Exercise 1: 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

Readings and Videos. Detailed coverage of sniffing and spoofing can be found in the following:

- Chapter 15 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 2 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 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 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. The current version of the SEED VM may not have Scapy installed for Python3. We can use the following command to install Scapy for Pyhon3.

```
$ sudo pip3 install scapy
```

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 ①), we should import all Scapy's modules.

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

from scapy.all import *    ①

a = IP()
a.show()

$ sudo python3 mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  ...


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

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

2.1 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/python3
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 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
$ sudo ./sniffer.py

// Run the program without the root privilege
$ 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.

2.2 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()               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. 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)
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 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.

2.3 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, 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. Either way is acceptable, as long as you get the result.

2.4 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 VMs on the same LAN. From VM A, 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 VM B, 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.

3 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets (for extra points)

3.1 Task 2.1: Writing Packet Sniffing Program

Sniffer programs can be easily written 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.

The SEED book "*Computer Security: A Hands-on Approach*" provides a sample code in Chapter 12, showing how to write a simple sniffer program using `pcap`. We include the sample code in the following (see the book for detailed explanation).

```
#include <pcap.h>
#include <stdio.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[] = "ip proto icmp";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  //         Students needs to change "eth3" to the name
  //         found on their own machines (using ifconfig).
  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);
```

```
  pcap_setfilter(handle, &fp);

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);

  pcap_close(handle);   //Close the handle
  return 0;
}

// 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. Students can type in the above code or download the sample code from the SEED book's website (`https://www.handsonsecurity.net/figurecode.html`). Students should provide screenshots as evidences to show that their sniffer program can run successfully and produces expected results. In addition, please 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. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.
- **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?
- **Quesiton 3.** Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

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.

3.2 Task 2.2: Spoofing

When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). OSes will set most of the fields, while

only allowing users to set a few fields, such as the destination IP address, the destination port number, etc. However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called packet spoofing, and it can be done through *raw sockets*.

Raw sockets give programmers the absolute control over the packet construction, allowing programmers to construct any arbitrary packet, including setting the header fields and the payload. Using raw sockets is quite straightforward; it involves four steps: (1) create a raw socket, (2) set socket option, (3) construct the packet, and (4) send out the packet through the raw socket. There are many online tutorials that can teach you how to use raw sockets in C programming. We have linked some tutorials to the lab's web page. Please read them, and learn how to write a packet spoofing program. We show a simple skeleton of such a program.

```
int sd;
struct sockaddr_in sin;
char buffer[1024]; // You can change the buffer size

/* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
 * tells the sytem that the IP header is already included;
 * this prevents the OS from adding another IP header.  */
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if(sd < 0) {
    perror("socket() error"); exit(-1);
}

/* This data structure is needed when sending the packets
 * using sockets. Normally, we need to fill out several
 * fields, but for raw sockets, we only need to fill out
 * this one field */
sin.sin_family = AF_INET;

// Here you can construct the IP packet using buffer[]
//    - construct the IP header ...
//    - construct the TCP/UDP/ICMP header ...
//    - fill in the data part if needed ...
// Note: you should pay attention to the network/host byte order.


/* Send out the IP packet.
 * ip_len is the actual size of the packet. */
if(sendto(sd, buffer, ip_len, 0, (struct sockaddr *)&sin,
             sizeof(sin)) < 0) {
      perror("sendto() error"); exit(-1);
}
```

Task 2.2A: Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidences (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

Task 2.2B: Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

Questions. Please answer the following questions.

- **Question 4.** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
- **Question 5.** Using the raw socket programming, do you have to calculate the checksum for the IP header?
- **Question 6.** Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

3.3 Task 2.3: Sniff and then Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two VMs on the same LAN. From VM A, 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 VM B, 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 write such a program in C, and include screenshots in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

4 Guidelines

4.1 Filling in Data in Raw Packets

When you send out a packet using raw sockets, you basically construct the packet inside a buffer, so when you need to send it out, you simply give the operating system the buffer and the size of the packet. Working directly on the buffer is not easy, so a common way is to typecast the buffer (or part of the buffer) into structures, such as IP header structure, so you can refer to the elements of the buffer using the fields of those structures. You can define the IP, ICMP, TCP, UDP and other header structures in your program. The following example show how you can construct an UDP packet:

```
struct ipheader {
   type  field;
   .....
}

struct udpheader {
   type field;
   ......
}

// This buffer will be used to construct raw packet.
char buffer[1024];

// Typecasting the buffer to the IP header structure
struct ipheader *ip = (struct ipheader *) buffer;

// Typecasting the buffer to the UDP header structure
```

```
struct udpheader *udp = (struct udpheader *) (buffer
                              + sizeof(struct ipheader));

// Assign value to the IP and UDP header fields.
ip->field = ...;
udp->field = ...;
```

4.2 Network/Host Byte Order and the Conversions

You need to pay attention to the network and host byte orders. If you use x86 CPU, your host byte order uses *Little Endian*, while the network byte order uses *Big Endian*. Whatever the data you put into the packet buffer has to use the network byte order; if you do not do that, your packet will not be correct. You actually do not need to worry about what kind of Endian your machine is using, and you actually should not worry about if you want your program to be portable.

What you need to do is to always remember to convert your data to the network byte order when you place the data into the buffer, and convert them to the host byte order when you copy the data from the buffer to a data structure on your computer. If the data is a single byte, you do not need to worry about the order, but if the data is a `short`, `int`, `long`, or a data type that consists of more than one byte, you need to call one of the following functions to convert the data:

```
htonl(): convert unsigned int from host to network byte order.
ntohl(): reverse of htonl().
htons(): convert unsigned short int from host to network byte order.
ntohs(): reverse of htons().
```

You may also need to use `inet_addr()`, `inet_network()`, `inet_ntoa()`, `inet_aton()` to convert IP addresses from the dotted decimal form (a string) to a 32-bit integer of network/host byte order. You can get their manuals from the Internet.

5 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Deadline: 9 Feb 11:59pm

Exercise 2: ARP Cache Poisoning Attack Lab

1 Overview

The Address Resolution Protocol (ARP) is a communication protocol used for discovering the link layer address, such as the MAC address, given an IP address. The ARP protocol is a very simple protocol, and it does not implement any security measure. The ARP cache poisoning attack is a common attack against the ARP protocol. Using such an attack, attackers can fool the victim into accepting forged IP-to-MAC mappings. This can cause the victim's packets to be redirected to the computer with the forged MAC address, leading to potential man-in-the-middle attacks.

The objective of this lab is for students to gain the first-hand experience on the ARP cache poisoning attack, and learn what damages can be caused by such an attack. In particular, students will use the ARP attack to launch a man-in-the-middle attack, where the attacker can intercept and modify the packets between the two victims A and B. Another objective of this lab is for students to practice packet sniffing and spoofing skills, as these are essential skills in network security, and they are the building blocks for many network attack and defense tools. Students will use Scapy to conduct lab tasks. This lab covers the following topics:

- The ARP protocol
- The ARP cache poisoning attack
- Man-in-the-middle attack
- Scapy programming

Videos. Detailed coverage of the ARP protocol and attacks can be found in the following:

- Section 3 of the SEED Lecture at Udemy, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Task 1: ARP Cache Poisoning

The objective of this task is to use packet spoofing to launch an ARP cache poisoning attack on a target, such that when two victim machines A and B try to communicate with each other, their packets will be intercepted by the attacker, who can make changes to the packets, and can thus become the man in the middle between A and B. This is called Man-In-The-Middle (MITM) attack. In this lab, we use ARP cahce poisoning to conduct an MITM attack.

The following code skeleton shows how to construct an ARP packet using Scapy.

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

```
E = Ether()
A = ARP()

pkt = E/A
sendp(pkt)
```

The above program constructs and sends an ARP packet. Please set necessary attribute names/values to define your own ARP packet. We can use `ls(ARP)` to see the attribute names of the ARP class. If a field is not set, a default value will be used (see the third column of the output):

```
$ python3
>>> from scapy.all import *
>>> ls(ARP)
hwtype     : XShortField                          = (1)
ptype      : XShortEnumField                      = (2048)
hwlen      : ByteField                            = (6)
plen       : ByteField                            = (4)
op         : ShortEnumField                       = (1)
hwsrc      : ARPSourceMACField                    = (None)
psrc       : SourceIPField                        = (None)
hwdst      : MACField                             = ('00:00:00:00:00:00')
pdst       : IPField                              = ('0.0.0.0')
```

In this task, we have three VMs, A, B, and M. We would like to attack A's ARP cache, such that the following results is achieved in A's ARP cache.

```
B's IP address --> M's MAC address
```

There are many ways to conduct ARP cache poisoning attack. Students need to try the following three methods, and report whether each method works or not.

- **Task 1A (using ARP request).** On host M, construct an ARP request packet and send to host A. Check whether M's MAC address is mapped to B's IP address in A's ARP cache.
- **Task 1B (using ARP reply).** On host M, construct an ARP reply packet and send to host A. Check whether M's MAC address is mapped to B's IP address in A's ARP cache.
- **Task 1C (using ARP gratuitous message).** On host M, construct an ARP gratuitous packets. ARP gratuitous packet is a special ARP request packet. It is used when a host machine needs to update outdated information on all the other machine's ARP cache. The gratuitous ARP packet has the following characteristics:
  - The source and destination IP addresses are the same, and they are the IP address of the host issuing the gratuitous ARP.
  - The destination MAC addresses in both ARP header and Ethernet header are the broadcast MAC address (`ff:ff:ff:ff:ff:ff`).
  - No reply is expected.

3 Task 2: MITM Attack on Telnet using ARP Cache Poisoning

Hosts A and B are communicating using Telnet, and Host M wants to intercept their communication, so it can make changes to the data sent between A and B. The setup is depicted in Figure 1.

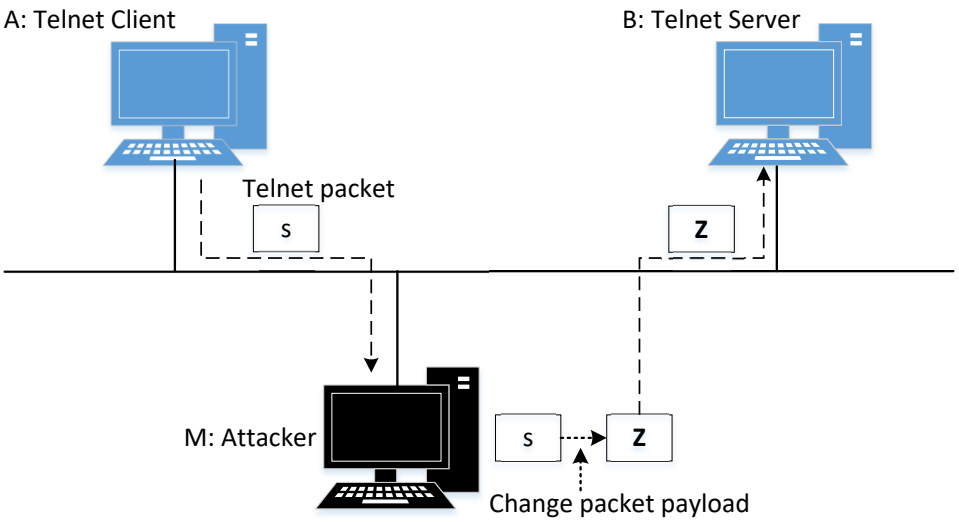

Figure 1: Man-In-The-Middle Attack against telnet

Step 1 (Launch the ARP cache poisoning attack). First, Host M conducts an ARP cache poisoning attack on both A and B, such that in A's ARP cache, B's IP address maps to M's MAC address, and in B's ARP cache, A's IP address also maps to M's MAC address. After this step, packets sent between A and B will all be sent to M. We will use the ARP cache poisoning attack from Task 1 to achieve this goal.

Step 2 (Testing). After the attack is successful, please try to ping each other between Hosts A and B, and report your observation. Please show Wireshark results in your report.

Step 3 (Turn on IP forwarding). Now we turn on the IP forwarding on Host M, so it will forward the packets between A and B. Please run the following command and repeat Step 2. Please describe your observation.

```
$ sudo sysctl net.ipv4.ip_forward=1
```

Step 4 (Launch the MITM attack). We are ready to make changes to the Telnet data between A and B. Assume that A is the Telnet client and B is the Telnet server. After A has connected to the Telnet server on B, for every key stroke typed in A's Telnet window, a TCP packet is generated and sent to B. We would like to intercept the TCP packet, and replace each typed character with a fixed character (say Z). This way, it does not matter what the user types on A, Telnet will always display Z.

From the previous steps, we are able to redirect the TCP packets to Host M, but instead of forwarding them, we would like to replace them with a spoofed packet. We will write a sniff-and-spoof program to accomplish this goal. In particular, we would like to do the following:

- We first keep the IP forwarding on, so we can successfully create a Telnet connection between A to B. Once the connection is established, we turn off the IP forwarding using the following command. Please type something on A's Telnet window, and report your observation:

  ```
  $ sudo sysctl net.ipv4.ip_forward=0
  ```

- We run our sniff-and-spoof program on Host M, such that for the captured packets sent from A to B, we spoof a packet but with TCP different data. For packets from B to A (Telnet response), we do not make any change, so the spoofed packet is exactly the same as the original one.

To help students get started, we provide a skeleton sniff-and-spoof program in the following. The program capture all the TCP packets, and then for packets from A to B, it makes some changes (the modification part is not included, because that is part of the task). For packets from B to A, the program simply forward the original packets.

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

VM_A_IP = "10.0.2.6"
VM_B_IP = "10.0.2.7"

def spoof_pkt(pkt):
  if pkt[IP].sr == VM_A_IP and pkt[IP].dst == VM_B_IP \
                           and pkt[TCP].payload:

     # Create a new packet based on the captured one.
     # (1) We need to delete the checksum fields in the IP and TCP headers,
     #     because our modification will make them invalid.
     #     Scapy will recalculate them for us if these fields are missing.
     # (2) We also delete the original TCP payload.
     newpkt = IP(pkt[IP])
     del(newpkt.chksum)
     del(newpkt[TCP].chksum)
     del(newpkt[TCP].payload)

     ########################################################################
     # Construct the new payload based on the old payload.
     # Students need to implement this part.

     olddata = pkt[TCP].payload.load   # Get the original payload data
     newdata = olddata  # No change is made in this sample code
     ########################################################################

     # Attach the new data and set the packet out
     send(newpkt/newdata)

  elif pkt[IP].src == VM_B_IP and pkt[IP].dst == VM_A_IP:
     send(pkt[IP]) # Forward the original packet

pkt = sniff(filter='tcp',prn=spoof_pkt)
```

It should be noted that the code above captures all the TCP packets, including the one generated by the program itself. That is undesirable, as it will affect the performance. Students needs to change the filter, so it does not capture its own packets.

Behavior of Telnet. In Telnet, typically, every character we type in the Telnet window triggers an individual TCP packet, but if you type very fast, some characters may be sent together in the same packet. That is why in a typical Telnet packet from client to server, the payload only contains one character. The character

sent to the server will be echoed back by the server, and the client will then display the character in its window. Therefore, what we see in the client window is not the direct result of the typing; whatever we type in the client window takes a round trip before it is displayed. If the network is disconnected, whatever we typed on the client window will not displayed, until the network is recovered. Similarly, if attackers change the character to Z during the round trip, Z will be displayed at the Telnet client window, even though that is not what you have typed.

4 Task 3: MITM Attack on Netcat using ARP Cache Poisoning

This task is similar to Task 2, except that Hosts A and B are communicating using `netcat`, instead of `telnet`. Host M wants to intercept their communication, so it can make changes to the data sent between A and B. You can use the following commands to establish a `netcat` TCP connection between A and B:

```
On Host B (server, IP address is 10.0.2.7), run the following:
$ nc -l 9090

On Host A (client), run the following:
$ nc 10.0.2.7 9090
```

Once the connection is made, you can type messages on A. Each line of messages will be put into a TCP packet sent to B, which simply displays the message. Your task is to replace every occurrence of your first name in the message with a sequence of A's. The length of the sequence should be the same as that of your first name, or you will mess up the TCP sequence number, and hence the entire TCP connection. You need to use your real first name, so we know the work is done by you.

5 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Deadline: 9 Feb 11:59pm