Austin Peterson
UIN: 926006358

Homework 1: Packet Sniffing and Spoofing

Sniffing:

The packet sniffing and spoofing assignment was a great way to get hands on experience manipulating network packets. This assignment started off with packet sniffing, which for us involved capturing packets on a given network, dissecting the packets and manipulating data to display information about each captured packet, like the way Wireshark operates. Packet sniffing comes down to a handful of essential steps and commands contained in the pcap library. The first essential step is establishing the device that is to be used to sniff, which can be found by either manually entering the device name or by using the call pcap_lookup(char* errbuff). The next step is to initialize the sniffing session using the call pcap_openlive(...), which takes arguments for the sniff device, length, promiscuity, timeout and error buffer. Next, if you don't want to sniff every packet on the network, you have to create a filter using pcap_compile(...) which creates a compiled bpf_program filter based on a string for the filter expression, and pcap_setfilter() which simply enforces the filter. After that, you can use functions like pcap_next, pcap_loop and pcap_dispatch to process each packet, which can be done using callback functions.

Sniffing network traffic involves using raw socket programming. You need root privileges to run packet sniffing programs because you're using the raw sockets to intercept traffic. If you don't have root privilege, then you fail when you try to call the pcap_openlive() command to start the sniffing session. Another important aspect to packet sniffing is promiscuous capturing, which is enabled or disabled using one of the arguments in pcap_open(). Promiscuous mode will determine whether you capture packets just going to your device (prom=0) or if you sniff all traffic on the network (prom=1). With promiscuous mode enabled, the packet sniffer picks up packets that are being send not only to the vm hosting the sniffer but also any other machines (vms in this case) that are connected to the network. Of course, promiscuous capturing can of course result in many captured packets, especially on larger network. To combat this, pcap has functions to selectively choose what kinds of packets you want to capture using filters.

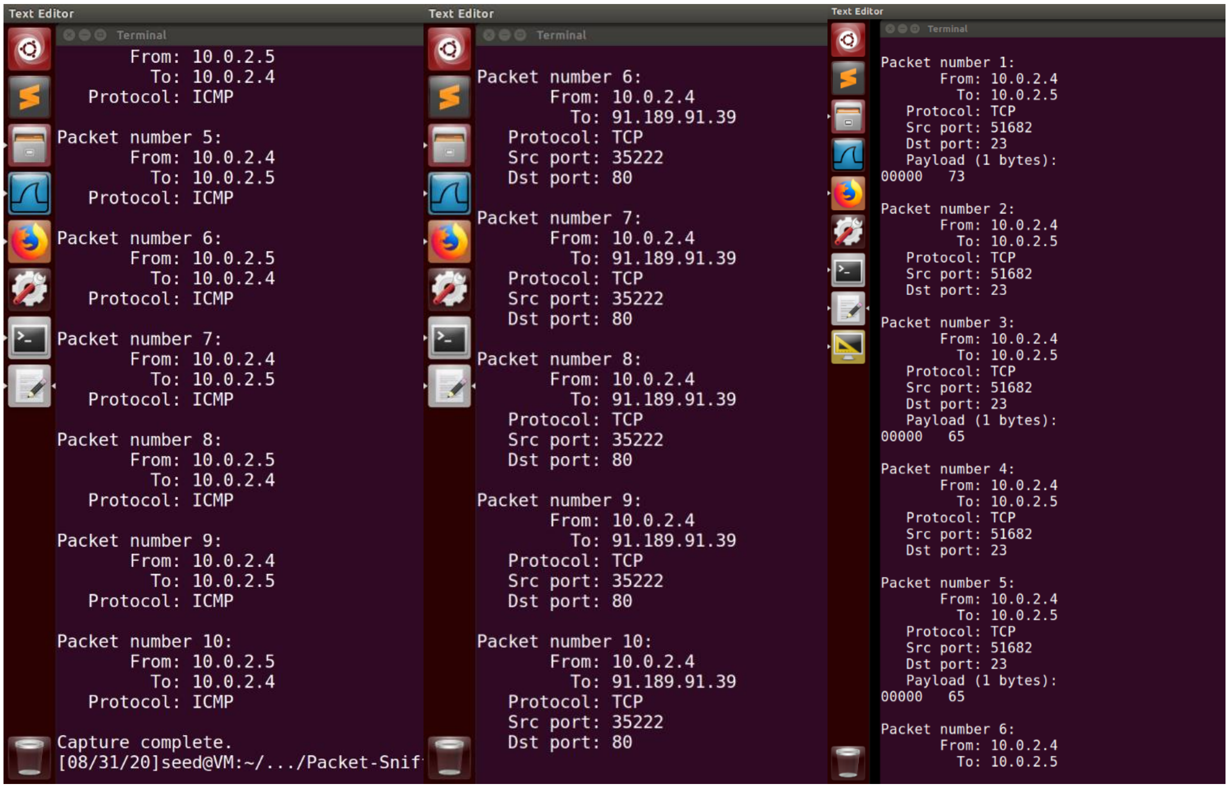
Part of manipulating packet sniffer programs is enabling specific filters in order to narrow down the kinds of packets that you can view. As an example of this, two primary filters were written and used during the packet capturing process of this assignment and are listed as follows:

ICMP filter expression: char filter_exp[]= "((icmp) and ((dst host 10.0.2.5) and (src host 10.0.2.4)) or ((dst host 10.0.2.4) and (src host 10.0.2.5)))";*

TCP port filter expression: char filter_exp[] = "(dst portrange 10-100)";*

The first filter captures icmp packets, such as echo requests/pings, between two specific hosts, in this case the local NAT addresses of the virtual machines used to test the programs. The latter is a more basic filter used to capture packets on a specific range of ports, in this case 10-100. Filters like these can be used on specific ports as well to capture http packets on port 80 or RDP packets on port 3389. Additionally, port filters can be used to capture traffic on reserved ports such as port 23, which is reserved for telnet connections. Analyzing these packets can be used to capture telnet passwords. Screenshots of these filters in action can be seen below. From left to right, you can see ICMP packets captured between two hosts, incoming packets captured on ports 10-100, and a telnet password capture. On the right, you can see the payload data contains the Hex values for “d e s”, the password for the target telnet server.

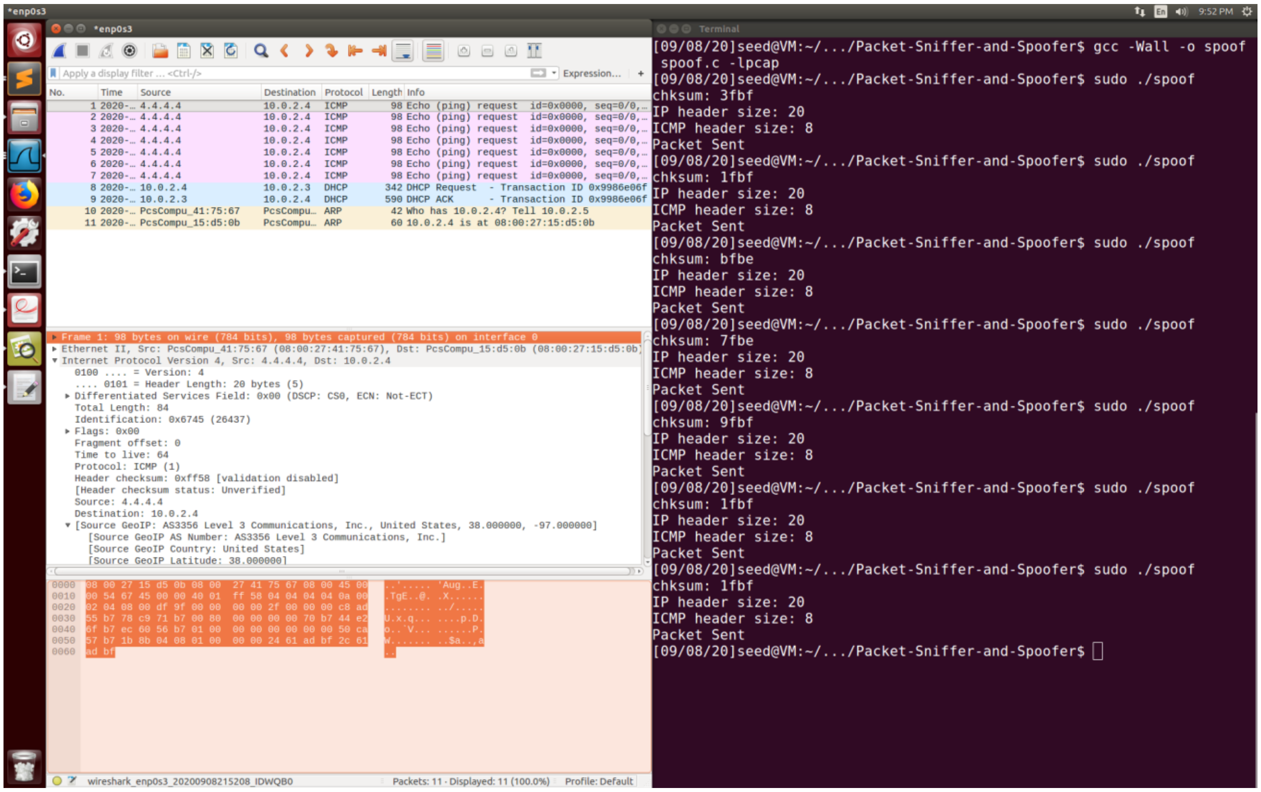
Austin Peterson
UIN: 926006358



Spoofing:

Spoofing packets is a matter of assembling a collection of headers, copying them into a buffer and sending the packet from a created and assigned socket. Usually you can create a full packet with custom made ethernet, ip, UDP/TCP, ICMP etc headers that are chosen based on the purpose that you want them to fulfill. For this assignment we focused on simple ICMP ECHO requests and replies, also known as “pings”, which did not really require hand made ethernet headers. The first two steps are sending a spoofed packet and spoofing a ping request from a different machine, which can be done at once. Below you can see spoofed packets being sent from a fake source IP address, 4.4.4.4, to a test virtual machine whos IP is 10.0.2.4. Spoofing packets from fake addresses is a matter of manually setting the ip_src and ip_dst fields in the IP header. Screenshots can be found below.

Austin Peterson
UIN: 926006358



As you can see above, the spoofer sends out fake echo requests to the victim virtual machine at IP 10.0.2.4 from a fake ip address 4.4.4.4. Unfortunately, the target machine does not respond with an echo reply, an issue that will be discussed more later.

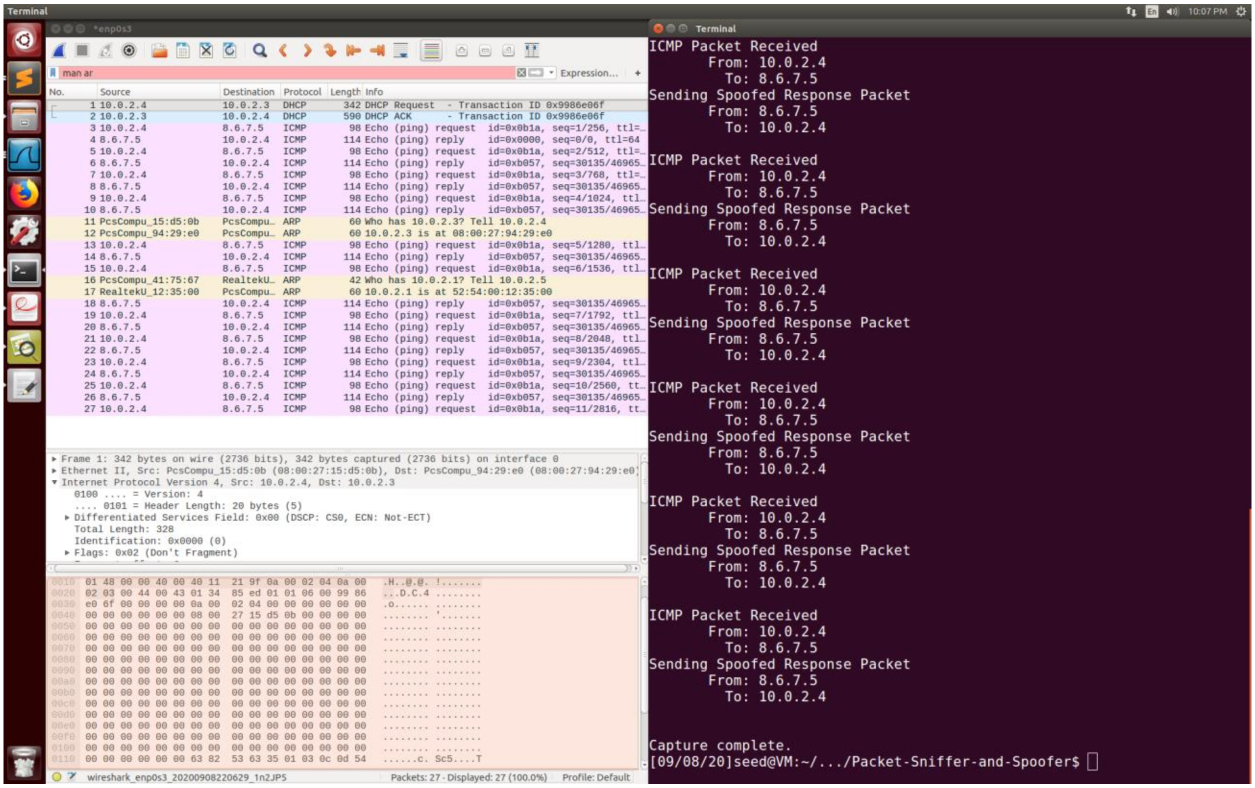
Along the way there were also several questions to consider while developing the packet spoofer program. The IP packet length field didn't seem to have much of an effect when sending these packets. The fields ip_v and ip_hl however did have a substantial effect on the processing of the packets during testing. While developing these spoofed packets one of the biggest bugs was getting the packet to properly show up on Wireshark as opposed to "Bogus IP version 3 (or 0), should be 4" as well as "Bogus header length". The two fields as well as something else I'm sure were vital in fixing this bug.

Sniffing and Spoofing:

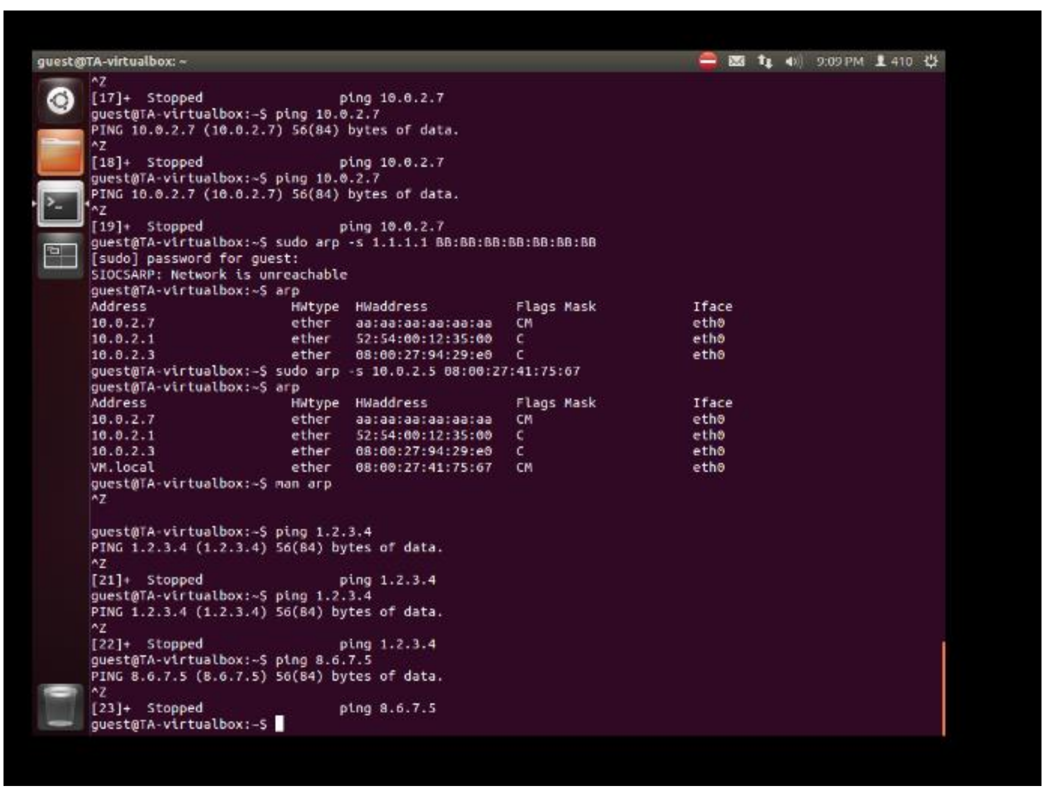
The logical next step is combining the first two sections of this assignment to intercept and respond to echo requests destined for any IP address. This is a matter of capturing ICMP packets from a given ip address, pulling the source and destination addresses from the intercepted packets and building a spoofed ping response packet to send back to the machine. What this means is that we can make any ip address look like a connection, alive machine even if its not. There isn't much that's unique to this part as it's mostly just the same functions and concepts that were used to sniff and spoof put together. As you can see from demonstration screenshots below, machine 10.0.2.4 repeatedly pings machine 10.0.2.7 (which does not exist) and the program sends back responses to give the illusion that the dummy machine is active. The only required step unique to this part of the assignment was adding a fake mac and IP to the arp cache of the victim machine, which was done using sudo arp -s <ip> <mac>. Below you can see demonstration screenshots of the sniff and spoof process:

Austin Peterson
UIN: 926006358

Program Host:



Victim Machine:



Austin Peterson
UIN: 926006358

As you can see above, when ping requests are sent to the dead address, 8.6.7.5, the packets are picked up by the program and responded to using dummy ICMP packets. While the packets are sent, they aren't necessarily registered as complete, which brings us to everyone's favorite part of the report.

Known Bugs:

There's only one outstanding bug in this program, which is that unfortunately the outbound ICMP packets do not register as complete/valid by the receiving system. I believe that the cause of this is the fact that the checksum field in the ICMP header is not properly calculated, although not without better effort. I tried changing the source IP to one that exists within the network, a multiple checksum functions, and several other things but the checksum calculation performed by Wireshark never matches up with the value that's assigned by the program. While I'm not 100% sure, I'm 95% sure that this is the cause of the shortcomings of this program. As far as I can tell everything else works exactly as it should. The ping requests and responses are both victim to this bug, as am I, who spent several hours on this bug alone over the last couple days.

Code Screenshots:

Since the pictures are big because the text is small, the screenshot doesn't fit on this page, so I'll use this opportunity to explain what you'll be looking at. For starters, sniffing the packets didn't really take much code outside of the filters since we were just supposed to download and use the snffex, there are screenshots of it running on the second page though and the filter code on the first.

Images one and 2 are both taken from the spoof.c program. The first screenshot contains the buildIPHeader and buildICMPHeader functions. They basically assign values to each field of the IP and ICMP header structures that are contained in the netinet library. The buildIPHeader function also takes in character pointers for the source and destination IP addresses. Both functions also compute (or attempt to compute) the checksum field. In the second picture you can see the main function, which allocates memory for the packet to be sent, creates the two packet headers, and copies them into the buffer using memcpy(). Additionally, the main function creates, assigns and binds the socket before sending the packet using the sendto() function.

The third and fourth screenshots are from the sniff-and-spoof.c program, and contains many of the same functions. Image #3 contains the main function and most of the code that handles the capturing and filtering of packets as well as the printing of information about incoming packets. As you can see at the beginning, this program was designed to capture icmp packets from IP 10.0.2.4, the dummy VM used during testing. The program retrieves info about the capture device, opens a capture session, compiles and applies the filters, and calls the got_packet() function every time a packet is "sniffed". The fourth screenshot shows the got_packet(), which handles each captured packet. When a packet is retrieved by the program, the source and destination addresses are retrieved, and a new packet is created using that information. This allows us to send responses as if we were the machine that the ping requests were intended for. The packet header building functions, socket creation and packet sending are all the same as the spoofer, except this one prints out the source and destination of the response packets that it sends out.

Improvements: If I were to redesign this program for higher packet throughput, I would create and manage the sockets used to send the packets outside of the handler function in sniff-and-spoof. This would allow for a higher output in less time for sending response packets as there isn't constant creation and management being done on the outputting sockets.

Austin Peterson
UIN: 926006358

acket-Sniffer-and-Spoofer/spoof.c • (Packet-Sniffer-and-Spoofer) - Sublime Text (UNREGISTERED)

sniffex.c × | spoof.c ● | sniff-and-spoof.c ×

```c

// struct eth_header* buildEthernetHeader(char* source_mac, char* dest_mac){
// }

struct ip* buildIPHeader(char* source_addr, char* dest_addr){
        struct ip* ipHeader;
        int size = sizeof(struct icmphdr) + sizeof(struct ip)+1;

        ipHeader = (struct ip*)malloc(sizeof(struct ip));
        ipHeader->ip_tos = 0;
        ipHeader->ip_v = 4;
        ipHeader->ip_hl = (sizeof(struct ip))/4;
        ipHeader->ip_len = htons(size);
        ipHeader->ip_id = rand();
        ipHeader->ip_off = 0;
        ipHeader->ip_ttl = 64;
        ipHeader->ip_p = IPPROTO_ICMP;
        inet_aton(source_addr, &ipHeader->ip_src);
        inet_aton(dest_addr, &ipHeader->ip_dst);
        ipHeader->ip_sum = checkSum((char*) ipHeader, ipHeader->ip_len);
        //printf("IP header sizeof: %s\n", ipHeader->ip_hl);
        return ipHeader;
}

struct icmphdr* buildICMPHeader(){
        struct icmphdr* icmp = (struct icmphdr*)malloc(sizeof(struct icmphdr));
        icmp->type = ICMP_ECHO;
        icmp->code = 0;
        icmp->checksum = 0;

        icmp->checksum = checkSum((char *)&icmp, 2);
  printf("chksum: %x\n",checkSum((char *)&icmp, sizeof(icmp)));
  //icmp->checksum = htons(0x8336);
        //icmp->checksum = in_cksum((unsigned short*)icmp, sizeof(struct icmphdr));
        return icmp;
}


u_short checkSum(char *addr, int len){
      long sum = 0;  /* assume 32 bit long, 16 bit short */
       while(len > 1){
         int temp = *((unsigned short*)addr);
         sum += temp++;
         if(sum & 0x80000000)   /* if high order bit set, fold */
           sum = (sum & 0xFFFF) + (sum >> 16);
         len -= 2;
       }

       if(len)       /* take care of left over byte */
         sum += (unsigned short) *(unsigned char *)addr;

       while(sum>>16)
         sum = (sum & 0xFFFF) + (sum >> 16);

       return ~sum;
}

int main(){

int sd;
struct sockaddr_in sin;
char buffer[BUFFER_SIZE]; //You can change the buffer size
const int on =1;

char* source_string = "4.4.4.4";    //spoof source address
char* dest_string = "10.0.2.4";

//struct eth_header* eth_hdr = buildEthernetHeader(source_mac, dest_mac); //didnt get used
struct ip* ip_hdr = buildIPHeader(source_string, dest_string);      //build IP header
struct icmphdr* icmp_hdr = buildICMPHeader();                       //build icmp header

```

```
--- /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:06:35 2020
+++ /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:09:24 2020
@@ -110,7 +110,7 @@
```

: 180, Column 73

Austin Peterson
UIN: 926006358

acket-Sniffer-and-Spoofer/spoof.c • (Packet-Sniffer-and-Spoofer) - Sublime Text (UNREGISTERED)

sniffex.c × | spoof.c ● | sniff-and-spoof.c ×

```
        if(len)        /* take care of left over byte */
            sum += (unsigned short) *(unsigned char *)addr;

        while(sum>>16)
            sum = (sum & 0xFFFF) + (sum >> 16);

        return ~sum;
}

int main(){

int sd;
struct sockaddr_in sin;
char buffer[BUFFER_SIZE]; //You can change the buffer size
const int on =1;

char* source_string = "4.4.4.4";     //spoof source address
char* dest_string = "10.0.2.4";

//struct eth_header* eth_hdr = buildEthernetHeader(source_mac, dest_mac); //didnt get used
struct ip* ip_hdr = buildIPHeader(source_string, dest_string);      //build IP header
struct icmphdr* icmp_hdr = buildICMPHeader();                       //build icmp header

printf("IP header size: %d\n", sizeof(struct ip));
printf("ICMP header size: %d\n", sizeof(struct icmphdr));

// memcpy(buffer, eth_hdr, SIZE_ETHERNET);
// memcpy(buffer + SIZE_ETHERNET, ip_hdr, sizeof(struct ip));
// memcpy(buffer+SIZE_ETHERNET+sizeof(struct ip), icmp_hdr, sizeof(struct icmphdr));

memcpy(buffer, ip_hdr, sizeof(struct ip)+1);                //copy headers into packet buffer
memcpy(buffer + sizeof(struct ip), icmp_hdr, sizeof(struct icmphdr));


sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);    //create socket
if(sd<0){
    perror("socket() error");
    exit(-1);
}
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = ip_hdr->ip_dst.s_addr;                    //set socket source addr

bind(sd, (struct sockaddr*)&sin, sizeof(sin));
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    perror("setsockopt");
    exit(1);
  }

if(sendto(sd, buffer, BUFFER_SIZE, 0, (struct sockaddr*)&sin,sizeof(sin))<0){
    perror("sendto() error");                              //send packet
    exit(-1);
}
else{
        printf("Packet Sent\n");
}
return 0;
}
```

```
--- /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:06:35 2020
+++ /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:09:24 2020
@@ -110,7 +110,7 @@
```

e 138, Column 2

Austin Peterson
UIN: 926006358

Packet-Sniffer-and-Spoofer/sniff-and-spoof.c (Packet-Sniffer-and-Spoofer) - Sublime Text (UNREGISTERED)

sniffex.c × | spoof.c × | sniff-and-spoof.c ×

```c

int main()
{
    char *dev = NULL;               /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
    pcap_t *handle;                 /* packet capture handle */
    const int on =1;

    char filter_exp[] = "icmp and (src host 10.0.2.4)"; /* filter expression [3] */
    struct bpf_program fp;       /* compiled filter program (expression) */
    bpf_u_int32 mask;            /* subnet mask */
    bpf_u_int32 net;             /* ip */
    int num_packets = 10;        /* number of packets to capture */

    //print_app_banner();

    /* check for capture device name on command-line */
    dev = pcap_lookupdev(errbuf);
    if (dev == NULL)
    {
        fprintf(stderr, "Couldn't find default device: %s\n",
                errbuf);
        exit(EXIT_FAILURE);
    }

    /* get network number and mask associated with capture device */
    if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1)
    {
        fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
                dev, errbuf);
        net = 0;
        mask = 0;
    }

    /* print capture info */
    printf("Device: %s\n", dev);
    printf("Number of packets: %d\n", num_packets);
    printf("Filter expression: %s\n", filter_exp);

    /* open capture device */
    handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
    if (handle == NULL)
    {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        exit(EXIT_FAILURE);
    }

    /* make sure we're capturing on an Ethernet device [2] */
    if (pcap_datalink(handle) != DLT_EN10MB)
    {
        fprintf(stderr, "%s is not an Ethernet\n", dev);
        exit(EXIT_FAILURE);
    }
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1)
    {
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
    }

    /* apply the compiled filter */
    if (pcap_setfilter(handle, &fp) == -1)
    {
        fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
    }

    /* now we can set our callback function */
    pcap_loop(handle, num_packets, got_packet, NULL);

```

```
--- /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:06:35 2020
+++ /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:09:24 2020
@@ -110,7 +110,7 @@
```

Line 41, Column 1

Austin Peterson
UIN: 926006358

acket-Sniffer-and-Spoofer/sniff-and-spoof.c (Packet-Sniffer-and-Spoofer) - Sublime Text (UNREGISTERED)

sniffex.c × | spoof.c × | sniff-and-spoof.c ×

```c
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    const struct ip *ip; /* The IP header */

    int size_ip;

    printf("ICMP Packet Received\n");
    const int on =1;


    /* define/compute ip header offset */
    ip = (struct ip *)(packet + SIZE_ETHERNET);
    size_ip = ip->ip_hl * 4;
    if (size_ip < 20)
    {
        printf("   * Invalid IP header length: %u bytes\n", size_ip);
        return;
    }

    /* print source and destination IP addresses */
    printf("       From: %s\n", inet_ntoa(ip->ip_src));
    printf("         To: %s\n", inet_ntoa(ip->ip_dst));

    //Create and send spoof packet
    int sd;
    struct sockaddr_in sin;
    char buffer[100];

    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sd < 0)
    {
        perror("socket() error");
        exit(-1);
    }

    sin.sin_family = AF_INET;

    struct ip* ipHeader = buildIPHeader(inet_ntoa(ip->ip_dst), inet_ntoa(ip->ip_src));
    ipHeader->ip_dst = ip->ip_src;
    struct icmphdr* icmpHeader = buildICMPHeader();

    memcpy(buffer, ipHeader, sizeof(struct ip));
    memcpy(buffer+sizeof(struct ip), icmpHeader, sizeof(struct icmphdr));

    size_t packet_len = sizeof(buffer);

    sin.sin_addr.s_addr = ip->ip_dst.s_addr;

if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    perror("setsockopt");
    exit(1);
  }

    printf("Sending Spoofed Response Packet\n");
    printf("       From: %s\n", inet_ntoa(ipHeader->ip_src));
    printf("         To: %s\n\n", inet_ntoa(ipHeader->ip_dst));

    if (sendto(sd, buffer, packet_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    {
        perror("sendto() error");
        exit(-1);
    }

    return;
}

int main()
{
    char *dev = NULL;               /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE];  /* error buffer */
```

```
--- /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:06:35 2020
+++ /home/seed/Documents/Packet-Sniffer-and-Spoofer/spoof.c Tue Sep  8 20:09:24 2020
@@ -110,7 +110,7 @@
```

e 41, Column 1