Sniffing/Spoofing Attacks (CS 915)

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, but what is more important 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, you will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs.

Lab setup

- Download the lab setup files from here (save them in any folder under /home/seed **EXCEPT** the shared folder if you use Virtual).
- Read the manual on docker here (for background learning)
- Download the script file from here

In this lab, we will use two machines that are connected to the same LAN. Instead of using two VMs, we will use two containers to simulate the two machines in a docker environment. Figure 1 depicts the lab environment setup using containers. We will do all the attacks on the attacker container, while using the other container as the user machine. This lab is based on the Packet Sniffing and Spoofing lab with some adaptations.

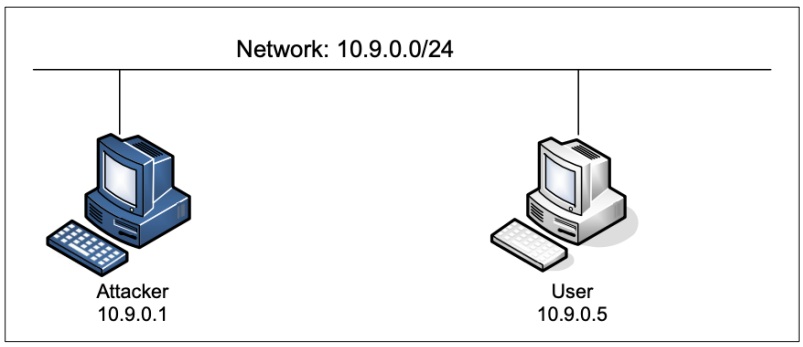

Figure 1: Lab environment setup

Under the **Labsetup** folder, run the following to build and start up the containers

```
$ dcbuild
$ dcup
```

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.

There are two ways to get the network interface name.

The first way is by using the **docker** command, as below. Find the network ID with the name net-10.9.0.0. Concatenate "br-" with the network ID "07950545de5e" to get "br-07950545de5e", which is the network interface name. Take note of this name as you will need it later.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
781929cd08e0        bridge              bridge              local
b3581338a28d        host                host                local
07950545de5e        net-10.9.0.0        bridge              local
77acecccbe26        none                null                local
```

The second way is by using **ifconfig**. When we use ifconfig to list network interfaces, we will see quite a few. Look for the IP address 10.9.0.1. You should get the same network interface name "br-07950545de5e" as above.

```
$ ifconfig
br-07950545de5e: 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::42:4cff:fee3:19f8  prefixlen 64  scopeid 0x20<link>
        ether 02:42:4c:e3:19:f8  txqueuelen 0  (Ethernet)
        RX packets 36089  bytes 1590496 (1.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4899819  bytes 264473220 (264.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Accessing Containers

If you want to access a particular container, first, run **dockps** to get containers IDs. Then, run **docksh <ID>** to get a shell of the specified container. (It's sufficient to type the first few characters of the container ID as long as it uniquely identifies the container)

1. Get Containers IDs

```
$ dockps
ad38fae170cf  host-10.9.0.5
157877bfeb53  seed-attacker
```

2. Get a shell on container

```
$ docksh ad
root@ad38fae170cf:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
root@ad38fae170cf:/#
```

Overview of the lab tasks

- Task 1: Sniffing Packets
- Task 2: Spoofing ICMP Packets
- Task 3: Sniffing and-then Spoofing

Task 1: Sniffing Packets

In this task, we will learn three methods to sniff packets, using 1) Scapy; 2) Wireshark; 3) tcpdump.

Using Scapy

First, we will learn to write our own sniffer program by using Scapy in Python. 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. You will need to change it to the correct network interface name that you have obtained from the lab setup instruction. Save the above code (with the correct network interface name) as **sniffer.py**.

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

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

While the sniffer program is running, log into the victim's machine (using **dockps** and **docksh** commands as explained in "Accessing Containers"). Try to ping an external website (say bbc.com) and observe what's captured by the sniffer program.

Using Wireshark

Instead of using Scapy, we will repeat the sniffing exercise by using Wireshark. Terminate the sniffer.py program (Ctrl+C), and open Wireshark. Choose the correct network interface to sniff. As before, from the victim's container, try to ping an external website. Observe what's captured by Wireshark.

Using tcpdump

Another tool to sniff the traffic is **tcpdump**. In fact, Wireshark uses **tcpdump** to capture data at the lower layer, but it presents the data in a more readable manner at the higher layer. Repeat the same sniffing exercise by using tcpdump in the command line.

```
$ sudo tcpdump -n src
```

It's possible to define filters for **tcpdump** to capture more specific traffic (like you would do in Wireshark). The following are some examples.

```
tcpdump -n src 192.168.1.1 and dst port 21
tcpdump dst 10.1.1.1 && !icmp
```

Task 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 packet.

```
>>> from scapy.all import *
>>> a = IP()                    ①
>>> a.dst = '10.9.0.5'          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. 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 ② shows how to set the destination IP address field. If a field is not set, a default value will be used.

```
>>> ls(a)  # show attributes of the IP packet a.2
```

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. We can now send out this packet using send() in Line ⑤. Please 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. You can use Wireshark to capture the spoofed request.

Task 3: 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 (named host)**. 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. A sample Python program using Scapy is provided in the lab files with the name **sniff_spoof.py** (you will need to change **iface** value to the correct network interface name for the program to work).

Run the attack

On the Attacker's machine or VM

```
$  sudo ./sniff_spoof.py
```

On the user Container

Try to ping the following three IP addresses from **the user container** (use **dockps** and **docksh <ID>** to gain root access to the user container).

```
# 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
```

Observe the output of sniff_spoof.py and what is being captured by Wireshark.

Answer the following questions (included in the assignment):

- **Question 1**: Why can you get the echo reply from 1.2.3.4 which does not exist on the Internet?
- **Question 2**: why can't you get the echo reply from 10.9.0.99? (Hint: you need to understand how the ARP protocol works; also observe Wireshark capture)

After the lab

- Stop the docker container and run **dcdown** under Labsetup/
- Complete the after-lab-assignment