

# Packet-sniffing tutorial

Daniel A. Nobuto

February 8, 2004

## Contents

<b>1</b>	<b>Framework</b>	<b>2</b>
<b>2</b>	<b>Pre-capture setup</b>	<b>4</b>
	2.0.1 Get a socket descriptor . . . . .	4
	2.0.2 Prepare for <code>ioctl</code> call . . . . .	4
	2.0.3 Set interface state . . . . .	5
<b>3</b>	<b>Capture packets</b>	<b>6</b>
<b>4</b>	<b>Miscellaneous functions</b>	<b>6</b>
<b>5</b>	<b>Command-line parsing</b>	<b>7</b>
<b>6</b>	<b>Miscellany</b>	<b>8</b>
<b>7</b>	<b>Defines</b>	<b>9</b>
	7.1 Global data . . . . .	9

As an example of a simple (and thus, not very efficient) TCP packet sniffer, the following program can be used as a minimal tutorial for dealing with promiscuous interfaces.

The document/program has been written as a literate program using the `noweb` package from Norman Ramsey.

The structure of the program is as follows:

```
1  <unet.c 1>≡
    <include files 13>
    <macros and defines 14>
    <prototypes 2>
    <bss variables 15>
    <local prototypes 9>
    <functions 3>
```

There are sections for both normal and local prototypes. This distinction is made to allow this program to be modified and used as a basis for other packet sniffers. As such, this code is made freely available under version 2.1 of the LGPL. A copy of the license can be found at the Free Software Foundation's web page at <http://www.fsf.org>.

## 1 Framework

The entry point into all standard C programs is the `main` function. The general flow of the program is shown in the code block below.

The local variables that are defined are used for the following purposes:

`if_fd` The file descriptor used to refer to the interface being monitored.

`res` This variable is used to store the return value of various functions so the return code can be checked.

`nr_read` Used to store the return value of `recvfrom` so the number of bytes read can be determined.

`off` The really simple example this code contains to print out the packets keeps track of the current position in the buffer being printed with this variable.

`ifreq` The data returned by the `ioctl` call is stored in this variable.

```
2  <prototypes 2>≡ (1)
    int main(int argc, char *argv[]);
    Uses main 3.
```

```

3  <functions 3>≡ (1) 10▷
    int main(int argc, char *argv[])
    {
    int    if_fd, /* interface file descriptor */
          res,   /* "result" */
          nr_read,
          off;   /* offset into result */
    struct ifreq  ifr;

        parse_command_line(argc, argv);

        <acquire socket descriptor 4>
        <initialize ifreq struct 5>
        <acquire ifreq state 6>
        <modify ifreq state 7>
        <capture packets 8>

        return 0;
    }

```

Defines:

if\_fd, used in chunks 4 and 6-8.

ifr, used in chunks 5-7.

main, used in chunk 2.

Uses parse\_command\_line 12.

## 2 Pre-capture setup

Before the packets can be sniffed, some work has to be done to get things ready. The following code snippets shows the required actions.

### 2.0.1 Get a socket descriptor

The `socket` call is told to create an IP-based socket. Rather than creating a socket that is specifically for TCP, UDP, ICMP, and such, it is told to create a raw socket that can parse anything. The last argument, 0, tells the socket call under Linux only to bother with TCP.

Why, then, do we use `SOCK_RAW` instead of `SOCK_STREAM`? Well, using a RAW socket, we can sniff packets, since a `SOCK_STREAM` parameter requires we attach the socket to some address and port.

Of course, there's the all-important error checking.

```
4  <acquire socket descriptor 4>≡ (3)
    if_fd = socket(PF_INET, SOCK_RAW, 0);
    if(if_fd == -1) {
        fprintf(stderr, "Unable to acquire a socket:%s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }
```

Uses `if_fd` 3.

### 2.0.2 Prepare for `ioctl` call

The `ioctl` call that is required to get the current state of the network interface needs a pointer to a `struct ifreq`. In order to avoid trashing the nicely configured interface too much, the current state of the interface needs to be read in.

In order to do so, we must first prepare the `struct ifreq` variable `ifr`. To ensure nothing weird is in there, a `memset` is done to clear all the fields including any padding that may be present.

The name of the interface to sniff, in this case “eth0”, is put into the structure.

```
5  <initialize ifreq struct 5>≡ (3)
    memset(&ifr, 0, sizeof(ifr));
    strncpy(ifr.ifr_name, "eth0", IFNAMSIZ);
```

Uses `ifr` 3.

Once the `ifr` variable is set up, an `ioctl` call is issued to acquire the current configuration state of the interface identified by the `ifr` variable.

`SIOCGIFFLAGS` is an `ioctl` value that means: "Socket I/O Control, Get Interface FLAGS". See? There's a meaning to that jumble of gibberish that the `ioctl` calls use for the second parameter after all!

The address of the `ifr` variable is passed in via the address-of operator since the `ioctl` call expects a pointer to an area of memory to write out the `struct ifreq` data.

```
6  <acquire ifreq state 6>≡ (3)
    res = ioctl(if_fd, SIOCGIFFLAGS, &ifr);
    if(res == -1) {
        perror("");
        exit(EXIT_FAILURE);
    }
```

Uses `if_fd` 3 and `ifr` 3.

### 2.0.3 Set interface state

Once the data from the `ioctl` call has been acquired, the state of the interface is changed. If the interface is already promiscuous, then this code won't really do much other than eat up CPU cycles.

In order to avoid issuing a system call when there is no need (mostly to keep the overhead down, though this isn't really much of an optimization...), the interface has the promiscuous flag added to the current flags only if it is not already set.

If the flag had to be set, the `ioctl` call is executed once again to tell the kernel to put the interface into promiscuous mode.

```
7  <modify ifreq state 7>≡ (3)
    if( (ifr.ifr_flags & IFF_PROMISC) == 0 ) {
        ifr.ifr_flags |= IFF_PROMISC;
        res = ioctl(if_fd, SIOCSIFFLAGS, &ifr);
        if(res == -1) {
            perror("");
            exit(EXIT_FAILURE);
        }
    }
```

Uses `if_fd` 3 and `ifr` 3.

### 3 Capture packets

This code is what you've been waiting for. Yup, this is it. The code that actually reads the packets!

The `recvfrom` code reads from the socket descriptor and writes the data it receives to the memory pointed to by `buf`. The number of bytes read is stored in `res` so the contents of the buffer can be printed out one byte at a time in somewhat human-readable form.

```
8  <capture packets 8>≡ (3)
    while(1) {
        res = recvfrom(if_fd, buf, sizeof(buf), 0, NULL, NULL);
        if(res == -1) {
            fprintf(stderr, "Unable to read data from socket:%s\n",
                strerror(errno));
            exit(EXIT_FAILURE);
        }
        for(off = 0; off < res; off++)
            printf("%x\n", buf[off]);
    }
```

Uses `buf` 15 and `if_fd` 3.

### 4 Miscellaneous functions

There are some functions that various programs need even though they are not concerned with the main purpose of the program.

One such function that is commonly used is a usage function that prints out the free-form help message many programs have. Unlike really broken piles of crap such as `cvs` and `mpg123`, the following usage function actually prints the message to `stdout` instead of blindly using `stderr`.

Oddly enough, this function is called `usage`.

Once called, `usage` prints out the short, useless help to the appropriate output stream then gracefully causes a program termination using the error code that was passed in.

```
9  <local prototypes 9>≡ (1) 11▷
    static void usage(int exit_value) __attribute__((noreturn));
```

Defines:

`usage`, used in chunk 12.

```

10  <functions 3>+≡ (1) <3 12>
    static void usage(int exit_value)
    {
        FILE *f;

        f = (exit_value == EXIT_SUCCESS) ? stdout : stderr;
        fprintf(f, PROG_NAME " " PROG_VERSION "\n");
        fprintf(f, "Usage:\n");
        fprintf(f, "\t-v\tVersion. Prints version.\n");
        fprintf(f, "\t-h\tHelp. Prints this message.\n");
        exit(exit_value);
    }

```

Defines:

- f, never used.
- usage, used in chunk 12.

Uses PROG\_NAME 14 and PROG\_VERSION 14.

## 5 Command-line parsing

What program would be complete without some measure of command-line options?

Of course, this program has very few, since it's meant more to be a tutorial on writing a sniffer than one on making user-friendly programs. Nevertheless, a stub function with some functionality is included here to parse the minimal options this program understands.

In fact, there are only two options to this program:

-v Version. Prints out the program name and version.

-h Help. Prints out the wonderfully verbose and lengthy help.

```

11  <local prototypes 9>+≡ (1) <9
    static void parse_command_line(int argc, char *argv[]);

```

Uses parse\_command\_line 12.

```

12  <functions 3>+≡ (1) <10
    static void parse_command_line(int argc, char *argv[])
    {
        int curr_opt;

        while( (curr_opt = getopt(argc, argv, "vhc:")) != -1) {

            switch(curr_opt) {
                case 'v':
                    printf(PROG_NAME " " PROG_VERSION "\n");
                    exit(EXIT_SUCCESS);
                case 'h':
                    usage(EXIT_SUCCESS);
                case '?':
                case ':':
                    usage(EXIT_FAILURE);
                default:
                    fprintf(stderr, "Ayieeee!\n");
                    abort();
            }
        }
    }

```

Defines:

`curr_opt`, never used.

`parse_command_line`, used in chunks 3 and 11.

Uses `PROG_NAME` 14, `PROG_VERSION` 14, and `usage` 9 10.

## 6 Miscellany

Include files. Nothing to see here.

```

13  <include files 13>≡ (1)
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <sys/ioctl.h>
    #include <net/if.h>
    #include <errno.h>
    #include <string.h>
    #include <stdlib.h>
    #include <netinet/tcp.h>
    #include <stdio.h>

```

## 7 Defines

Nothing too interesting here. Yes, the program is called waka-waka. Long live Fozzy! :-)

```
14  <macros and defines 14>≡ (1)
    #define PROG_NAME      "waka-waka"
    #define PROG_VERSION   "1.0"
    #define RBUF_SIZ       1024
```

Defines:

PROG\_NAME, used in chunks 10 and 12.

PROG\_VERSION, used in chunks 10 and 12.

RBUF\_SIZ, used in chunk 15.

### 7.1 Global data

Okay, I got lazy.

```
15  <bss variables 15>≡ (1)
    char buf[RBUF_SIZ];
```

Defines:

buf, used in chunk 8.

Uses RBUF\_SIZ 14.