

Packet Sniffing and Spoofing Lab

Ethical Hacking 2021/22, University of Padua

Eleonora Losiouk, Alessandro Brighente, Denis Donadel, Gabriele Orazi

Based on a work of Wenliang Du. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

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.

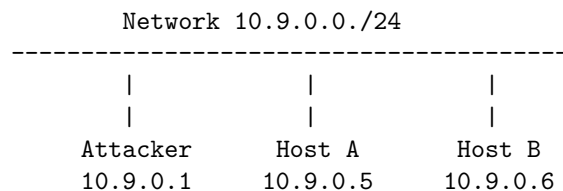
In this lab you will understand how packet sniffing works and how to build a simple sniffer. This lab covers the following topics:

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

Lab environment: This lab have been tested on the Ubuntu 20.04 VM and on a plain installation of Ubuntu 21.04.

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. The following figure 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.



2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your machine (or in you VM, if you are using it) from Moodle, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. You can find more details and ways to resolve some problems in [this manual](#).

In the following, we list some of the commonly used commands related to docker-compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (already configured in our provided VM, but you can easily add them to your local favourite `rc` 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. 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    hostA-10.9.0.5
0af4ea7a3e2e    hostB-10.9.0.6
9652715c8e0a    hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#
```

Note that 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 want to use your local `rc` file, you can simply paste at the end the following:

```
# Aliases for the Docker Compose
alias dcbuild='sudo docker-compose build' # Alias for: docker-compose build
alias dcup='sudo docker-compose up' # Alias for: docker-compose up
alias dcdown='sudo docker-compose down' # Alias for: docker-compose down
alias dockps='sudo docker ps --format "{{.ID}} {{.Names}}"'
alias docksh='f(){ sudo docker exec -it $1 /bin/bash; unset -f f; }; f'
```

Then, remember to relaunch the terminal or source your `rc` file.

2.2 About the Attacker Container

In this lab, we can either use the VM/your machine 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.

2.3 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` or `ip a` 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
e99b370eb525        host                host                local
df62c6635eae        none                null                local
c93733e9f913        seed-net            bridge              local
```

3 Task 1: 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, 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, we should import all Scapy's modules.

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

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 with its beautiful GUI. 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='eth0', filter='icmp', prn=print_pkt)
```

The code above will sniff the packets on the `eth0` 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=['eth0', 'wlan0']
```

Task 1.1A. 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; describe and explain your observations.

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.

Hint: Berkeley Packet Filter are used in Wireshark as well, so you can easily try filters on a network capture in Wireshark.

Hint: if filter is not working you may need to install `tcpdump` packet (on Debian-based distro: `sudo apt install tcpdump`).

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.

You can build ICMP packets from scratch using Scapy. You have to create one layer at the time and stack them using the `/` operator:

```
ip = IP()
ip.dst = '10.0.0.1'
icmp = ip/ICMP()
sendp(icmp)
```

You can find an introduction to Scapy on the [official docs](#). A class of attributes is defined for each IP header field and you can use `ls(IP)` or `IP.show()` to see all the attribute names/values. If a field is not set, a default value will be used.

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.

3.3 Task 1.3: Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the `traceroute` tool. In this task, we will write our own tool. The idea is quite straightforward: just send an 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 us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reach 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)
```

If you are an experienced Python programmer (but also if you are not so experienced, it is really easy!), you can write your tool to perform the entire procedure automatically. If you are new to Python programming,

you can do it by manually changing the TTL field in each round, and record the IP address based on your observation from Wireshark.

Warning: eduroam blocks your ICMP requests: try this at home!

3.4 Task 1.4: Sniffing and-then Spoofing

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. 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`

Hint: try to craft the ICMP response to be as similar as possible to the original one. You have, for instance, to add the right `Raw` data and to set the correct ICMP parameters.

4 Task 2: Writing Programs to Sniff and Spoof Packets (ADVANCED and not mandatory)

For this set up of tasks, you should compile the C code inside the host VM, and then run the code inside the container. You can use the “`docker cp`” command to copy a file from the host VM to a container. See the following example (there is no need to type the docker ID in full):

```
$ dockps
f4501a488a69 hostA-10.9.0.5
85058cbdee62 hostB-10.9.0.6
24cbc879e371 seed-attacker
// Copy a.out to the seed-attacker container's /tmp folder
$ docker cp a.out 24cbc879e371:/tmp
```

Hint: you may need to install libpcap (sudo apt install libpcap-dev).

4.1 Task 2.1: Writing Packet Sniffing Program

Sniffer programs can be easily written in C using the `pcap` library. With `pcap`, the task of sniffers becomes invoking a simple sequence of procedures in the `pcap` library. At the end of the sequence, packets will be put in buffer for further processing as soon as they are captured. All the details of packet capturing are handled by the `pcap` library. In the following there is a scratch code to start with:

```
#include <pcap.h>
#include <stdio.h>
```

```

#include <stdlib.h>
/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function. */

void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
    printf("Got a packet\n");
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp";
    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name eth3.
    //          Students need to change "eth3" to the name found on their own
    //          machines (using ifconfig). The interface to the 10.9.0.0/24
    //          network has a prefix "br-" (if the container setup is used).
    handle = pcap_open_live("eth3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);
    return 0;
    //Close the handle
}

// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap

```

Tim Carstens has also written a tutorial on how to use pcap library to write a sniffer program. The tutorial is available at <http://www.tcpdump.org/pcap.htm>.

Task 2.1A: Understanding How a Sniffer Works In this task, students need to write a sniffer program to print out the source and destination IP addresses of each captured packet. We advise you to answer the following questions:

- Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs.
- 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?
- 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):

```

# ip -d link show dev br-f2478ef59744
1249: br-f2478ef59744: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...

```

```
link/ether 02:42:ac:99:d1:88 brd ff:ff:ff:ff:ff:ff promiscuity 1 ...
```

Task 2.1B: Writing Filters. Please write filter expressions for your sniffer program to capture each of the followings. You can find online manuals for pcap filters. In your lab reports, you need to include screenshots to show the results after applying each of these filters. - Capture the ICMP packets between two specific hosts. - Capture the TCP packets with a destination port number in the range from 10 to 100.

Task 2.1C: Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using `telnet` on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (`telnet` uses TCP). It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

5 References

- [Berkeley Packets Filter \(BPF\) guide](#)
- [Scapy Docs](#)
- [pcap.h lib guide](#)