CS528 Lab 1 Report

Mohammad Haseeb
mhaseeb@purdue.edu

Task 1: Writing a Packet Sniffing Program

Task 1.a: Understanding sniffex

Problem 1

This is the sequence of pcap library function calls that are essential for the sniffex program:

1. pcap_lookupdev(errbuf); // find a capture device
    - If the device name on which to sniff has not been passed as a command line argument, then this pcap library function will return a pointer to a string containing the name of the first device that that is suitable for listening on.
2. pcap_lookupnet(dev, &net, &mask, errbuf); // get network number and mask associated with capture device
    - This function gets the network address and the subnet mask of the device mentioned in "dev". It stores the values of the network address and the subnet mask in two separate bpf_u_int32 that we pass as argument.
3. pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf); // open capture device
    - Configures, opens, and returns a new 'handle' for the live capture session according to the arguments passed in the function. The current arguments will set the capturing to be done in promiscuous mode.
4. pcap_datalink(handle) // make sure that the capturing is being done on an Ethernet device
    - Determines and returns the link-layer type of the device on which we want to sniff e.g. Ethernet, Wi-Fi etc.
5. pcap_compile(handle, &fp, filter_exp, 0, net) // compile the filter expression
    - Compiles the filter expression mentioned in the filter_exp char array and sets its value to fp, which is the compiled filter program (expression)
6. pcap_setfilter(handle, &fp); // apply the compiled filter
    - Applies the compiled filter expression to the device handler. Only the packets that match the expression will now be sniffed, others will not be captured by pcap
7. pcap_loop(handle, num_packets, got_packet, NULL); // set the callback function for each new packet sniffed
    - Allows the user to pass a reference to a callback function that will be called every time a new packet is sniffed on the device. This will keep 'looping' until the number of packets sniffed reaches the num_packets argument. In this case, the name of the callback function is got_packet. The last argument is normally

set to NULL unless the user wants to pass in arguments to the callback function itself.

8. pcap_freecode(&fp); // for cleaning up
    - Frees up the memory from the compiled filter.
9. pcap_close(handle); // for cleaning up
    - Closes the 'handle' or the session for the packet capture on the device.

Problem 2

- If the program is run without passing the device name as a command line argument, it fails at the library call to **pcap_lookupdev()**
- If the device name on which to sniff packets is passed as a command line argument to the program, then it fails at the library call to **pcap_open_live()**

The root privilege is needed to run sniffex because the program directly accesses the low-level network interfaces (called a 'device' in this program) of the machine. This is normally done by the OS itself. And because sniffex runs in the user space, the OS automatically prohibits this due to security. For example, if root access were not required than any user program could access the network interfaces, set expression filters or change the firewall to allow malicious packets into the machine and thereby, cause many security issues.

Problem 3

The pcap_open_live() function has the following function prototype:

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)
```

Setting the `promisc` argument to 1 (True) turns the promiscuous mode on. This can be verified by setting the filter expression to "icmp" and then pinging a remote host e.g. google.com. The output of the program shows that it has captured the icmp packets although these were not directed to this machine.

Setting the `promisc` argument to 0 (False) turns the promiscuous mode off. This can be verified by keeping the filter expression to "icmp" and then again pinging a remote host e.g. google.com. In this case, the sniffer did not show any captured icmp packets. However, pinging the host that is running the sniffer program shows that it has captured the icmp packets. This verifies that the promiscuous mode is indeed off.

Task 1.b: Writing Filters

1. ICMP packets between two specific hosts
    - IP of sniffing machine: 192.168.15.4
    - Src host IP address: 192.168.15.5
    - Dst host IP address: 128.10.126.33 (mssn2.cs.purdue.edu)
    - Filter expression: `"(icmp) and ((src net 192.168.15.5) or (src net 128.10.126.33)) and ((dst net 192.168.15.5) or (dst net 128.10.126.33))"`

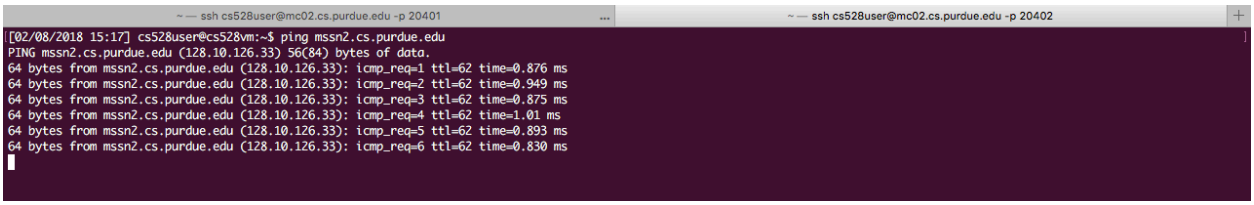

Figure 1

- Figure 1 shows the source machine 192.168.15.5 pinging the destination machine 128.10.126.33 (mssn2.cs.purdue.edu).

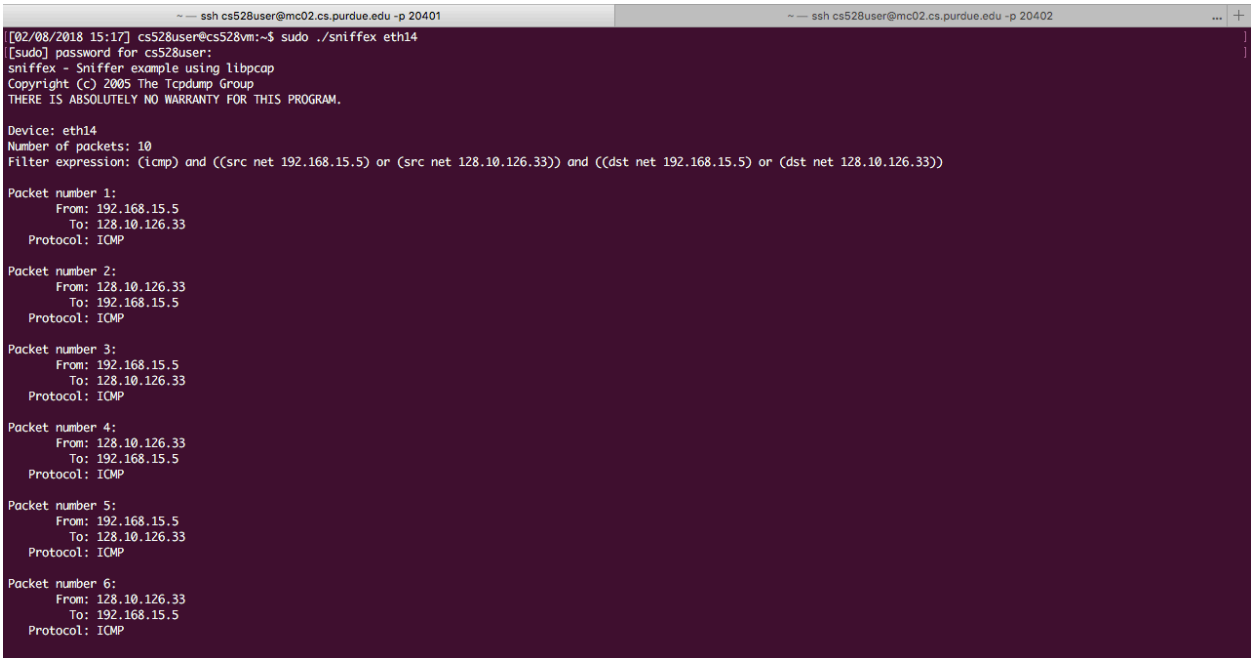

Figure 2

- Figure 2 shows the sniffing machine (192.168.15.4) sniffing the ICMP packets between the two hosts.

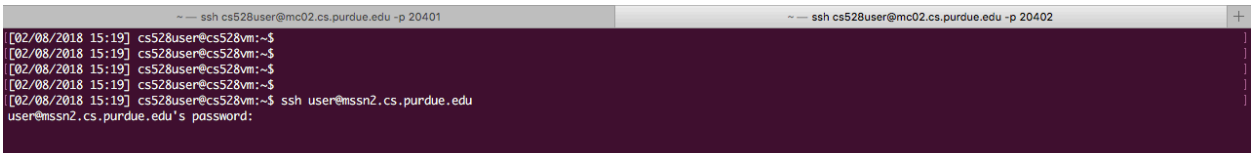

Figure 3

- Figure 3 shows the source machine now trying to SSH to the destination.

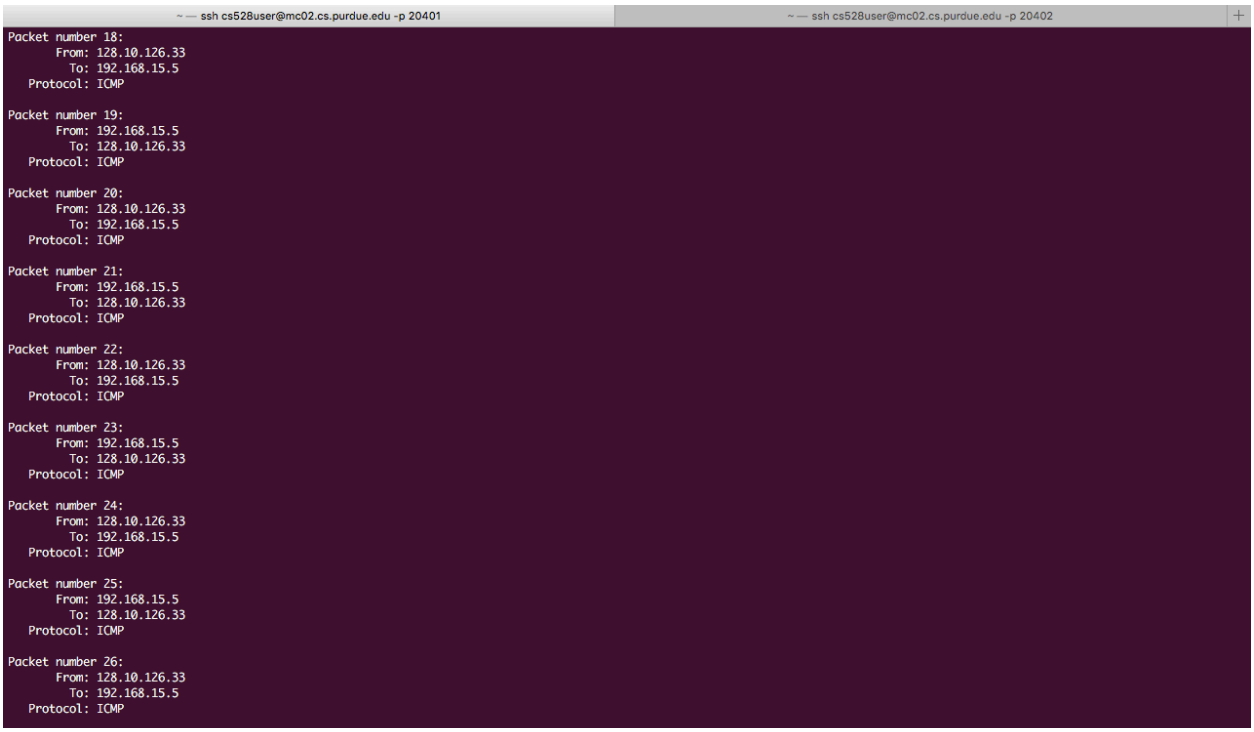

Figure 4

- Figure 4 shows that the sniffing machine did not capture the corresponding SSH packets because SSH uses TCP, not ICMP.

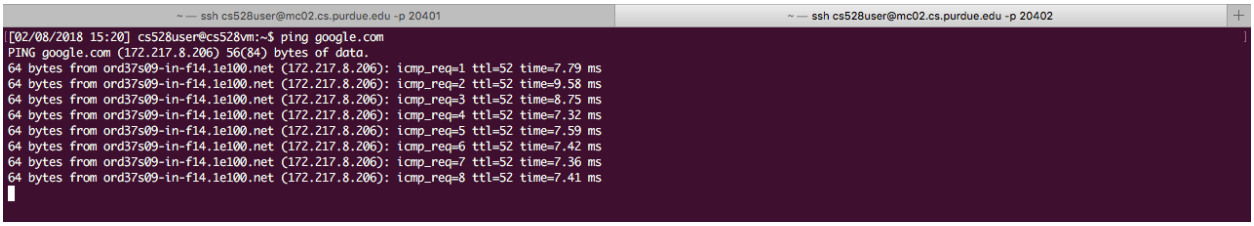

Figure 5

- Figure 5 shows the source machine now pinging some other machine other than the specified destination machine.

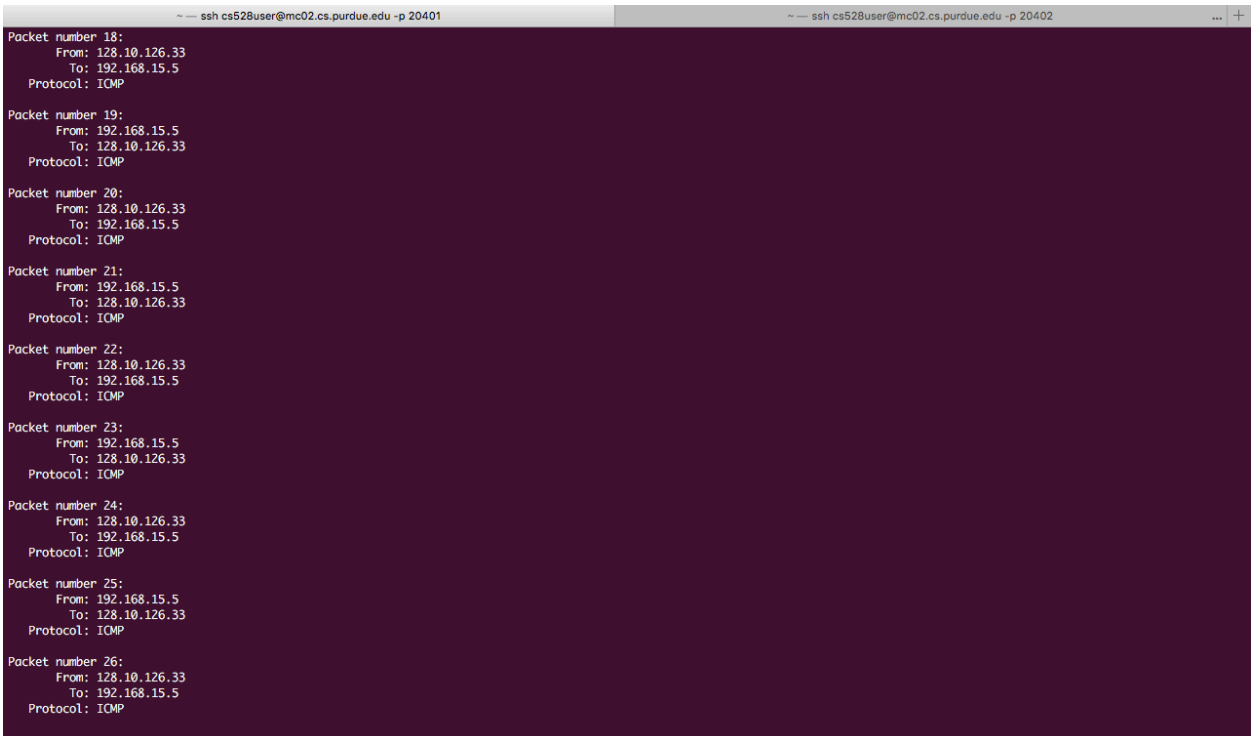

Figure 6

- Figure 6 shows that the sniffing machine again did not capture any ICMP packets because the destination machine is different from the one specified.

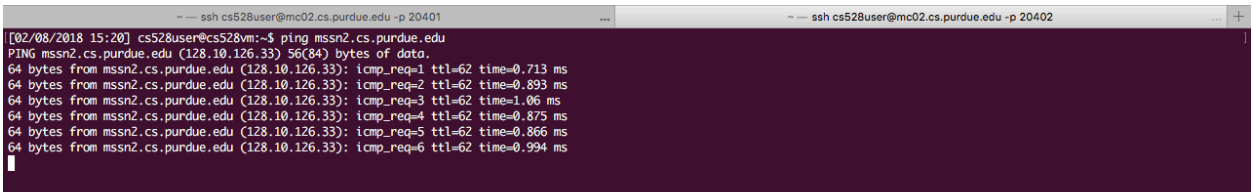

Figure 7

- Figure 7 shows the source machine now pinging the specified destination machine again.

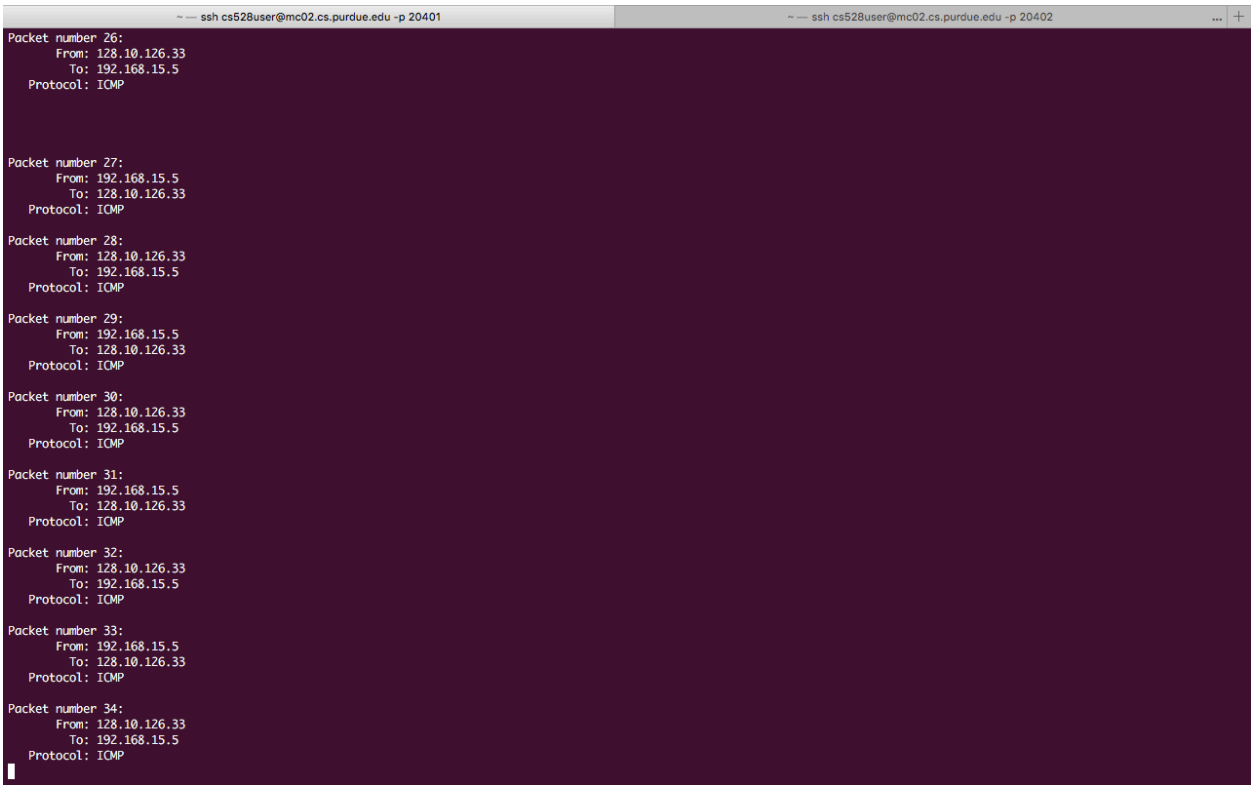

Figure 8

- Figure 8 shows the sniffing machine now capturing the relevant ICMP packets.

2. TCP packets that have a destination port range 50-100
   - IP of sniffing machine: 192.168.15.4
   - Src host IP address: 192.168.15.5
   - Dst host IP address: 128.10.126.33 (mssn2.cs.purdue.edu)
   - Filter expression: "`(tcp) and (dst portrange 50-100)`"

Figure 9

- Figure 9 shows the source machine trying to SSH the destination machine at port 49.

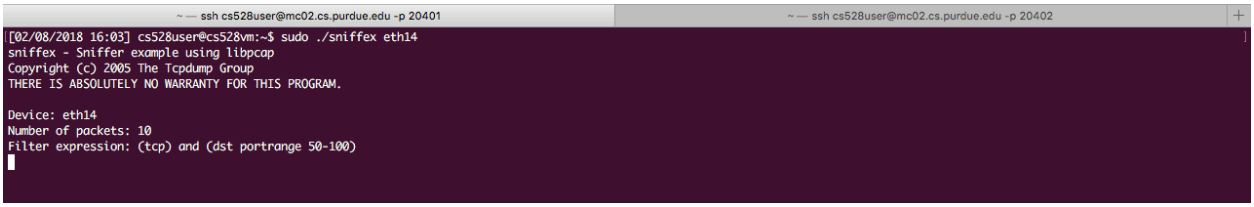

Figure 10

- Figure 10 shows the sniffer machine not capturing the SSH packets because they are on port 49 (out of range) although using TCP.

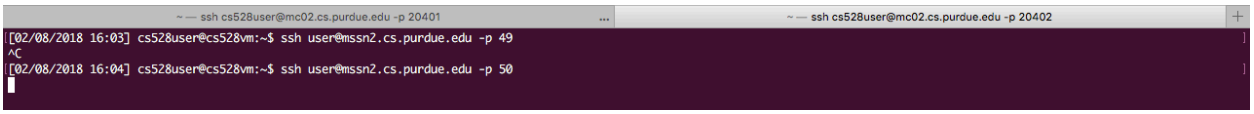

Figure 11

- Figure 11 shows the source machine now trying to SSH on port 50.

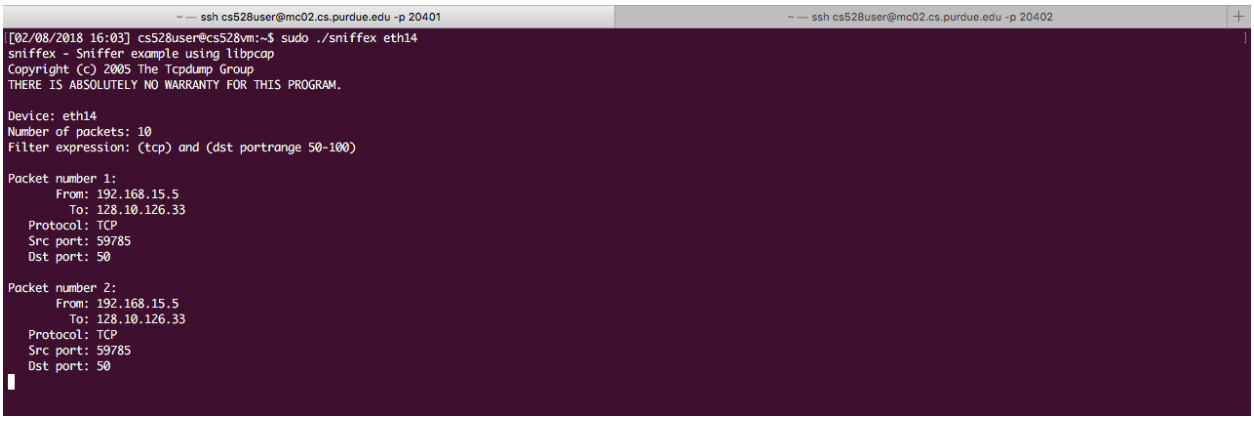

Figure 12

- Figure 12 shows the sniffer machine now capturing the corresponding packets because they are using TCP and within the specified port range.
- Figures 13 - 16 show the same behavior.

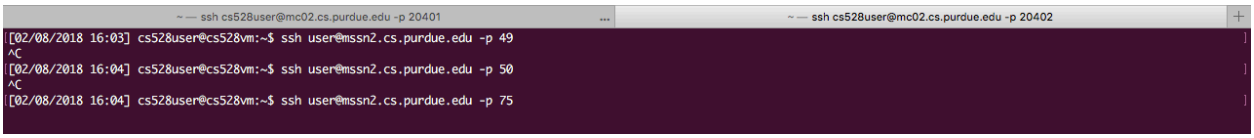

Figure 13

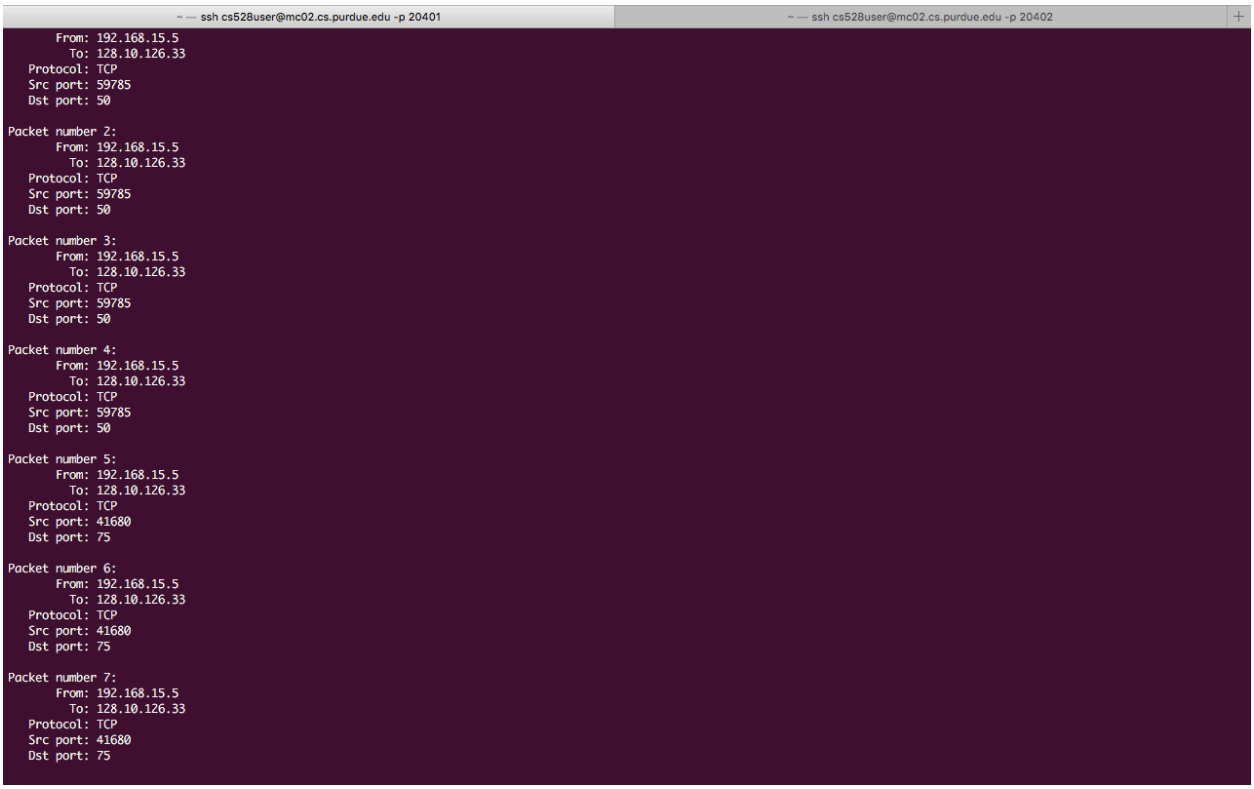

Figure 14

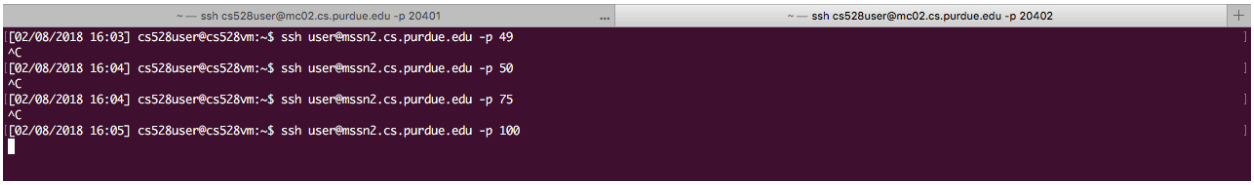

Figure 15

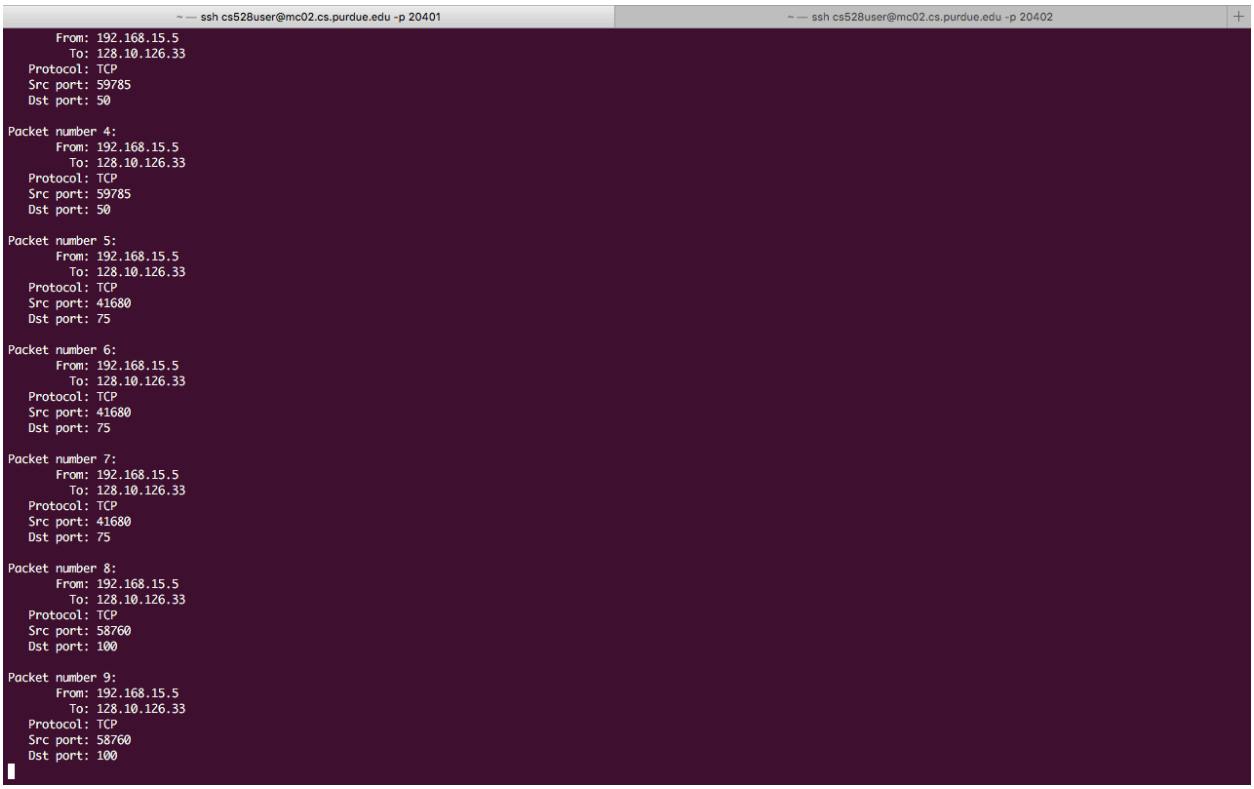

Figure 16

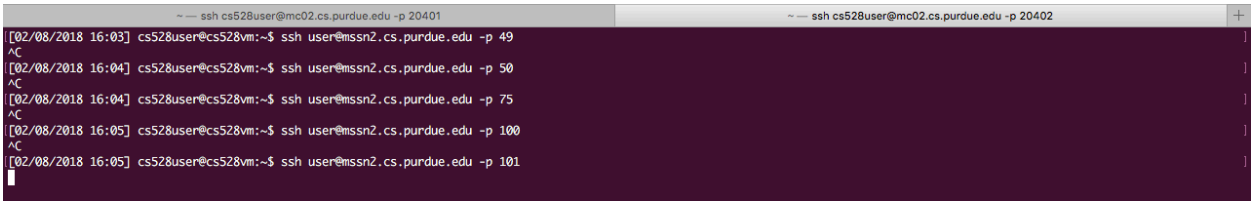

Figure 17

- Figure 17 shows the source machine trying to SSH the destination machine at port 101.

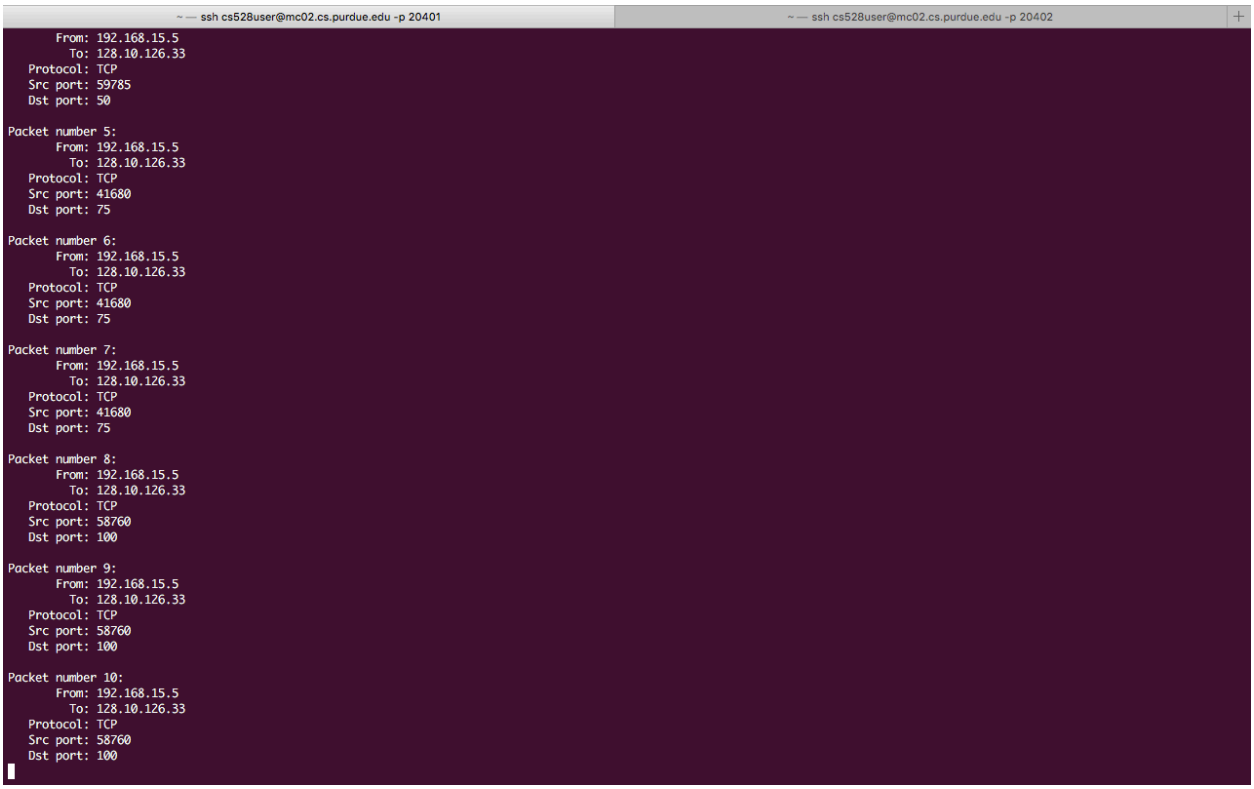

Figure 18

- Figure 18 shows the sniffer machine not capturing the SSH packets because they are on port 101 (out of range) although using TCP.

Task 1.c: Sniffing Passwords

- Telnetd server running on machine with IP 192.168.15.4
- Telnet client IP: 192.168.15.5
- Filter expression: "`port 23`" – because Telnet uses port 23.

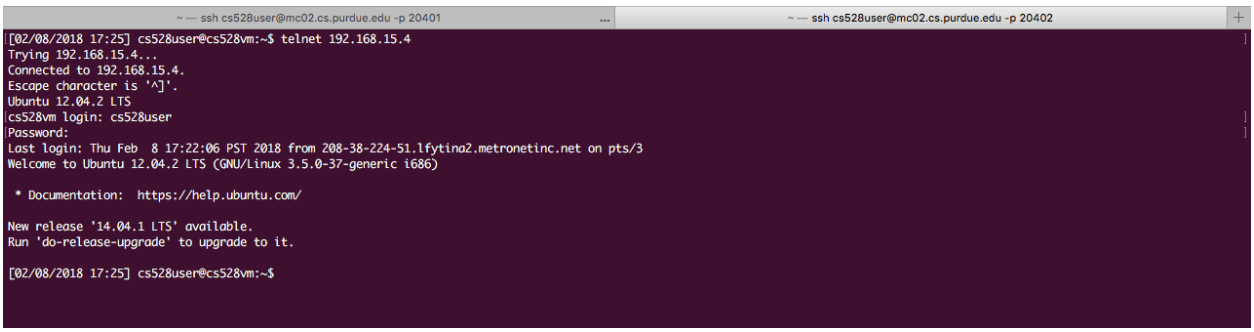

Figure 19

- Figure 19 shows the Telnet client connecting to the Telnet server. The username and password has been entered and the login has been successful.

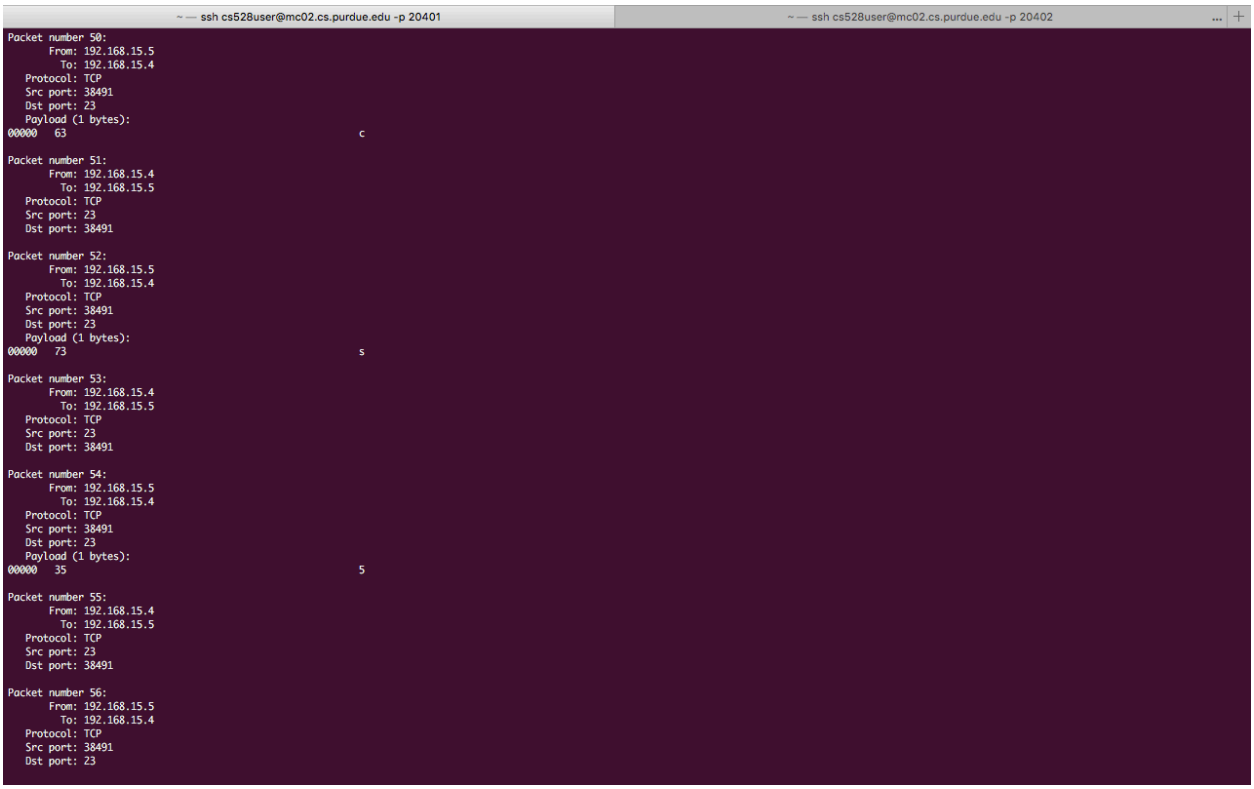

Figure 20

- Figures 20 – 22 show the password of the user as it is typed character-by-character. Note that Telnet sends the username and password as a character-by-character, not as one string. We can get the whole password as a string using Wireshark's "Follow TCP Stream" feature.

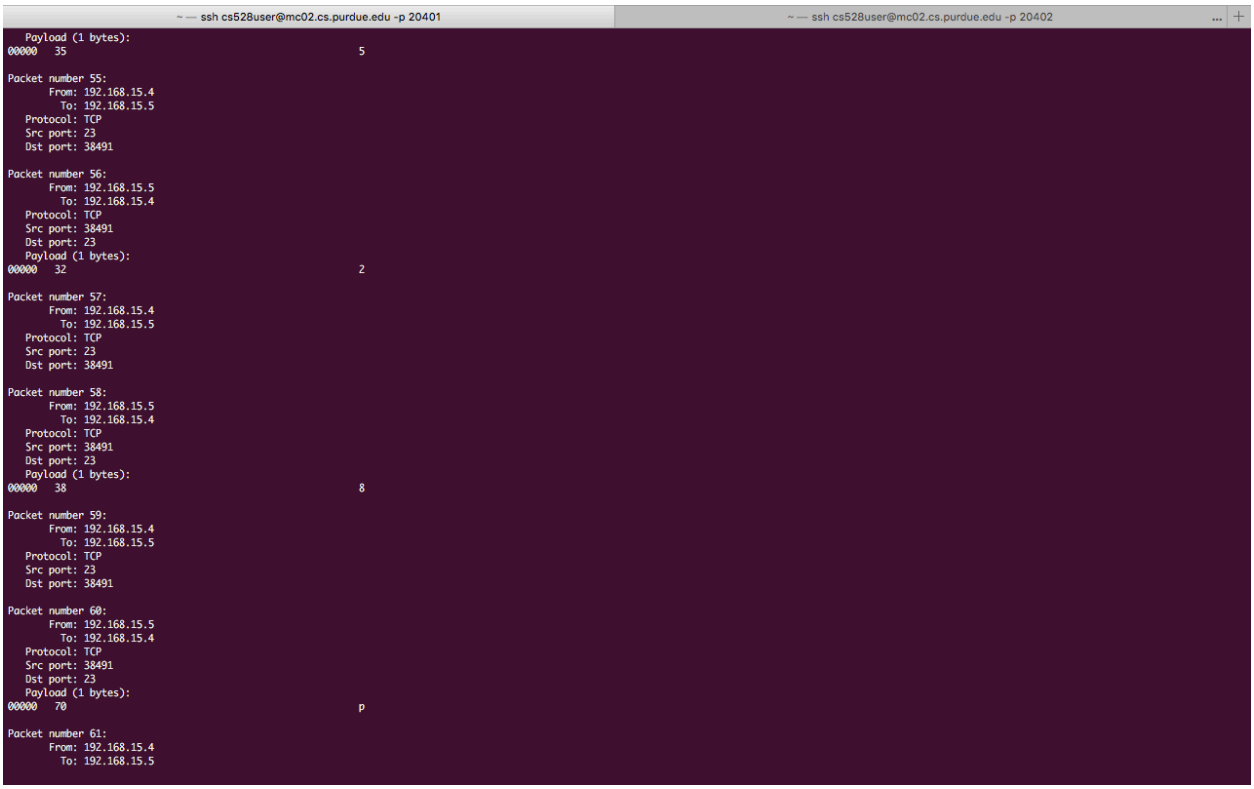

Figure 21

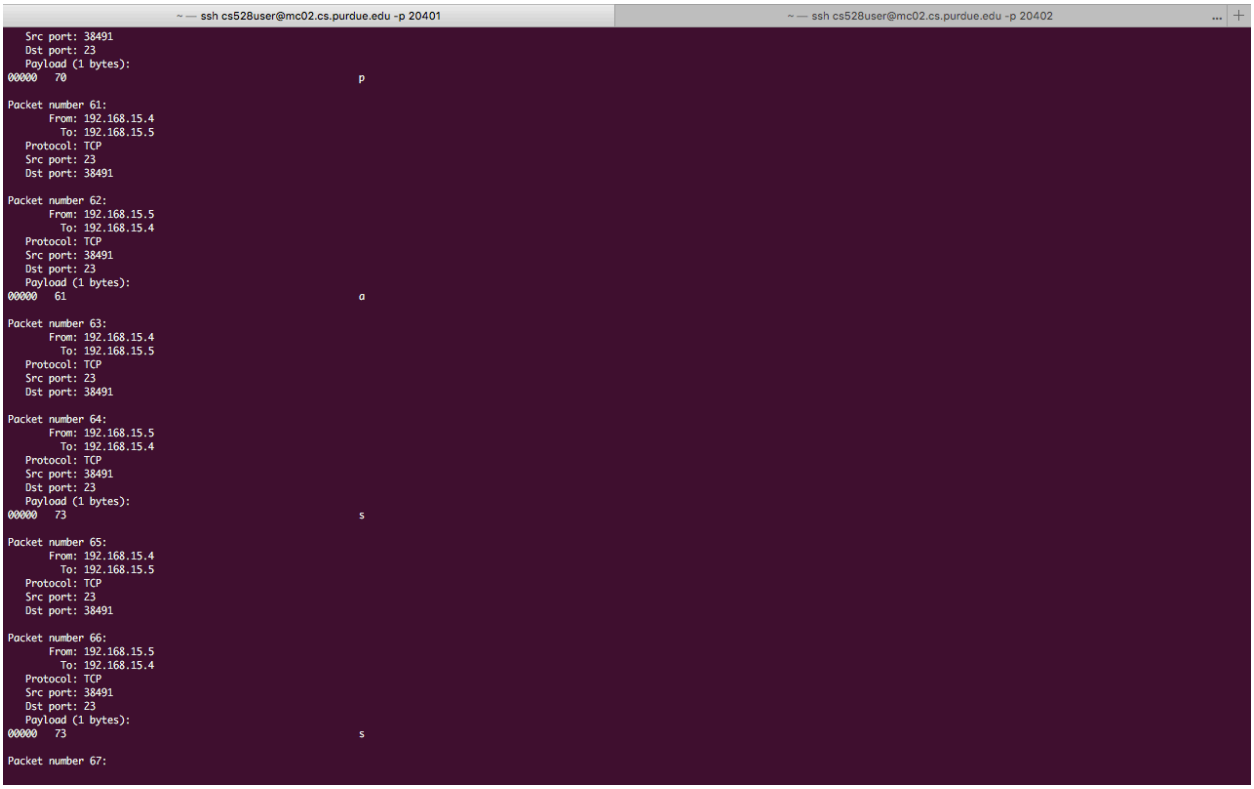

Figure 22

Task 2: Spoofing

Task 2.a: Write a spoofing program

The program is named `task2_spoofer.c`
I have done both parts b and c in this program. You can choose to run the ICMP spoofer by passing 1 as a command line argument. You can choose to run the Ethernet spoofer by passing 2 as a command line argument. Both options will call separate functions.

Task 2.b: Spoof an ICMP Echo Request

- Spoofing machine IP: 192.168.15.4
- Victim machine IP (spoofed IP): 192.168.15.92
- Wireshark being run on machine with IP: 192.168.15.5
- Remote machine IP: 172.217.8.174

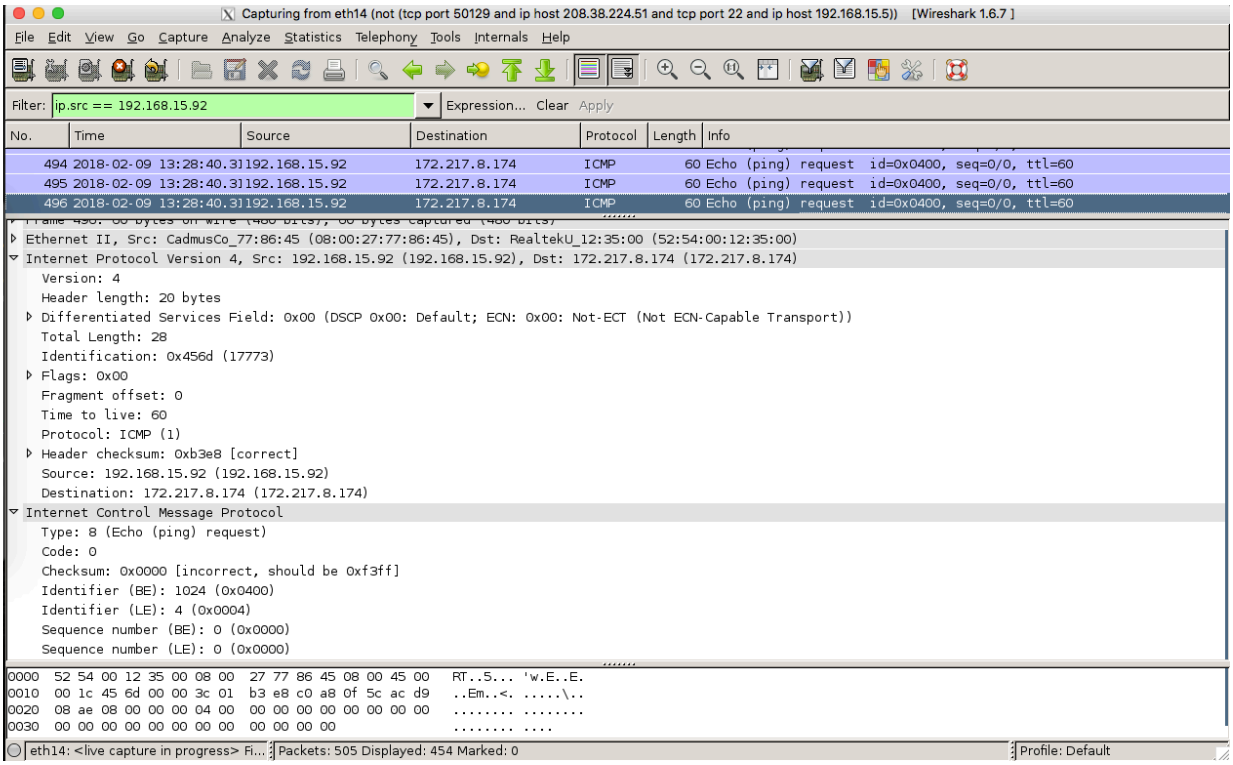

Figure 23

- Figure 23 shows a spoofed ICMP Echo packet. The spoofed IP is 192.168.15.92. This packet was originally sent from the machine 192.168.15.4.

Task 2.c: Spoof an Ethernet Frame

- Spoofing machine IP: 192.168.15.4

- Victim machine MAC address (spoofed MAC address): 01:02:03:04:05:06
- Wireshark being run on machine with IP: 192.168.15.5
- The Ethernet frame is still carrying an ICMP packet within its IP payload

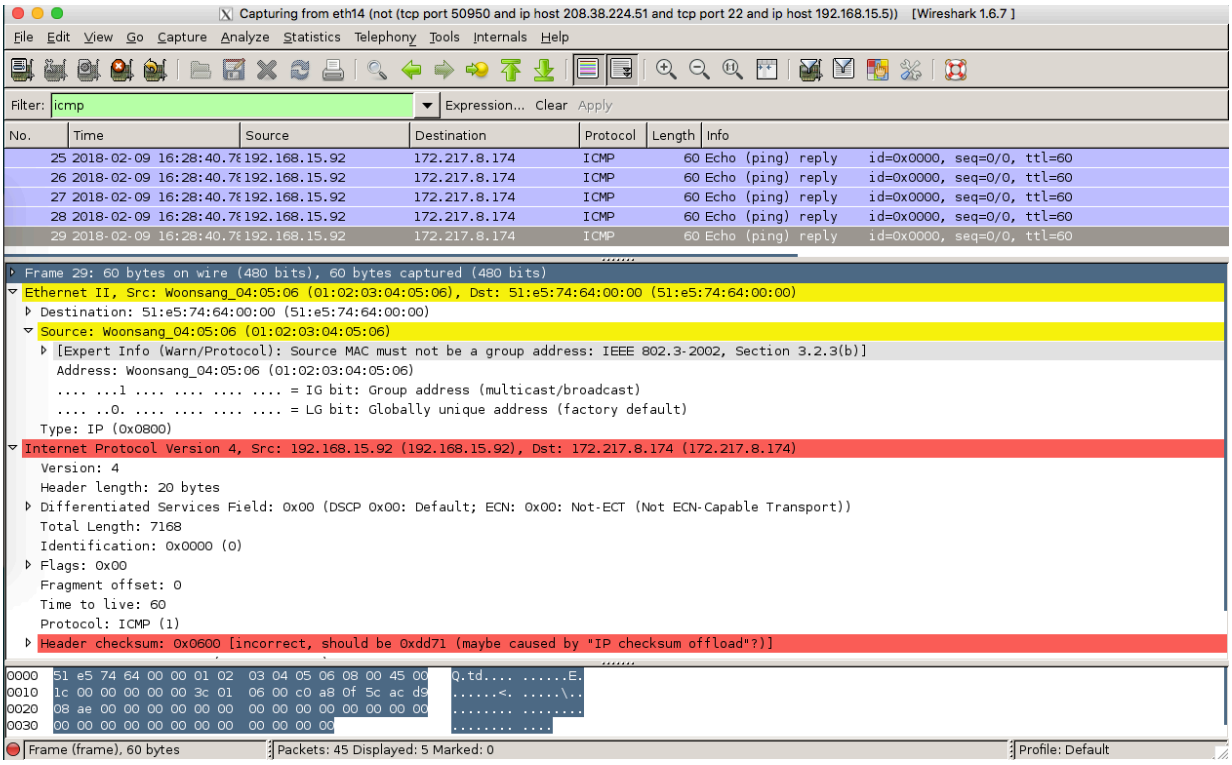

Figure 24

- Figure 24 shows the spoofed Ethernet frame. We have spoofed the source address of the frame and set it to 01:02:03:04:05:06. The frame is carrying an IP payload which itself is carrying an ICMP payload

Question 4

Yes, the IP packet length field can be set to any arbitrary value, regardless of the actual packet size. I tested this by setting the `ip->tot_len` property of the IP header to `1044` and the packet still went through without a problem.

Question 5

No, setting a bogus checksum value for the checksum field or not setting it at all still gets the packet through. It seems like the Linux raw socket library's code handles the case of incorrect or missing checksum and fills it up accordingly. [Reference: https://stackoverflow.com/questions/10932408/ip-checksum-changes-automatically]

Question 6

The program fails at the point we try to create a raw socket. The program needs root privilege to run because raw sockets allow us to build and specify the fields of a packet in whatever

way we want. This can lead to many security concerns like spoofing. The OS wants to ensure that not any user program can do this.

Question 7

1. socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
   - Creates the raw socket with IP protocol. Indicates to the system that the IP header will be provided by the programmer, so the OS need add its own header.
2. sin.sin_family = AF_INET;
   - Not a library call but this field needs to be set in order to send packets out a socket.
3. inet_addr(ip)
   - Needed to convert an IP address represented in string format to internet address format needed by the `iphdr` struct that defines the header of an IP packet
4. sendto(sd, buffer, ip_header->tot_len, 0, (struct sockaddr *)&sin, sizeof(sin))
   - Sends out `ip_header->tot_len` bytes from the `buffer` out the socket.

Task 3: Sniff and then Spoof

- Machine A IP: 192.168.15.5
- Machine B IP: 192.168.15.4
- Machine X IP: depends on what IP is pinged from machine A

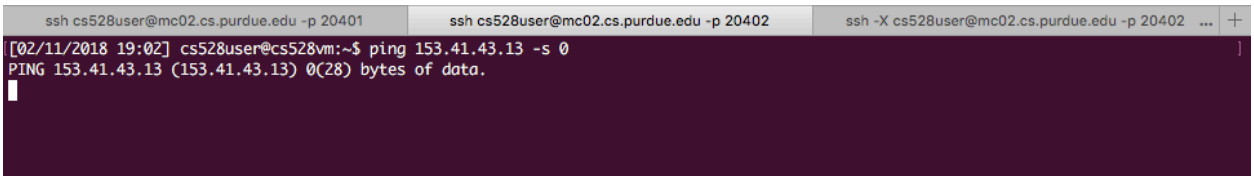

Figure 25

- Figure 25 shows machine B pinging an unreachable host and not running any reply. Our sniff-and-spoof program is not running right now.

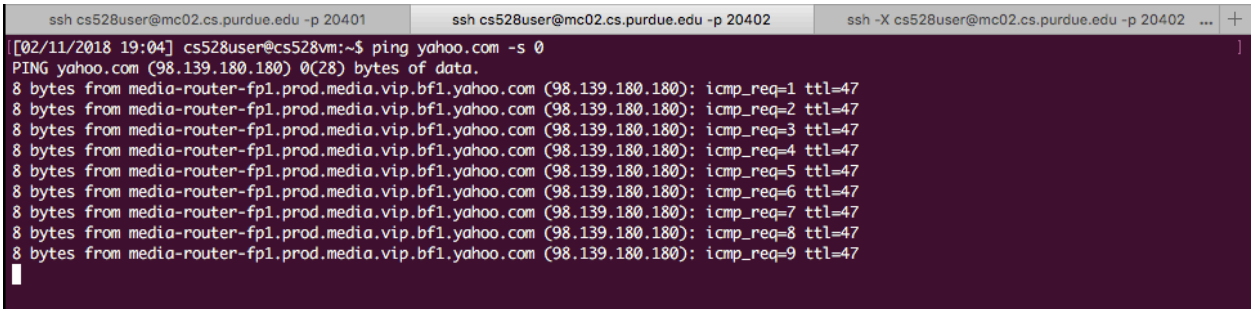

Figure 26

- Figure 26 shows machine B pinging a reachable host. Our sniff-and-spoof program is not running right now.

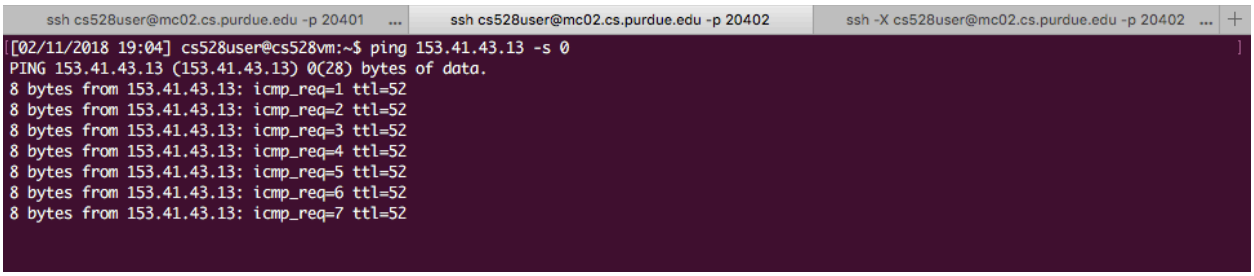

Figure 27

- Figure 27 shows machine B receiving a ping reply from the unreachable machine. Our sniff-and-spoof program is running now.

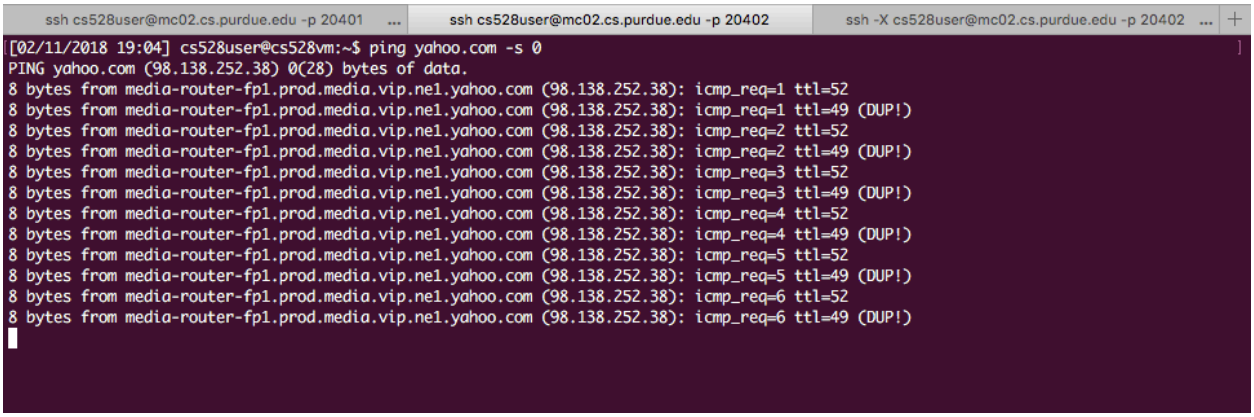

Figure 28

- Figure 28 shows machine B receiving two ping replies. One is from our sniff-and-spoof program and the other is the actual reply from the reachable host.