The first part of the Lab1 was more of getting started with the intricacies of raw socket programming unlike the cooked version that was used in the previous lab.

I had fun exploring how we could modify the packets on each layer and how the attacks can be performed by utilizing the raw sockets.

Task0.a

For Task 0.a of building an ICMP echo request and trying to mimic the ping functionality, I first researched about the composition of a ICMP echo request and got to know that it will need an ICMP packet to be sent which would have the type 8 and code 0[1] along with other attributes related to the ICMP protocol.
First I set up a raw socket of type ICMP using the below command:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_ICMP)
```

As I was only interested in sending the ping request, I did not explicitly write the IP headers and Ethernet headers to encapsulate this packet. I let the kernel do it for me.
I used the struct.pack property in python to configure all the icmp details together.
To calculate the checksum, I referred the RFC[2] and resources[3] to get to a sweet spot and move ahead with the request.

First, I created the dummy icmp header without the checksum and passed it to the checksum function to calculate the total checksum and then populated the actual icmp packet.

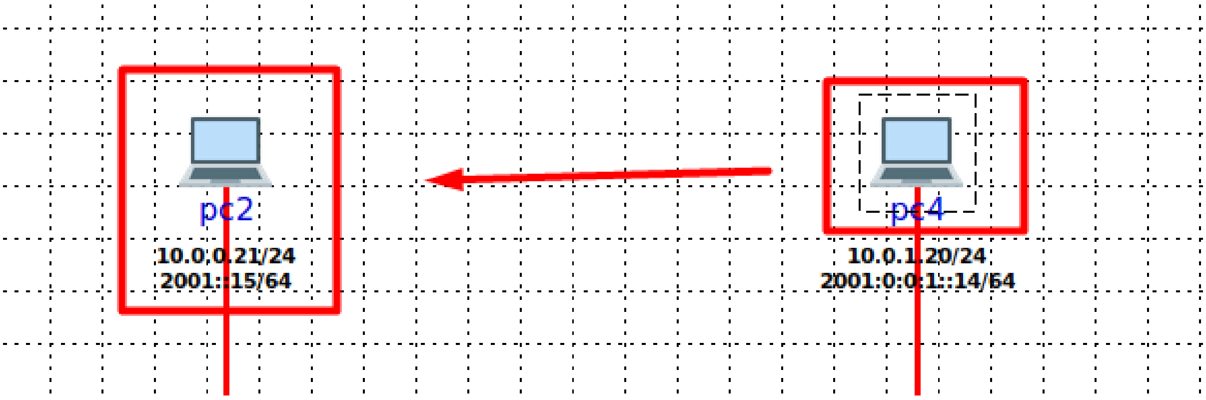
Learning wise, I feel it did not take me much time for the actual implementation of icmp packet but there were some issues with the checksum calculations that took some time.

To run Task0.a:
Files needed:
icmp_basic.py
test-net.xml

Modifications in the file: Edit the IP address in the code to the destination IP address you want to send the icmp request to

```
        return

if __name__ == '__main__':
    #Manually add the destination IP address, easy to change later
    dest_addr = "10.0.0.20"
    icmp_echo(dest_addr)
```

Execution:



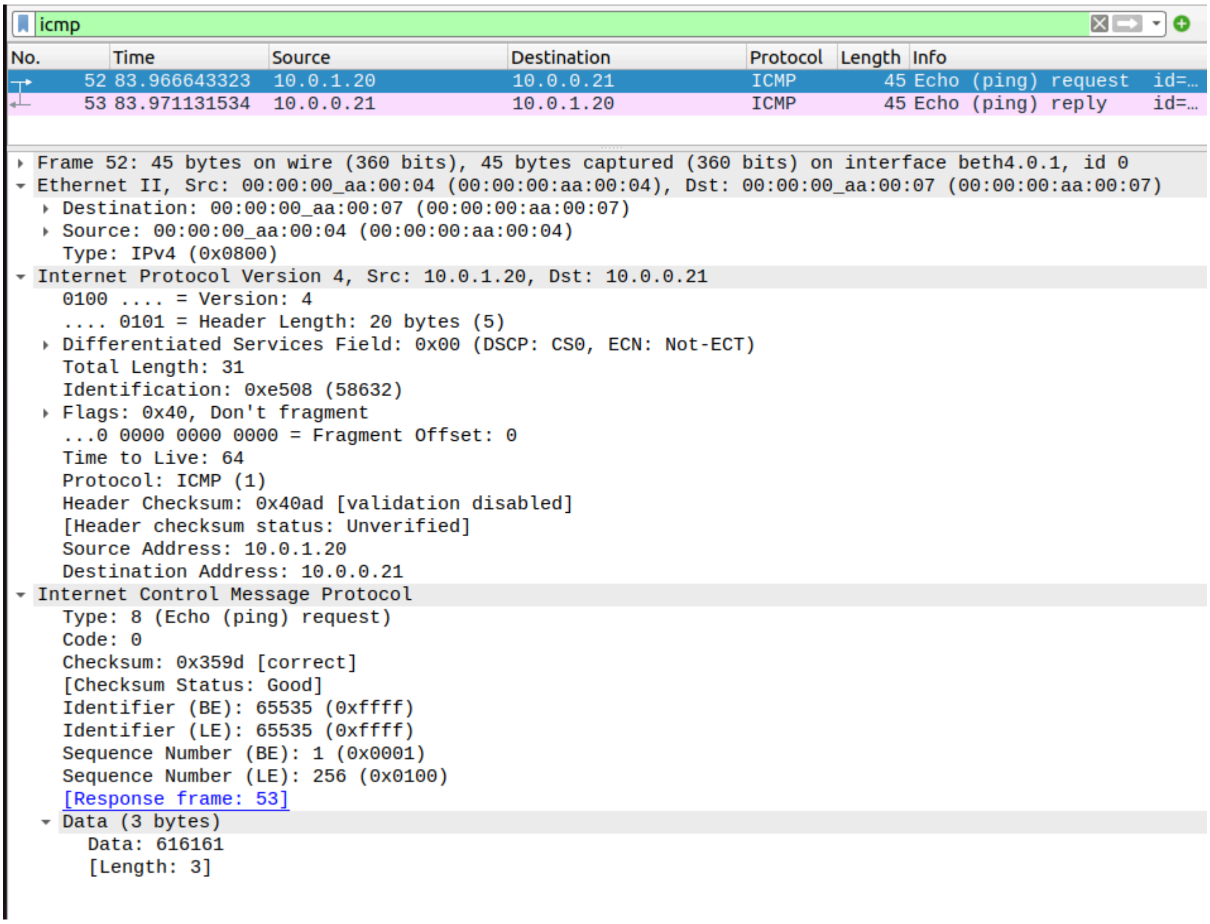
First spin up wireshark on the node you are sending request from.
In my case, I wanted to send a request from 10.0.1.20(pc4) to 10.0.0.21(pc2) so I first opened wireshark using sudo wireshark and then using the interface beth4.0.1

Once the wireshark is up, filter for icmp packets

After the modification and spinning up wireshark, open the terminal on pc4 and navigate to the directory of the file and run using python3 icmp_basic.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 icmp_basic.py
Sending ICMP Echo Request to 10.0.0.21
Packet sent
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1#
```

Checking on wireshark for the response:



Task0.b

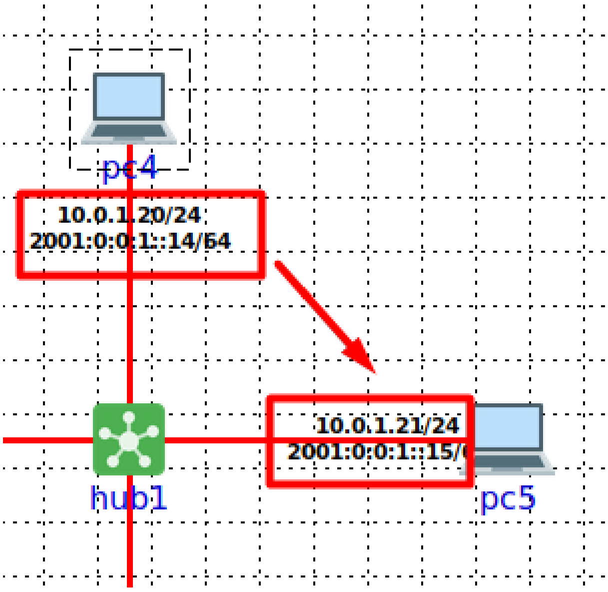
For the Task0.b, the goal was to send an ARP request packet. As I had not previously built the custom ethernet header and ARP deals with Ethernet addresses, as explained in the class I leveraged the fcntl and ioctl[4] utility to get the mac address of my node.
To form the ARP packet I referred the lecture notes and some online resources[5] for the parameters to be populated for performing the request.
I first created a raw socket for arp packet using the command:

```
sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
socket.htons(ETH_P_ARP))
```

As this was an ARP request the request was of type 1 and there was the ethernet type for ARP of 0x0806[6] to be included while building the packet.
As we were creating the manual ARP packet, it required Source IP, source mac, destination ip, destination mac. I found the source ip and source mac using the ioctl utility. I knew the destination IP but did not know the destination mac(reason why using ARP). So I populated the destination mac address while packing my arp struct with a broadcast address: ff:ff:ff:ff:ff:ff.
This would send the arp packet to all the devices in my subnet and then only the destination with the IP mentioned would respond back.
The request would be broadcast and the response unicast.
I then packed the ARP packet with the ethernet header and sent the request and successfully got the response

Execution:
To run this task, you will need the following files:
arp.py
test-net.xml

For this example, I will be using the two nodes in the same subnet:



In the program, you will need to modify the ip address of the node you want to get the mac address of. In my case I am sending the request from pc4 with ip 10.0.1.20 to get the mac address of 10.0.1.21

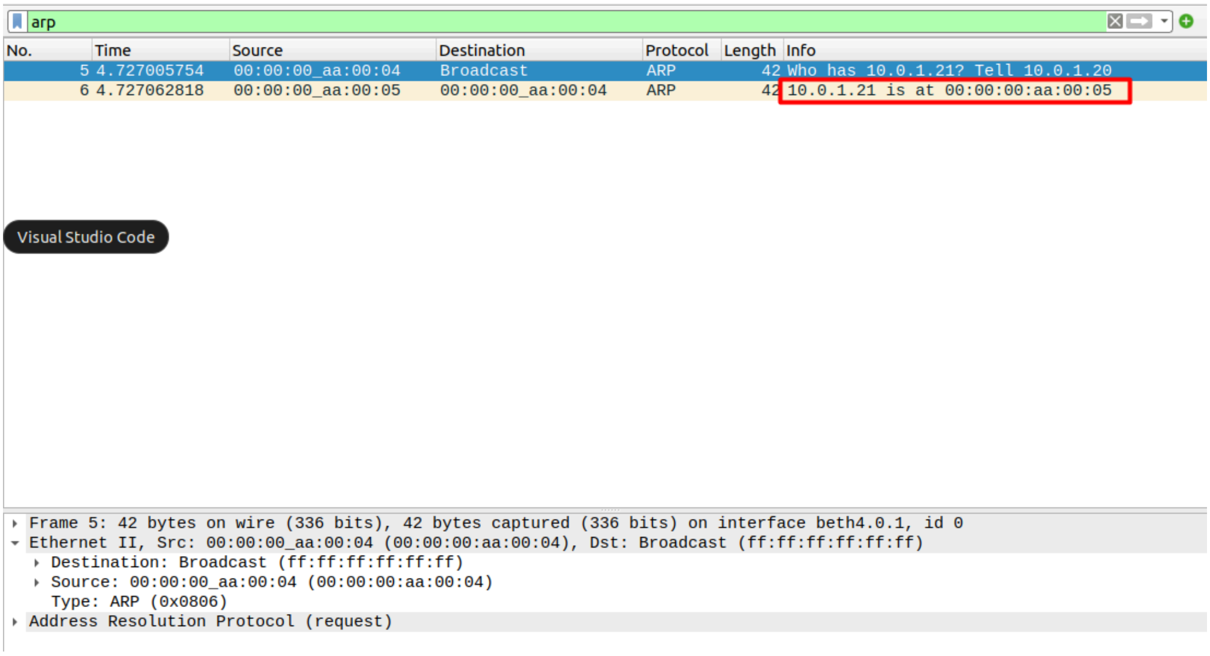
```
    #Manually add the destination IP address, easy to change later
    dest_ip = "10.0.1.21"
    send_arp_request(dest_ip)
```

Modify this accordingly in the code.

After the modification, open up the wireshark on the interface linked to pc4 and start a capture for arp packets
Once wireshark is up, now you will have to open the terminal on pc4 and navigate to the directory where the program is and run it usign python3 arp.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 arp.py
ARP Request sent to 10.0.1.21
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1#
```

Check wireshark for the response:



Task0.c

For the last part of task0, we have to send a TCP syn message. Here I had to modify the checksum function to match the tcp pseudo header calculations, tcp header details and the payload(null in this experiment). Length of the header was also taken into consideration for padding while calculating the proper checksum[8] and then using one's complement to sum up the details.
For the raw socket, I had to create a socket of type TCP so I used the below command to initialise the socket:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_TCP)
```

I used the functions from the last task to get the source IP and part of the checksum. For the creation of the TCP packets I referred to the lecture and resources[7] for the offset, flags and then packaging the TCP in IP header.

As here I manually created the IP header, I used the following command to tell the kernel to not add IP header on its own:

```
sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
```

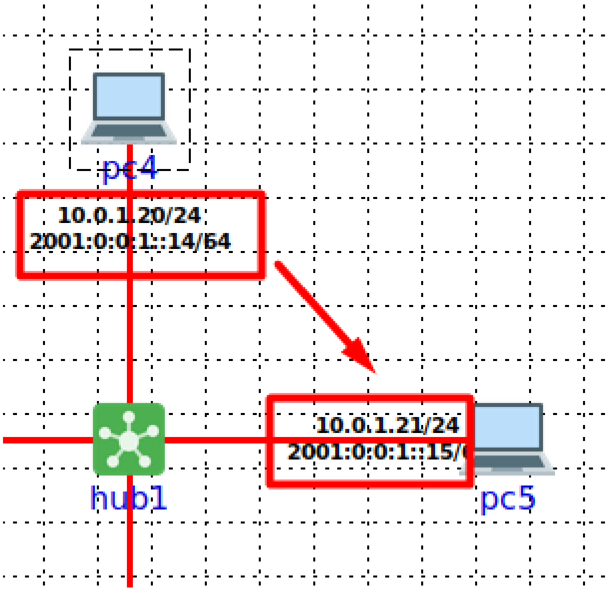
Once all the things were in place, I first proceeded to create a TCP packet and packed it inside an IP header and sent the packet to the destination address with the port.

Execution:
To run Task0.c you will need the files:
tcp.py
test-net.xml

First modify the program to set the destination where the packet has to be sent,the port number of the receiver and the port number of the sender.

```
#Manually add the destination IP address, ports to be used, easy to change later
dest_ip = "10.0.1.21"
# Random source port
source_port = 0x6789
# Port 80
dest_port = 0x50
```

For this example, I am sending a syn packet from pc4 to pc5



So the destination is 10.0.1.21, I have set the port to 80(destination) and random port on sender side

Open wireshark on pc4 interface and start capture for tcp packets.

To run the program, open a terminal on pc4, navigate to the directory where the program is and run using python3 tcp.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 tcp.py
TCP SYN packet sent to 10.0.1.21:80 on eth0.
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1#
```

On the wireshark side:

tcp

No.	Time	Source	Destination	Protocol	Length	Info
922	1513.2502437…	10.0.1.20	10.0.1.21	TCP	54	26505 → 80 [SYN] Seq=0 Win=53270 Len=0
923	1513.2502842…	10.0.1.21	10.0.1.20	TCP	54	80 → 26505 [RST, ACK] Seq=1 Ack=1 Win=0 L…

As we can see, we were able to send the SYN message and the pc5 device responded with rst, ack message as the node is not actively listening on the port 80.

To check this with the active listening port- Open a terminal on pc5 and open a listener on the port 80 using the command

```
root@pc5:/tmp/pycore.1/pc5.conf# nc -l -k 80 > /dev/null
```

Once done, head back to the pc4 terminal and resend the tcp syn packet and check wireshark

tcp

No.	Time	Source	Destination	Protocol	Length	Info
14	21.727448905	10.0.1.20	10.0.1.21	TCP	54	26505 → 80 [SYN] Seq=0 Win=53270 Len=0
15	21.727533164	10.0.1.21	10.0.1.20	TCP	58	80 → 26505 [SYN, ACK] Seq=0 Ack=1 Win=642…
16	21.727543523	10.0.1.20	10.0.1.21	TCP	54	26505 → 80 [RST] Seq=1 Win=0 Len=0

As we can see the node sent a syn ack message in response to our message and we later sent the rst message.

Task 1

To first set up my socket for listening I used the ETH_P_ALL properly to basically get all the packets and then decapsulate them one by one according to the tags.
Building a sniffer was a challenge because, I had to do everything in reverse that I was doing till now and some of the functionalities like decapsulating TCP headers was not working properly. But after some trials, I was able to figure out that it was how the packets were being packed and stored/transmitted in form of arrays. Thus I had to jump 34 index for the info of TCP as first 34 was filled with Ethernet and IP header info. I was able to then parse the information and display it in the opposite was as to how I packed it in the previous tasks with some reference[9].

Execution:
To run the sniffer you will need the below files:
sniffer.py
test-net.xml

No changes in the code are required and the sniffer can be run directly.
For this example, I am running my sniffer on pc4.
I first open up a terminal on pc4 and then navigate to the directory where the program is and run it using python3 sniffer.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 sniffer.py
sniffer started. Press Ctrl+C to stop.

Ethernet Header:
Destination MAC: 01:00:5e:00:00:05
Source MAC: 00:00:00:aa:00:07
EtherType: 0x800

IP Header:
Version: 4
IHL: 5 words (20 bytes)
TOS: 192
Total Length: 64
Protocol: 89
Source IP: 10.0.1.1
Destination IP: 224.0.0.5
```

To verify the logs, I tried sending a tcp packet using a new terminal on pc4 and checking it:

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 tcp.py
TCP SYN packet sent to 10.0.1.21:80 on eth0.
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1#
```

The packet was captured on wireshark:

```
No.  Time          Source      Destination  Protocol  Length  Info
  6  8.424262  Time (format as specified)   10.0.1.21    TCP       54  26505 → 80 [SYN] Seq=0 Win=53270 Len=0
  7  8.424323858   10.0.1.21   10.0.1.20    TCP       58  80 → 26505 [SYN, ACK] Seq=0 Ack=1 Win=642…
  8  8.424331310   10.0.1.20   10.0.1.21    TCP       54  26505 → 80 [RST] Seq=1 Win=0 Len=0

▸ Frame 6: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface beth4.0.1, id 0
▾ Ethernet II, Src: 00:00:00_aa:00:04 (00:00:00:aa:00:04), Dst: 00:00:00_aa:00:05 (00:00:00:aa:00:05)
  ▸ Destination: 00:00:00_aa:00:05 (00:00:00:aa:00:05)
  ▸ Source: 00:00:00_aa:00:04 (00:00:00:aa:00:04)
    Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 10.0.1.20, Dst: 10.0.1.21
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0xffff (65535)
  ▸ Flags: 0x00
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: TCP (6)
    Header Checksum: 0x64a8 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 10.0.1.20
    Destination Address: 10.0.1.21
▾ Transmission Control Protocol, Src Port: 26505, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 26505
    Destination Port: 80
    [Stream index: 0]
    [Conversation completeness: Incomplete (35)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 160069179
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
```

Output on the sniffer:

```
-------------------------

Ethernet Header:
Destination MAC: 00:00:00:aa:00:05
Source MAC: 00:00:00:aa:00:04
EtherType: 0x800

IP Header:
Version: 4
IHL: 5 words (20 bytes)
TOS: 0
Total Length: 40
Protocol: 6
Source IP: 10.0.1.20
Destination IP: 10.0.1.21

TCP Header:
Source Port: 26505
Destination Port: 80
Sequence Number: 160069179
Acknowledgment Number: 0
```

Task 2.1

For the task 2 part 1, to flood the messages, I used the program from Task0.a and modified it a bit. The idea was to send icmp echo requests till the user stopped using a interrupt. I looped the send request function to continuously send the icmp echo messages to the target. Other than everything else was the same.

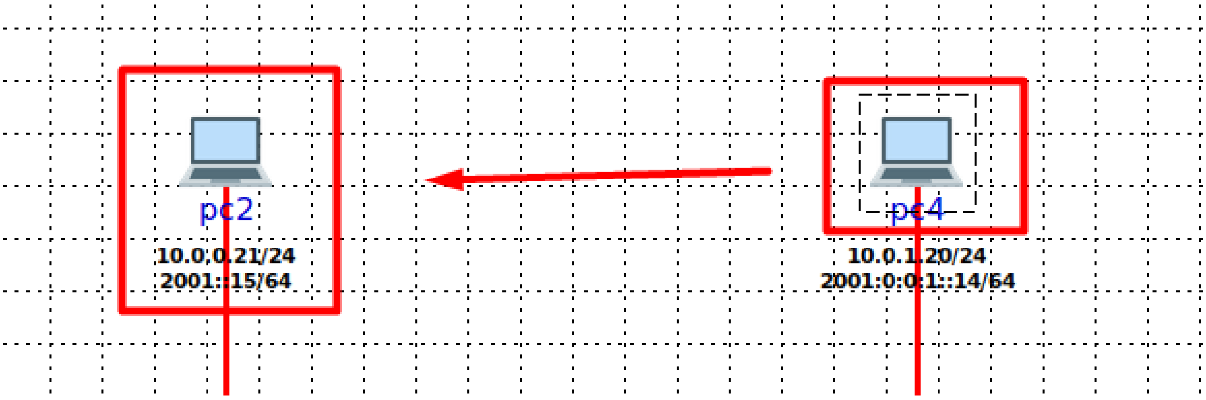
Execution:
To run task2.1, you will need the following files:
icmp_flood_1.py
test-net.xml

For this experiment, I will be using pc4, so first open up wireshark on the pc4 interface and start capture for icmp packets
If you want you can change the destination address for the request in the program:

```
if __name__ == '__main__':
    #Manually add the destination IP address, easy to change later
    dest_addr = "10.0.0.20"
    icmp_echo(dest_addr)
```

Once capture is in place, then open a terminal on pc4 and then navigate the the directory of the program and run it using python3 icmp_flood_1.py

In this experiment, I am sending echo request from pc4 to pc2 as in task0.a:




```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 icmp_flood_1.py
Starting ICMP flood to 10.0.0.20... Press Ctrl+C to stop.
Sent ICMP Echo Request #1 to 10.0.0.20
Sent ICMP Echo Request #2 to 10.0.0.20
Sent ICMP Echo Request #3 to 10.0.0.20
Sent ICMP Echo Request #4 to 10.0.0.20
Sent ICMP Echo Request #5 to 10.0.0.20
Sent ICMP Echo Request #6 to 10.0.0.20
Sent ICMP Echo Request #7 to 10.0.0.20
Sent ICMP Echo Request #8 to 10.0.0.20
Sent ICMP Echo Request #9 to 10.0.0.20
Sent ICMP Echo Request #10 to 10.0.0.20
```

Wireshark:

icmp

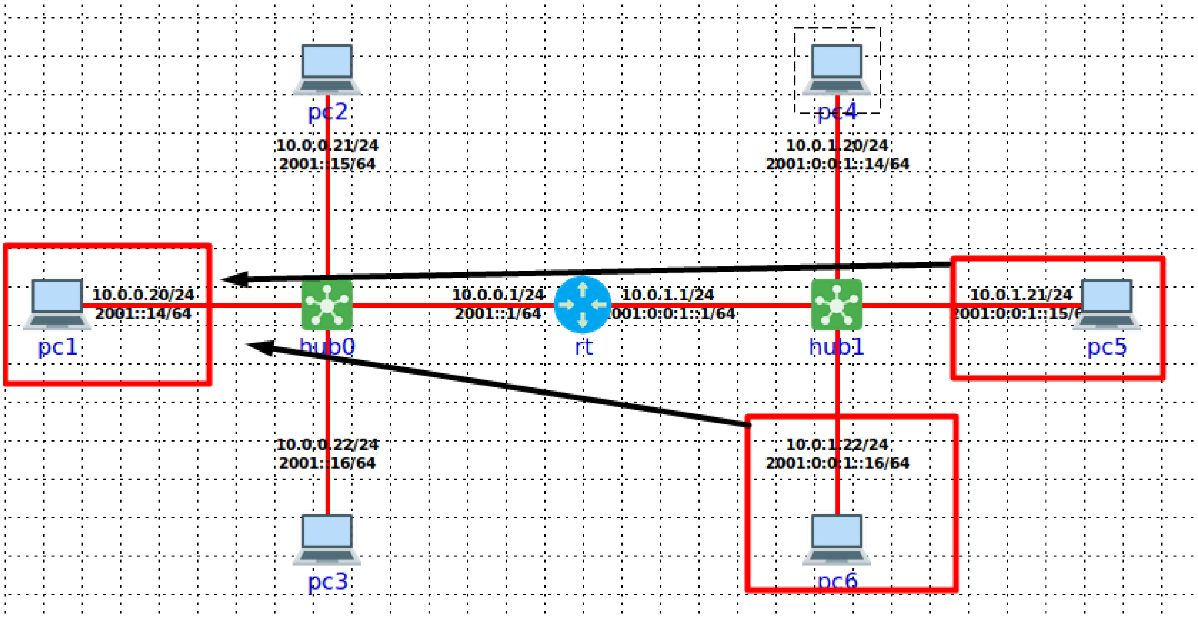
No.	Time	Source	Destination	Protocol	Length	Info
495	802.598366053	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=1/25…
496	802.598671565	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=1/25…
497	802.701096831	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=2/51…
498	802.701169782	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=2/51…
499	802.812896034	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=3/76…
500	802.813133784	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=3/76…
501	802.916532192	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=4/10…
502	802.916588504	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=4/10…
503	803.021795778	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=5/12…
504	803.022162219	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=5/12…
505	803.122786099	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=6/15…
506	803.122860582	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=6/15…
507	803.224809839	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=7/17…
508	803.224899423	10.0.0.20	10.0.1.20	ICMP	45	Echo (ping) reply id=0xffff, seq=7/17…
509	803.326923169	10.0.1.20	10.0.0.20	ICMP	45	Echo (ping) request id=0xffff, seq=8/20…

Task2.2

For the task2.2, we will be using smurf attacks where the target will be getting icmp echo requests from different nodes other than my node. To do this, I used the methodology of Task2.1 but instead of using my own IP address for the echo message, I created a list of IP address of other node that would be used in amplification and used their IP address to send the echo request to the target node.

Execution:
To run task2.2, we will need the below files:
icmp_flood_2_smurf.py
test-net.xml

First we will need to populate the program with amplification IP address from our network and the destination Ip address.
Here I have taken 2 hosts pc5 and pc6 as the amplification hosts and I will be creating IP header using this spoofed IP address to send icmp echo message to pc1



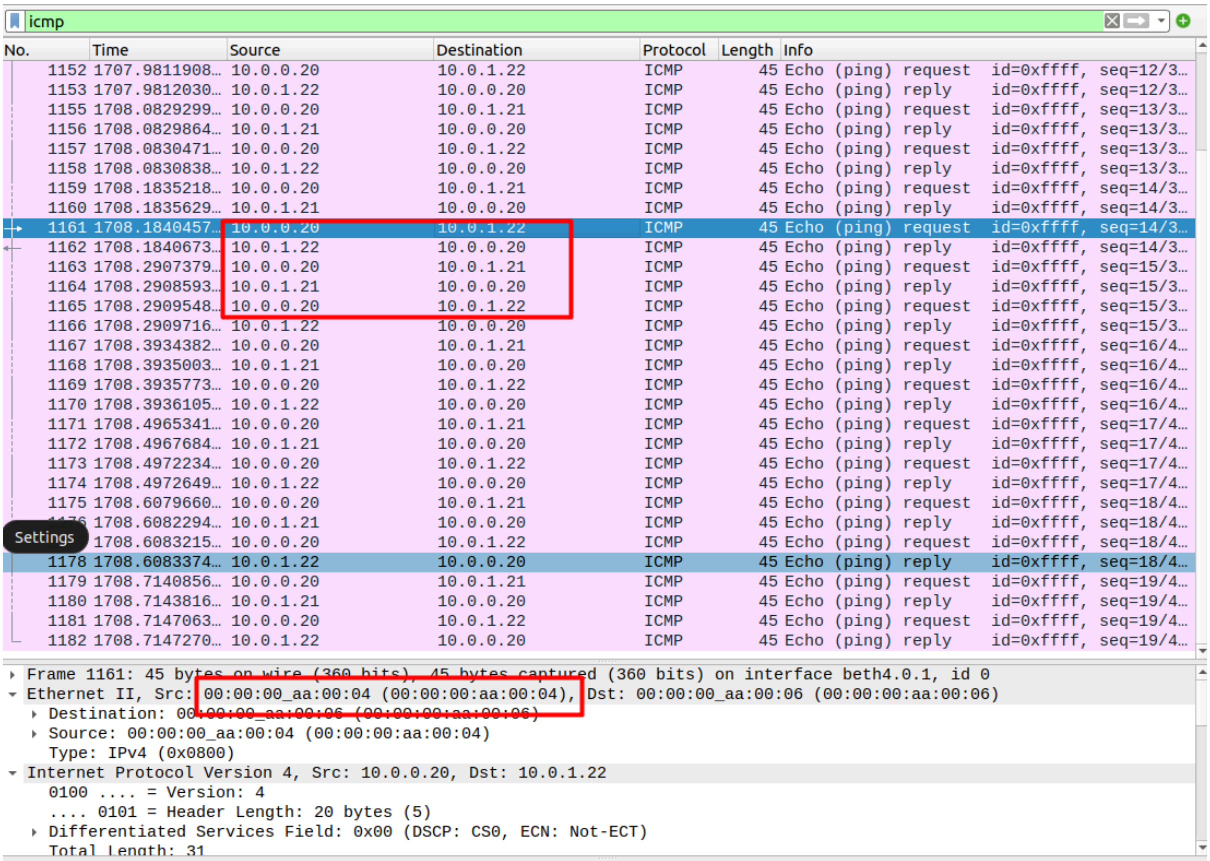

```
# The destination IP address of the target
dest_ip = "10.0.0.20"

# List of nodes in network for amplification
amplification_ips = [
    #"10.0.0.20",   #not including the original IP(own IP)
    "10.0.1.21",
    "10.0.1.22",
]

smurf_attack(dest_ip,amplification_ips )
```

Once modified, open wireshark on pc4(this is where we will initiate the attack from) and start capturing icmp packets.

Open terminal on pc4 and navigate to the directory and run the program using
python3 icmp_flood_2_smurf.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 icmp_flood_2_smurf.p

Starting Smurf attack with 10.0.0.20... Press Ctrl+C to stop.
Sent spoofed ICMP Echo Request #1 from 10.0.0.20 to 10.0.1.21
Sent spoofed ICMP Echo Request #1 from 10.0.0.20 to 10.0.1.22
Sent spoofed ICMP Echo Request #2 from 10.0.0.20 to 10.0.1.21
Sent spoofed ICMP Echo Request #2 from 10.0.0.20 to 10.0.1.22
Sent spoofed ICMP Echo Request #3 from 10.0.0.20 to 10.0.1.21
Sent spoofed ICMP Echo Request #3 from 10.0.0.20 to 10.0.1.22
```

Wireshark:



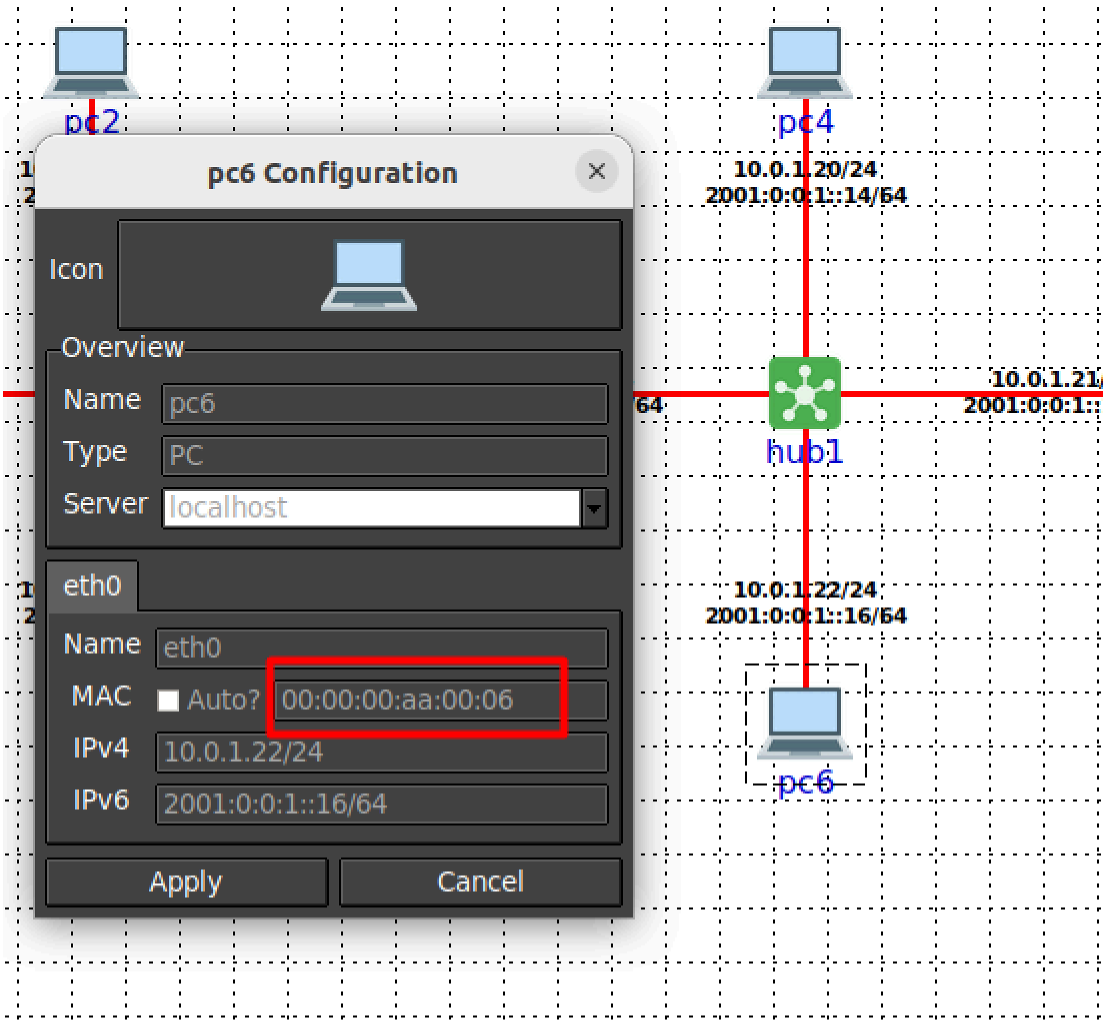
As we can see here, the requests are going from pc1 to pc5 and pc6, but the mac address is still of pc4

Task 3

As we saw, the smurf attack using amplification was successful but the mac address was still from pc4(attacker). To counter this, earlier we were not building the ethernet header in the previous smurf method and only had spoofed IP address in the IP header. The ethernet header would allow us to spoof the mac address too, so now I did that. I hard-coded the mac addresses of the amplification IP's and used it to create an ethernet packet which has the spoofed mac address. This will make the destination directly send packets to the spoofed nodes and the attacker pc4 will not get the packets and the mac address will also will not be visible

Execution:
To run task 3 we will need the below files:
icmp_smart.py
test-net.xml

First I chose pc5 and pc6 for my amplification and checked their mac address from core-gui. In addition to this, I also hard coded the mac address of the target(pc1).



I then updated the IP addresses and their corresponding mac addresses

```
# The destination IP address of the target
dest_ip = "10.0.0.20"

#List of nodes in network for amplification
amplification_ips = [
    "10.0.1.22",
    "10.0.1.21"
]
```

P1 mac address

```
    # Spoofed MAC address of the victim
    spoofed_mac_address = b'\x00\x00\x00\xaa\x00\x01'
```

P5 and p6 mac address

```python
def get_mac_for_ip(ip_address):
    # Mapping of IP addresses used for amplification to MAC addresses to be included in the Ethernet frame
    if ip_address == "10.0.1.22":
        return b'\x00\x00\x00\xaa\x00\x06'
    elif ip_address == "10.0.1.21":
        return b'\x00\x00\x00\xaa\x00\x05'
```

As we are attacking pc1, now we can open wireshark on pc1 interface and then start capturing for icmp packets. Also open one more wireshark capture for pc4 interface

Open terminal on p4(attacker) and navigate to the folder of the program and run the program Python3 icmp_smart.py

```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 icmp_smart.py
Starting Smurf attack... Press Ctrl+C to stop.
Sent spoofed ICMP Echo Request #1 from 10.0.0.20 to 10.0.1.22 with MAC b'\x00\x0
0\x00\xaa\x00\x01' to MAC: b'\x00\x00\x00\xaa\x00\x06'
Sent spoofed ICMP Echo Request #1 from 10.0.0.20 to 10.0.1.21 with MAC b'\x00\x0
0\x00\xaa\x00\x01' to MAC: b'\x00\x00\x00\xaa\x00\x05'
Sent spoofed ICMP Echo Request #2 from 10.0.0.20 to 10.0.1.22 with MAC b'\x00\x0
0\x00\xaa\x00\x01' to MAC: b'\x00\x00\x00\xaa\x00\x06'
```

Wireshark on pc4:

icmp

No.	Time	Source	Destination	Protocol	Length	Info
128	26.623176977	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
129	26.623192776	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
130	26.723379986	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
131	26.723935808	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
132	26.724172108	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
133	26.724191597	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
134	26.825892975	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
135	26.825944562	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
136	26.826026258	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
137	26.826040497	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
138	26.939583636	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
139	26.939622915	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
140	26.940148231	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
141	26.940165106	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
142	27.041882845	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
143	27.041930063	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
144	27.042014597	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
145	27.042031201	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
146	27.145420278	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
147	27.147225792	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
148	27.152079644	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
149	27.152200369	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
150	27.253231663	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
151	27.253399036	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
152	27.253608497	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
153	27.253633270	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
154	27.355031280	10.0.0.20	10.0.1.22	ICMP	45	Echo (ping) request id=0xffff, se
155	27.355088004	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se
156	27.355180014	10.0.0.20	10.0.1.21	ICMP	45	Echo (ping) request id=0xffff, se
157	27.355193654	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply id=0xffff, se

```
› Frame 16: 45 bytes on wire (360 bits), 45 bytes captured (360 bits) on interface beth4.0.1, id 0
▾ Ethernet II, Src: 00:00:00_aa:00:01 (00:00:00:aa:00:01), Dst: 00:00:00_aa:00:06 (00:00:00:aa:00:06)
  › Destination: 00:00:00_aa:00:06 (00:00:00:aa:00:06)
  › Source: 00:00:00_aa:00:01 (00:00:00:aa:00:01)
    Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 10.0.0.20, Dst: 10.0.1.22
    0100 .... = Version: 4
```

As we can see the IP addresses have been spoofed and the mac addresses too have been spoofed.

Wireshark on pc1:

icmp

No.	Time	Source	Destination	Protocol	Length	Info
22	33.767829076	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
23	33.767932091	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
24	33.868500543	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
25	33.868576633	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
26	33.969362068	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
27	33.969679299	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
28	34.070749754	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
29	34.071003032	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
30	34.172871061	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
31	34.173123471	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
32	34.273891034	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
33	34.274047748	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
35	34.376313475	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
36	34.376857346	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
37	34.478863887	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
38	34.491152561	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
39	34.591504802	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
40	34.592362126	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
41	34.694336216	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
42	34.694648173	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
43	34.798764110	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
44	34.799005945	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
45	34.899636627	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
46	34.899998556	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
47	35.002055701	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
48	35.002192887	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply
49	35.103976898	10.0.1.22	10.0.0.20	ICMP	45	Echo (ping) reply
50	35.131585426	10.0.1.21	10.0.0.20	ICMP	45	Echo (ping) reply

Help

```
Frame 22: 45 bytes on wire (360 bits), 45 bytes captured (360 bits) on interface beth1.0.1, id 0
Ethernet II, Src: 00:00:00_aa:00:03 (00:00:00:aa:00:03), Dst: 00:00:00_aa:00:00 (00:00:00:aa:00:0
Internet Protocol Version 4, Src: 10.0.1.22, Dst: 10.0.0.20
Internet Control Message Protocol
```

Pc1 being overwhelmed with echo replies it did not request

Task4:

In this task we have to perform a tcp-syn attack similar to that of task2.2. So here first I create a raw socket of type TCP and tell the kernel not to include the IP header as we are gonna manually create it using the command:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_TCP)
```

And

```
sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
```

To send multiple syn messages, I used the combination of techniques to build TCP header and IP header from task 0.c and to use amplification like task 2.2 to send many TCP syn messages to the target. The target is then overwhelmed with the requests

Execution:
To run task4, you will need the following files:
tcp_synflood.py
test-net.xml

For this simulation I will be using pc5 as the target and pc2, pc1 as the amplification nodes.

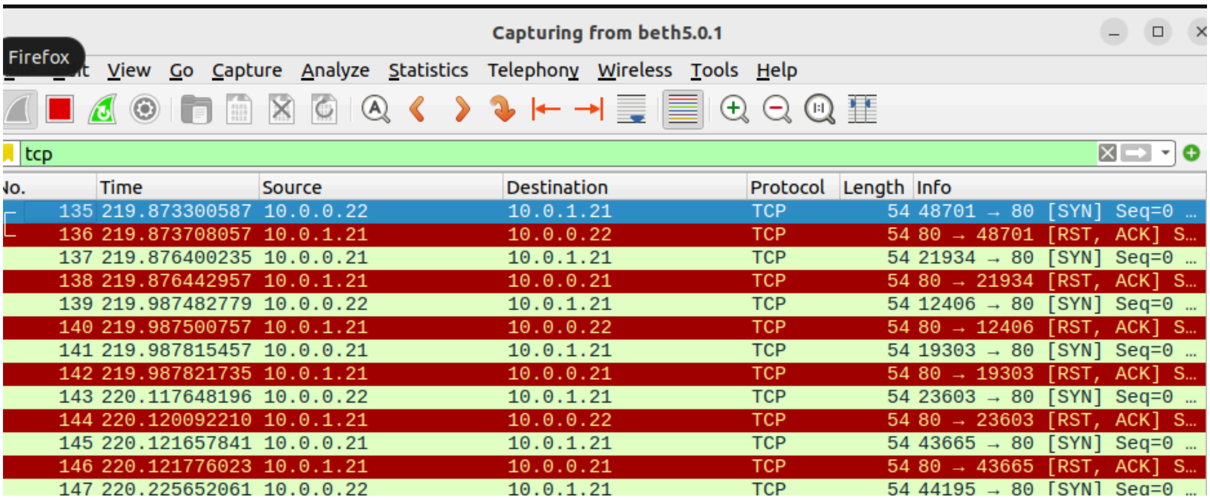
First modify the program to include this IP’s

```
# Port 80
dest_port = 0x50

# List of spoofed source IP addresses that will act as amplifiers
spoofed_source_ips = [
    "10.0.0.22",
    "10.0.0.21"
]

syn_flood(dest_ip,dest_port ,spoofed_source_ips )
```

Now as we are attacking pc5 ,open a wireshark capture on pc5 interface
Now open pc4(attacker) and navigate to the folder of the program and start the attack by Python3 tcp_synflood.py

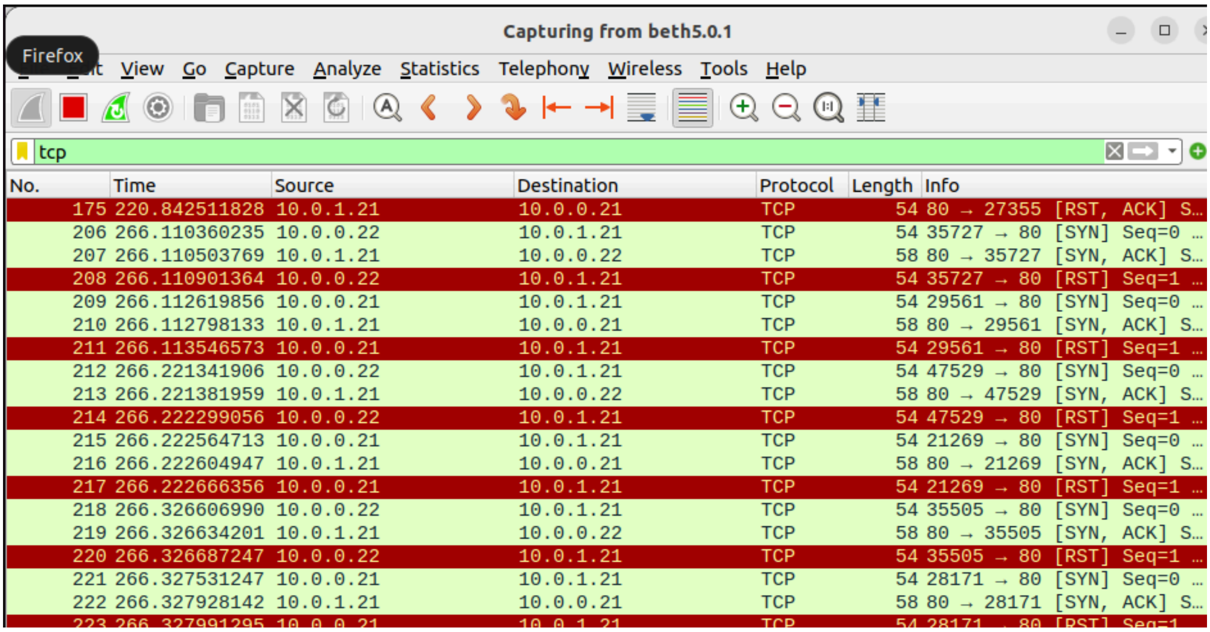
```
root@pc4:/home/inzi/Desktop/SNS/s25-lab1-inzamam1# python3 tcp_synflood.py
 Starting TCP SYN flood... Press Ctrl+C to stop.
Sent spoofed message from 10.0.0.22:43081 to 10.0.1.21:80
Sent spoofed message from 10.0.0.21:39391 to 10.0.1.21:80
Sent spoofed message from 10.0.0.22:2627 to 10.0.1.21:80
Sent spoofed message from 10.0.0.21:14131 to 10.0.1.21:80
Sent spoofed message from 10.0.0.22:21906 to 10.0.1.21:80
Sent spoofed message from 10.0.0.21:37010 to 10.0.1.21:80
Sent spoofed message from 10.0.0.22:3873 to 10.0.1.21:80
Sent spoofed message from 10.0.0.21:14682 to 10.0.1.21:80
```

Now cheking wireshark



As we can see that the IP’s are spoofed and the target is responding to our request with rst, ack as it is not listening.

Wireshark after starting to listen on port 80:



No.	Time	Source	Destination	Protocol	Length	Info
175	220.842511828	10.0.1.21	10.0.0.21	TCP	54	80 → 27355 [RST, ACK] S…
206	266.110360235	10.0.0.22	10.0.1.21	TCP	54	35727 → 80 [SYN] Seq=0 …
207	266.110503769	10.0.1.21	10.0.0.22	TCP	58	80 → 35727 [SYN, ACK] S…
208	266.110901364	10.0.0.22	10.0.1.21	TCP	54	35727 → 80 [RST] Seq=1 …
209	266.112619856	10.0.0.21	10.0.1.21	TCP	54	29561 → 80 [SYN] Seq=0 …
210	266.112798133	10.0.1.21	10.0.0.21	TCP	58	80 → 29561 [SYN, ACK] S…
211	266.113546573	10.0.0.21	10.0.1.21	TCP	54	29561 → 80 [RST] Seq=1 …
212	266.221341906	10.0.0.22	10.0.1.21	TCP	54	47529 → 80 [SYN] Seq=0 …
213	266.221381959	10.0.1.21	10.0.0.22	TCP	58	80 → 47529 [SYN, ACK] S…
214	266.222299056	10.0.0.22	10.0.1.21	TCP	54	47529 → 80 [RST] Seq=1 …
215	266.222564713	10.0.0.21	10.0.1.21	TCP	54	21269 → 80 [SYN] Seq=0 …
216	266.222604947	10.0.1.21	10.0.0.21	TCP	58	80 → 21269 [SYN, ACK] S…
217	266.222666356	10.0.0.21	10.0.1.21	TCP	54	21269 → 80 [RST] Seq=1 …
218	266.326606990	10.0.0.22	10.0.1.21	TCP	54	35505 → 80 [SYN] Seq=0 …
219	266.326634201	10.0.1.21	10.0.0.22	TCP	58	80 → 35505 [SYN, ACK] S…
220	266.326687247	10.0.0.22	10.0.1.21	TCP	54	35505 → 80 [RST] Seq=1 …
221	266.327531247	10.0.0.21	10.0.1.21	TCP	54	28171 → 80 [SYN] Seq=0 …
222	266.327928142	10.0.1.21	10.0.0.21	TCP	58	80 → 28171 [SYN, ACK] S…
223	266.327991295	10.0.0.21	10.0.1.21	TCP	54	28171 → 80 [RST] Seq=1

We can also take this forward and modify the ethernet header to spoof the mac address as well but I did not do that as I have other conflicting deadlines. I am happy that I was able to complete till here and gain handson experience with raw socket programming.

Also, I tried to disable and enable the syncookie using the root commands, but it was throwing some warnings but I could see that the cookie value was being changed.

```
oot@pc5:/tmp/pycore.1/pc5.conf# sudo sysctl -w net.ipv4.tcp_syncookies=0
sudo: unable to resolve host pc5: Temporary failure in name resolution
net.ipv4.tcp_syncookies = 0
```

I don't know if the effect took place as after running for both the iterations I was able to perform the attack. I will dig into this later but after going through the document and from the resource[10]. I can infer that the cookies when enabled can help minimize the resources that are allocated for the handshake. It helps in case of a synflood attack where the attacker is only sending syn messages but no ack to the server's reply, the server does not allocate resources. The allocation of resources is basically dependent on the reply being received from the sender.

Reference-

1. “ICMP (Internet Control Message Protocol),” NetworkLessons.com, Jul. 22, 2015. https://networklessons.com/cisco/ccie-routing-switching-written/icmp-internet-control-message-protocol
2. R. T. Braden, D. A. Borman, and C. Partridge, “Computing the Internet checksum,” Sep. 1988, doi: https://doi.org/10.17487/rfc1071.
3. Python and socket library - Raspberry Pi Forums,” Raspberrypi.com, 2024. https://forums.raspberrypi.com/viewtopic.php?t=362742
4. The fcntl and ioctl System Calls in Python,” Tutorialspoint.com, 2019. https://www.tutorialspoint.com/the-fcntl-and-ioctl-system-calls-in-python
5. Address Resolution Protocol (ARP) Parameters,” www.iana.org. https://www.iana.org/assignments/arp-parameters/arp-parameters.xhtml
6. ARP Protocol Packet Format,” GeeksforGeeks, Feb. 14, 2023. https://www.geeksforgeeks.org/arp-protocol-packet-format/
7. GeeksforGeeks – "TCP/IP Packet Format", https://www.geeksforgeeks.org/tcp-ip-packet-format/
8. GeeksforGeeks – "Calculation of TCP Checksum", https://www.geeksforgeeks.org/calculation-of-tcp-checksum/
9. “Packet sniffer in Python.” Available: https://www.uv.mx/personal/angelperez/files/2018/10/sniffers_texto.pdf
10. Wikipedia Contributors, “SYN cookies,” Wikipedia, Oct. 17, 2019. https://en.wikipedia.org/wiki/SYN_cookies
11.