Lab3 - SEED Labs – Packet Sniffing and Spoofing Lab

- *Lohith Kumar Bhambore*

Setup – Using Containers

- Used the SEED-Ubuntu 20.04 (64-bit) image provided on the website in VM on a windows 10 machine.
- Followed the instructions as demonstrated in the video shared by the TA
- Next, I used the docker-compose.yml file to set up the environment using the alias container commands as mentioned below –
    - $dcbuild: Alias for docker-compose build
    - $dcup: Alias for docker-compose up
    - $dcdown: Alias for docker-compose down
    - $dockps: Alias for docker ps –format “{{.ID}} {{.Names}}”
    - $docksh <id>: Alias for docker exec -it <id> /bin/bash

- The following is the setup of the containers –

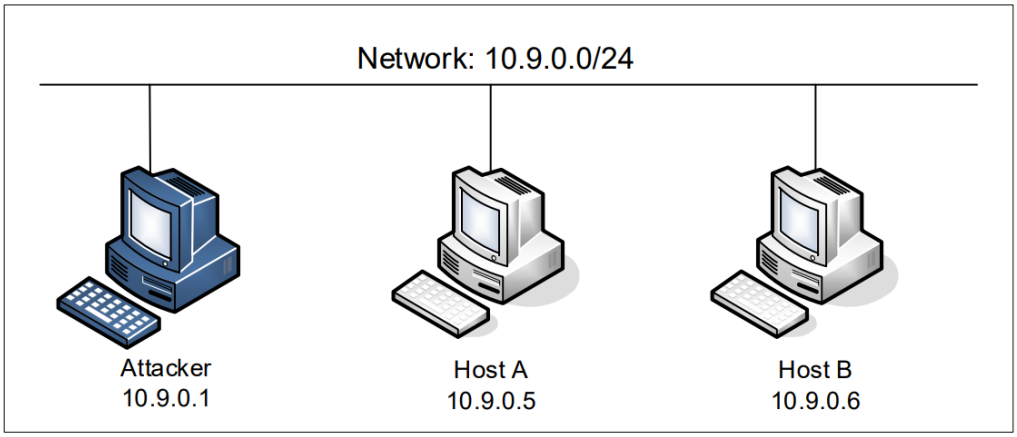
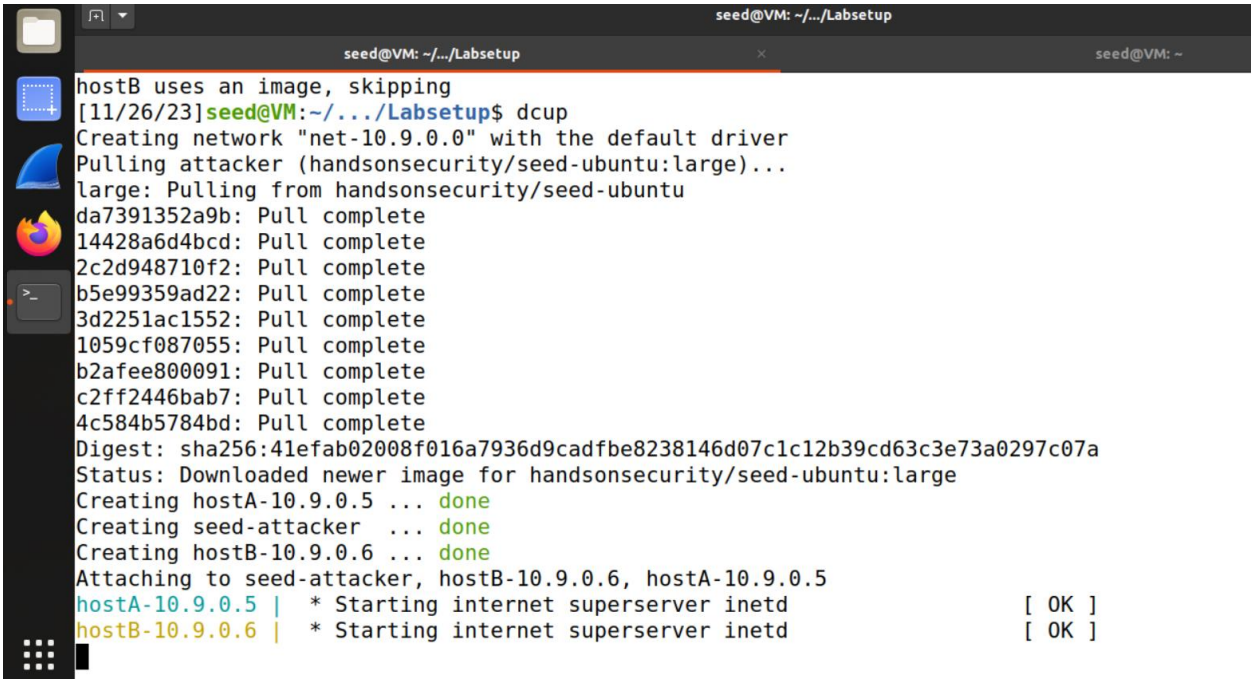


Figure 1: Lab environment setup

- We run these using docker containers using the commands mentioned above



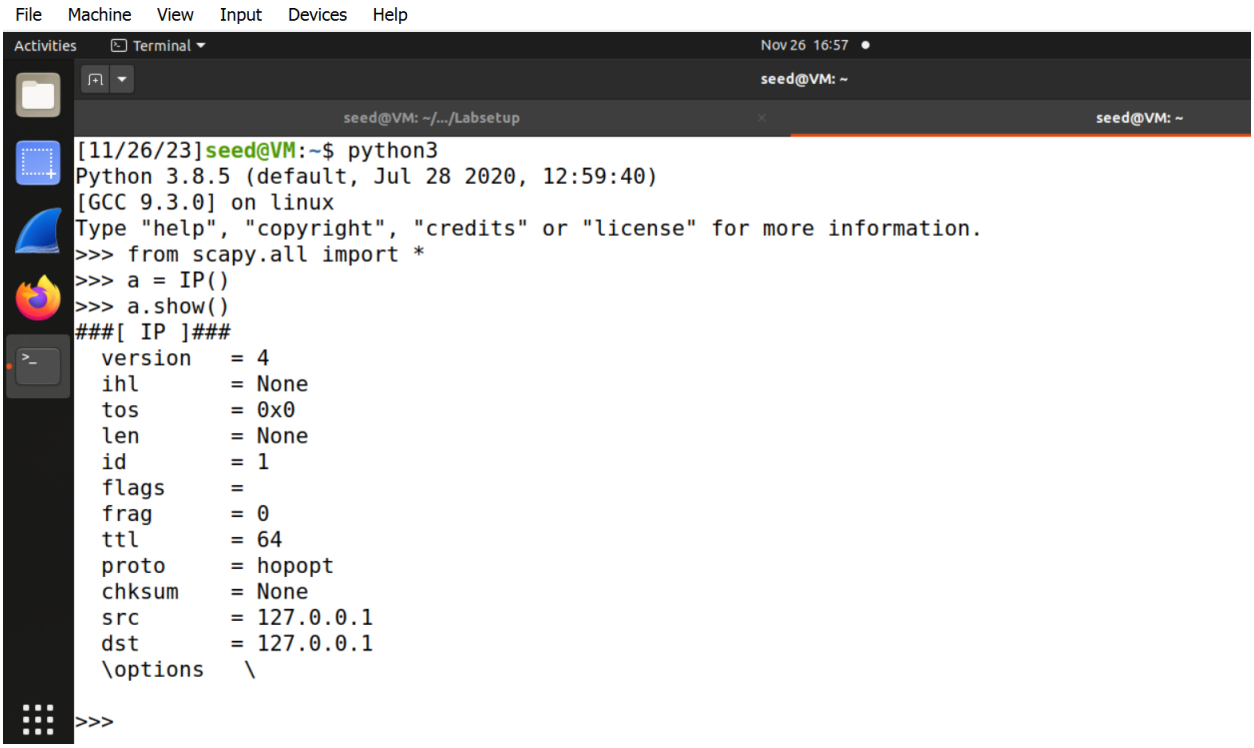
- Points to note about the attacker container –
    - **Shared Folder –** Cide editing inside a VM is easier than in containers as it is possible to use the editors we want. To allow the VM and the containers to share files, we must create a shared folder b/w them using Docker volumes. Using the docker compose file there is an entry which mounts ./volumes folder on the VM to the /volumes folder inside the container. Our code is written in ./volumes folder on the VM
    - **Host mode –** To run sniffer programs in a container, which because of being attached to a virtual switch can only check its own traffic and not of other containers, we make the attacker container a host. This allows this container to see all traffic using the entry – network_mode: host
- To get the network interface name – We need to find the name of our network interface on VM to use it in the programs using the ifconfig command as shown below –

```
[11/26/23]seed@VM:~$ ifconfig
br-2af6eaa1bbb1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:72ff:fe2c:ef18  prefixlen 64  scopeid 0x20<link>
        ether 02:42:72:2c:ef:18  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 47  bytes 7147 (7.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 1 – Using Scapy to Sniff and Spoof Packets

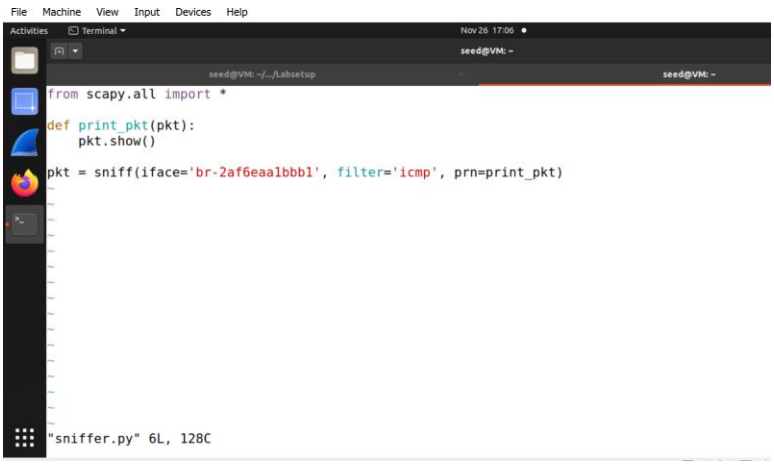
Scapy is used as it can be used not only as a tool but also as a building block to create other sniffing and spoofing tools.

We use scapy using a python program running on root privilege as it is required to spoof packets



Task 1.1 – Sniffing Packets

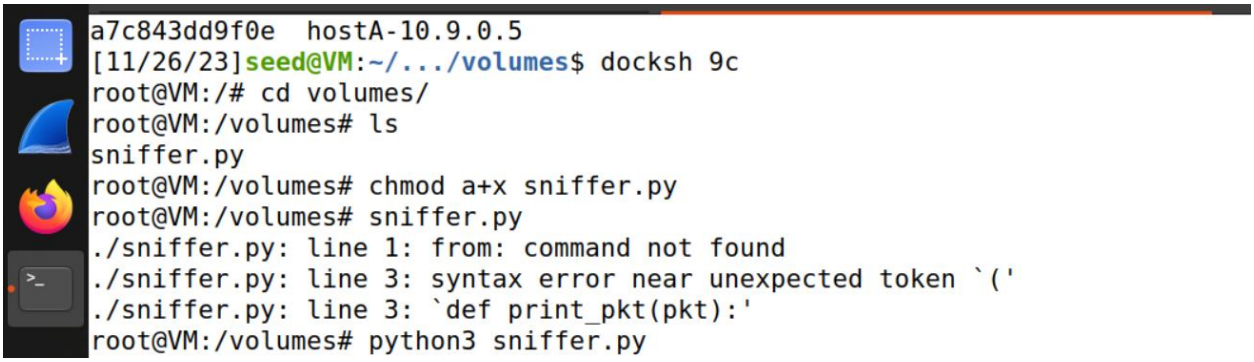
Though Wireshark is a very popular sniffing tool, it can't be used as building block to construct other tools, thus we use scapy. This task is about learning how to use Scapy to do packet sniffing in python program sniffer.py in volumes folder in VM, which is also accessible in container –



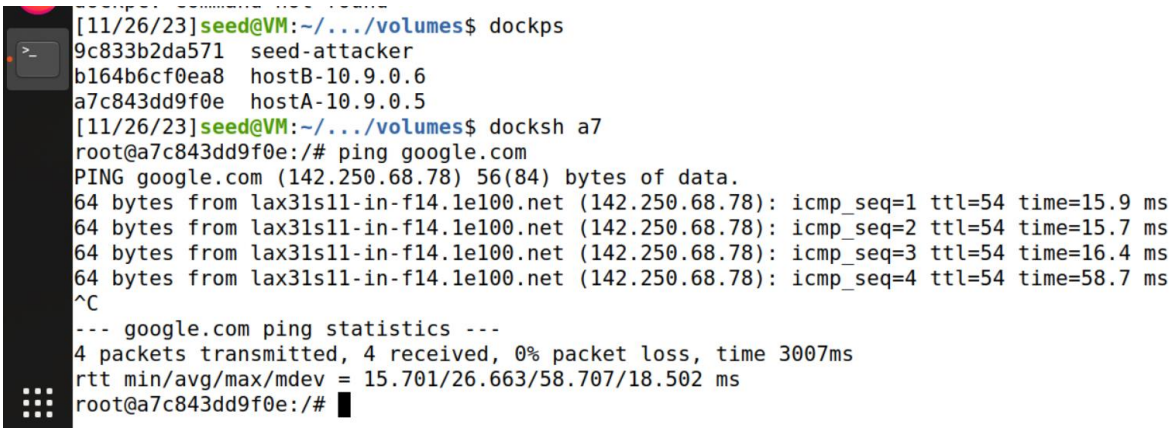
Task 1.1A

As can be seen in the code above, we call the function print_pkt() for each packet sniffed, which will then print out some information about the packet.

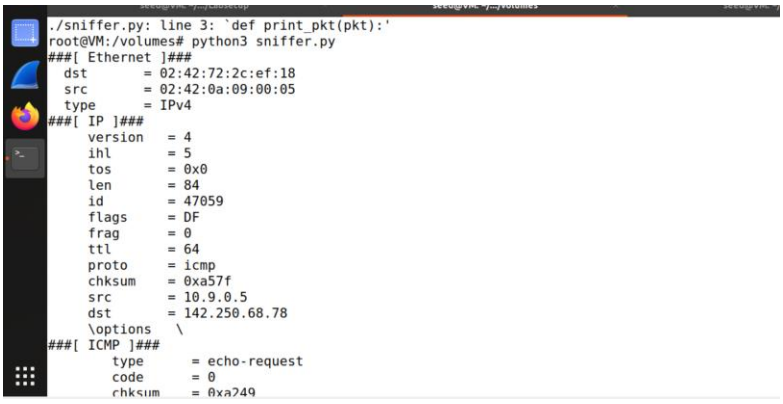
Let's run the program with **root** privileges in attacker container –



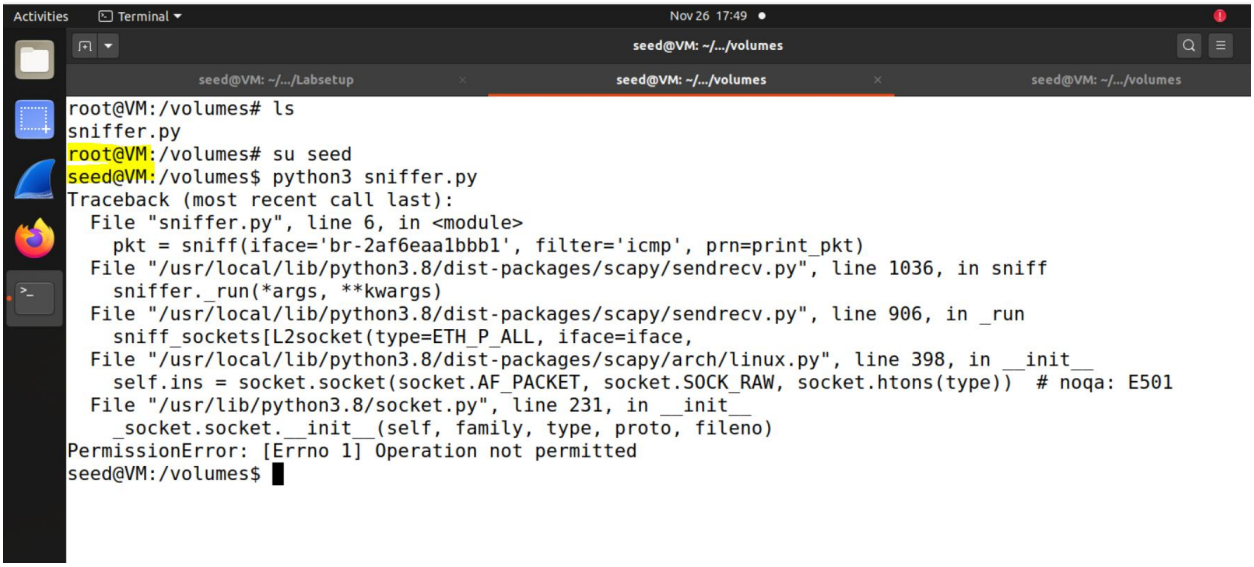
Let's ping google.com from hostA –



And we can see the packet information in the attacker container and since we used the ICMP filter in our python file, we see ICMP information of the packet. We are now able to sniff every packet being sent from hosts A and B. We can also see that echo-request is of type ICMP and the echo-reply type when we receive a reply from the ping –



When we try to run the sniffer.py file without root privileges (in seed account as follows) –

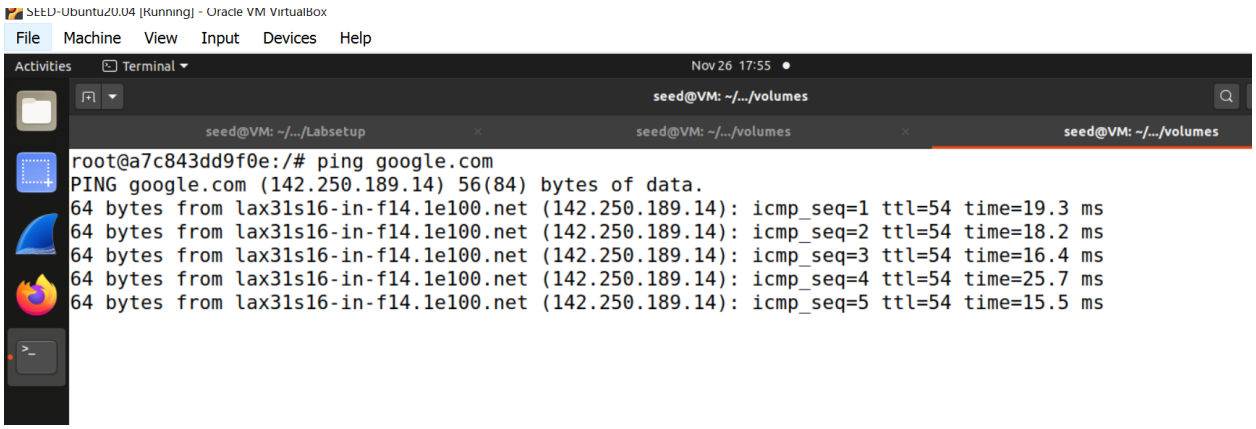


As expected, we get a permission error as sniffing packets can only be done when you have root privileges.

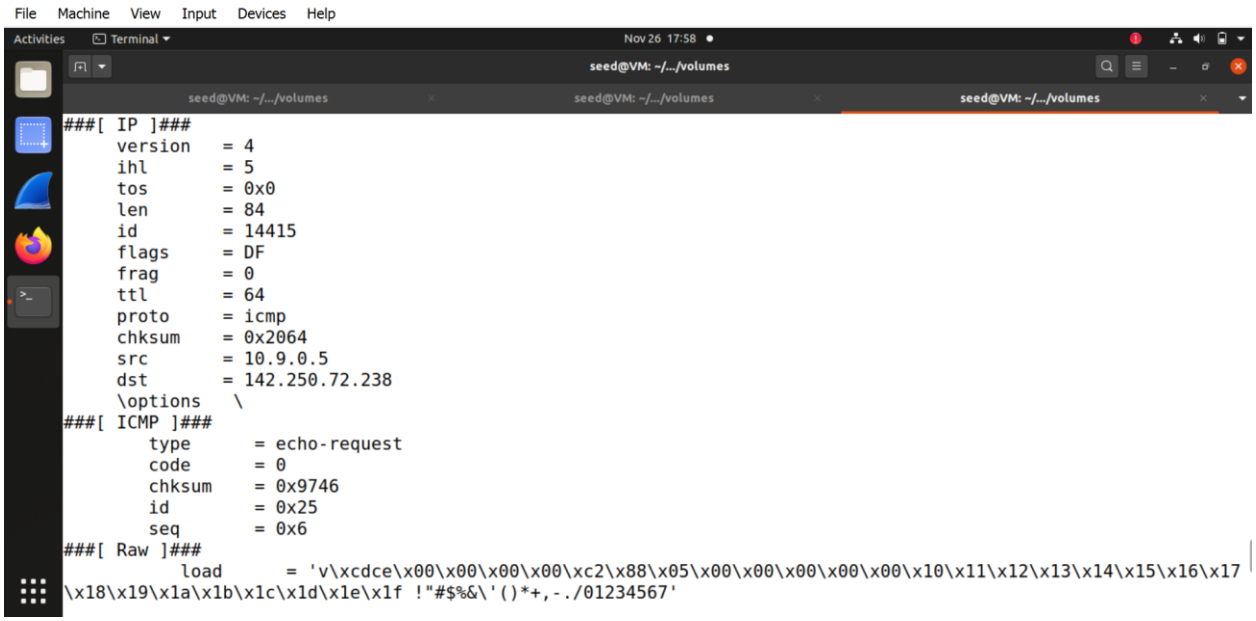
Task 1.1B

Using different filters

- Capture only ICMP packet –
  Ping google.com from Host A



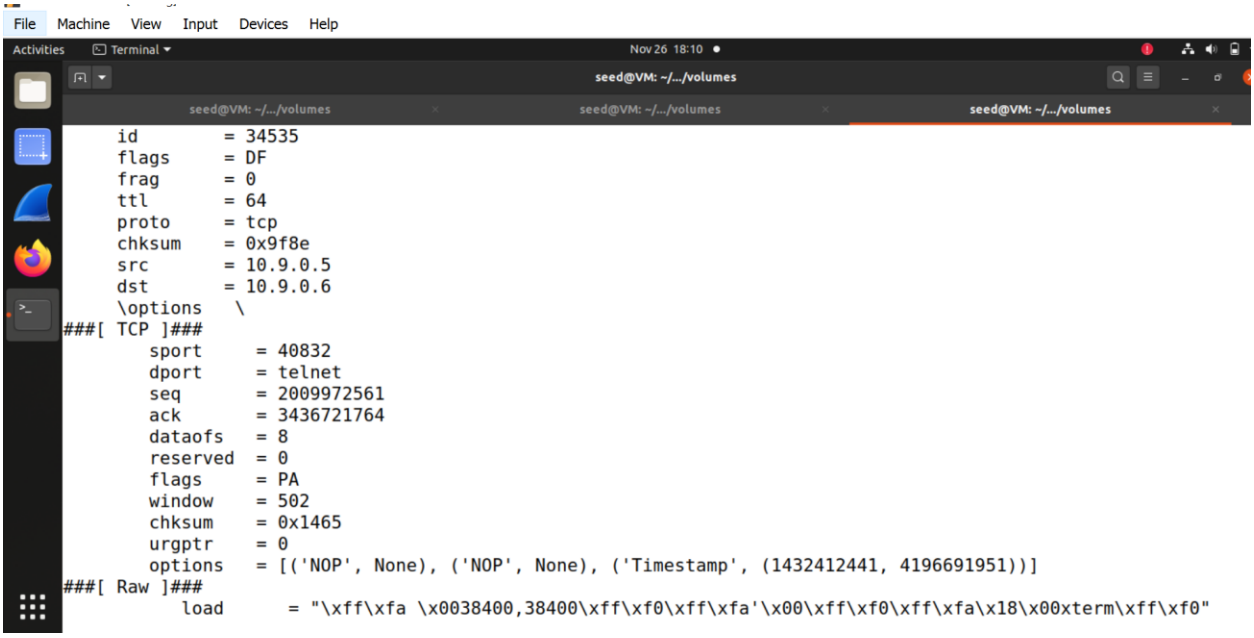
Run sniffer.py with ICMP filter in attacker container and capture packet information –



- Capture any TCP packet coming from a particular IP and with destination port 23 –
  For this we need to modify sniffer.py to change the filter to TCP with port 23 as shown below

Run a telnet command from 10.9.0.5 (Host A) to 10.9.0.6 (Host B)

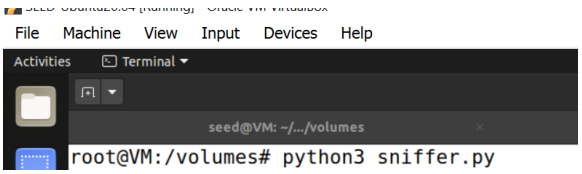
Run this new sniffer.py in attacker container and observe the TCP packets –



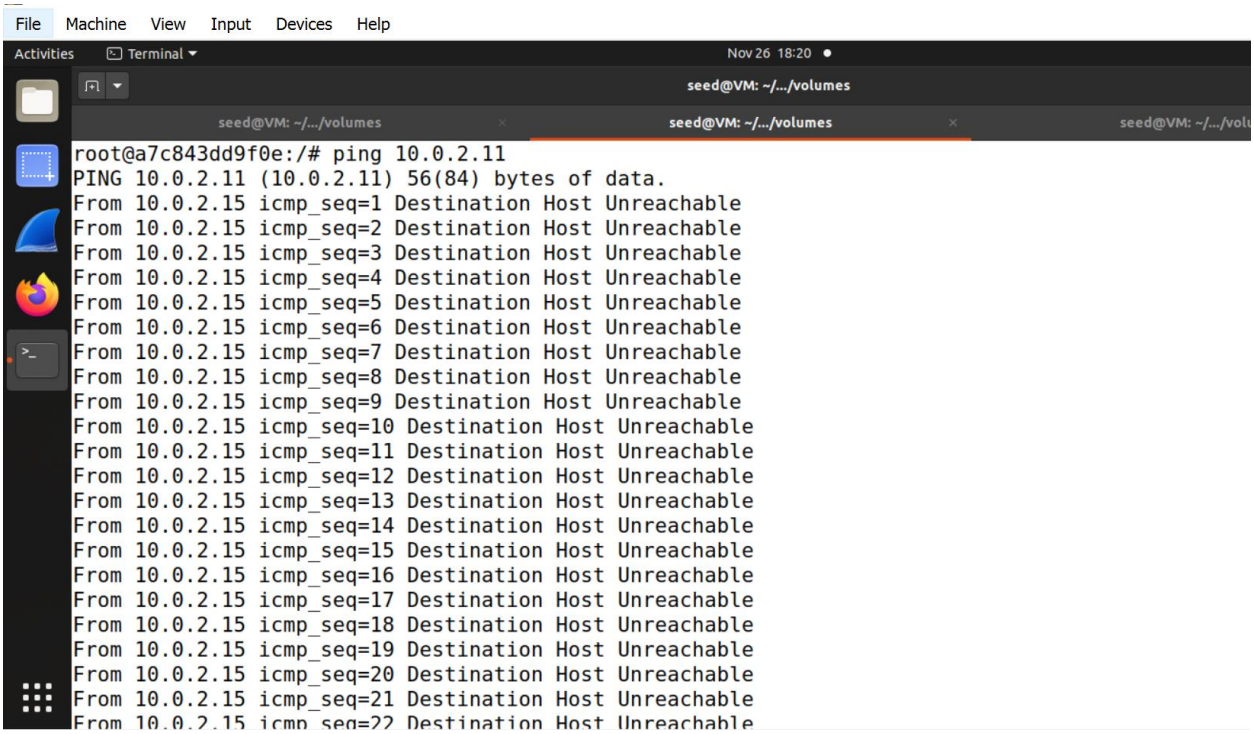
- Capture packets from or to a particular subnet

  Modify the sniffer.py to capture packets from the subnet 10.0.2.0/24 as shown below –

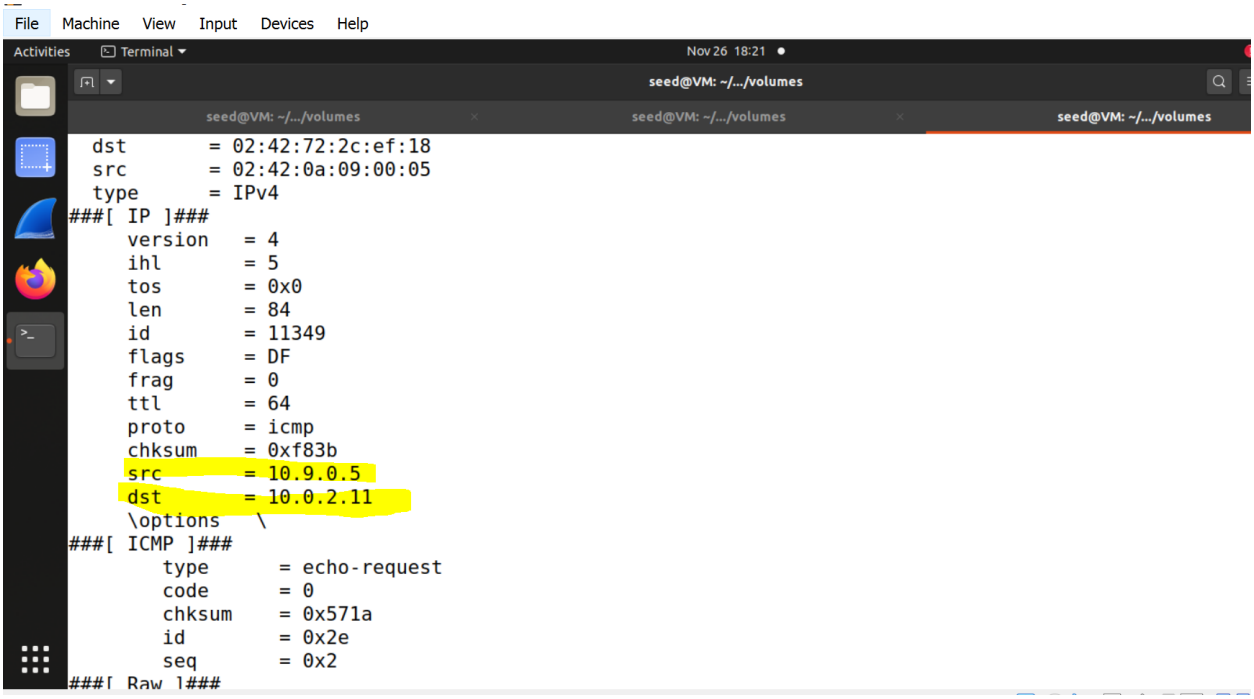
Now run the sniffer.py in attacker container with root privileges –



Next, let's ping 10.0.2.11 on the subnet from Host A –



We can observe these packets by running sniffer on attacker container as shown below –



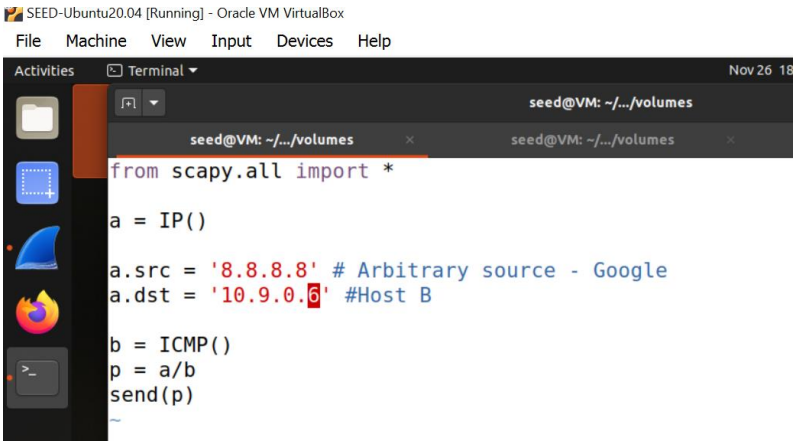
Task 1.2 – Spoofing ICMP Packets

Scapy has a feature to let us set the fields of IP packets to arbitrary values. Now we will spoof IP packets with irrelevant source IP addresses.

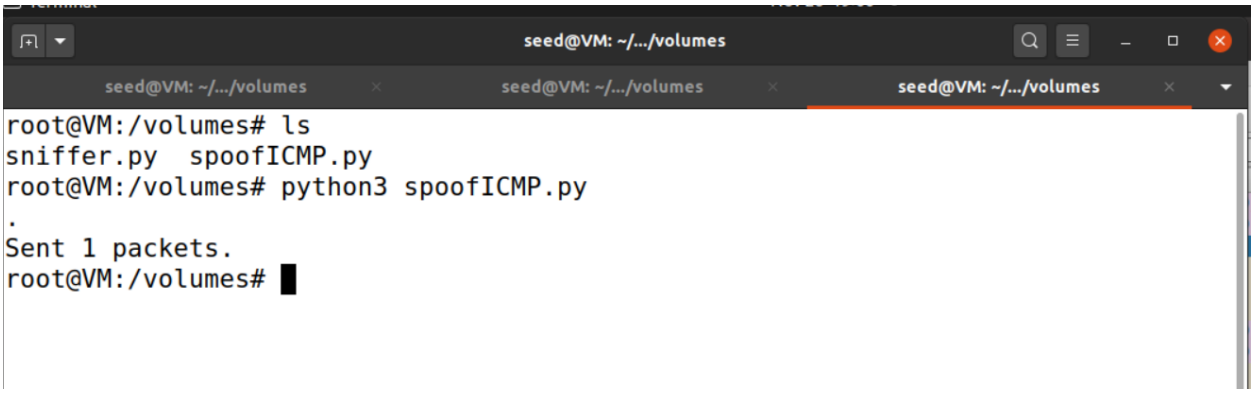
In particular, we will be spoofing ICMP echo-request packets and then send them to a different VM, though on the same network.

In this task, we use Wireshark to check if our request is accepted by the receiver. In case it is accepted, we get an echo-reply packet from the spoofed IP address.

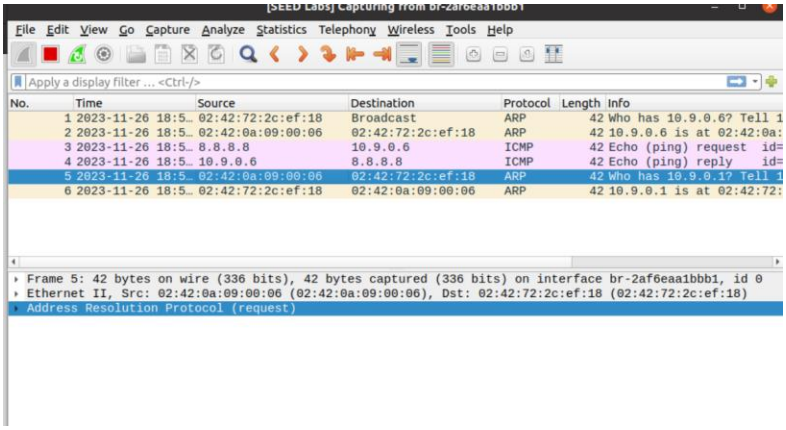
- First, create a new python file spoofICMP.py by creating an IP object a, and then specifying arbitrary src and destination IP address.
- Next, create an ICMP object, b with default request type being echo-request. Then stack a and b together to form a new object. It is basically adding b as the payload field of a and modifying the fields of a correspondingly to get a new ICMP packet
- We then send the packet.



- We run this code in attacker container –



- As can be seen below in Wireshark, the request was accepted by the receiver –



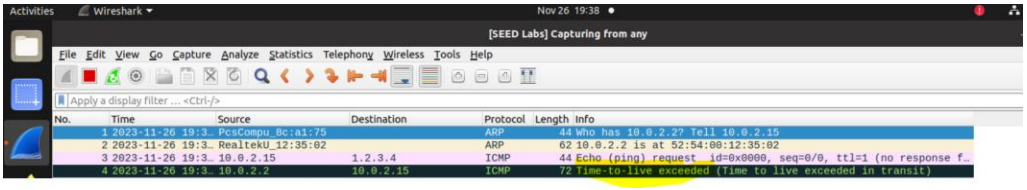
Task 1.3 – Traceroute

In this task, we want to use Scapy to estimate the distance in terms of number of nodes (routers) in between the VM and specified destination. This is done using traceroute.py (our own traceroute tool).

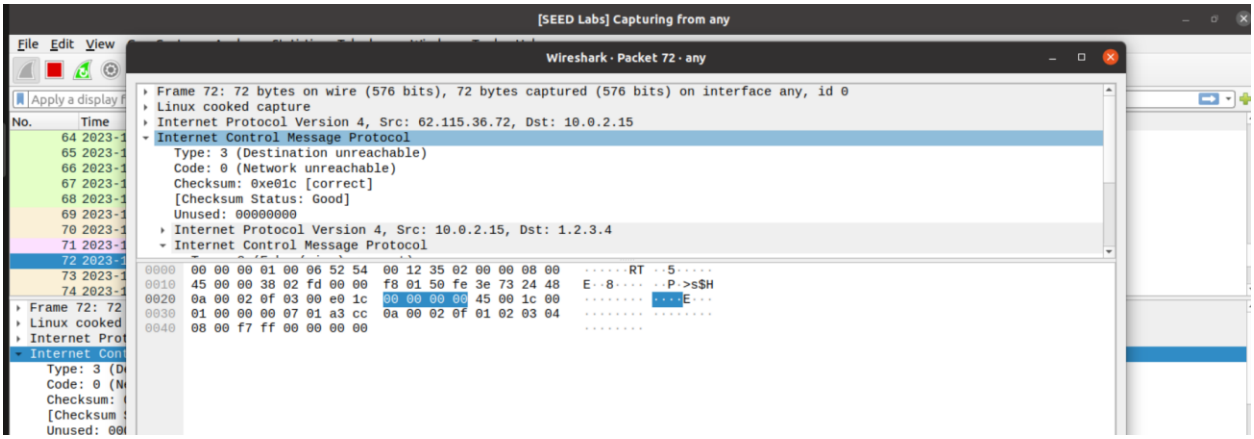
The basic principle –

- Send a packet to the destination with its TTL set to 1 at the start
- This will be sent to the first router and is subsequently dropped and send us an ICMP error message, due to the fact that it's time-to-live has exceeded.
- We can then increase the TTL field to 2 and send another packet which can go till second router
- We need to repeat this till we reach our destination.
- The code is written in file traceroute.py in volumes folder in VM as shown –

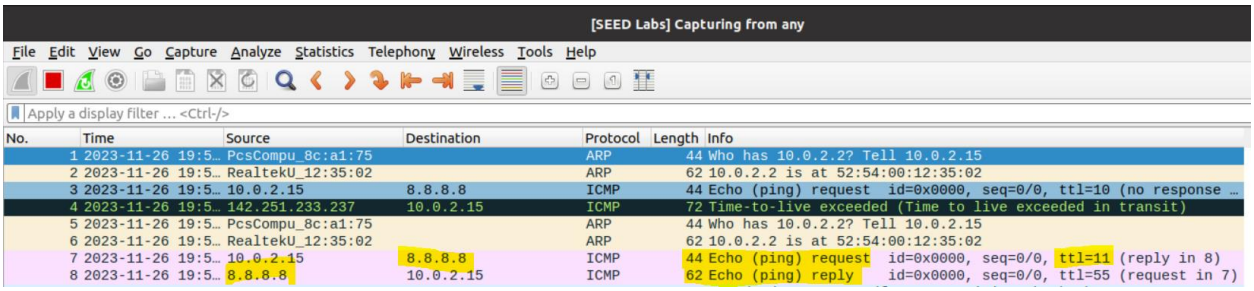
- For TTL = 1, we can see the packet is dropped due to the time-to-live exceeded as shown –



- For the destination 1.2.3.4, When TTL was 8, I got a message that the destination is not reachable. Which makes sense as it is a random made-up address as shown below



- Let's try a reachable address like 8.8.8.8 which is for google
- For this destination, I got TTL exceeded until 10 and finally succeeded at TTL = 11 as shown below –



Task 1.4 – Sniffing and then Spoofing

In this one we want to implement a sniff-and-then-spood program.

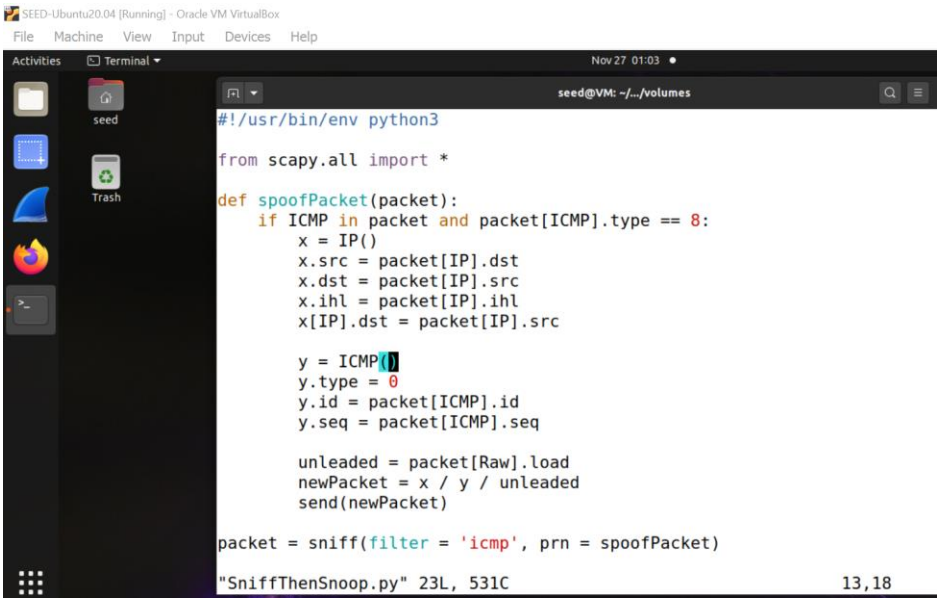
Requirement – 2 machines on one LAN (VM and User container)

Steps –

- Ping an IP X from user container. This creates an ICMP echo-request packet
- If X is alive, we get a ICMP echo-reply and print response
- The sniff-and-then-spoof program is run on the VM and which can monitors LAN through packet sniffing
- Whenever it comes across a an ICMP echo-request, regardless of the destination address, the program should immediately send an echo-reply using the spoofing technique

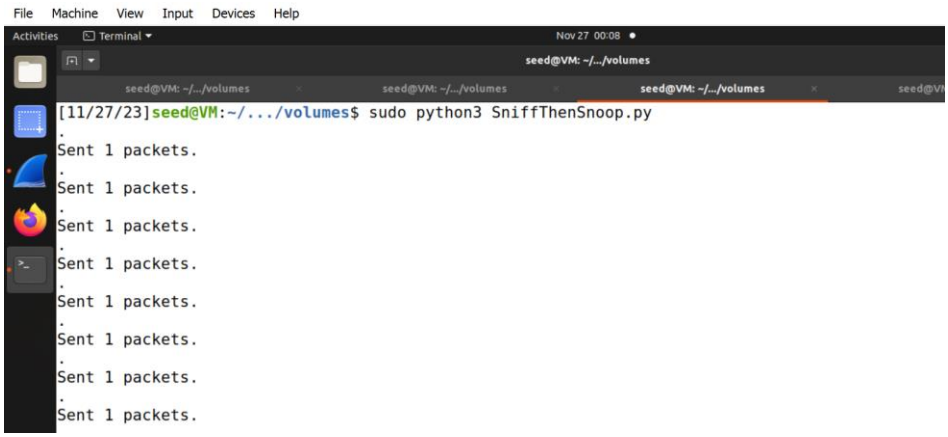
- It doesn’t matter if the machine X is alive or not, the ping program will always receive a reply, showing that X is alive
- The following address can be used are –
    - ping 1.2.3.4 # a host which doesn’t exist on the internet
    - ping 10.9.0.99 #a host which doesn’t exist on the LAN
    - ping 8.8.8.8 #a host which exists on internet
- We also need to understand the working of ARP (Address Resolution protocols) works in order. We can use this command can help us find the router for a specific destination – Eg: ip route get 1.2.3.4

The code which is run on the VM with root privileges –



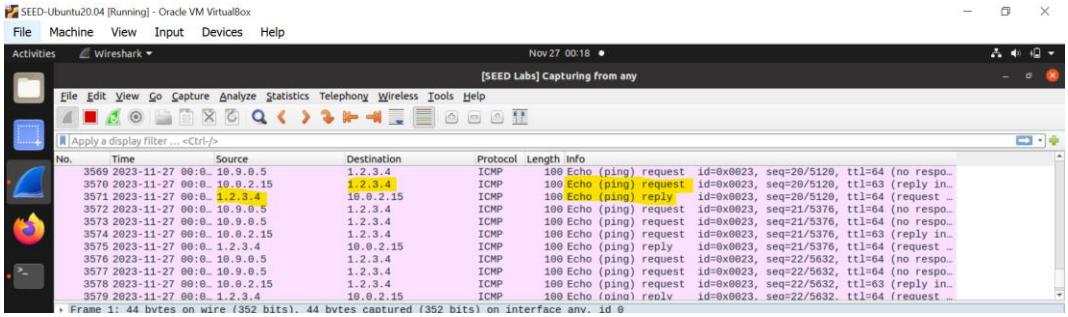
The observations clearly demonstrate that regardless of the state of machine X (alive or not) the host always receives an echo-reply for each of it’s echo-requests –

- For ping 1.2.3.4 (A host which doesn’t exist on the internet)
    - Run the code on the VM

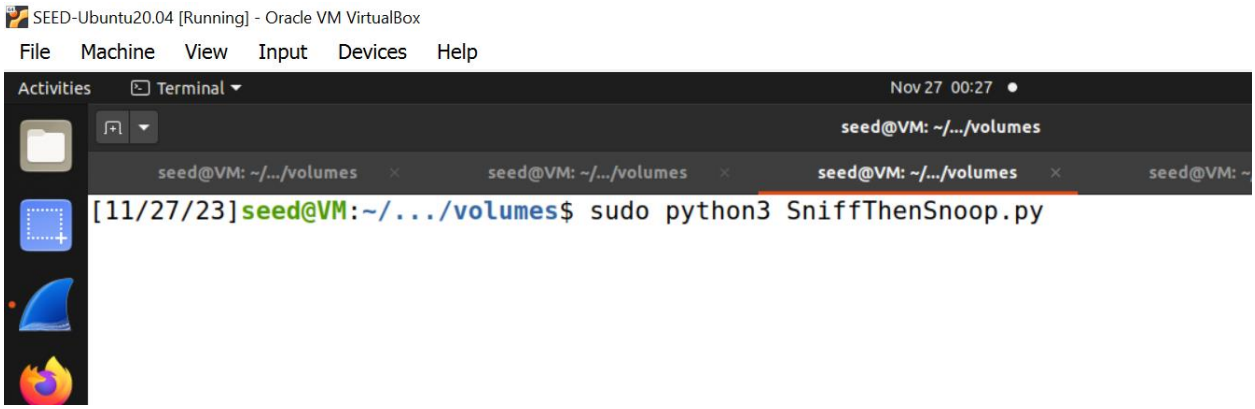


- Send a ping on a user container (Host A) to 1.2.3.4. We can see that the Packet is sent.

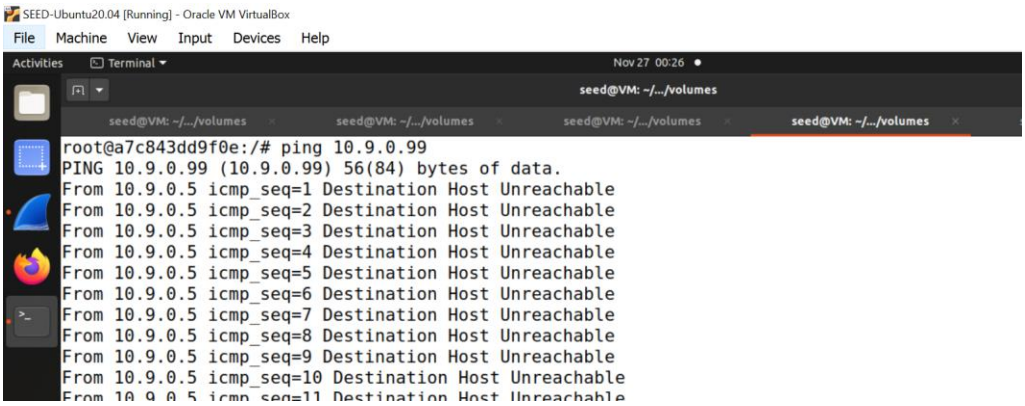
- We can see that we receive a echo-reply to the echo-request on wire shark



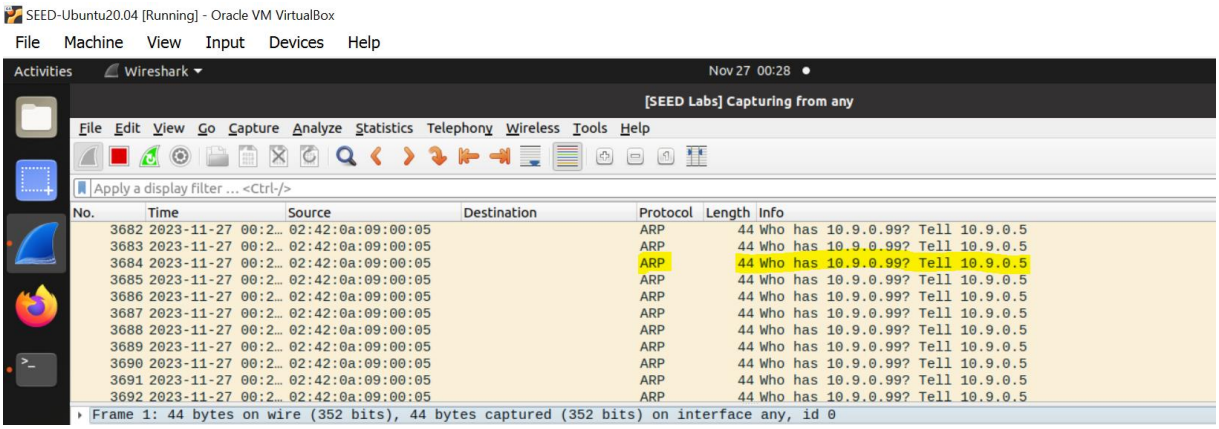
- For ping 10.9.0.99 (A host which doesn't exist on the LAN)
  - Run the code on the VM



  - Send a ping on a user container (Host A) to 10.9.0.99. Packets are not sent as host is unreachable

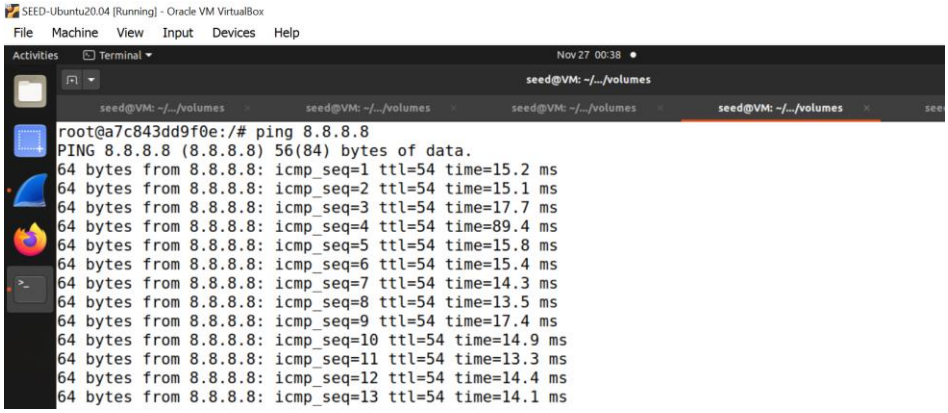


- In the Wireshark we see that there was no request being sent for 10.9.0.99. The Wireshark is flooded with ARP messages asking for the actual address of 10.9.0.99 and to tell 10.9.0.5 (Host A). We can see that the echo-request itself is not generated as ARP is not successful in finding the destination, hence we are unable to sniff nor spoof and send a response.



- For ping 8.8.8.8 (A host which exists on the internet - google)
  - Run the code on the VM

  - Send a ping on a user container (Host A) to 10.9.0.99. We can see that the packets are sent…



- We can see that we receive a echo-reply to the echo-request on Wireshark as shown below –