"I hereby declare that I have completed this individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used:
(i) Course textbook,

(ii) All material that is made available to me via Blackboard for this course,
(iii) Notes taken by me during lectures.
I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me."

71881, Sude Karagöl, 5.05.2023

Important Note: Since my computer has an apple m1 chip, I solved the problems I had in the other project using amazon. Since Amazon took money from my account last week and then my usage was restricted and I was having problems, I sent an e-mail to TA group for an extra day extension. On May 10th, it stated that it would be okay to upload and points would not be broken.

Comp 434/534 - Spring 2023

PROJECT 3 : Packet Sniffing and Spoofing Lab

Sude Karagöl 71881

1. Introduction

Tools such as Wireshark, Tcpdump and Netwox are tools that enable the use of two important security concepts such as packet sniffing and spoofing. In this project, we will understand and learn these two security elements by using wireshark. After understanding what packet sniffing and spoofing are and how they work, we will write simple sniffer and spoofing programs to complete the tasks as in the previous project. First of all, we need to understand what packet sniffing and spoofing are.

First, packet sniffing; means capturing the network traffic and data passing through the computer network and then examining this network and the data it contains. It is quite dangerous as it is a method that network security analysts can use, as well as attackers can use to gain access to unauthorized data.

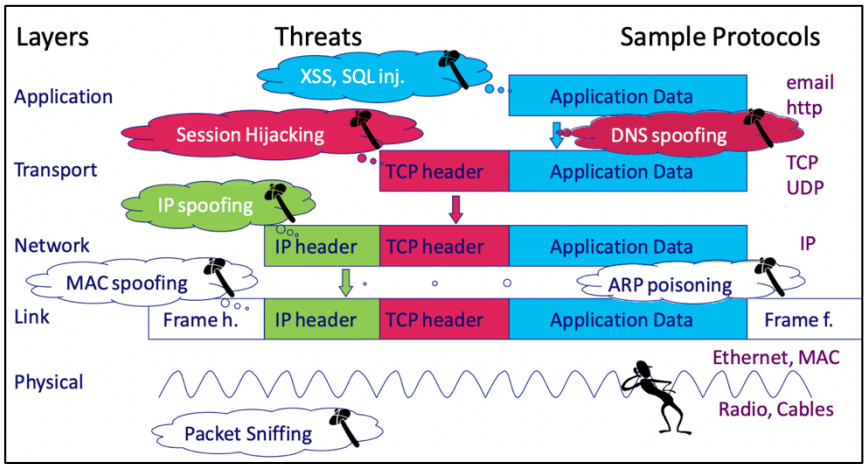
Second, spoofing is deception by making the attacker's source appear safe. With attacks on protocols that determine the way of communication between the server and the client, such as IP addresses, DNS, SMTP or ARP, which we talked about in the lesson, the receiver is deceived and is provided to trust a source that they should not trust.

The topics of this project are grouped under **4 main headings**:

i. How the sniffing and spoofing work (basic ones we talked in class)

Promiscuous mode: Sniffing can be accomplished by configuring the network interface to read all frames. Network interface card reads all passing data. This method allows it to capture all packets on the network regardless of the destination, not just the target data.

ARP Spoofing: The ARP protocol allows IP addresses to be mapped to MAC addresses that enable connection to devices. With this method, the attacker can send fake ARP messages, mapping them to different IP addresses, and have the data go to their destination instead of the recipient.



ii. Packet sniffing using the pcap library and Scapy

Pcap library enables us to capture and filter packets, so it is useful for sniffing. To do this, first we need to install these required libraries. :

pip install pypcap

pip install scapy

Next, a callback function must be created to examine every data that passes through the network and is captured. For this to happen, it must be ensured that every data passing through the network is captured. All these processes should repeat each other in the loop so that the capture and analysis of data is not interrupted. The methods we can use thanks to the pcap library: pcap.pcap() provides data capture. Scapy, on the other hand, is an open source library and is used to generate and send network packets. It can be installed on the terminal and for this project we will install scapy via the terminal. It lists the protocols we can use while doing the tasks in this project with the scapy "ls" command. This is how we will write and send the simple sniffer and spoofing programs mentioned at the beginning of the report.

iii. Packet spoofing using raw socket and Scapy

This time, we have to import socket to terminal. Socket is the communication channel created to manage the communication between the receiver and the sender. Thanks to this, we can create and send packets with fake headers. As I mentioned above, we can create these packages with scapy. We can also physically create packages using raw sockets. Unlike other sockets, it allows direct access and connection without any information, IP address. Therefore, packet spoofing is made easier using raw socket.

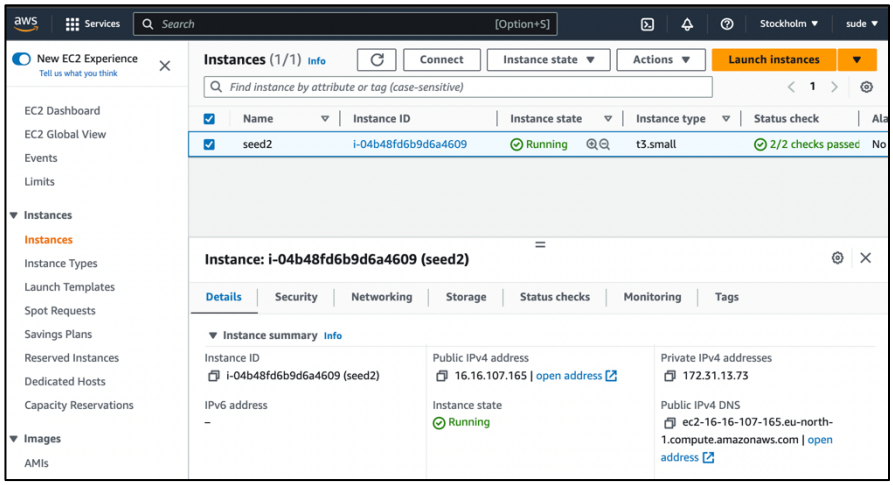
iv. Manipulating packets using Scapy

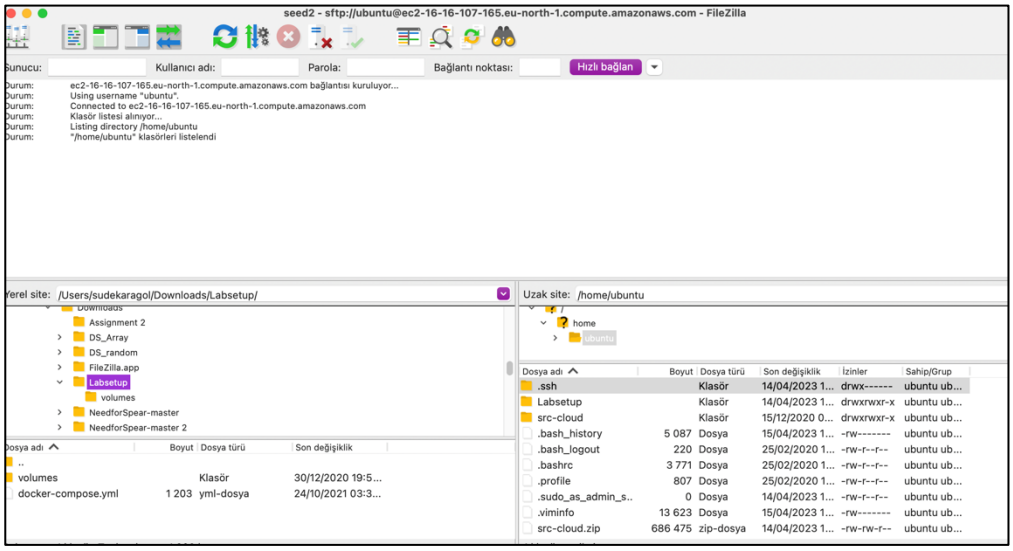
As I said in the first item, Scapy is an open source library that allows us to send packets over the network. After downloading Scapy, we can create the network packet and modify and send it as we want. Thanks to Scapy, we can change IP addresses or the data they contain.

2. Environment Setup using Container

The smallest network structure created by more than one computer is called a local area network (LAN). For this project, we need 3 different machines connected to a common LAN. One of them is attacker, one is host A, and the other is host B. We will make the mentioned sniffer attacks on the attacker machine. A and B will be two different users.

As in my other project, because of the computer I am using, VNC viewwer is not working and I am creating EC2 instance with amazon aws. Firstly, with AWS, I have established a connection between ubuntu and my computer from the terminal. Again, using FileZilla, we load the Labsetup.zip file given to us into the VM so that we can do the tasks given to us by using the files inside.





opened filezilla to transfer files

files transferred to ubuntu thanks to filezilla

```
*** System restart required ***
Last login: Sat Apr 15 11:13:16 2023 from 85.101.216.84
ubuntu@ip-172-31-13-73:~$ ls
Labsetup  Labsetup_  src-cloud  src-cloud.zip
ubuntu@ip-172-31-13-73:~$ cd Labsetup
ubuntu@ip-172-31-13-73:~/Labsetup$ ls
docker-compose.yml  volumes
ubuntu@ip-172-31-13-73:~/Labsetup$ cd docker-compose.yml
-bash: cd: docker-compose.yml: Not a directory
ubuntu@ip-172-31-13-73:~/Labsetup$
```

opened the files in ubuntu

$ docker-compose build # Build the container images

$ docker-compose up # Start the containers

$ docker-compose down # Shut down the containers

wrote sudo front of these codes, otherwise it gives error.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker-compose build
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker-compose up
WARNING: Found orphan containers (server-10.9.0.6, server-10.9.0.5) for this project. If you removed or renamed
 this service in your compose file, you can run this command with the --remove-orphans flag to clean it up.
Pulling attacker (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Already exists
14428a6d4bcd: Already exists
2c2d948710f2: Already exists
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating hostA-10.9.0.5 ... done
Creating hostB-10.9.0.6 ... done
Creating seed-attacker  ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
^CGracefully stopping... (press Ctrl+C again to force)
Stopping hostB-10.9.0.6 ... done
Stopping seed-attacker  ... done
Stopping hostA-10.9.0.5 ... done
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker-compose down
WARNING: Found orphan containers (server-10.9.0.6, server-10.9.0.5) for this project. If you removed or renamed
 this service in your compose file, you can run this command with the --remove-orphans flag to clean it up.
Removing hostB-10.9.0.6 ... done
Removing seed-attacker  ... done
Removing hostA-10.9.0.5 ... done
Removing network net-10.9.0.0
ubuntu@ip-172-31-13-73:~/Labsetup$
```

with “sudo docker ps” command to find out the ID of the container, and then use "sudo docker exec" to start a shell on that container. ID of the containers are: “53” and “6e”

```
enied
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED   STATUS   PORTS   NAMES
```

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker ps -a
CONTAINER ID   IMAGE                    COMMAND                  CREATED       STATUS                      PORTS     NAMES
53388dbf2c4d   seed-image-fmt-server-2   "/bin/sh -c ./server"   3 weeks ago   Exited (137) 11 days ago              server-10.9.0.6
6e8fbcc8d333   seed-image-fmt-server-1   "/bin/sh -c ./server"   3 weeks ago   Exited (137) 11 days ago              server-10.9.0.5
ubuntu@ip-172-31-13-73:~/Labsetup$
```

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker exec -it 53 /bin/bash
Error response from daemon: Container 53388dbf2c4df82c620c7832339e4df29b1a4365b77ef2c2480b34685d0ffa30 is
 not running
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker exec -it 6e /bin/bash
Error response from daemon: Container 6e8fbcc8d33303604a2691f6987987385e787a42699b495226af359e125eef41 is
 not running
```

As attacker machine, I will use attacker container in docker compose file and I will put the attack code inside it. In the docker compose file there is a “volumes” part which we can change that file part from the outside and cause changes in this file. Also, for this attacker container, I will use host mode which allows me to see all packet traffic. In the following Picture, “volumes” and “network_mode” parts are very important for this project.

(this screen opens with “vim docker-compose.yml”)

```
version: "3"

services:
    attacker:
        image: handsonsecurity/seed-ubuntu:large
        container_name: seed-attacker
        cap_add:
                - ALL
        privileged: true
        volumes:
                - ./volumes:/volumes
        network_mode: host

    hostA:
        image: handsonsecurity/seed-ubuntu:large
        container_name: hostA-10.9.0.5
        tty: true
        cap_add:
                - ALL
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.5
        command: bash -c "
                    /etc/init.d/openbsd-inetd start  &&
                    tail -f /dev/null
                "

    hostB:
        image: handsonsecurity/seed-ubuntu:large
        container_name: hostB-10.9.0.6
        tty: true
        cap_add:
                - ALL
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.6
        command: bash -c "
                    /etc/init.d/openbsd-inetd start  &&
                    tail -f /dev/null
                "

networks:
    net-10.9.0.0:
```

```
networks:
    net-10.9.0.0:
    net-10.9.0.0:
        name: net-10.9.0.0
        ipam:
            config:
                - subnet: 10.9.0.0/24
```

We need the IP for a new network which is created to connect the VM and the containers. IP is given in docker compose. 10.9.0.0/24

```
ubuntu@ip-172-31-13-73:~/Labsetup$ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:f7:1b:ec:35  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ens5: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 172.31.13.73  netmask 255.255.240.0  broadcast 172.31.15.255
        inet6 fe80::c91:8cff:fec0:269a  prefixlen 64  scopeid 0x20<link>
        ether 0e:91:8c:c0:26:9a  txqueuelen 1000  (Ethernet)
        RX packets 1724734  bytes 1746537540 (1.7 GB)
        RX errors 0  dropped 19  overruns 0  frame 0
        TX packets 753080  bytes 101223105 (101.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3594  bytes 366090 (366.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3594  bytes 366090 (366.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The IP address of the VM we are using is 10.9.0.1. The network interface represented by this IP address will be used in the programs. I will use "ifconfig" to see this network interface.

another way to learn network interface:

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
e91666e0e828   bridge    bridge    local
aedf57eabeca   host      host      local
6fbb6113a4e3   none      null      local
```

```
ubuntu@ip-172-31-13-73:~/Labsetup$ ifconfig
br-de383b854ce6: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:1bff:fea7:9d0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:1b:a7:09:d0  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 36  bytes 5039 (5.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

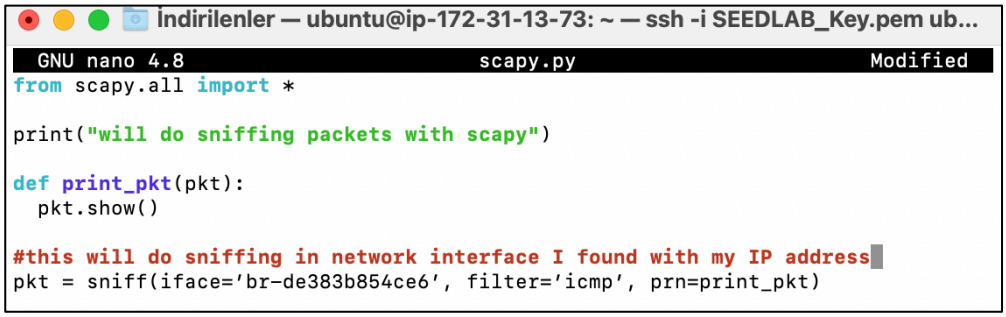
network interface is: br-de383b854ce6

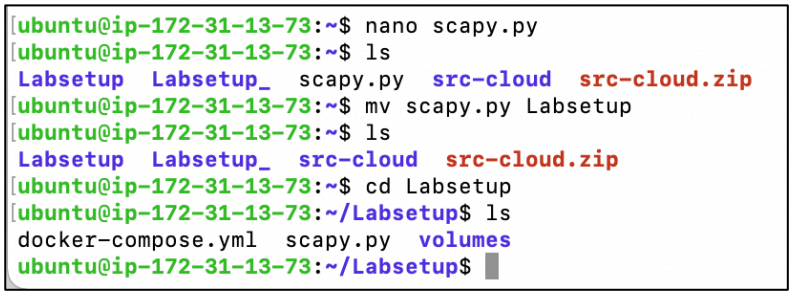
3. Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

As I mentioned in introduction, there are too many ways to do sniffing and spoofing but I will use Scapy for this tasks. Again, as I mentioned in the introduction part, we will implement it step by step. First of all, we need to import the packages required for scapy and write a phyton program to use scapy. For the first task we will learn how to import, use and write Scapy to do packet sniffing.

For this task, we need to use the network interface that corresponds to our IP address that we found in the environment setup section. After importing the necessary libraries and providing the phyton code, we write all the interfaces we want to sniff into "iface = [....]". Here we will write the network interface we found inside these brackets.

With writing “nano scapy.py” created the python script tto use scapy and import it.





I moved the python script I created into Labsetup because I created it in the wrong place

With this code, for each captured packet, the callback function print pkt() will be invoked and informations in packets will be printed.

Now, I will run the program with root privilege and show that packets can be captured this way.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker ps --format "{{.ID}} {{.Names}}"
960439e76883 hostB-10.9.0.6
f5a1d96b3b56 seed-attacker
14eac992a660 hostA-10.9.0.5
```

```
ubuntu@ip-172-31-13-73:~/Labsetup$ scapy
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().

                     aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.5.0
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | We are in France, we say Skappe
       scccccp///pSP///p          p//Y   | OK? Merci.
      sY/////////y  caa           S//P   |             -- Sebastien Chabal
       cayCyayP//Ya              pY/Ya   |
        sY/PsY////YCc          aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs

                                       using IPython 7.13.0

>>> a=IP()
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

we must write “docksh seed-attacker” , “docksh hostA-10.9.0.5” and “docksh hostB-10.9.0.6” to open all machines in different tabs.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ ./task1.1.py
Traceback (most recent call last):
  File "./task1.1.py", line 18, in <module>
    pkt = sniff(iface="br-de383b854ce6", filter="icmp", prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1311, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1171, in _run
    sniff_sockets[_RL2(iface)(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 484, in __init__
    self.ins = socket.socket(
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Sniffing task requires high privilege. Running task1.1 without privilege gives “operation not permitted” answer. But executing with root privileges allows Scapy to have the necessary privileges for packet manipulation and spoofing.

For task1.1a wrote “chmod a+x task1.1.py” and task1.1.py” to run the program with the root privilige.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ chmod a+x task1.1.py
ubuntu@ip-172-31-13-73:~/Labsetup$ task1.1.py
task1.1.py: command not found
ubuntu@ip-172-31-13-73:~/Labsetup$ ./task1.1.py
Traceback (most recent call last):
  File "./task1.1.py", line 18, in <module>
    pkt = sniff(iface="br-de383b854ce6", filter="icmp", prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1311, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1171, in _run
    sniff_sockets[_RL2(iface)(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 484, in __init__
    self.ins = socket.socket(
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Then, to switch account to “seed” account: write “su seed”. We must use the “sudo” command to gain root privileges.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ su seed
Password:
su: Authentication failure
ubuntu@ip-172-31-13-73:~/Labsetup$ su seed
Password:
su: Authentication failure
```

It does not accept the “dees” password for “su seed” command.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo su seed
seed@ip-172-31-13-73:/home/ubuntu/Labsetup$ ls
__pycache__  docker-compose.yml  task1.1.py  volumes
seed@ip-172-31-13-73:/home/ubuntu/Labsetup$
```

used “sudo su seed” because ubuntu does not utilize the su function for privileges.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker exec -it hostA-10.9.0.5 /bin/bash
root@14eac992a660:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.145 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.075 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.057 ms
64 bytes from 10.9.0.6: icmp_seq=11 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=64 time=0.061 ms
64 bytes from 10.9.0.6: icmp_seq=13 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=14 ttl=64 time=0.065 ms
64 bytes from 10.9.0.6: icmp_seq=15 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=16 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=17 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=18 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=19 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=20 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=21 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=22 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=23 ttl=64 time=0.059 ms
64 bytes from 10.9.0.6: icmp_seq=24 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=25 ttl=64 time=0.065 ms
64 bytes from 10.9.0.6: icmp_seq=26 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=27 ttl=64 time=0.072 ms
64 bytes from 10.9.0.6: icmp_seq=28 ttl=64 time=0.060 ms
^C
--- 10.9.0.6 ping statistics ---
28 packets transmitted, 28 received, 0% packet loss, time 27656ms
rtt min/avg/max/mdev = 0.057/0.064/0.145/0.015 ms
root@14eac992a660:/#
```

we can see that 28 packets transmitted and received

```
root@ip-172-31-13-73:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.097 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.053 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.049 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.050 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.048 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.051 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.050 ms
^C
--- 10.9.0.6 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6151ms
rtt min/avg/max/mdev = 0.048/0.056/0.097/0.016 ms
```

İndirilenler — ubuntu@ip-172-31-13-73: ~/Labsetup — ssh -i SEEDLAB_K...

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

def print_pkt(pkt):

    print_pkt.num_packets+=1
    print("\n========= packet: {} =======\n".format (print_pkt.num_packets))
    pkt.show()

print_pkt.num_packets=0
pkt = sniff(iface="br-de383b854ce6", filter="icmp", prn=print_pkt)
~
```

this code captures only ICMP packet because it has filter=”icmp”.

To capture any TCP packet that comes from a particular IP and with a destination port number 23: filter must be “filter=”tcp && src host 10.9.0.5 && dst port 23” After this update, write ping 10.9.0.6 again and execute it.

```
root@14eac992a660:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.119 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.081 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.064 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.080 ms
^C
--- 10.9.0.6 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6120ms
rtt min/avg/max/mdev = 0.060/0.075/0.119/0.019 ms
```

To capture packets comes from or to go to a particular subnet: subnet can be 128.230.0.0/16 . For this time in filter we must write: filter=”net 128.230.0.0/16” and write “ping 128.230.0.0/16” or “ping 128.230.0.11” (this number is dst)

```
seed@14eac992a660:/$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
14eac992a660 login: sudo seed
Password:
```

```
print_pkt.num_packets=0
pkt = sniff(iface="br-de383b854ce6", filter="net 128.230.0.0/16", prn=print_pkt)
```

```
seed@14eac992a660:/$ ping 128.230.0.11
PING 128.230.0.11 (128.230.0.11) 56(84) bytes of data.
From 128.230.61.170 icmp_seq=1 Destination Host Unreachable
From 128.230.61.170 icmp_seq=2 Destination Host Unreachable
From 128.230.61.170 icmp_seq=3 Destination Host Unreachable
From 128.230.61.170 icmp_seq=4 Destination Host Unreachable
From 128.230.61.170 icmp_seq=5 Destination Host Unreachable
From 128.230.61.170 icmp_seq=6 Destination Host Unreachable
^C
--- 128.230.0.11 ping statistics ---
8 packets transmitted, 0 received, +6 errors, 100% packet loss, time 7093ms
pipe 4
```

Task 1.2 is about spoofing ICMP packets. I wrote nano task1.2.py to create python script.

```
  GNU nano 4.8
#!/usr/bin/env python3

from scapy.all import *

a = IP()
a.dst = "10.0.2.3"
b = ICMP()
p = a/b
ls(a)
send(p)
```

```
root@ip-172-31-13-73:/volumes# chmod a+x task1.2.py
root@ip-172-31-13-73:/volumes# ./task1.2.py
version    : BitField  (4 bits)                  = 4               (4)
ihl        : BitField  (4 bits)                  = None            (None)
tos        : XByteField                          = 0               (0)
len        : ShortField                          = None            (None)
id         : ShortField                          = 1               (1)
flags      : FlagsField  (3 bits)                = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField  (13 bits)                 = 0               (0)
ttl        : ByteField                           = 64              (64)
proto      : ByteEnumField                       = 0               (0)
chksum     : XShortField                         = None            (None)
src        : SourceIPField                       = '172.31.13.73'  (None)
dst        : DestIPField                         = '10.0.2.3'      (None)
options    : PacketListField                     = []              ([])
.
Sent 1 packets.
```

This is the return of task1.2.py

```
#!/usr/bin/env python3

from scapy.all import *

a = IP()
a.dst = "1.2.3.4"
b = ICMP()
p = a/b
ls(a)
send(p,iface="br-de383b854ce6")
```

When I see that program is executing, I maket he necessary changes to spoof and ICMP echo request packet with source IP.

```
Sent 1 packets.
root@ip-172-31-13-73:/volumes# ./task1.2.py
version    : BitField  (4 bits)                  = 4               (4)
ihl        : BitField  (4 bits)                  = None            (None)
tos        : XByteField                          = 0               (0)
len        : ShortField                          = None            (None)
id         : ShortField                          = 1               (1)
flags      : FlagsField  (3 bits)                = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField  (13 bits)                 = 0               (0)
ttl        : ByteField                           = 64              (64)
proto      : ByteEnumField                       = 0               (0)
chksum     : XShortField                         = None            (None)
src        : SourceIPField                       = '172.31.13.73'  (None)
dst        : DestIPField                         = '1.2.3.4'       (None)
options    : PacketListField                     = []              ([])
.
Sent 1 packets.
```

For task 1.3, again created a python script with “nano task1.3.py” In this task, we will send any packet with any type to a.dst. a.tll will be 1 and it will give error message. I will learn IP with this error message.

```
 GNU nano 4.8                                task1.3.py
#!/usr/bin/env python3

from scapy.all import *
import sys

a = IP()
a.dst = "1.2.3.4"
a.ttl = int(sys.argv[1])
b = ICMP()
a = sr1(a/b)
print ("IP: " , a.src)

#we will print different values with changing time-to-live field.
```

We will make tll value 1 and 2 accordingly. 1 will give error and IP address of the first router. Then, 2 will get second router.

```
root@ip-172-31-13-73:/volumes# chmod a+x task1.3.py
root@ip-172-31-13-73:/volumes# ./task1.3.py 1
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  13.53.0.197
```

We can see that IP is 13.53.0.197 with a.tll = 1

```
root@ip-172-31-13-73:/volumes# ./task1.3.py 2
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  240.0.20.15
```

Here, we can see that IP is 240.0.20.15 with a.tll = 2

```
root@ip-172-31-13-73:/volumes# ./task1.3.py 3
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  240.0.20.30
```

When I set the destination address, a.dst ="8.8.8.8", I will aim to reach that value by changing a.tt.

```
IP:  142.250.171.248
root@ip-172-31-13-73:/volumes# ./task1.3.py 12
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  142.250.213.185
root@ip-172-31-13-73:/volumes# ./task1.3.py 13
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  142.251.48.45
root@ip-172-31-13-73:/volumes# ./task1.3.py 14
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
IP:  8.8.8.8
root@ip-172-31-13-73:/volumes#
```

In a.tll = 14, program reached the desired destination address 8.8.8.8

```
root@14eac992a660:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2041ms
```

Aim of task 1.4 is to combine sniffing and spoofing. With container, I will do “ping 1.2.3.4” which is going to generate an ICMP echo request packet. After that, we will do “ping 10.9.0.99”. An then, “ping 8.8.8.8”

```
root@14eac992a660:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
From 10.9.0.5 icmp_seq=4 Destination Host Unreachable
From 10.9.0.5 icmp_seq=5 Destination Host Unreachable
From 10.9.0.5 icmp_seq=6 Destination Host Unreachable
From 10.9.0.5 icmp_seq=7 Destination Host Unreachable
From 10.9.0.5 icmp_seq=8 Destination Host Unreachable
From 10.9.0.5 icmp_seq=9 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
10 packets transmitted, 0 received, +9 errors, 100% packet loss, time 9222ms
pipe 4
```

```
root@14eac992a660:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=109 time=1.83 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=109 time=1.86 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=109 time=1.83 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=109 time=1.88 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=109 time=1.85 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 1.830/1.851/1.883/0.019 ms
```

How the ARP protocol and routing works are important for this task. ARP protocol is when a machine will send data to another machine with the same network (LAN), it will want to checks its ARP cache to be sure that it knows the MAC address corresponding to the IP address of the destination machine.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ nano task1.4.py
ubuntu@ip-172-31-13-73:~/Labsetup$ chmod a+x task1.4.py
ubuntu@ip-172-31-13-73:~/Labsetup$ ./task1.4.py
Traceback (most recent call last):
  File "./task1.4.py", line 29, in <module>
    pkt = sniff(iface="br-de383b854ce6",filter='arp or icmp',prn=spoof_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1311, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1171, in _run
    sniff_sockets[_RL2(iface)(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 484, in __init__
    self.ins = socket.socket(
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

```
 GNU nano 4.8                                  task1.4.py
#!/usr/bin/env python3

from scapy.all import*


def print_pkt(pkt):
        a = IP()
        a.src = pkt[IP].dst
        a.dst = pkt[IP].src
        b = ICMP()
        b.type ="echo-reply"
        b.code =0
        b.id = pkt[ICMP].id
        b.seq = pkt[ICMP].seq
        p = a/b
        send(p)
```

```
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
```

4. Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

For this task, I will use pcap library I talked about in introduction. “sudo docker ps” lists the containers and their IP addresses used and will be used. “docker cp” command can be used to copy a file from host machine to another machine, container in this tasks.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ sudo docker ps --format "{{.ID}} {{.Names}}"
960439e76883 hostB-10.9.0.6
f5a1d96b3b56 seed-attacker
14eac992a660 hostA-10.9.0.5
```

In task 2.1, I will write a sniffing program with using ready-made protocols from pcap library. For this task, first create a program that writes the source and destination IPs of the packets captured by the machine.

• Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

As a write in the introduction:

Pcap library enables us to capture and filter packets, so it is useful for sniffing. To do this, first we need to install these required libraries. :

pip install pypcap

pip install scapy

Next, a callback function must be created to examine every data that passes through the network and is captured. For this to happen, it must be ensured that every data passing through the network is captured. All these processes should repeat each other in the loop so that the capture and analysis of data is not interrupted. The methods we can use thanks to the pcap library: pcap.pcap() provides data capture. Scapy, on the other hand, is an open source library and is used to generate and send network packets. It can be installed on the terminal and for this project we will install scapy via the terminal. It lists the protocols we can use while doing the tasks in this project with the scapy "ls" command. This is how we will write and send the simple sniffer and spoofing programs mentioned at the beginning of the report.

• Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

It is used to run a sniffer program, as we did in the previous task, as it requires low-level access to the operating system to capture network packets and access raw network interfaces. Root privilege allows us to read and write any files on the machine. We use the word "sudo" to achieve this. Without root privileges, the program cannot open the network interface and capture packets on the network, which is the main purpose of sniffer programs.

• Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the promiscuity's value).

Thanks to the promiscuous mode, the machine can capture all packets passing through the network. Turning this mode on allows all packets to be captured, while turning it off ensures that only packets at the target are captured. The "inconfig" command used in the above task also determines whether this mode is on or off, by looking at how many of the existing packages have been captured or whether "promisc" is written in the "flags=" value.

```
ubuntu@ip-172-31-13-73:~/Labsetup$ ifconfig
br-de383b854ce6: 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::42:1bff:fea7:9d0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:1b:a7:09:d0  txqueuelen 0  (Ethernet)
        RX packets 941  bytes 76300 (76.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 118  bytes 10949 (10.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:f7:1b:ec:35  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Second part for task 2.1 is writing filters. Capture the ICMP packets between two specific hosts. For this we will use : `char filter_exp[] = "icmp && host 10.0.2.129 && host 120.78.209.0"`

And to capture the TCP packets with a destination port number in the range from 10 to 100, we will use: `char filter_exp[] = "tcp && dst portrange 10-100"`

Task 2.2 is about writing a spoofing program and spoofing an ICMP echo request. For writing a spoofing program, there must be a variable for the length of the IP which will be incremented by the size of the IP header. We must create and set a raw socket network.

Question 4. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Yes, the length of the IP header can be set to arbitrary values, no matter how large the actual size. The IP length is changed to its original size, regardless of the size set by the user, thanks to the raw sockets we saw in the lesson.

Question 5. Using the raw socket programming, do you have to calculate the checksum for the IP header?

A raw socket interface is an interface that provides direct access to lower layer protocols such as IP and ICMP. Checksum calculations over IP are automatically output by the network. I think we did not calculate the IP header in the tasks we have done so far.

Question 6. Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

If a program using a raw socket is executed without root privilege, the raw socket cannot be created and the program will fail to run. Without root privilege and required permissions, the program will warn and no connection to the machine can be established.

For task 2.3, we must generate an ICMP echo request packet give reply using the packet spoofing technique. For this, we must generate a raw network socket, identify a source, destination and send the packet. An ARP man-in-the-middle attack capable of capturing network packets or sniffing in promocious mode can be used.

Resources:

Blackboard

Amazon aws

Tim Carstens tutorial for pcap : http://www.tcpdump.org/pcap.htm

https://www.handsonsecurity.net/files/slides/N04_Sniffing_Spoofing.pptx

Used:

Filezilla

AWS client

Terminal

Labsetup file

Project pdf

Lecture slides

scapy

pcap