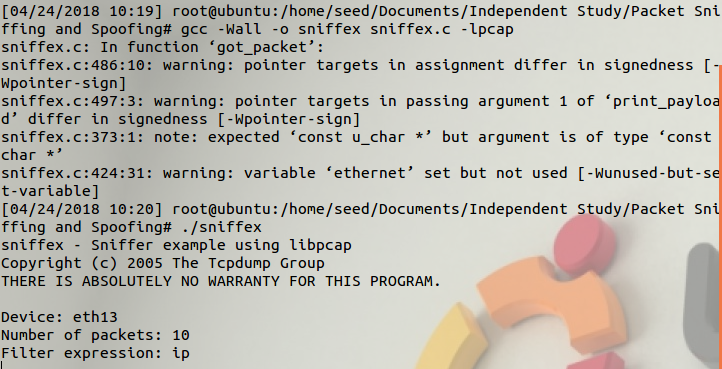
Daniel Oliveros

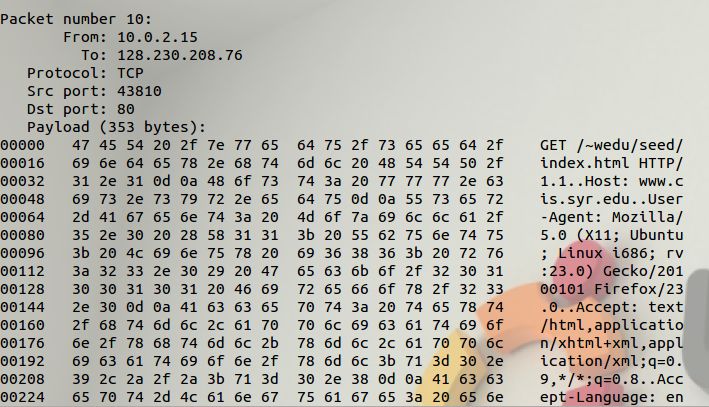
Garrett Bogart

Spring 2018 – Independent Study

Package Sniffing and Spoofing Lab

**Task 1.a:**

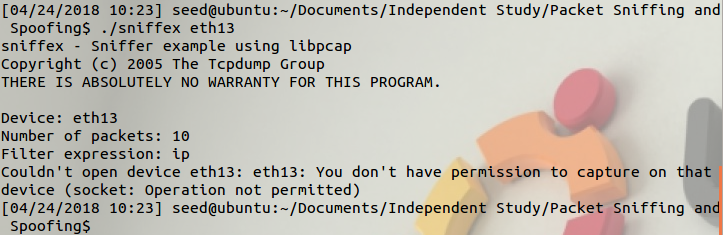
****

****

**Problem 1**

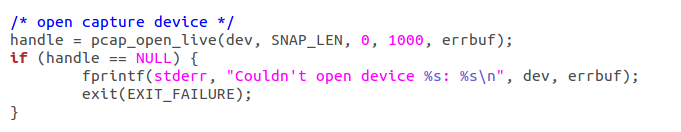
1. **Understanding Sniffex:**
   1. Open device for sniffing
      1. Handle = pcap\_t \*pcap\_open\_live(char \*device, int snaplen, int promisc, int to\_ms, char \*ebuf)
         1. Device name
         2. Max number of bytes
         3. Promiscuous mode status: 0 off, 1 on
         4. How long it should wait for packets to arrive
         5. String to store error messages
   2. Filtering traffic
      1. Compile filter
         1. int pcap\_compile(pcap\_t \*p, struct bpf\_program \*fp, char \*str, int optimize, bpf\_u\_int32 netmask)
            1. Pointer to our handle from open sniffing function
            2. Pointer to the place the compiled filter will be
            3. Filter expression
            4. Optimization flag
            5. Network mask of the network the filter will be applied to
         2. int pcap\_setfilter(pcap\_t \*p, struct bpf\_program \*fp)
            1. Pointer to our handle from open sniffing function
            2. Pointer to the location of our compiled filter
   3. Actual sniffing
      1. Single packet sniffing
         1. u\_char \*pcap\_next(pcap\_t \*p, struct pcap\_pkthdr \*h)
            1. Pointer to our handle from open sniffing function
            2. Pointer to a structure that holds our general information about our packet
      2. Looped packet sniffing
         1. int pcap\_loop(pcap\_t \*p, int cnt, pcap\_handler callback, u\_char \*user)
            1. Pointer to our handle from open sniffing function
            2. How many packets are going to be sniffed before returning
            3. Callback function name
            4. Additional callback function for use by user, generally NULL

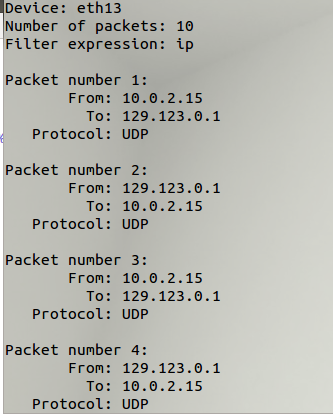
**Problem 2**

****

We lack the permission to capture on the device if we run the program without root permission.

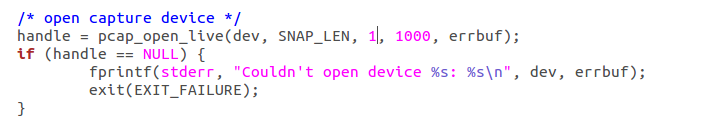
**Problem 3**

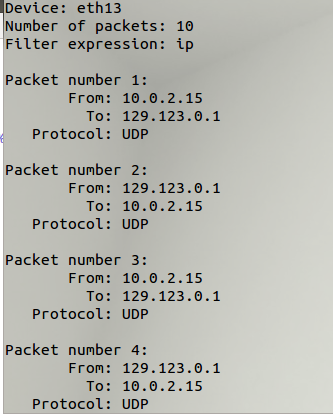
****

****

While promiscuous mode is off no packets are found. The only communication that exist is between the device and the DHCP server. This communication happens so that if your computer no longer needs its ip address the server can reassign your ip address to some other computer. That is why the From and To sections are always the same, one is your ip address the other is the DHCP server.

In theory when promiscuous mode is turned on the virtual machine will get information from your host computer. The virtual machine won’t get information from the outside network because it is a subnetwork of your host machine. Depending on how the virtual machine is set up an outside connection may not be possible. In our case the virtual machine wasn’t able to listen to the host machine so the only packets that we got were between the virtual machine and the server.

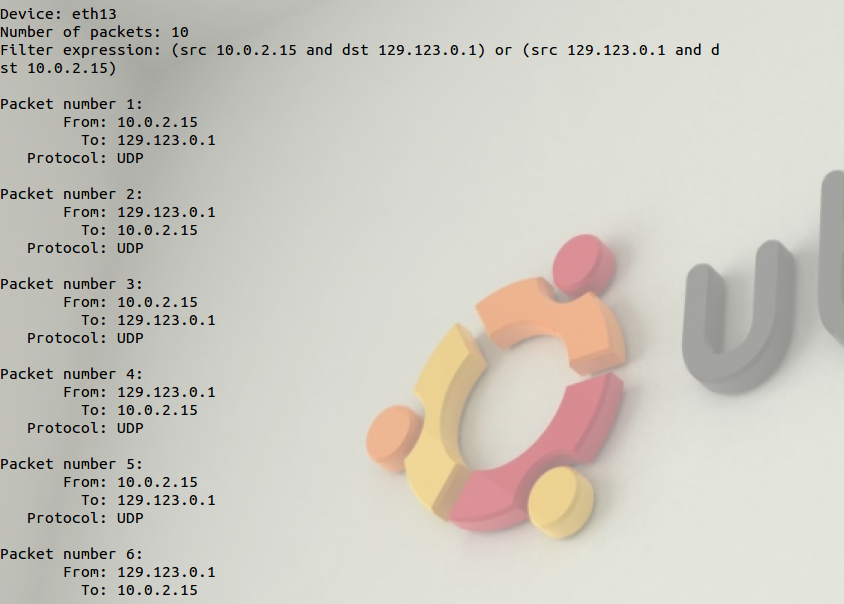




**Task 1.b:**

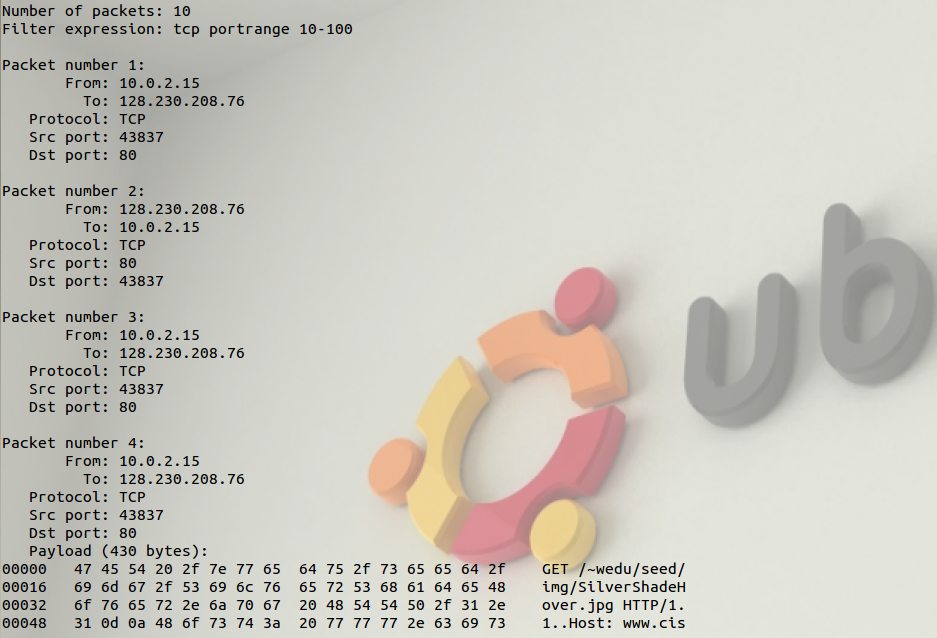
**Capture the ICMP packets**

****

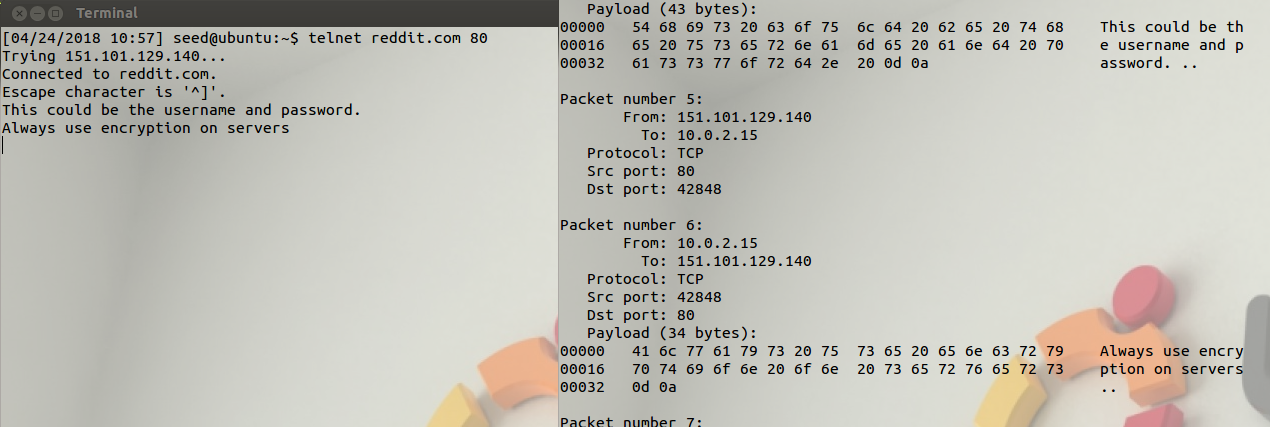
****

**Capture TCP packets from ports 10-100**

****

****

**Task 1.c:**

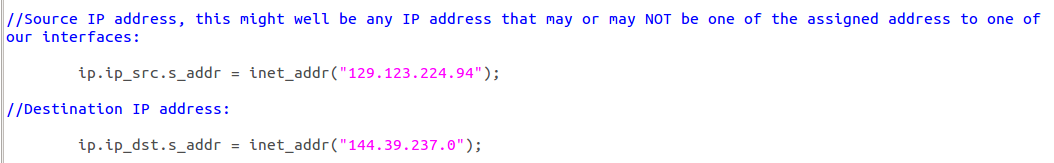
****

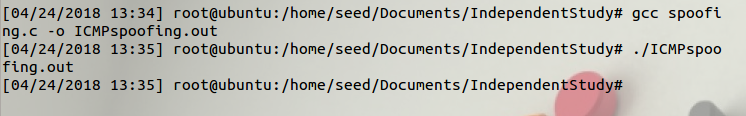
The text is sent as plain text. A sniffing program can pick this information up. If we had sent a username and password both of those could be seen and, we could use the data to gain access to the account.

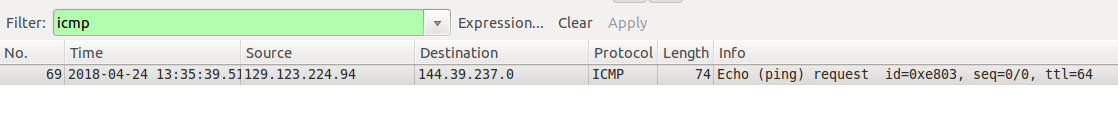
**Task 2:**

**Task 2a and 2b**

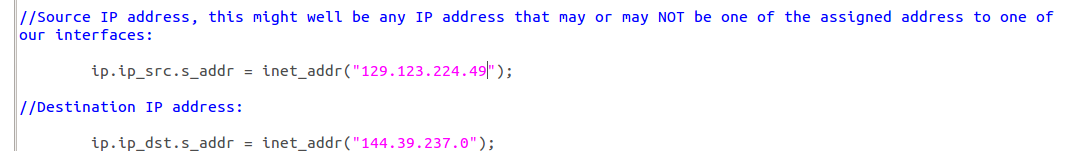
We used a spoofing program we found at <http://www.enderunix.org/docs/en/rawipspoof/>. We used our ip address, 129.123.224.94, as the source and the destination address of 144.39.237.0.

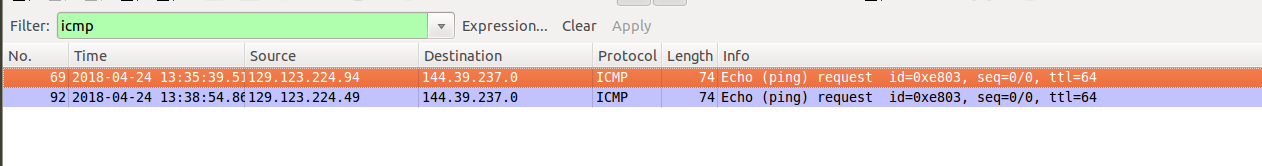






We successfully sent a packet with the listed information. Testing it again with a new sender ip address



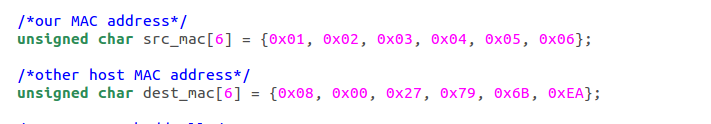


We can see that the source address has been changed.

**Task 2.c**

For this task we found an ethernet spoofing program <http://aschauf.landshut.org/fh/linux/udp_vs_raw/ch01s03.html> unforttunently the server is no longer operational but, we will include a copy of the file.

Changing the mac addresses

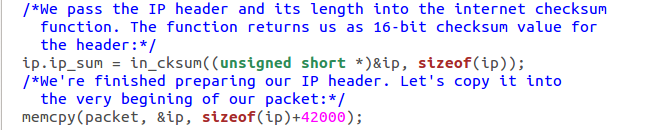


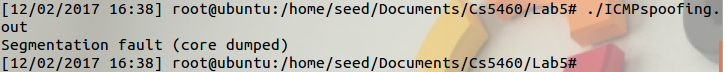
After running the program we can see that the packet was sent



**Question 4**

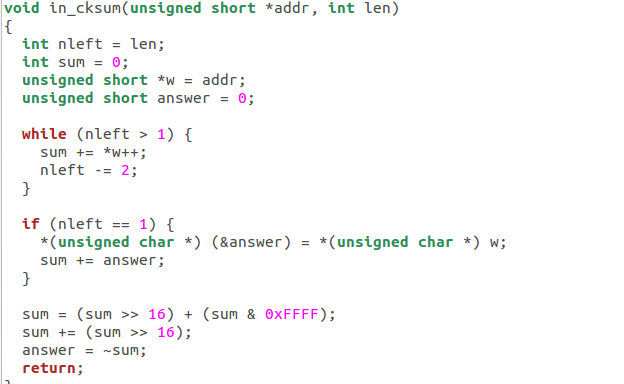
When the value is small enough, we can give the ip header an arbitrary length. A large enough ip header causes a segmentation fault to occur, this happened when we increased the header size to over 40000



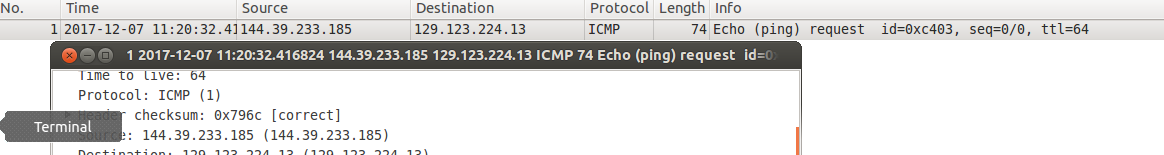


**Question 5**

To test this question we changed the checksum function to void and had it return nothing.

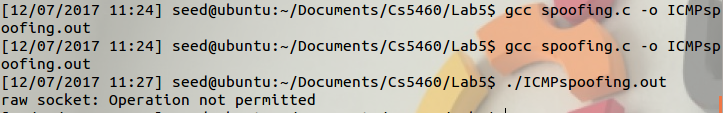
****

After running the spoofing program we checked on the checksum value. Surprisingly, it had a valid checksum value, meaning that the system will automatically calculate a checksum value if none is available



**Question 6**

Running the program as a normal user.



As a normal user we no longer have access to raw sockets.

**Issues:**

1. This lab takes a huge dive into low level network code, even after doing this lab for a second time, we didn’t feel prepared to write these programs. From the description of the lab itself, it seems completely okay with people using code from other sources online, and we think this is really the only way many people will even get through this assignment
2. For some reason, we still couldn’t get promiscuous mode to work, and honestly, it doesn’t feel completely needed when you can easily recreate the whole process with a single VM
3. Personally, we still find this our least favorite lab. There is some useful information in it, but without enough of a networking background, it really doesn’t feel as fulfilling to complete it as other labs do.

**Ideas for Improvement:**

1. Giving out programs for sniffing and spoofing would be extremely helpful, as students would get a lot out of learning the concepts behind these programs as opposed to writing the code from scratch