Packet Sniffing and Spoofing Lab

1 Overview

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 packet 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 lab is two-fold: learning to use the tools and understanding the technologies underlying these tools. For the second object, students will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs. This lab 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

ReadingsandVideos. Detailedcoverageofsniffingandspoofingcanbefoundinthefollowing:

Labenvironment. ThislabhasbeentestedontheSEEDUbuntu20.04VM.Youcandownloadapre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

Note for Instructors. There are two sets of tasks in this lab. The first set focuses on using tools to conduct packet sniffing and spoofing. It only requires a little bit of Python programming (usually a few lines of code); students do not need to have a prior Python programming background.

The second set of tasks is designed primarily for Computer Science/Engineering students. Students need to write their own C programs from the scratch to do sniffing and spoofing. This way, they can gain a deeper understanding on how sniffing and spoofing tools actually work. Students need to have a solid programming background for these tasks. The two sets of tasks are independent; instructors can choose to assign one set or both sets to their students, depending on their students' programming background.

2 Environment Setup using Container

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

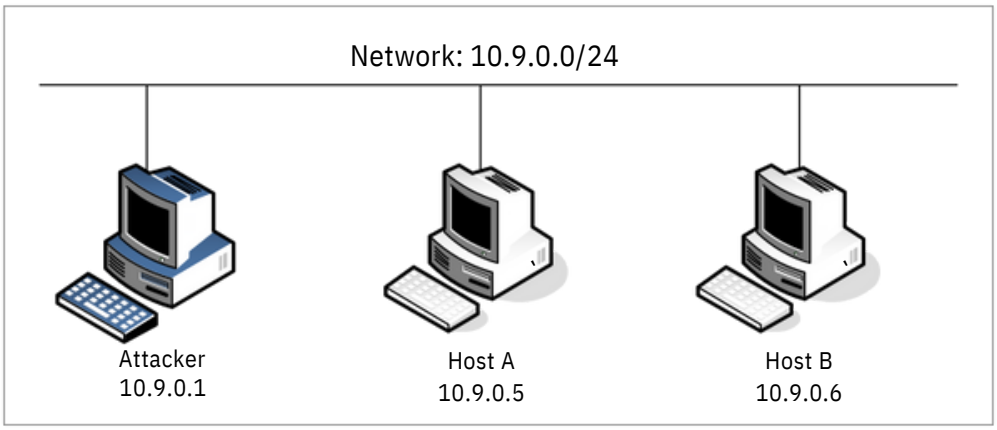

Figure 1: Lab environment setup

2.1 Container Setup and Commands

Please download the Labsetup.zip file to your VM from the lab's website, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment. Detailed explanation of the content in this file and all the involved Dockerfile can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.
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 (in our provided SEEDUbuntu 20.04 VM).

```
$ 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        #Aliasfor:docker-composebuild
$dcup           #Aliasfor:docker-composeup
$dcdown         #Aliasfor:docker-composedown
```

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. We 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 <id>   // Alias for: docker exec -it <id> /bin/bash
```

```
// The following example shows how to get a shell inside
hostC $ dockps b1004832e275 0af4ea7a3e2e 9652715c8e0a
              hostA-
              10.9.0.5
              hostB-
              10.9.0.6
$ docksh 96   hostC-
root@9652715c8e0a:/#
              10.9.0.7

// 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.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

2.2 About the Attacker Container

In this lab, 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, because we can use our favorite editors. 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. We will write our code in the ./volumes folder (on the VM), so they can be used inside the containers.

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

- Host mode. In this lab, 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, we 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 we use the provided Compose file to create containers for this lab, 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. We need to find the name of the corresponding network interface on our VM, because we need to use it in our programs. The interface name is the concatenation of br- and the ID of the network created by Docker. When we use ifconfig to list network interfaces, we will see quite a few. Look for the IP address 10.9.0.1.

```
$ ifconfig
br-c93733e9f913: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  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 ourselves (the name of the network is seed-net:

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

3 LabTaskSet1:UsingScapytoSniffandSpoofPackets

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, we will use Scapy for each task.

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

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


from scapy.all import *     À

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
```

We 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 we need to change our code frequently in an experiment.

```
# python3 >>> from scapy.all import *
>>> a = IP()
>>> a.show()
###[ IP ]###


  version   = 4
  ihl       =None
  ...
```

3.1 Task 1.1: Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. However, it is difficult to use Wireshark as a building block to construct other tools. We 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. Please read the instruction in the lab setup section regarding how to get the interface name. If we want to sniff on multiple interfaces, we can put all the interfaces in a list, and assign it to iface. See the following example:

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

Task 1.1A. If the program, for each captured packet, the callback function printpkt() 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; describe and explain your observations.

```
// 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
```

Task 1.1B. Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. Scapy's filter use the BPF (Berkeley Packet Filter) syntax; you can find the

BPF manual from the Internet. Please 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 comes from or to 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.

3.2 Task 1.2: Spoofing ICMP Packets

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. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our 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 an ICMP packets.

```
>>> from scapy.all import *

>>>a=IP()                    À
>>> a.dst = '10.0.2.3'       Á
>>>b=ICMP()                  Â
>>>p=a/b                     Ã
>>>send(p)                   Ä
.
Sent 1 packets.
```

In the code above, Line À creates an IP object from the IP class; a class attribute is defined for each IP header field. We 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 Á 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)   (None)
ihl       : BitField (4 bits)  =  None   =  (0)   (None)
tos       : XByteField         =  0      =  (1) (<Flag 0
len       : ShortField         =  None   =  ()>) (0) (64)
id        : ShortField         =  1      =  (0)   (None)
flags     : FlagsField (3 bits)=  <Flag   (None)
frag      : BitField (13 bits) =  0      ()>  (None) ([])
ttl       : ByteField          =  0   = 64
proto     : ByteEnumField      =  0      =
chksum    : XShortField        =  None
src       : SourceIPField      =
dst       : DestIPField        =  '127.0
options   : PacketListField       .0.1'  =
                                  '127.0
                                   .0.1'  =
[]
```

Line Â creates an ICMP object. The default type is echo request. In Line Ã, 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 represent an ICMP packet. We can now send out this packet using send() in Line Ä. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.