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 to learn how use the tools and understand the underlying technologies.

2 Lab Environment

This lab has been tested on the Ubuntu 16.04 VM that you should have installed on your computer during the first lab of this course.

3 Submission

Submit a PDF document with your answers to the tasks and questions in this lab. Include the task numbers and the questions. Place your answers immediately after the task number or question. Some answers may require you to include a screen shot. Paste any code that you write or use in the corresponding answer.

4 Lab Tasks

4.1 Lab Task Set 1: Using Tools 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 sniffingand 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.

```
#!/bin/bin/python

from scapy.all import
```

```
a = IP()

a.show()
```

To run the program, we execute following commands. We also show a part of the expected output.

```
$ sudo python mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  .
  .
  .
```

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. We show how to use the Python console in the following example.

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

4.1.1 Task 1.1: Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. 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/python

from scapy.all import*

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
```

Task 1.1A. The above program sniffs packets. 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 (i.e., `sudo`) 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. See example below.

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

// Run the program without the root privilege
$ python sniffer.py
```

Deliverable for Task 1.1A. Include a screen shot of the output of your Python script on the terminal showing the packets it captured while using the root privilege. 2-3 captured packets is enough. Include your answer to the following question: Why is the output of your program different without using the root privilege?

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 that your sniffer program only display the packets specified in the filters (each filter should be set separately):

1. Capture only ICMP packets
2. Capture any TCP packet that comes from a particular IP and with a destination port number 23. You can choose any IP address.
3. Capture packets that 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.

Deliverable for Task 1.1B. Include a screen shot of your sniffer only capturing the specified type of packets. Include one screen shot for each of the three items in the list above.

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

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

In the code above, Line 1 creates an IP object from the IP class; a class attribute is defined for each IPheader 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.

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 represent an ICMP packet. We can now send out this packet using `send()` in Line 5. 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.

Deliverable Task 1.2 For this task setup two VMs: VM1 and VM2. Use VM1 to craft the spoofed packet with an arbitrary source IP address. In VM2, set up the packet sniffer and capture the spoofed packet from VM1. Include a screen shot of VM1 sending the spoofed packet and a screen shot of VM2 receiving it.

We show an example output of the `ls(a)` command showing the packet class attributes below.

```
>>> 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
proto      : ByteEnumField           = 0               (0)chksum     : XShortField
src        : SourceIPField           = '127.0.0.1'     (None)
dst        : DestIPField             = '127.0.0.1'     (None)
options    : PacketListField         = []              ([])
```

4.2.1 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. Int his task, we will write our 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 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 following code snippet 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 are familiar with Python, you can write a script that performs 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 recording the IP address based on your observation from Wireshark. Either way is acceptable, as long as you get the result. However, the second approach may be more labor intensive.

Deliverable Task 1.3 A screen shot of the round trip times calculated by your Python script or by Wireshark. If you wrote a Python script include it here. Although there may be existing `traceroute` programs in the Internet, you are expected to write your own.

Bonus assignment Complete task set 2 in the Packet Sniffing and Spoofing lab in the Seed website for Ubuntu 16.