Programming with pcap

Tim Carstens
timcarst **at** yahoo **dot** com
Further editing and development by Guy Harris
guy **at** alum **dot** mit **dot** edu

Ok, let's begin by defining who this document is written for. Obviously, some basic knowledge of C is required, unless you only wish to know the basic theory. You do not need to be a code ninja; for the areas likely to be understood only by more experienced programmers, I'll be sure to describe concepts in greater detail. Additionally, some basic understanding of networking might help, given that this is a packet sniffer and all. All of the code examples presented here have been tested on FreeBSD 4.3 with a default kernel.

Getting Started: The format of a pcap application

The first thing to understand is the general layout of a pcap sniffer. The flow of code is as follows:

1. We begin by determining which interface we want to sniff on. In Linux this may be something like eth0, in BSD it may be xl1, etc. We can either define this device in a string, or we can ask pcap to provide us with the name of an interface that will do the job.
2. Initialize pcap. This is where we actually tell pcap what device we are sniffing on. We can, if we want to, sniff on multiple devices. How do we differentiate between them? Using file handles. Just like opening a file for reading or writing, we must name our sniffing "session" so we can tell it apart from other such sessions.
3. In the event that we only want to sniff specific traffic (e.g.: only TCP/IP packets, only packets going to port 23, etc) we must create a rule set, "compile" it, and apply it. This is a three phase process, all of which is closely related. The rule set is kept in a string, and is converted into a format that pcap can read (hence compiling it.) The compilation is actually just done by calling a function within our program; it does not involve the use of an external application. Then we tell pcap to apply it to whichever session we wish for it to filter.
4. Finally, we tell pcap to enter it's primary execution loop. In this state, pcap waits until it has received however many packets we want it to. Every time it gets a new packet in, it calls another function that we have already defined. The function that it calls can do anything we want; it can dissect the packet and print it to the user, it can save it in a file, or it can do nothing at all.
5. After our sniffing needs are satisfied, we close our session and are complete.

This is actually a very simple process. Five steps total, one of which is optional (step 3, in case you were wondering.) Let's take a look at each of the steps and how to implement them.

Setting the device

This is terribly simple. There are two techniques for setting the device that we wish to sniff on.

The first is that we can simply have the user tell us. Consider the following program:

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
        char *dev = argv[1];

        printf("Device: %s\n", dev);
        return(0);
}
```

The user specifies the device by passing the name of it as the first argument to the program. Now the string "dev" holds the name of the interface that we will sniff on in a format that pcap can understand (assuming, of course, the user gave us a real interface).

The other technique is equally simple. Look at this program:

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
       char *dev, errbuf[PCAP_ERRBUF_SIZE];
```

```
        dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
                return(2);
        }
        printf("Device: %s\n", dev);
        return(0);
}
```

In this case, pcap just sets the device on its own. "But wait, Tim," you say. "What is the deal with the errbuf string?" Most of the pcap commands allow us to pass them a string as an argument. The purpose of this string? In the event that the command fails, it will populate the string with a description of the error. In this case, if pcap_lookupdev() fails, it will store an error message in errbuf. Nifty, isn't it? And that's how we set our device.

Opening the device for sniffing

The task of creating a sniffing session is really quite simple. For this, we use pcap_open_live(). The prototype of this function (from the pcap man page) is as follows:

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

The first argument is the device that we specified in the previous section. snaplen is an integer which defines the maximum number of bytes to be captured by pcap. promisc, when set to true, brings the interface into promiscuous mode (however, even if it is set to false, it is possible under specific cases for the interface to be in promiscuous mode, anyway). to_ms is the read time out in milliseconds (a value of 0 means no time out; on at least some platforms, this means that you may wait until a sufficient number of packets arrive before seeing any packets, so you should use a non-zero timeout). Lastly, ebuf is a string we can store any error messages within (as we did above with errbuf). The function returns our session handler.

To demonstrate, consider this code snippet:

```
#include <pcap.h>
...
pcap_t *handle;

handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        return(2);
}
```

This code fragment opens the device stored in the strong "dev", tells it to read however many bytes are specified in BUFSIZ (which is defined in pcap.h). We are telling it to put the device into promiscuous mode, to sniff until an error occurs, and if there is an error, store it in the string errbuf; it uses that string to print an error message.

A note about promiscuous vs. non-promiscuous sniffing: The two techniques are very different in style. In standard, non-promiscuous sniffing, a host is sniffing only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer. Promiscuous mode, on the other hand, sniffs all traffic on the wire. In a non-switched environment, this could be all network traffic. The obvious advantage to this is that it provides more packets for sniffing, which may or may not be helpful depending on the reason you are sniffing the network. However, there are regressions. Promiscuous mode sniffing is detectable; a host can test with strong reliability to determine if another host is doing promiscuous sniffing. Second, it only works in a non-switched environment (such as a hub, or a switch that is being ARP flooded). Third, on high traffic networks, the host can become quite taxed for system resources.

Not all devices provide the same type of link-layer headers in the packets you read. Ethernet devices, and some non-Ethernet devices, might provide Ethernet headers, but other device types, such as loopback devices in BSD and OS X, PPP interfaces, and Wi-Fi interfaces when capturing in monitor mode, don't.

You need to determine the type of link-layer headers the device provides, and use that type when processing the packet contents. The pcap_datalink() routine returns a value indicating the type of link-layer headers; see the list of link-layer header type values. The values it returns are the `DLT_` values in that list.

If your program doesn't support the link-layer header type provided by the device, it has to give up; this would be done with code such as

```
if (pcap_datalink(handle) != DLT_EN10MB) {
        fprintf(stderr, "Device %s doesn't provide Ethernet headers - not supported\n", dev);
```

```
        return(2);
}
```

which fails if the device doesn't supply Ethernet headers. This would be appropriate for the code below, as it assumes Ethernet headers.

Filtering traffic

Often times our sniffer may only be interested in specific traffic. For instance, there may be times when all we want is to sniff on port 23 (telnet) in search of passwords. Or perhaps we want to highjack a file being sent over port 21 (FTP). Maybe we only want DNS traffic (port 53 UDP). Whatever the case, rarely do we just want to blindly sniff *all* network traffic. Enter pcap_compile() and pcap_setfilter().

The process is quite simple. After we have already called pcap_open_live() and have a working sniffing session, we can apply our filter. Why not just use our own if/else if statements? Two reasons. First, pcap's filter is far more efficient, because it does it directly with the BPF filter; we eliminate numerous steps by having the BPF driver do it directly. Second, this is a *lot* easier :)

Before applying our filter, we must "compile" it. The filter expression is kept in a regular string (char array). The syntax is documented quite well in the man page for tcpdump; I leave you to read it on your own. However, we will use simple test expressions, so perhaps you are sharp enough to figure it out from my examples.

To compile the program we call pcap_compile(). The prototype defines it as:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
    bpf_u_int32 netmask)
```

The first argument is our session handle (pcap_t *handle in our previous example). Following that is a reference to the place we will store the compiled version of our filter. Then comes the expression itself, in regular string format. Next is an integer that decides if the expression should be "optimized" or not (0 is false, 1 is true. Standard stuff.) Finally, we must specify the network mask of the network the filter applies to. The function returns -1 on failure; all other values imply success.

After the expression has been compiled, it is time to apply it. Enter pcap_setfilter(). Following our format of explaining pcap, we shall look at the pcap_setfilter() prototype:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

This is very straightforward. The first argument is our session handler, the second is a reference to the compiled version of the expression (presumably the same variable as the second argument to pcap_compile()).

Perhaps another code sample would help to better understand:

```
#include <pcap.h>
...
pcap_t *handle;                 /* Session handle */
char dev[] = "rl0";             /* Device to sniff on */
char errbuf[PCAP_ERRBUF_SIZE]; /* Error string */
struct bpf_program fp;          /* The compiled filter expression */
char filter_exp[] = "port 23"; /* The filter expression */
bpf_u_int32 mask;               /* The netmask of our sniffing device */
bpf_u_int32 net;                /* The IP of our sniffing device */

if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
        fprintf(stderr, "Can't get netmask for device %s\n", dev);
        net = 0;
        mask = 0;
}
handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        return(2);
}
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return(2);
}
if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return(2);
}
```

This program preps the sniffer to sniff all traffic coming from or going to port 23, in promiscuous mode, on the device rl0.

You may notice that the previous example contains a function that we have not yet discussed. pcap_lookupnet() is a function that, given the name of a device, returns one of its IPv4 network numbers and corresponding network mask (the network number is the IPv4 address ANDed with the network mask, so it contains only the network part of the address). This was essential because we needed to know the network mask in order to apply the filter. This function is described in the Miscellaneous section at the end of the document.

It has been my experience that this filter does not work across all operating systems. In my test environment, I found that OpenBSD 2.9 with a default kernel does support this type of filter, but FreeBSD 4.3 with a default kernel does not. Your mileage may vary.

The actual sniffing

At this point we have learned how to define a device, prepare it for sniffing, and apply filters about what we should and should not sniff for. Now it is time to actually capture some packets.

There are two main techniques for capturing packets. We can either capture a single packet at a time, or we can enter a loop that waits for *n* number of packets to be sniffed before being done. We will begin by looking at how to capture a single packet, then look at methods of using loops. For this we use pcap_next().

The prototype for pcap_next() is fairly simple:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

The first argument is our session handler. The second argument is a pointer to a structure that holds general information about the packet, specifically the time in which it was sniffed, the length of this packet, and the length of his specific portion (incase it is fragmented, for example.) pcap_next() returns a u_char pointer to the packet that is described by this structure. We'll discuss the technique for actually reading the packet itself later.

Here is a simple demonstration of using pcap_next() to sniff a packet.

```
#include <pcap.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        pcap_t *handle;                 /* Session handle */
        char *dev;                      /* The device to sniff on */
        char errbuf[PCAP_ERRBUF_SIZE];  /* Error string */
        struct bpf_program fp;          /* The compiled filter */
        char filter_exp[] = "port 23";  /* The filter expression */
        bpf_u_int32 mask;               /* Our netmask */
        bpf_u_int32 net;                /* Our IP */
        struct pcap_pkthdr header;      /* The header that pcap gives us */
        const u_char *packet;           /* The actual packet */

        /* Define the device */
        dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
                return(2);
        }
        /* Find the properties for the 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;
        }
        /* Open the session in promiscuous mode */
        handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
        if (handle == NULL) {
                fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
                return(2);
        }
        /* Compile and apply the filter */
        if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
                fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp, pcap_geterr(handle));
                return(2);
        }
        if (pcap_setfilter(handle, &fp) == -1) {
                fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp, pcap_geterr(handle))
                return(2);
```

```
        }
        /* Grab a packet */
        packet = pcap_next(handle, &header);
        /* Print its length */
        printf("Jacked a packet with length of [%d]\n", header.len);
        /* And close the session */
        pcap_close(handle);
        return(0);
}
```

This application sniffs on whatever device is returned by pcap_lookupdev() by putting it into promiscuous mode. It finds the first packet to come across port 23 (telnet) and tells the user the size of the packet (in bytes). Again, this program includes a new call, pcap_close(), which we will discuss later (although it really is quite self explanatory).

The other technique we can use is more complicated, and probably more useful. Few sniffers (if any) actually use pcap_next(). More often than not, they use pcap_loop() or pcap_dispatch() (which then themselves use pcap_loop()). To understand the use of these two functions, you must understand the idea of a callback function.

Callback functions are not anything new, and are very common in many API's. The concept behind a callback function is fairly simple. Suppose I have a program that is waiting for an event of some sort. For the purpose of this example, let's pretend that my program wants a user to press a key on the keyboard. Every time they press a key, I want to call a function which then will determine that to do. The function I am utilizing is a callback function. Every time the user presses a key, my program will call the callback function. Callbacks are used in pcap, but instead of being called when a user presses a key, they are called when pcap sniffs a packet. The two functions that one can use to define their callback is pcap_loop() and pcap_dispatch(). pcap_loop() and pcap_dispatch() are very similar in their usage of callbacks. Both of them call a callback function every time a packet is sniffed that meets our filter requirements (if any filter exists, of course. If not, then *all* packets that are sniffed are sent to the callback.)

The prototype for pcap_loop() is below:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

The first argument is our session handle. Following that is an integer that tells pcap_loop() how many packets it should sniff for before returning (a negative value means it should sniff until an error occurs). The third argument is the name of the callback function (just its identifier, no parentheses). The last argument is useful in some applications, but many times is simply set as NULL. Suppose we have arguments of our own that we wish to send to our callback function, in addition to the arguments that pcap_loop() sends. This is where we do it. Obviously, you must typecast to a u_char pointer to ensure the results make it there correctly; as we will see later, pcap makes use of some very interesting means of passing information in the form of a u_char pointer. After we show an example of how pcap does it, it should be obvious how to do it here. If not, consult your local C reference text, as an explanation of pointers is beyond the scope of this document. pcap_dispatch() is almost identical in usage. The only difference between pcap_dispatch() and pcap_loop() is that pcap_dispatch() will only process the first batch of packets that it receives from the system, while pcap_loop() will continue processing packets or batches of packets until the count of packets runs out. For a more in depth discussion of their differences, see the pcap man page.

Before we can provide an example of using pcap_loop(), we must examine the format of our callback function. We cannot arbitrarily define our callback's prototype; otherwise, pcap_loop() would not know how to use the function. So we use this format as the prototype for our callback function:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet);
```

Let's examine this in more detail. First, you'll notice that the function has a void return type. This is logical, because pcap_loop() wouldn't know how to handle a return value anyway. The first argument corresponds to the last argument of pcap_loop(). Whatever value is passed as the last argument to pcap_loop() is passed to the first argument of our callback function every time the function is called. The second argument is the pcap header, which contains information about when the packet was sniffed, how large it is, etc. The pcap_pkthdr structure is defined in pcap.h as:

```
struct pcap_pkthdr {
        struct timeval ts; /* time stamp */
        bpf_u_int32 caplen; /* length of portion present */
        bpf_u_int32 len; /* length this packet (off wire) */
};
```

These values should be fairly self explanatory. The last argument is the most interesting of them all, and the most confusing to the average novice pcap programmer. It is another pointer to a u_char, and it points to the first byte of a chunk of data containing the entire packet, as sniffed by pcap_loop().

But how do you make use of this variable (named "packet" in our prototype)? A packet contains many attributes, so as you can imagine, it is not really a string, but actually a collection of structures (for instance, a TCP/IP packet would have an Ethernet header, an IP header, a TCP header, and lastly, the packet's payload). This u_char pointer points to the serialized version of these structures. To make any use of it, we must do some interesting typecasting.

First, we must have the actual structures define before we can typecast to them. The following are the structure definitions that I use to describe a TCP/IP packet over Ethernet.

```
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

        /* Ethernet header */
        struct sniff_ethernet {
                u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
                u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
                u_short ether_type; /* IP? ARP? RARP? etc */
        };

        /* IP header */
        struct sniff_ip {
                u_char ip_vhl;          /* version << 4 | header length >> 2 */
                u_char ip_tos;          /* type of service */
                u_short ip_len;         /* total length */
                u_short ip_id;          /* identification */
                u_short ip_off;         /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
                u_char ip_ttl;          /* time to live */
                u_char ip_p;            /* protocol */
                u_short ip_sum;         /* checksum */
                struct in_addr ip_src,ip_dst; /* source and dest address */
        };
        #define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
        #define IP_V(ip)                (((ip)->ip_vhl) >> 4)

        /* TCP header */
        typedef u_int tcp_seq;

        struct sniff_tcp {
                u_short th_sport;       /* source port */
                u_short th_dport;       /* destination port */
                tcp_seq th_seq;         /* sequence number */
                tcp_seq th_ack;         /* acknowledgement number */
                u_char th_offx2;        /* data offset, rsvd */
        #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
                u_char th_flags;
        #define TH_FIN 0x01
        #define TH_SYN 0x02
        #define TH_RST 0x04
        #define TH_PUSH 0x08
        #define TH_ACK 0x10
        #define TH_URG 0x20
        #define TH_ECE 0x40
        #define TH_CWR 0x80
        #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
                u_short th_win;         /* window */
                u_short th_sum;         /* checksum */
                u_short th_urp;         /* urgent pointer */
};
```

So how does all of this relate to pcap and our mysterious u_char pointer? Well, those structures define the headers that appear in the data for the packet. So how can we break it apart? Be prepared to witness one of the most practical uses of pointers (for all of those new C programmers who insist that pointers are useless, I smite you).

Again, we're going to assume that we are dealing with a TCP/IP packet over Ethernet. This same technique applies to any packet; the only difference is the structure types that you actually use. So let's begin by defining the variables and compile-time definitions we will need to deconstruct the packet data.

```
/* ethernet headers are always exactly 14 bytes */
#define SIZE_ETHERNET 14

        const struct sniff_ethernet *ethernet; /* The ethernet header */
        const struct sniff_ip *ip; /* The IP header */
        const struct sniff_tcp *tcp; /* The TCP header */
        const char *payload; /* Packet payload */

        u_int size_ip;
        u_int size_tcp;
```

And now we do our magical typecasting:

```
        ethernet = (struct sniff_ethernet*)(packet);
        ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
        size_ip = IP_HL(ip)*4;
        if (size_ip < 20) {
                printf("   * Invalid IP header length: %u bytes\n", size_ip);
                return;
        }
        tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
        size_tcp = TH_OFF(tcp)*4;
        if (size_tcp < 20) {
                printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
                return;
        }
        payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
```

How does this work? Consider the layout of the packet data in memory. The u_char pointer is really just a variable containing an address in memory. That's what a pointer is; it points to a location in memory.

For the sake of simplicity, we'll say that the address this pointer is set to is the value X. Well, if our three structures are just sitting in line, the first of them (sniff_ethernet) being located in memory at the address X, then we can easily find the address of the structure after it; that address is X plus the length of the Ethernet header, which is 14, or SIZE_ETHERNET.

Similarly if we have the address of that header, the address of the structure after it is the address of that header plus the length of that header. The IP header, unlike the Ethernet header, does **not** have a fixed length; its length is given, as a count of 4-byte words, by the header length field of the IP header. As it's a count of 4-byte words, it must be multiplied by 4 to give the size in bytes. The minimum length of that header is 20 bytes.

The TCP header also has a variable length; its length is given, as a number of 4-byte words, by the "data offset" field of the TCP header, and its minimum length is also 20 bytes.

So let's make a chart:

Variable	Location (in bytes)
sniff_ethernet	X
sniff_ip	X + SIZE_ETHERNET
sniff_tcp	X + SIZE_ETHERNET + {IP header length}
payload	X + SIZE_ETHERNET + {IP header length} + {TCP header length}

The sniff_ethernet structure, being the first in line, is simply at location X. sniff_ip, who follows directly after sniff_ethernet, is at the location X, plus however much space the Ethernet header consumes (14 bytes, or SIZE_ETHERNET). sniff_tcp is after both sniff_ip and sniff_ethernet, so it is location at X plus the sizes of the Ethernet and IP headers (14 bytes, and 4 times the IP header length, respectively). Lastly, the payload (which doesn't have a single structure corresponding to it, as its contents depends on the protocol being used atop TCP) is located after all of them.

So at this point, we know how to set our callback function, call it, and find out the attributes about the packet that has been sniffed. It's now the time you have been waiting for: writing a useful packet sniffer. Because of the length of the source code, I'm not going to include it in the body of this document. Simply download sniffex.c and try it out.

Wrapping Up

At this point you should be able to write a sniffer using pcap. You have learned the basic concepts behind opening a pcap session, learning general attributes about it, sniffing packets, applying filters, and using callbacks. Now it's time to get out there and sniff those wires!