Basic Packet-Sniffer Construction from the Ground Up

By Chad Renfro

Packet sniffers are applications used by network administrators to monitor and validate network traffic. Sniffers are programs used to read packets that travel across the network at various levels of the OSI layer. And like most security tools sniffers too can be used for both good and destructive purposes.

On the light-side of network administration sniffers help quickly track down problems such as bottlenecks and misplaced filters. However on the dark-side sniffers can be used to reap tremendous amounts of havoc by gathering legitimate user names and passwords so that other machines can be quickly compromised. Hopefully this paper will be used to help administrators gain control of their networks by being able to analyze network traffic not only by using preconstructed sniffers but by being able to create their own. This paper will look at the packet sniffer from the bottem up, looking in depth at the sniffer core and then gradualy adding functionality to the application.

The example included here will help illustrate some rather cumbersome issues when dealing with network programing. In no way will this single paper teach a person to write a complete sniffing application like tcpdump or sniffit. It will however teach some very fundamental issues that are inherent to all packet sniffers. Like how the packets are accessed on the network and how to work with the packets at different layers.

The most basic sniffer...

Sniffer #1.

This sniffer will illustrate the use of the SOCK_RAW device and show how to gather packets from the network and print out some simple header information to std_out.

Although the basic premise is that packet sniffers operate in a promiscuous mode which listens to all packets weather or not the packet is destined for the machines mac address, this example will collect packets in a non-promiscuous mode . This will let usconcentrate on the SOCK_RAW device for the first example. To operate this same code in a promiscous mode the network card may be put in a promiscous mode manually. To do this type this in after the log in :

```
> su -
Password : ********
# ifconfig eth0 promisc
```

This will now set the network interface eth0 in promiscous mode.

```
/***********************simple_Tcp_sniff.c********************/

1.      #include <stdio.h>
2.      #include <sys/socket.h>
3.      #include <netinet/in.h>
4.      #include <arpa/inet.h>

5.      #include "headers.h"

6.      int main()
7.      {
8.          int sock, bytes_recieved, fromlen;
9.          char buffer[65535];
10.         struct sockaddr_in from;
11.         struct ip  *ip;
12.         struct tcp *tcp;
```

```
13.

14.         sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);

15.     while(1)
16.      {
17.             fromlen = sizeof from;
18.             bytes_recieved = recvfrom(sock, buffer, sizeof buffer, 0,
                                          (struct sockaddr *)&from,
&fromlen);
19.             printf("\nBytes received ::: %5d\n",bytes_recieved);
20.             printf("Source address ::: %s\n",inet_ntoa(from.sin_addr));
21.             ip = (struct ip *)buffer;
22.             printf("IP header length ::: %d\n",ip->ip_length);
23.             printf("Protocol ::: %d\n",ip->ip_protocol);
24.             tcp = (struct tcp *)(buffer + (4*ip->ip_length));
25.             printf("Source port ::: %d\n",ntohs(tcp->tcp_source_port);
26.             printf("Dest port  ::: %d\n",ntohs(tcp->tcp_dest_port));

27.              }
28. }
/**********************EOF***********************************/
```

What this means :

Line 1-4 :
These are the header files required to use some needed c functions we will use later

```
        <stdio.h>       =     functions like printf and std_out
        <sys/socket.h> =      this will give access to the SOCK_RAW and the
                              IPPROTO_TCP defines
        <netinet/in.h> =      structs like the sockaddr_in
        <arpa/inet.h>  =      lets us use the functions to do network to host byte
                              order conversions
```

Line 5 :
This is the header file headers.h that is also included with this program to give standard structures to access the ip and tcp fields. The structures identify each field in the ip and tcp header for instance :

```
        struct ip {
                unsigned int        ip_length:4;         /* length of ip-header in
32-bit
                                                            words*/
                unsigned int        ip_version:4;        /* set to "4", for Ipv4 */
                unsigned char       ip_tos;              /* type of service*/
                unsigned short      ip_total_length;     /* Total length of ip
datagram in
                                                            bytes */
                unsigned short      ip_id;               /*identification field*/
                unsigned short      ip_flags;
                unsigned char       ip_ttl;              /*time-to-live, sets upper
limit
                                                           for max number of
routers to
                                                           go through before the
packet is
                                                           discarded*/
```

```
                unsigned char       ip_protocol;         /*identifies the correct
transport
                                                          protocol */
                unsigned short      ip_cksum;            /*calculated for the ip
header ONLY*/
                 unsigned int       ip_source;           /*source ip */
                 unsigned int       ip_dest;             /*dest ip*/
        };



        struct tcp {
                  unsigned short     tcp_source_port; /*tcp source port*/
                  unsigned short     tcp_dest_port;   /*tcp dest port*/
                  unsigned int       tcp_seqno;       /*tcp sequence number,
                                                        identifies the byte in the
                                                        stream of data*/
                  unsigned int       tcp_ackno;       /*contains the next seq num
that
                                                        the sender expects to
recieve*/
                  unsigned int       tcp_res1:4,      /*little-endian*/
                                     tcp_hlen:4,      /*length of tcp header in 32-
bit
                                                        words*/
                                     tcp_fin:1,       /*Finish flag "fin"*/
                                     tcp_syn:1,       /*Synchronize sequence
                                                        numbers to start a
connection
                                    tcp_rst:1,       /*Reset flag */
                                     tcp_psh:1,       /*Push, sends data to the
                                                        application*/
                                     tcp_ack:1,       /*acknowledge*/
                                     tcp_urg:1,       /*urgent pointer*/
                                     tcp_res2:2;
                   unsigned short    tcp_winsize;     /*maxinum number of bytes
able
                                                       to recieve*/
                  unsigned short     tcp_cksum;       /*checksum to cover the tcp
                                                        header and data portion of
the
                                                        packet*/

                  unsigned short     tcp_urgent;     /*vaild only if the urgent flag
is
                                                      set, used to transmit
                                                       emergency data */
        };
```

Line 8-13 :
This is the variable declaration section

```
        integers :
             sock                = socket file descriptor
             bytes_recieved      = bytes read from the open socket "sock"
```

```
      fromlen              = the size of the from structure char :
      buffer               = where the ip packet that is read off the
                             wire will be held buffer will hold a datagram
                             of 65535 bytes which is the maximum length
                             of an ip datagram.

Struct sockaddr_in :

    struct sockaddr_in {
        short int          sin_family;  /* Address family   */
        unsigned short int sin_port;    /* Port number      */
        struct in_addr     sin_addr;    /* Internet address */
        unsigned char      sin_zero[8]; /* Same size as struct sockaddr */
     };
```

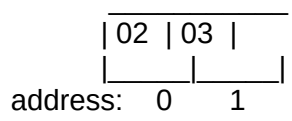
Before we go any further two topics should be covered,byte-ordering and sockaddr structures. Byte-ordering,is the way that the operating system stores bytes in memory.

There are two ways that this is done first with the low-order byte at the starting address this is known as "little-endian" or host-byte order. Next bytes can be stored with the high order byte at the starting address, this is called "big-endian" or network byte order.

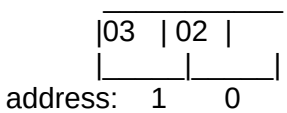
The Internet protocol uses >>>>>> network byte order.

This is important because if you are working on an intel based linux box you will be programming on a little-endian machine and to send data via ip you must convert the bytes to network-byte order. For examle lets say we are going to store a 2-byte number in memory say the value is (in hex) 0x0203

First this is how the value is stored on a big-endian machine:



And here is the same value on a little-endian machine:



The same value is being represented in both examples it is just how we order the bytes that changes.

The next topic that you must understand is the sockaddr vs. the sockaddr_in structures. The struct sockaddr is used to hold information about the socket such as the family type and other address information it looks like :

```
    struct sockaddr {
              unsigned short sa_family;         /*address family*/
```

```
            char            sa_data[14];        /*address data*/
};
```

The first element in the structure "sa_family" will be used to reference what the family type is for the socket, in our sniffer it will be AF_INET. Next the "sa_data" element holds the destination port and address for the socket. To make it easier to deal with the sockaddr struct the use of the sockaddr_in structure is commonly used. Sockaddr_in makes it easier to reference all of the elements that are contained by sockaddr.

Sockaddr_in looks like:

```
struct sockaddr_in {
          short int          sin_family;    /* Address family               */
          unsigned short int sin_port;      /* Port number                  */
          struct in_addr     sin_addr;      /* Internet address             */
          unsigned char      sin_zero[8];   /* Same size as struct sockaddr */
};
```

We will use this struct and declare a variable "from" which will give us the information on the packet that we will collect from the raw socket. For instance the var "from.sin_addr" will give access to the packets source address (in network byte order). The thing to mention here is that all items in the sockaddr_in structure must be in network-byte order. When we receive the data in the sockaddr_in struct we must then convert it back to Host-byte order. To do this we can use some predefined functions to convert back and forth between host and network byteorder.

Here are the functions we will use:

ntohs : this function converts network byte order to host byte order for a 16-bit short

ntohl : same as above but for a 32-bit long

inet_ntoa : this function converts a 32-bit network binary value to a dotted decimal ip address

inet_aton : converts a character string address to the 32-bit network binary value

inet_addr : takes a char string dotted decimal addr and returns a 32-bit network binary value

To further illustrate ,say I want to know the port number that this packet originated from:

```
int packet_port; packet_port=ntohs(from.sin_port);
                                  ^^^^^
```

If I want the source IP address of the packet we will use a special function to get it to the 123.123.123.123 format:

```
char *ip_addr; ip_addr =inet_ntoa(from.sin_addr)
                        ^^^^^^^^
```

Line 11-12:

```
struct ip *ip :
 struct tcp *tcp :
```

This is a structure that we defined in our header file "headers.h". This structure is declared so that we can access individual fields of the ip/tcp header. The structure is like a transparent slide with predefined fields drawn on it. When a packet is taken off the wire it is a stream of bits, to make sense of it the "transparency" (or cast) is laid on top of or over the bits so the individual fields can be referenced.

Line 14 :

```
sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
```

This is the most important line in the entire program. Socket() takes three arguments in this form:

```
sockfd = socket(int family, int type, int protocol);
```

The first argument is the family. This could be either AF_UNIX which is used so a process can communicate with another process on the same host or AF_INET which is used for internet communication between remote hosts. In this case it will be AF_INET . Next is the type, the type is usually between 1 of 4 choices (there are others that we will not discuss here) the main four are :

```
1.   SOCK_DRAM      : used for udp datagrams
 2.  SOCK_STREAM    : used for tcp packets
 3.  SOCK_RAW       : used to bypass the transport layer
                      and directly access the IP layer

 4.  SOCK_PACKET    : this is linux specific, it is similuar to
                       SOCK_RAW except it accesses the DATA LINK Layer
```

For our needs we will use the SOCK_RAW type. You must have root acces to open a raw socket. The last parameter is the protocol,the protocol value specifies what type of traffic the socket should receive , for normal sockets this value is usally set to "0" because the socket can figure out if for instance the "type" of SOCK_DGRAM is specified then the protocol should be UDP.In our case we just want to look at tcp traffic so we will specify IPPROTO_TCP.

Line 15 :

```
while (1)
```

The while (1) puts the program into an infinite loop this is necessary so that after the first packet is processed we will loop around and grab the next.

Line 18:

```
bytes_recieved = recvfrom(sock, buffer, sizeof buffer, 0, (struct sockaddr
*)&from, &fromlen);
```

Now here is where we are actually reading data from the open socket "sock".The from struct is also filled in but notice that we are casting "from" from a "sockaddr_in" struct to a "sockaddr" struct. We do this because the recvfrom() requires a sockaddr type but to access the separate fields we will

continue to use the sockaddr_in structure. The length of the "from" struct must also be present and passed by address. The recvfrom() call will return the number of bytes on success and a -1 on error and fill the global var errno.

This is what we call "blocking-I/O" the recvfrom() will wait here forever until a datagram on the open socket is ready to be processed. This is opposed to Non-blocking I/O which is like running a process in the background and move on to other tasks.

Line 20:

```
printf("Source address ::: %s\n",inet_ntoa(from.sin_addr));
```

This printf uses the special function inet_ntoa() to take the value of "from.sin_addr" which is stored in Network-byte order and outputs a value in a readable ip form such as 192.168.1.XXX.

Line 21:

```
ip = (struct ip *)buffer;
```

This is where we will overlay a predefined structure that will help us to individually identify the fields in the packet that we pick up from the open socket.

Line 22:

```
printf("IP header length ::: %d\n",ip->ip_length);
```

The thing to notice on this line is the "ip->ip_length" this will access a pointer in memory to the ip header length the important thing to remember is that the length will be represented in 4-byte words this will be more important later when trying to access items past the ip header such as the tcp header or the data portion of the packet.

Line 23:

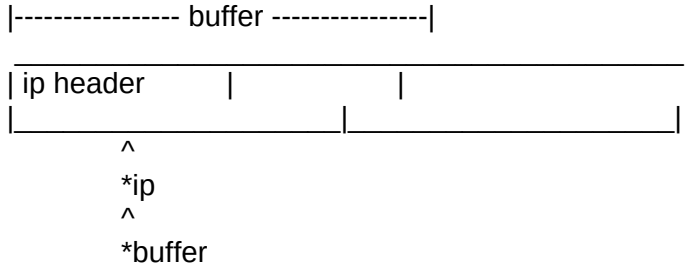
```
printf("Protocol ::: %d\n",ip->ip_protocol);
```

This gives access to the type of protocol such as 6 for tcp or 17 for udp.

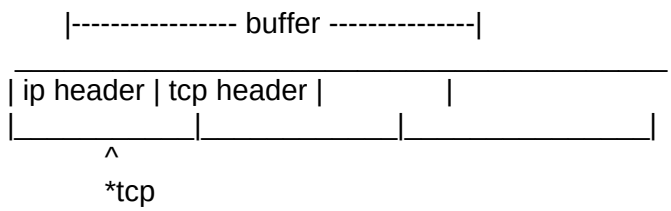
Line 24:

```
tcp = (struct tcp *)(buffer + (4*ip->ip_length));
```

Remember earlier it was mentioned that the ip header length is stored in 4 byte words, this is where that bit of information becomes important. Here we are trying to get access to the tcp header fields, to do this we must overlay a structure that has the fields predefined just as we did with ip. There is one key difference here the ip header fields were easy to access due to the fact that the beginning of the buffer was also the beginning of the ip header as so :



So to get access to the ip header we just set a pointer casted as an ip structure to the beginning of the buffer like "ip = (struct ip *)buffer;". To get access to the tcp header is a little more difficult due to the fact that we must set a pointer and cast it as a tcp structure at the beginning of the tcp header which follows the ip header in the buffer as so :



This is why we use 4*ip->ip_length to find the start of the tcp header.

Line 25-26:

```
   printf("Source port ::: %d\n",ntohs(tcp->tcp_source_port);
     printf("Dest port  ::: %d\n",ntohs(tcp->tcp_dest_port));
```

We can now access the source and dest ports which are located in the tcp header via the structure as defined above.

This will conclude our first very simple tcp sniffer. This was a very basic application that should help define how to access packets passing on the network and how to use sockets to access the packets. Hopefully this will be the first of many papers to come, which each proceeding paper we will add a new or more complex feature to the sniffer. I should also mention that there a number of great resources on the net that should aid you in further research in this area :

1. Beej's Guide to Network Programming
   This is an awesome paper that really helps
   clear up any misconceptions about network programming.
   [http://www.ecst.csuchico.edu/~beej/guide/net]

2. TCP/IP Illustrated Vol 1,2,3
   W.Richard Stevens

To use the above program, cut out the above code and strip off all of the line numbers. Save the edited file as sniff.c. Next cut out the header file headers.h (below) and save it to a file headers.h in the same directory. Now just compile: gcc -o sniff sniff.c. You should now have the executable "sniff", to run it type #./sniff

```
/************************headers.h*************************/
/*structure of an ip header             */
struct ip {
        unsigned int        ip_length:4;    /*little-endian*/
        unsigned int        ip_version:4;
        unsigned char       ip_tos;
        unsigned short      ip_total_length;
        unsigned short      ip_id;
        unsigned short      ip_flags;
        unsigned char       ip_ttl;
        unsigned char       ip_protocol;
        unsigned short      ip_cksum;
        unsigned int        ip_source;
```

```
        unsigned int        ip_dest;
};

/* Structure of a TCP header */
struct tcp {
        unsigned short      tcp_source_port;
        unsigned short      tcp_dest_port;
        unsigned int        tcp_seqno;
        unsigned int        tcp_ackno;
        unsigned int        tcp_res1:4,     /*little-endian*/
        tcp_hlen:4,
        tcp_fin:1,
        tcp_syn:1,
        tcp_rst:1,
        tcp_psh:1,
        tcp_ack:1,
        tcp_urg:1,
        tcp_res2:2;
        unsigned short      tcp_winsize;
        unsigned short      tcp_cksum;
        unsigned short      tcp_urgent;
};
/*********************EOF*************************************/
```

Packet Sniffer Construction
Part II

In the previous paper we discussed the use of the SOCK_RAW device for accessing packets from the network layer, and how to interpret packets in a logical manor. This will serve as the basis for the next set of topics for constructing a more complete packet sniffer.

The first topic will be error checking the socket function calls. Error checking will become invaluable as the code evolves into a more complete application. The second topic that this paper will concentrate on is the use of the "ioctl" function for selecting and manipulating the network interface.

By looking at the code below you will notice that it has grown substantially since the last issue. First the use of functions to has beenimplemented to modularize the code, due to the fact that one of the main topics of this issue error checking. Modular coding is very helpful in quickly tracking down the problem in evolving code. This also helps the growing pains of adding new functions without backtracking and debugging the code base. Second all of the socket call have been "wrapped". Also remember in the first article the code was not really a true packet sniffer.

This was due to the fact that the sniffer did not set the interface into promiscuous "PROMISC" mode. Promiscuous mode on a network interface enables an interface that is intended to look at traffic addressed only to its 6 byte mac address to look at ALL traffic on the broadcast medium. This sniffer will utilize the Set_Promisc function to set the promiscuous flag on the network interface. In the original sniffer,the sniffer would get packets from the first non-loopback interface. All of the manipulation to the interface in this project will be done via the ioctl function call. The ioctl function call is "used to manipulate the underlying device parameters for special files", as stated by the BSD man pages. These special file are usually terminals, sockets and interfaces. Our concern is using ioctl to manipulate socket and the network interface. The interface in this project will be chosen by hard coding it in the "headers.h" file.

The ioctl function is the all in one function for ansii c when it comes to gathering and manipulating interface attributes. Although the following example shows how to retrieve and set certain flags on a given interface ioctl has many other uses that should be looked at as well. For a better understanding of ioctl other features look at the ioctls.h and ioctl-types.h files.

```
/************************Tcp_sniff_2.c*******************/
1.#include <stdio.h>
2.#include <sys/socket.h>
3.#include <socketbits.h>
4.#include <sys/ioctl.h>
5.#include <net/if.h>
6.#include <netinet/in.h>
7.#include <arpa/inet.h>
8.#include <unistd.h>
9.#include "headers.h"

/*Prototype area*/

10.int Open_Raw_Socket(void);
11.int Set_Promisc(char *interface, int sock);

12.int main() {
13.    int sock, bytes_recieved, fromlen;
14.    char buffer[65535];
15.    struct sockaddr_in from;
16.    struct ip  *ip;
17.    struct tcp *tcp;

18.  sock = Open_Raw_Socket();

    /*now since the socket has been created,
      set the interface into promiscuous mode*/

19.  Set_Promisc(INTERFACE, sock);


20.        while(1)
22.        {
23.               fromlen = sizeof from;
24.               bytes_recieved = recvfrom(sock, buffer, sizeof buffer,
                                 0, (struct sockaddr *)&from, &fromlen);
25.               printf("\nBytes received ::: %5d\n",bytes_recieved);
26.               printf("Source address ::: %s\n",inet_ntoa(from.sin_addr));
27.               ip = (struct ip *)buffer;
                 /*See if this is a TCP packet*/
28.               if(ip->ip_protocol == 6) {
                    /*This is a TCP packet*/
29.                     printf("IP header length ::: %d\n",ip->ip_length);
30.                     printf("Protocol ::: %d\n",ip->ip_protocol);
31.                     tcp = (struct tcp *)(buffer + (4*ip->ip_length));
32.                     printf("Source port ::: %d\n",ntohs(tcp-
>tcp_source_port));
33.                     printf("Dest port  ::: %d\n",ntohs(tcp->tcp_dest_port));
34.               }

35.        }
36.}
```

```
37.int Open_Raw_Socket() {
38. int sock;
39.   if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
        /*Then the socket was not created properly and must die*/
40.        perror("The raw socket was not created");
41.        exit(0);
42.   };
43.       return(sock);
44. }

45.int Set_Promisc(char *interface, int sock ) {
46.       struct ifreq ifr;
47.       strncpy(ifr.ifr_name, interface,strnlen(interface)+1);
48.       if((ioctl(sock, SIOCGIFFLAGS, &ifr) == -1)) {
                /*Could not retrieve flags for the interface*/
49.               perror("Could not retrive flags for the interface");
50.               exit(0);
51.       }
52.       printf("The interface is ::: %s\n", interface);
53.       perror("Retrieved flags from interface successfully");

        /*now that the flags have been retrieved*/
        /* set the flags to PROMISC */
54.       ifr.ifr_flags |= IFF_PROMISC;
55.       if (ioctl (sock, SIOCSIFFLAGS, &ifr) == -1 ) {
                /*Could not set the flags on the interface */
56.               perror("Could not set the PROMISC flag:");
57.               exit(0);
58.       }
59.       printf("Setting interface ::: %s ::: to promisc", interface);

60.       return(0);
61. }

/**********************EOF***********************************/
```

Now we will examine the code line by line. Most of the code base has not changed and therefore we will not spend time going over the original code. Lets get started.

```
18.  sock = Open_Raw_Socket();
       Here is the call to open the raw socket. Now
       jump down and look at the Open_Raw_Socket
       function.

       37.int Open_Raw_Socket() {
       38. int sock;
       39.   if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
               /*Then the socket was not created properly and must die*/
       40.        perror("The raw socket was not created");
       41.        exit(0);
       42.   };
       43.       return(sock);
       44. }
```

Allot of this function should look familiar from the first paper. The socket call is the same only this time the socket function is wrapped in an if statement to test for an error return. Remember all the socket call does is create an endpoint for communication if successful it will return a socket descriptor (an integer) and if the call fails it will return a "-1". This is what line 39 is testing for, if the socket returns a value less than 0 it must have failed. If it fails two other tasks must take place. First report the error using perror, this will print the error to std_out and will also print the "errno" value that will describe where the last call failed. Second the exit call is used to halt the execution of the program. Now most good programs will also close any open file/socket descriptors before the exit is called. This is not really necessary due to the fact that the exit call will close all open descriptors before it closes.

Hopefully the socket will succeed and return an open descriptor If it does the function will perform the return(sock) call and send back an open descriptor to the call on line 18 and store the descriptor in "sock". If the socket call fails this could be caused by a defective interface or the user running the program not having the correct permissions. Remember to open a socket the user must have root access.

```
Set_Promisc(INTERFACE, sock);
```

Now that the socket has been successfully created the interface can be chosen and manipulated. For this example the interface has been preselected and hard coded into "headers.h" it reads,

```
#define INTERFACE "eth0"
```

This is not the optimal way to choose an interface due to the fact that there are calls to query for all network interfaces using ioctl. Given that this is our first exercise using ioctl the emphasis will be placed on the use of ioctl to manipulate the flags for a predefined interface.

```
45.int Set_Promisc(char *interface, int sock ) {
46.       struct ifreq ifr;
47.       strncpy(ifr.ifr_name, interface,strnlen(interface)+1);
48.       if((ioctl(sock, SIOCGIFFLAGS, &ifr) == -1)) {
                /*Could not retrieve flags for the interface*/
49.               perror("Could not retrive flags for the interface");
50.               exit(0);
51.       }
52.       printf("The interface is ::: %s\n", interface);
53.       perror("Retrieved flags from interface successfully");
          /*now that the flags have been retrieved*/
          /* set the flags to PROMISC */
54.       ifr.ifr_flags |= IFF_PROMISC;
55.       if (ioctl (sock, SIOCSIFFLAGS, &ifr) == -1 ) {
                /*Could not set the flags on the interface */
56.               perror("Could not set the PROMISC flag:");
57.               exit(0);
58.       }
59.       printf("Setting interface ::: %s ::: to promisc", interface);
60.       return(0);
61. }
```

Starting at line 45, look at the beginning of the Set_Promisc function. As the name implies the sole purpose of this function is to set a network interface into promiscuous mode. The function takes two parameters, the a char pointer to the interface and the integer that references the open raw socket. Now starting with line 46 we will introduce ioctl.

46. struct ifreq ifr;

This is an interface request structure used for socket ioctl. The ifreq structure is a rather large structure the main members that will be used in this structure are the members that hold the interface name and the interface flags.

47. strncpy(ifr.ifr_name, interface,strnlen(interface)+1);

Earlier we addressed the fact that we had the interface predetermined and hard coded in "headers.h". Here the value held in interface must be copied into the ifr structure into the ifr.ifr_name member. This is due to the fact that the ioctl call will require an address to a interface request structure with the name of the interface in the structure.

48. if((ioctl(sock, SIOCGIFFLAGS, &ifr) == -1))

Here is the first ioctl call that will get the flags of the interface name that was placed in the ifr struct earlier look at the internal ioctl call :

ioctl(sock, SIOCGIFFLAGS, &ifr)

the first parameter is an open socket descriptor "sock" the second is the request that is to be performed. In this case the call is requesting "SOCGIFFLAGS" which will get the flags of the "eth0" interface. The third parameter is an address of an interface request structure ifr which hold the name of the interface that will be queried. This step must be performed before the promisc flag can be set. Now look at the entire line, the ioctl call is all being tested for its return. On success the call will return "0" and if the call fails a "-1" will be returned.

54. ifr.ifr_flags |= IFF_PROMISC;

This is where the interface flags are changed. Here the promiscuous flag is being applied to the interface structure ifr. Notice the notation "|=" this is what applies the promiscuous flag to the ifr structure. Although the promisc flag is applied here this is not the final step there is still one final call to set the new flags into place. This is what is called bit testing, to see if a certain bit is set. There is a very special notation for altering and testing bits.

To set a specific bit use a binary "or" "|" to combine the bit var with the needed bit mask:

x = x|mask;

To unset a specific bit use the binary "and" "&" with the complement sign "~" of the mask :

x= x & ~mask;

To just test if a bit is on, use the "&" sign and evaluate the result for a non zero value :

result=x & mask;

55. if (ioctl (sock, SIOCSIFFLAGS, &ifr) == -1)

This is the final ioctl call to put the device into promiscuous mode. Just as the first call retrieved the flags of the interface, this call sets the new revised flags that were set in the ifr struct to the physical interface. Also notice that just as in the first ioctl call that the return value here is being tested for success.

59. printf("Setting interface ::: %s ::: to promisc", interface);

If all goes well this message should be sent to std_out letting the end_user know that the socket was created and that the interface was set into promiscuous mode properly.

60. return(0);

Finally the value of "0" is sent back to the original call to signify that the function completed successfully.

There is still allot of functionally that could be added to the sniffer to make it more complete but most of that is parsing the raw packetsinto some desired output form.

The next step in constructing a more complete sniffer would be to be able to capture not just a single protocol but all packets. This is the major drawback to this project is the fact that only Tcp based packets can be viewed. To work around that the SOCK_RAW device should be replaced with SOCK_PACKET.

However this is only for linux based operating systems. A better solution wouldbe to use a preconstructed API like libpcap written by Lawrence Berkeley National Laboratory. The use of libpcap was designed to make the sniffer code more portable across different operating systems. Now guess what the next issue will deal with.

This is the end of part two

1. Beej's Guide to Network Programming
This is an awesome paper that really helps
clear up any misconceptions about network programming.
[http://www.ecst.csuchico.edu/~beej/guide/net

2. TCP/IP Illustrated Vol 1,2,3, Unix Network Programing
W.Richard Stevens

```
/************************headers.h**************************/
/*structure of an ip header*/
struct ip {
        unsigned int        ip_length:4;    /*little-endian*/
        unsigned int         ip_version:4;
        unsigned char       ip_tos;
        unsigned short      ip_total_length;
        unsigned short      ip_id;
        unsigned short      ip_flags;
        unsigned char       ip_ttl;
        unsigned char       ip_protocol;
        unsigned short      ip_cksum;
        unsigned int          ip_source;
        unsigned int          ip_dest;
};
```

```
/* Structure of a TCP header */
struct tcp {
        unsigned short      tcp_source_port;
        unsigned short      tcp_dest_port;
        unsigned int         tcp_seqno;
        unsigned int         tcp_ackno;
        unsigned int         tcp_res1:4,     /*little-endian*/
        tcp_hlen:4,
        tcp_fin:1,
        tcp_syn:1,
        tcp_rst:1,
        tcp_psh:1,
        tcp_ack:1,
        tcp_urg:1,
        tcp_res2:2;
        unsigned short      tcp_winsize;
        unsigned short      tcp_cksum;
        unsigned short      tcp_urgent;
};
/*********************EOF*************************************/
```