Implementing Packet Sniffing and Spoofing

By: Shivam Pandit

**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.**

Solution: The following library calls are within sniffex.c:

1. Setting up the device: Sniffex.c contains pcap\_lookupdev that sets the sniffer by determining interface which will be like eth0 in Linux and xl1 in BSD.

2. Initialize PCAP: pcap\_lookupnet call in sniffex.c tells pcap what device we will be sniffing on.

3. Traffic Filtering: Here we start the sniffing and even apply set of rules if required like pcap\_open\_live call in sniffex.c which starts sniffing on the capture device, pcap\_datalink returns the kind of device we are capturing on, pcap\_compile compiles the filter expression to set the filter and pcap\_setfilter sets the compiled filter

4. Execution /actual sniffing: We can either sniff one packet at a time (pcap\_next) or continuously sniff (pcap\_loop). In sniffex.c we have used pcap\_loop.

5. Ending Session: After our sniffing needs are satisfied, we close our session and are complete using pcap\_freecode in sniffex.c that frees up allocated memory generated by pcap\_compile and pcap\_close call which closes the sniffing session

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

Solution: We need root privilege to run sniffex because sniffex will need to access a network device which is not accessible to a non-root user. Till one is not a root he cannot access NIC which is required here.

The below mentioned code from sniffex.c causes this to fail:

/\* find a capture device if not specified on command-line \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

fprintf(stderr, "Couldn't find default device: %s\n",

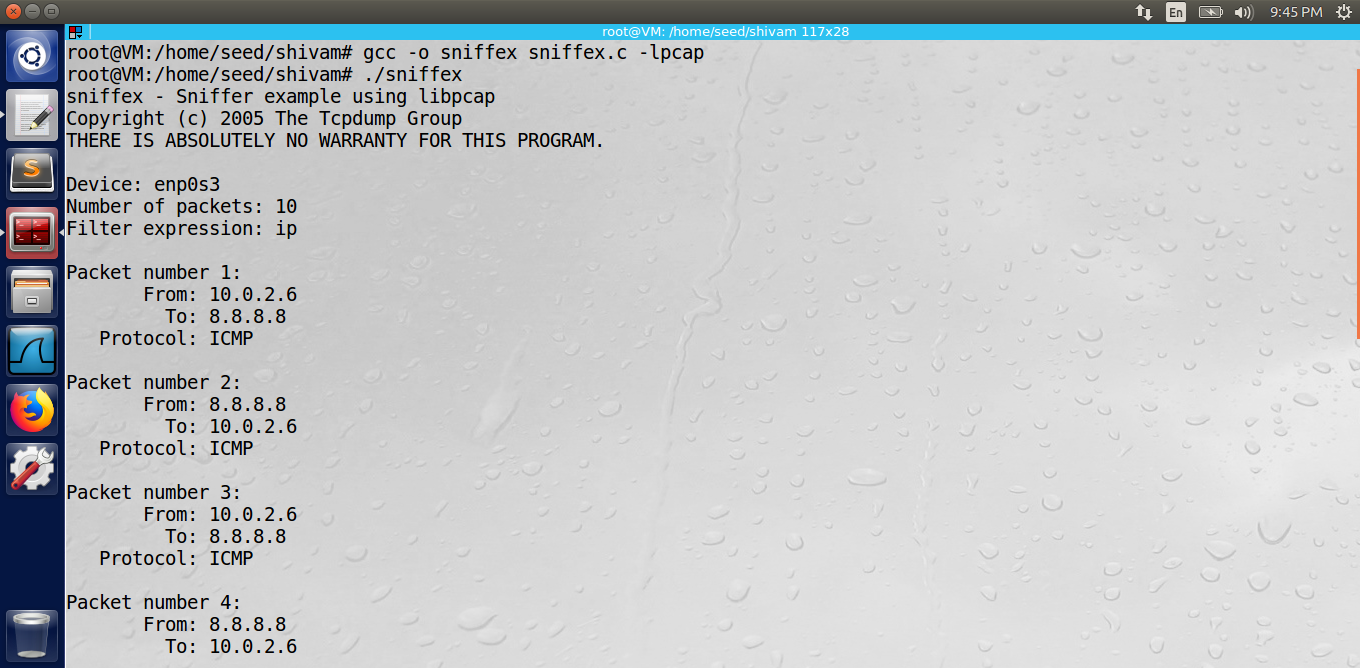
errbuf);

exit(EXIT\_FAILURE);

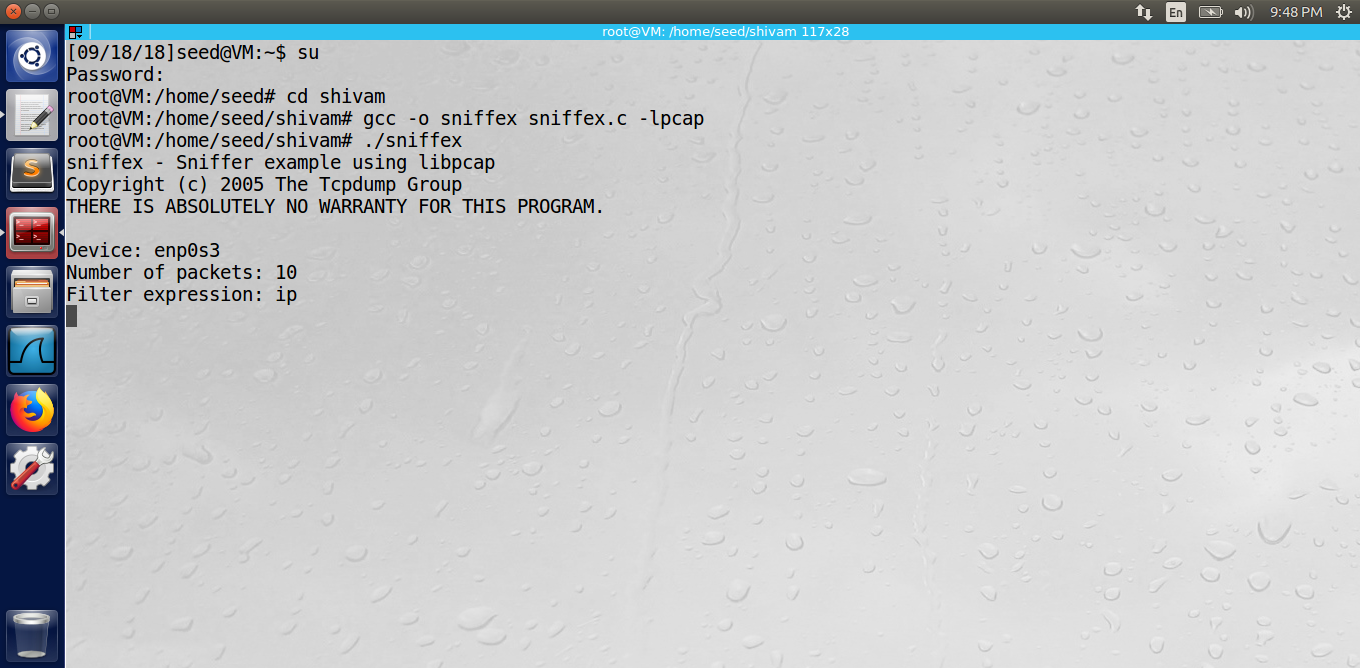
**Problem 3: Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.**

Solution: In promiscuous mode all the packets are sent to a computer or sniffed by sniffer irrespective of whether they were addressed to it or not. On the contrary, in non-promiscuous mode only those packets are sent to the computer or sniffed by sniffer which are addressed to it. By default, sniffex sniffs in promiscuous mode till we change it manually which is a Boolean flag i.e. 0 or 1 inside pcap\_open\_live call.

For verifying this, i used two virtual machines say A and B. Sniffex was running on machine A set in non-promiscuous mode and when i pinged some random ip in machine B, there was no packet sniffed at A other than ones addressed to it. However, same thing i did in promiscuous mode, some packets were received at A that were not addressed to it. I have attached the screenshots below for the same.



With Promiscuous Mode



With Non-Promiscuous Mode

**Problem 4: Please write filter expressions to capture each of the followings. In your lab reports, you need to include screen dumps to show the results of applying each of these filters.**

**A. Capture the ICMP packets between two specific hosts.**

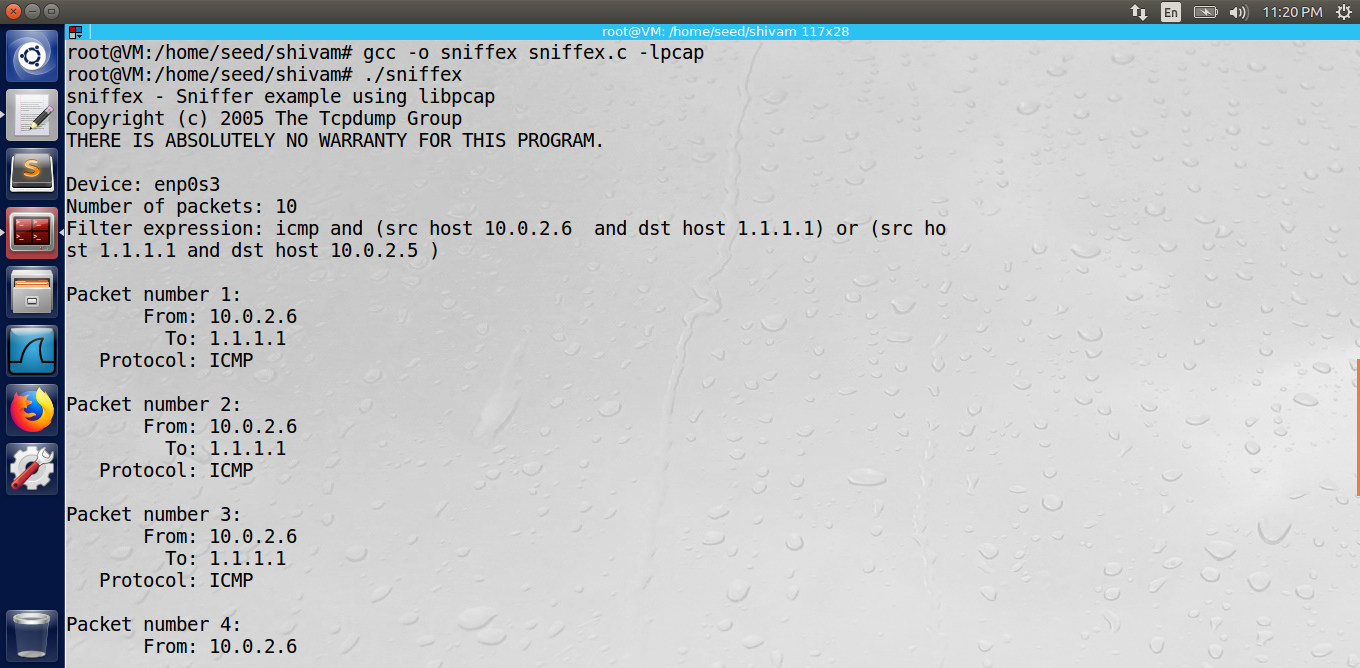
**B. Capture the TCP packets that have a destination port range from to port 10 - 100.**

Solution 4A. Here, my sniffer was running on ip 10.0.2.5 and sender on ip 10.0.2.6. We can filter only the ICMP packets by modifying the filter\_exp[] string as mentioned below.

// ICMP packets between this host 10.0.2.5 and 1.1.1.1

char filter\_exp[] = "icmp and (src host 10.0.2.6 and dst host 1.1.1.1) or (src host 1.1.1.1 and dst host 10.0.2.5 )";

Running both sniffex and ping 1.1.1.1 results in the following output which shows our filter has successfully captured packets that were coming from sender ip 10.0.2.6 to host ip 1.1.1.1. If pinged any random ip not mentioned in the filter it will not catch which serves the purpose of a filter here.



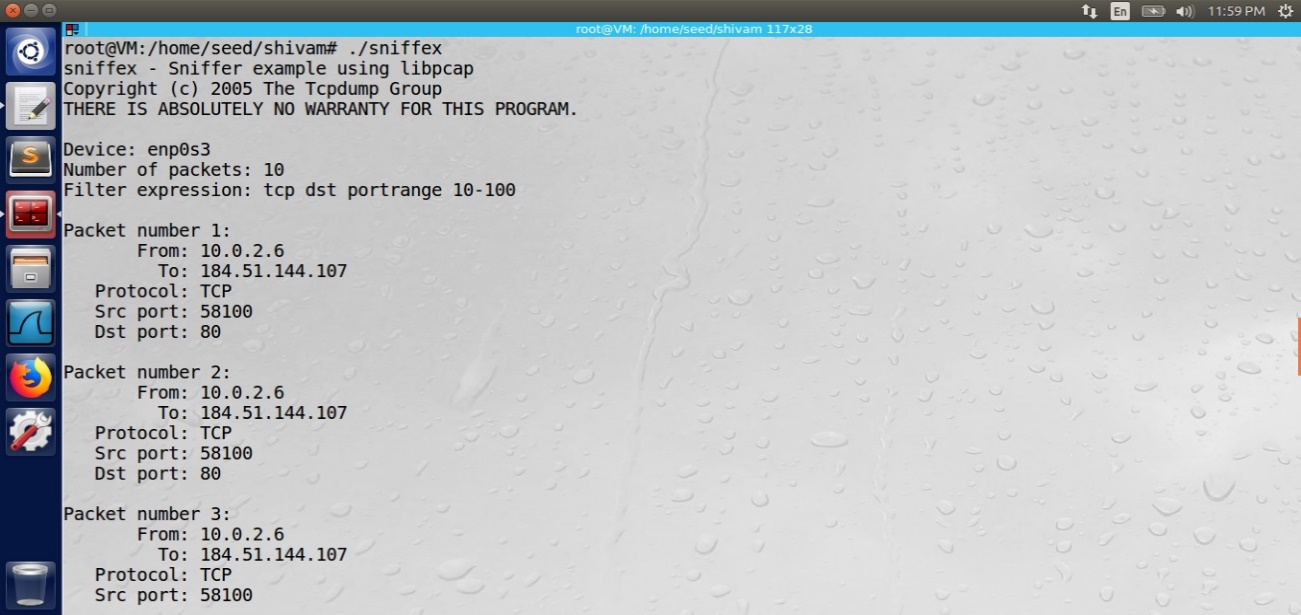
Sniffed packets from sender ip 10.0.2.6 to host ip 1.1.1.1.

Solution 4B: We have to capture Packets within TCP port range from 10-100. To achieve same, I applied following filter:

// TCP packets with dest port 10-100 PROBLEM 4B

char filter\_exp[] = "tcp dst portrange 10-100";

Sniffer running on 10.0.2.5 caught the packet sent from 10.0.2.6 to 184.51.144.107 that has a port number 80 which falls in the range of 10-100.



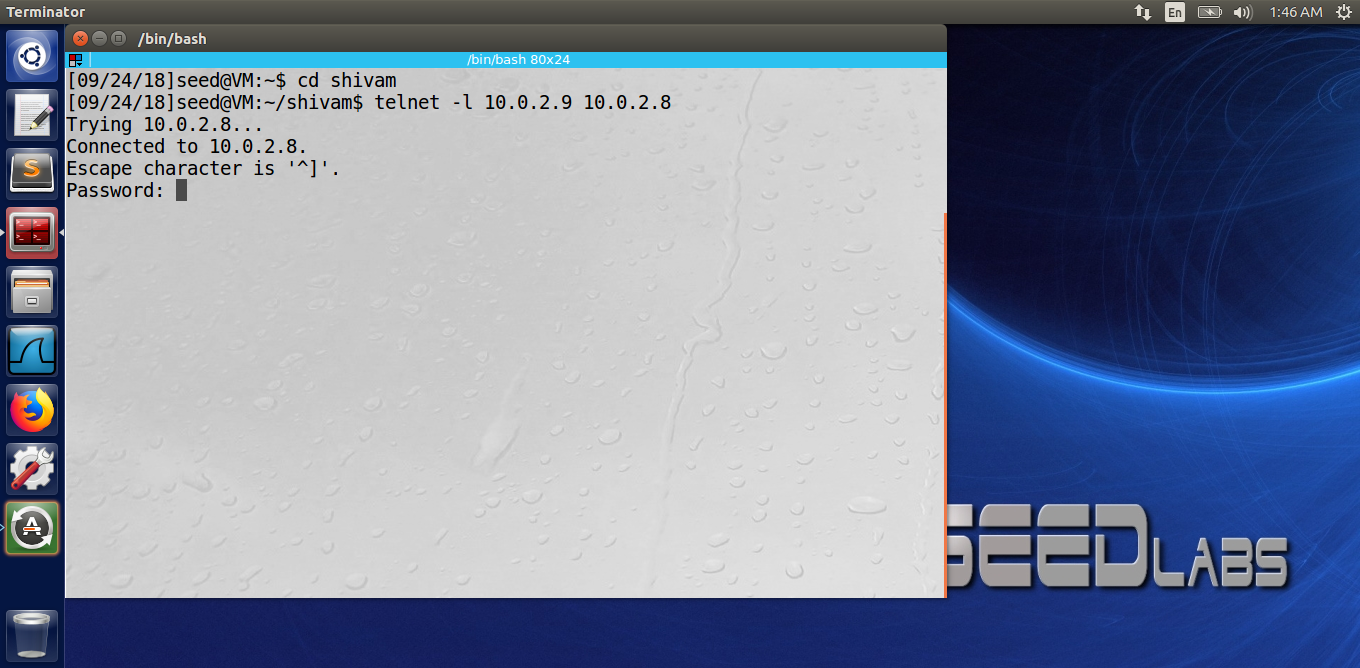
Sniffed packets with Port 80(HTTP protocol)

**Problem 5: 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, thetelnetd server is already installed; just type the following command to start it. % sudo service openbsd-inetd start**

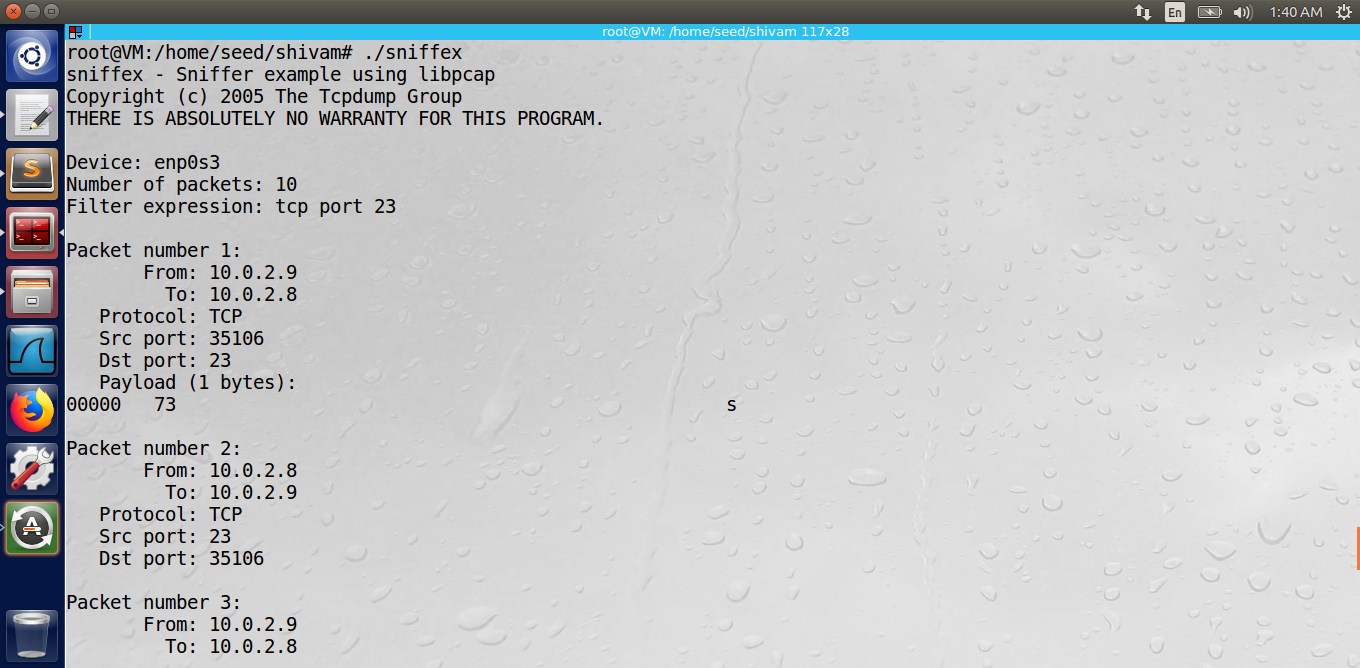
Solution: To demonstrate this, we take here two machines say A and B. Since we're sniffing telnet passwords, we can just look for tcp packets on port 23 modifying small portion of code in sniffex.c as written below:

char filter\_exp[] = "tcp port 23";

When A tries to telnet to B on port 23, B requests A for login credentials. Now, when A enters the login and password requested by B, these details are easily captured by the sniffer on the network. To see if we have completely received password we have to look for “..” sign which marks the end of the password. Below Screenshot shows how password was sent character by character and caught by the sniffer.



Screenshot above: Machine A telnets to machine B and user enters password



Screenshot above: Sniffer running on host machine records password

**Problem 6: Please use your own words to describe the sequence of the library calls that are essential for packet spoofing. This is meant to be a summary.**

Solution: Four necessary library calls necessary for spoofing are:

1. Create a raw socket.

2. Set socket option.

3. Construct the packet.

4. Sending out packet through the raw socket.

**1 Creating a raw socket**

Most important step in doing spoofing is the creation of raw sockets which allow direct sending of packets bypassing all applications in operating system. We need raw sockets for injecting packets into the network to perform spoofing. Unix provides two types of sockets that provides direct access to the network. One is SOCK\_PACKET and other is SOCK\_RAW. SOCK\_PACKET works on device link layer and contains NIC specific header. However, we will be using SOCK\_RAW, which includes the IP headers and all subsequent protocol headers and data. We have to create a datagram socket using command: socket (PF\_INET, SOCK\_RAW, and IPPROTO\_UDP);

**2. Setting Socket option**

Even though Socket is interface to the IP header, it is transport layer specific. So, we have to create 3 separate raw sockets, using IPPROTO\_UDP, IPPROTO\_TCP and IPPROTO\_ICMP (the protocol numbers are 0 or 6 for TCP, 17 for UDP and 1 for ICMP).

**3. Constructing the packet**

To construct the packet for injecting, we must know the protocols that are required. There are defined structures of IP, ICMP, TCP and UDP headers. The data type sizes vary such as unsigned int - 4 bytes (32bits), unsigned short int - 2 bytes (16 bits), etc. UDP, TCP and IP headers are used to create packet that are later injected into network for spoofing.

**3. Sending out packet through raw socket**

Once we have created raw sockets and with our knowledge of protocol headers and basic programming skills, we have to create datagrams to inject into the network.

**Problem 7: 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?**

Solution: Generally, we are not allowed to receive raw packets unless we are a root. This restriction is imposed by the OSes to ensure security since raw packets can gain access to all other processes and users using that interface. For a non-root user Oses only allow us to set some fields in protocol headers (such as destination ip address, destination port, etc.) while it itself sets other fields (such as UDP, TCP and IP headers). For spoofing to happen we should be able to create raw sockets that is only possible for a root user who can set any arbitrary fields in packet headers and injecting the created packets in the networks. So, the program will fail since a non-root user can’t create raw sockets needed to inject raw packets into the network which is must for spoofing to happen.