Server!\n”, ports 7070->9090, and send packets changing the ID.

I wrote another python script to run our code multiple times to affect the server a little more.

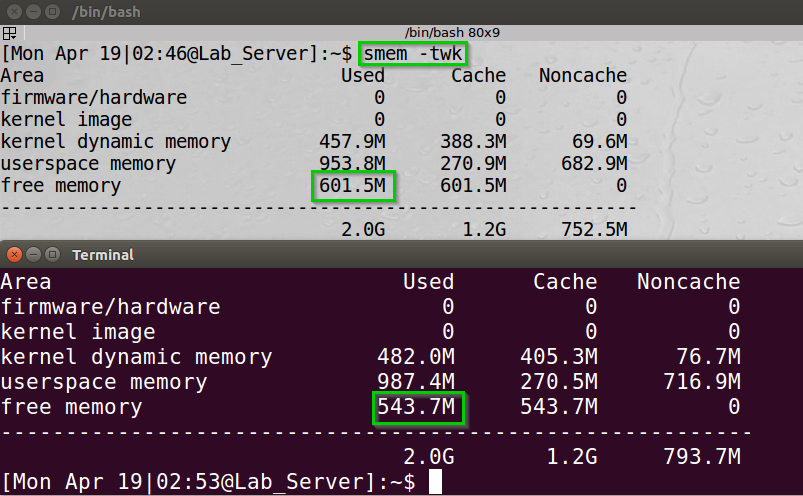


  
Ran the script.

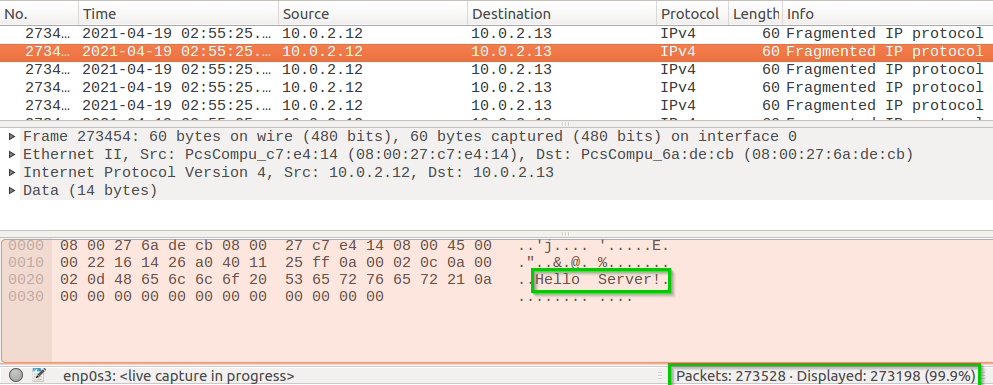
On the Netcat server the output was empty – since the packet was not completed.

smem is a tool for reports on memory usage on Linux.   
‘t’ shows the total amount.  
‘w’ shows summary of memory usage.  
‘k’ shows the unit suffixes.

By executing “smem -twk” on the server before and after the attack we can see the changes



We can see the free memory has dropped, however, not significantly to crash the system. The reason is that all the fragments stuck in the kernel waiting to be completed.

On Wireshark we can see the “Hello Server!” sent in the fragments and 273,528 packets sent from our client.

I could give the Client more memory and give the Server less memory to try and see the server crush.

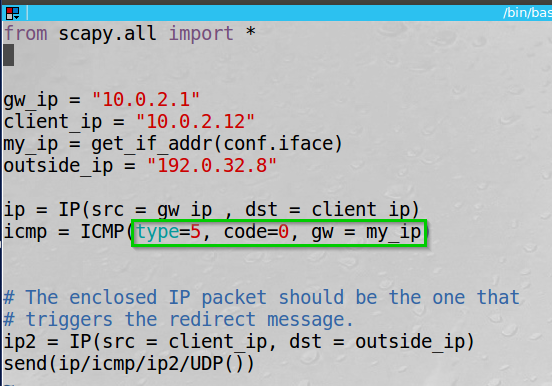
**Task 1 Summary**

* I have succeeded task. The screenshots, Wireshark and terminal output 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, this part of the lab was the hardest and most unintuitive.

**Task 2: ICMP Redirect Attack**

**Task Description:**

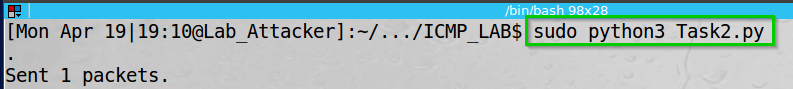
An ICMP redirect is an error message sent by a router to the sender of an IP packet. Redirects are used when a router believes a packet is being routed incorrectly, and it would like to inform the sender that it should use a different router for the subsequent packets sent to that same destination.

I wrote the following code in python using Scapy:

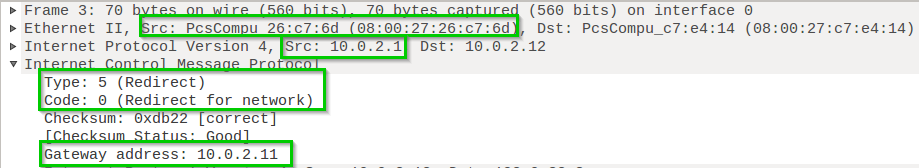
I set the relevant ip addresses.:  
gw\_ip is the original gateway ip.  
client\_ip is the victim.  
my\_ip is my ip got using Scapy get\_if\_addr function  
outside\_ip is the IP address of [www.iana.org](http://www.iana.org)

On the client I accept ipv4 redirects.

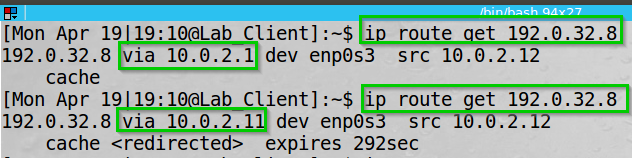
We can see the command executed successfully.

On the attacker I run the code.

We can see the packet sent.

On Wireshark we can see the packet sent.

The source IP is of the real gateway. However, the mac address is ours.  
we can see the ICMP type 5 (Redirect) and code 0(Redirect for network) and we specify the gateway address to the attacker IP.

On the client we can see the ip route before and after the attack.

We can see 192.0.32.8 ([www.iana.org](http://www.iana.org)) being redirected through 10.0.2.11 – the attacker.

Questions:  
1. Can you use ICMP redirect attacks to redirect to a remote machine?  
  
No. Redirect message supposed to send to a machine inside the LAN, so we try to tell the client he should use for example “220.2.2.12” the OS would not accept it since it does not belong to the same network.

2. Can you use ICMP redirect attacks to redirect to a non-existing machine on the same network?  
  
No. Again, the ip route table remains the same and using the real default gateway.

The machine checks if it can reach the ip address from the ICMP redirect message and decides if it should update the ip route table (It uses ARP request to check for Reply).

**Task 2 Summary**

* I have succeeded task. The screenshots of Wireshark and terminal can show it.
* 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.

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

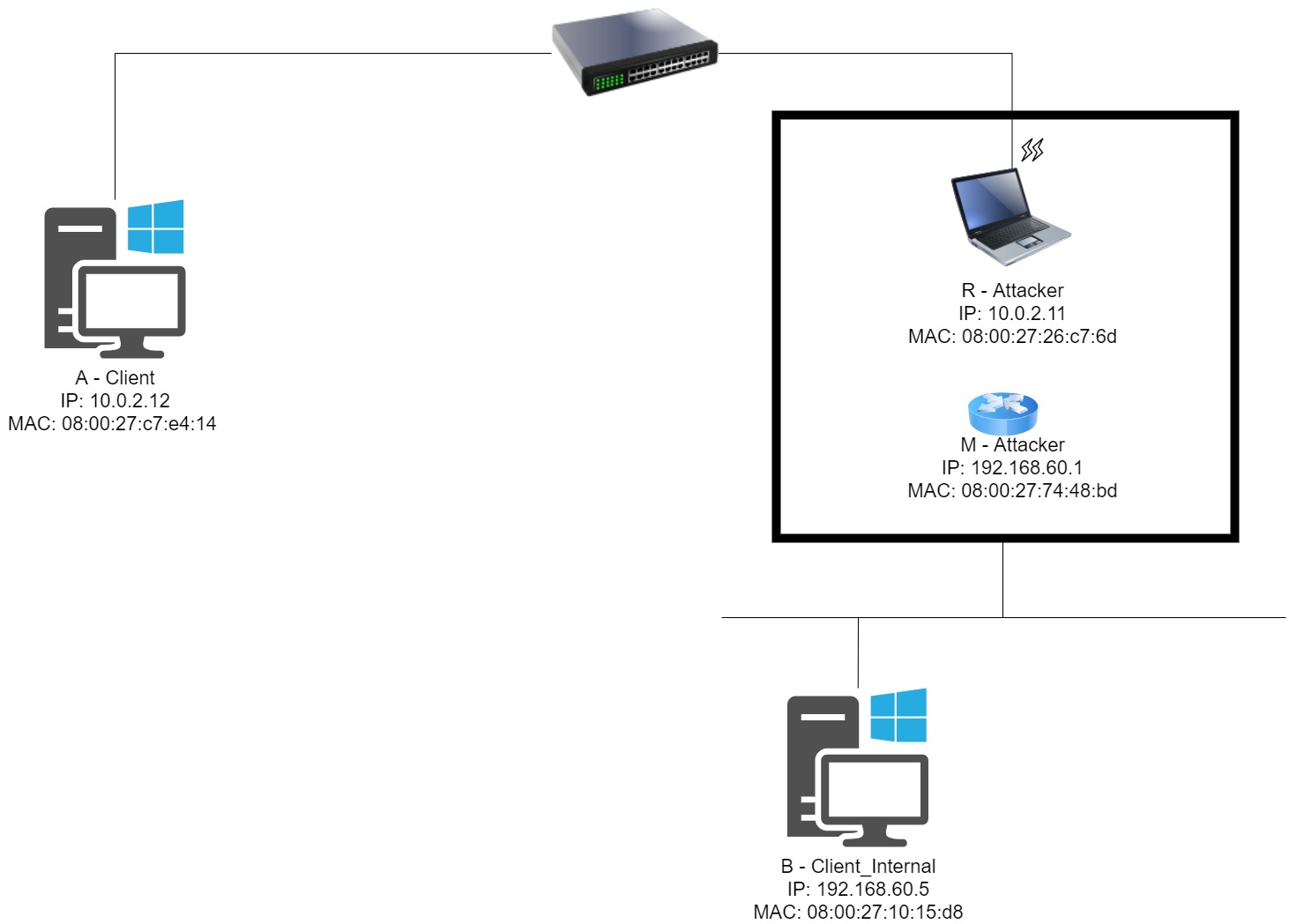
**Task Description:**

The objective of this task is to get familiar with routing and understand a spoof-prevention mechanism called reverse path filtering, which prevents outside from spoofing.

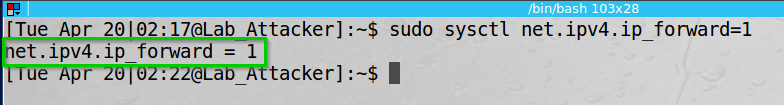
**Task 3.a: Network Setup**

I set the environment as requested.

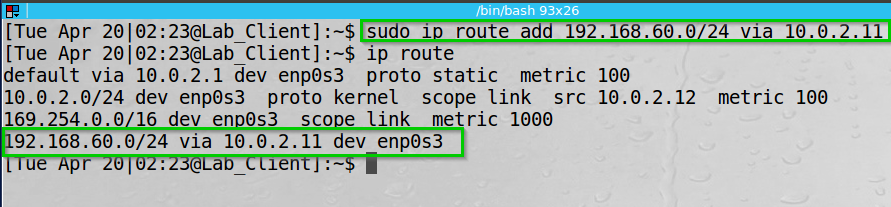
Here is the topology of the network:



**Task 3.b: Routing Setup**

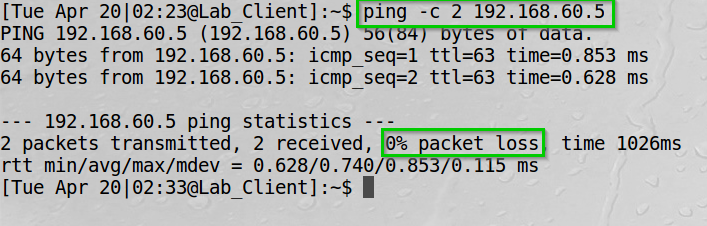
Enabled ip forwarding on the attacker.

We can see the command executed successfully.

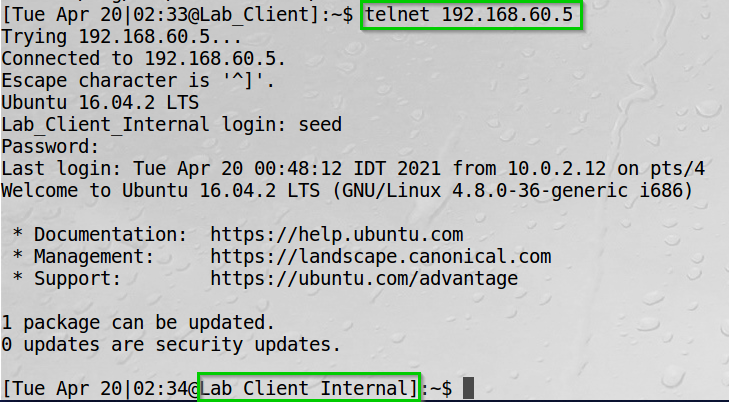
I added a new route for any communication to 192.168.60.0/24 via the attacker.

We can see the ip route table.

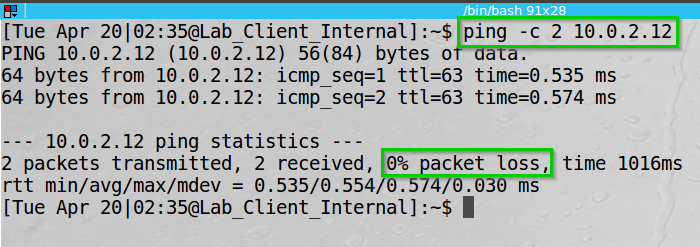
Demonstrate the ping and telnet from each of the clients.

Ping from the client (A) to the client\_internal (B)

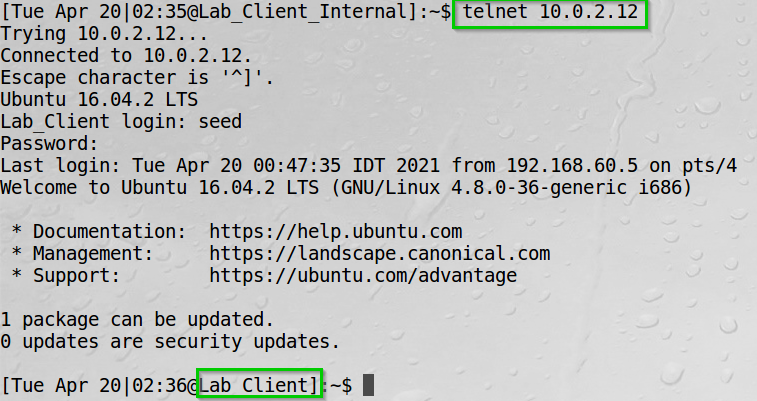
We can see there in not packet loss.

Telnet from the client (A) to the client\_internal (B)

We can see we connected successfully.

Ping from the client\_internal (B) to the client (C)

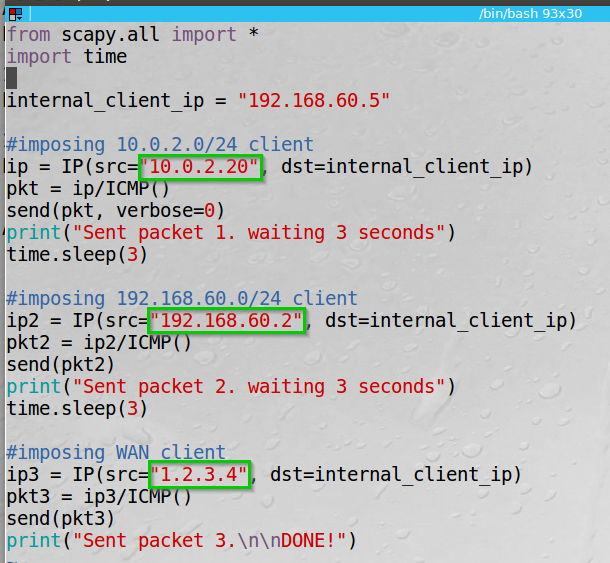
We can see there in not packet loss.

Telnet from the client\_internal (B) to the client (A)

We can see we connected successfully.

**Task 3.c: Reverse Path Filtering**

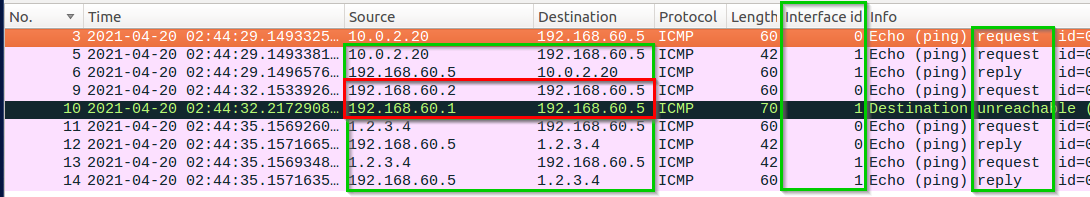
I sent 3 spoofed packets from client (A) to internal\_client (B) to demonstrate the “reverse path filtering” rule.

I wrote the following code in python using Scapy.

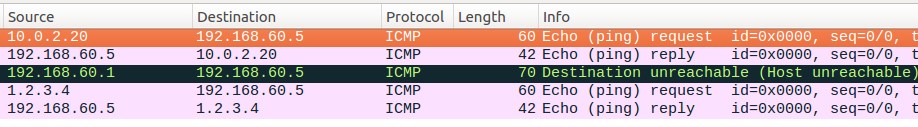
I send 3 ICMP packets to the internal client (192.168.60.5 – Host B) from the client (10.0.2.12 – Host A).

The reason I used time sleep was to see the packets apart to better understand what happens.

On the client, Executed the python script and sent the 3 packets.  
Packet 1: from 10.0.2.0/24   
Packet 2: from 192.168.60.0/24  
Packet 3: from 1.2.3.0/24

By exploring Wireshark on the attacker (R) we can see both the interfaces and see the behavior of “reverse path filtering”.

We can see that the first and third ICMP packets were requested and replied successfully.  
We can see they both routed through the same interface – hence complying with the requirements of the kernel rules of “reverse path filtering”.  
The second ICMP request however was dropped since the interfaces are different.

On the client\_internal (Host B) we can see the behavior we see on Interface id 1 from the Attacket (R) host.

**Task 2 Summary**

* I have succeeded task. The Wireshark screenshots can show it.
* 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, this part of the lab was very intuitive and fun.

**Lab Summary**

Task 1 was challenging. It took me time to remember how fragmentation works (and I am still ambiguous about it) and the fact that I should consider the multiplication of 8 when thinking about the overlapping.

Task 2 was quite straight forward. I changed the route of the packets to another “router”.

Task 3 was interesting, and I have learned about routing tables and filtering rule “reverse path filtering”.

In conclusion I think the lab was informative and very interesting – however not intuitive.