# 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`, 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 for students to master the technologies underlying most of the sniffing and spoofing tools. Students will play with some simple sniffer and spoofing programs, read their source code, modify them, and eventually gain an in-depth understanding of the technical aspects of these programs. At the end of this lab, students should be able to write their own sniffing and spoofing programs.

**Note for Instructors.** If the instructor plans to hold lab sessions for this lab, it is suggested that the following should be covered:

- Socket and raw socket programming
- Programming using the `pcap` library

## 2 Lab Tasks

### 2.1 Task 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. Tim Carstens has 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 1.a: Understanding `sniffex`.** Please download the `sniffex.c` program from the tutorial mentioned above, compile and run it. You should provide screendump evidence to show that your program runs successfully and produces expected results.

**Problem 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.

**Problem 2:** Why do you need the root privilege to run `sniffex`? Where does the program fail if executed without the root privilege?

**Problem 3:** Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.

**Task 1.b: Writing Filters.** Please write filter expressions for your sniffer program to capture each of the followings. In your lab reports, you need to include screendumps to show the results of applying each of these filters.

- Capture the ICMP packets between two specific hosts.
- Capture the TCP packets that have a destination port range from to port 10 - 100.

**Task 1.c: Sniffing Passwords.** Please show how you can use `sniffex` to capture the password when somebody is using `telnet` on the network that you are monitoring. You may need to modify the `sniffex.c` a little bit if needed. You also need to start the `telnetd` server on your VM. If you are using our pre-built VM, the `telnetd` server is already installed; just type the following command to start it.

```
% sudo service openbsd-inetd start
```

## 2.2 Task 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 arbitary 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 spoonfing 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.a: Write a spoofing program.** You can write your own packet program or download one. You need to provide evidences (e.g., Wireshark packet trace) to show us that your program successfully sends out spoofed IP packets.

**Task 2.b: 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 alieve). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

**Task 2.c: Spoof an Ethernet Frame.** Spoof an Ethernet Frame. Set 01:02:03:04:05:06 as the source address. To tell the system that the packet you constract already includes the Ethernet header, you need to create the raw socket using the following parameters:

```
sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
```

When constructing the packets, the beginning of the `buffer[]` array should now be the Ethernet header.

**Questions.** Please answer the following questions.

**Question 4:** Can you set the IP packet length field to an arbitary 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?

### 2.3 Task 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, and include screendumps in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

## 3 Guidelines

### 3.1 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 structres 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 = ...;
```

## 3.2 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.

## 3.3 VirtualBox Network Configuration for Task 3

The configuration mainly depends on the IP address you ping. Please make sure finish the configuration before the task. The details are attached as follows:

- When you ping a non-existing IP in the same LAN, an ARP request will be sent first. If there is no reply, the ICMP requests will not be sent later. So in order to avoid that ARP request which will stop the ICMP packet, you need to change the ARP cache of the victim VM by adding another MAC to the IP address mapping entry. The command is as follows:

  ```
  % sudo arp -s 192.168.56.1 AA:AA:AA:AA:AA:AA
  ```

  IP `192.168.56.1` is the non-existing IP on the same LAN. `AA:AA:AA:AA:AA:AA` is a spoofed MAC address.

- When you ping a non-existing IP outside the LAN, the Network settings will make a difference.

  If you are using the new “NAT Network”, no further changes are needed. If your VMs are connected to two virtual networks (one through the “NAT” adapter, and the other through the “Host-only” adapter), there are two things you need to keep in mind to finish the task:

  - If the victim user is trying to ping a non-existing IP outside the local network, the traffic will go to the NAT adapter. VirtualBox permanently disables the promiscuous mode for NAT by default.

Here we explain how to address these issues:

Redirect the traffic of the victim VM to the gateway of your "Host-only" network by changing the routing table in the victim VM. You can use the command `route` to configure the routing table. Here is an example:

```
% sudo route add -net 1.1.2.0 netmask 255.255.255.0 gw 192.168.56.1
```

The subnet `1.1.2.0/24` is where the non-existing IP sits. The traffic is redirected to `192.168.56.1`, which is on the same LAN of the victim but does not exist. Before sending out the ICMP request, again the victim VM will send an ARP request to find the gateway. Similarly, you should change the ARP cache as stated in the previous example.

Alternatively, you can setup your two VMs using "Bridged Network" in VirtualBox to do the task. "Bridged Network" in VirtualBox allows you to turn on the promiscuous mode. The following procedure on the VirtualBox achieves it:

```
Settings -> Network -> Adapter 1 tab -> "Attach to" section:
Select "Bridged Adapter"

In "Advanced" section -> "Promiscuous Mode":
Select "Allow All"
```

## 4 Submission

You need to submit a detailed lab report 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.