# COS430 – Project Deliverable 4 – Packet Sniffing and Spoofing – Due Date: April 30th, 2022



**Note:** The materials presented in this project are inspired by and partially taken from SEED labs, with permission from the author, Dr. Du.

## Description:

Packet sniffing and spoofing are two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking. There are many packets sniffing and spoofing tools, such as `Wireshark`, `Tcpdump`, `Netwox`, `Scapy`, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this project is two-fold: learning to use the tools and understanding the technologies underlying these tools. For the second object, you will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs.

This project covers the following topics:

- How the sniffing and spoofing work
- Packet sniffing using the `pcap` library and Scapy
- Packet spoofing using raw socket and Scapy
- Manipulating packets using Scapy

## Project Environment:

This project has been tested on the pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

## Task 0 – Setting Up the Project Environment – Docker

In this project, we will use two machines that are connected to the same LAN. We can either use two VMs or use two containers. Figure 1 depicts the project environment setup using containers. We will do all the attacks on the attacker container while using the other container as the user machine.

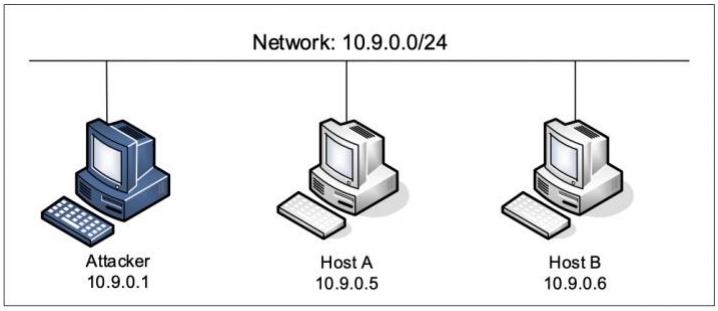


*Figure 1 - Project Environment Setup*

**Container Setup and Commands:**
In the `Projectsetup` folder, use the `docker-compose.yml` file to set up the project environment.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file.

```
$ docker-compose build     # Build the container image
$ docker-compose up        # Start the container
$ docker-compose down      # Shut down the container

// Aliases for the Compose commands above
$ dcbuild     # Alias for: docker-compose build
$ dcup        # Alias for: docker-compose up
$ dcdown      # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. You first need to use the `"docker ps"` command to find out the ID of the container, and then use `"docker exec"` to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps      // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh      // Alias for: docker exec -it /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

**About the Attacker Container:**
In this project, we can either use the VM or the attacker container as the attacker machine. If you look at the `Docker Compose` file, you will see that the attacker container is configured differently from the other containers. Here are the differences:

- *Shared folder*. When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. You will write your code in the `./volumes` folder (on the VM), so they can be used inside the containers.

  ```
  volumes:
            - ./volumes:/volumes
  ```

- *Host mode*. In this project, the attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. To solve

this problem, you use the host mode for the attacker container. This allows the attacker container to see all the traffics. The following entry used on the attacker container:

```
network_mode: host
```

When a container is in the `host` mode, it sees all the host's network interfaces, and it even has the same IP addresses as the host. Basically, it is put in the same network namespace as the host VM. However, the container is still a separate machine, because its other namespaces are still different from the host.

**Getting the network interface name.** When you use the provided `compose` file to create containers for this project, a new network is created to connect the VM and the containers. The IP prefix for this network is `10.9.0.0/24`, which is specified in the `docker-compose.yml` file. The IP address assigned to our VM is `10.9.0.1`. You need to find the name of the corresponding network interface on the VM, because you need to use it in the programs. The interface name is the concatenation of `br-` and the ID of the network created by Docker. When you use `ifconfig` to list network interfaces, you will see quite a few. Look for the IP address `10.9.0.1`.

```
$ ifconfig
br-c93733e9f913: flags=4163<UP, BROADCAST,RUNNING, MULTICASE>  mtu  1500
        inet 10.9.0.1 netmask  255.255.255.0  broadcast 10.9.0.255
        ...
```

Another way to get the interface name is to use the "`docker network`" command to find out the network ID yourselves, the name of the network is `seed-net`:

```
$ docker network ls
NETWORK ID          NAME         DRIVER              SCOPE
a82477ae4e6b        bridge       bridge              local
e99b370eb525        host         host                local
df62c6635eae        none         null                local
c93733e9f913        seed-net     bridge              local
```

**Using Scapy to Sniff and Spoof Packets:**

Many tools can be used to do sniffing and spoofing, but most of them only provide fixed functionalities. *Scapy* is different: it can be used not only as a tool but also as a building block to construct other sniffing and spoofing tools, i.e., we can integrate the *Scapy* functionalities into our own program. In this set of tasks, you will use *Scapy* for each task.

To use *Scapy*, you can write a `Python` program, and then execute this program using Python. See the following example. You should run `Python` using the `root` privilege because the privilege is required for spoofing packets. At the beginning of the program (`Line 1`), you should import all *Scapy*'s modules.

```
# view mycode.py
#!/usr/bin/env python3

from scapy.all import *          (1)

a = IP()
a.show()

# python3 mycode.py
###[ IP ]###
       version = 4
```

```
       ihl = None
       ...

// Make mycode.py executable (another way to run python programs)
# chmod a+x mycode.py
# mycode.py
```

You can also get into the interactive mode of Python and then run our program one line at a time at the Python prompt. This is more convenient if you need to change the code frequently in an experiment.

```
# python3
>>> from scapy.all import *
>>> a = IP()
>>> a.show()
###[ IP ]###
       version = 4
       ihl = None
       ...
```

**Task 1 – Sniffing Packets (25 Points):**

Wireshark is the most popular sniffing tool, and it is easy to use. You will use it throughout the project. However, it is difficult to use Wireshark as a building block to construct other tools. You will use *Scapy* for that purpose.

The objective of this task is to learn how to use *Scapy* to do packet sniffing in `Python` programs. A sample code is provided in the following:

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-c93733e9f913', filter='icmp', prn=print_pkt)
```

The code above will sniff the packets on the `br-c93733e9f913` interface. Read the instruction in the setup section regarding how to get the interface name. If you want to sniff on multiple interfaces, you can put all the interfaces in a list, and assign it to `iface`. See the following example:

```
iface=['br-c93733e9f913', 'enp0s3']
```

**Task 1. A (10 Points).** In the above program, for each captured packet, the callback function `print_pkt()` will be invoked; this function will print out some of the information about the packet. Run the program with the `root` privilege and demonstrate that you can indeed capture packets.

After that, run the program again, but without using the `root` privilege, and observe and document the result.

```
// Make the program executable
# chmod a+x sniffer.py

// Run the program with the root privilege
# sniffer.py

// Switch to the "seed" account, and
```

```
// run the program without the root privilege
# su seed
$ sniffer.py
```

**Submission:** In your report, you need to show the steps and then describe and explain your observations.

**Task 1. B (15 Points).** Usually, when you sniff packets, you are only interested in certain types of packets. You can do that by setting filters in sniffing. *Scapy*'s filter uses the BPF (Berkeley Packet Filter) syntax; you can find the BPF manual on the Internet.

Set the following filters and demonstrate your sniffer program again (each filter should be set separately):

- Capture only the ICMP packet.
- Capture any TCP packet that comes from a particular IP and with a destination port number 23.
- Capture packets come from or go to a particular subnet. You can pick any subnet, such as `128.230.0.0/16`; you should not pick the subnet that your VM is attached to.

**Submission:** In your report, you need to show the steps and then describe and explain your observations.

**<u>Task 2 – Spoofing ICMP Packets (20 Points):</u>**

As a packet spoofing tool, *Scapy* allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. You will spoof ICMP echo request packets and send them to another VM on the same network. You need to use *Wireshark* to observe whether the request will be accepted by the receiver. If it is accepted, an `echo` reply packet will be sent to the spoofed IP address.

The following code shows an example of how to spoof ICMP packets.

```
>>> from scapy.all import *
>>> a = IP()                (1)
>>> a.dst = '10.0.2.3'      (2)
>>> b = ICMP()              (3)
>>> p = a/b                 (4)
>>> send(p)                 (5)
.
Sent 1 packets.
```

In the code above, `Line (1)` creates an IP object from the IP class; a class attribute is defined for each IP header field. You can use `ls(a)` or `ls(IP)` to see all the attribute names/values. We can also use *a.show()* and *IP.show()* to do the same.

`Line (2)` shows how to set the destination IP address field. If a field is not set, a default value will be used.

```
>>> ls(a)
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           = '127.0.0.1'     (None)
dst        : DestIPField             = '127.0.0.1'     (None)
options    : PacketListField         = []              ([])
```

`Line (3)` creates an ICMP object. The default type is echo request. In `Line (4)`, we stack a and b together to form a new object. The `/` operator is overloaded by the IP class, so it no longer represents division; instead, it means adding `b` as the payload field of `a` and modifying the fields of `a` accordingly. As a result, we get a new object that represents an ICMP packet. You can now send out this packet using `send()` in `Line (5)`.

Make any necessary changes to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

**Submission:** In your report, you need to show the changes that you made in the code, demonstrate the spoofing, and describe your observations.

## Task 3 – Traceroute (25 Points):

The objective of this task is to use *Scapy* to estimate the distance, in terms of the number of routers, between your VM and a selected destination. This is basically what is implemented by the `traceroute` tool.

In this task, you will write your own tool. The idea is quite straightforward: just send a packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send you an ICMP error message, telling you that the time-to-live has exceeded. That is how you get the IP address of the first router. You then increase the TTL field to 2, send out another packet, and get the IP address of the second router. You will repeat this procedure until your packet finally reaches the destination. It should be noted that this experiment only gets an estimated result because, in theory, not all these packets take the same route (but in practice, they may within a short period of time). The code in the following shows one round in the procedure.

```
a = IP()
a.dst = '1.2.3.4'
a.ttl = 3
b = ICMP()
send(a/b)
```

**Submission:** You need to write your tool in Python to perform the entire procedure automatically and record the IP address. Then in your report, you need to describe the procedure, your findings, and observations.

**Task 4 – Sniffing and then Spoofing (30 Points):**

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two machines on the same LAN: the VM and the user container. From the user container, you `ping` an IP X. This will generate an ICMP echo request packet. If X is alive, the `ping` program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the VM, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the `ping` program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task.

**Submission:** In your report, you need to provide evidence to demonstrate that your technique works. In your experiment, you should `ping` the following three IP addresses from the user container. Report your observation and explain the results.

```
ping 1.2.3.4       # a non-existing host on the Internet
ping 10.9.0.99     # a non-existing host on the LAN
ping 8.8.8.8       # an existing host on the Internet
```

**Hint:** You need to understand how the ARP protocol works in order to correctly explain your observation. You also need to know a little bit about routing. The following command help you find the router for a specified destination:

```
ip route get 1.2.3.4
```