The Sniffer's Guide to Raw Traffic

(a libpcap tutorial)

- Download **libpcap** source from www.tcpdump.org here
- Download **libpcap** for win32 from www.winpcap.org
- Check out a better pcap tutorial here

Front matter: This is a slightly modified and extended version of my older pcap tutorial. Revisiting this work five years later, I am necessarily dumber (age and beer) yet hopefully somewhat more knowledgeable. Contact information has changed, please send your hate-mail to casado *at* cs.stanford.edu.

Contents

Who this is for: This tutorial assumes a cursory knowledge in networks; what a packet is, Ethernet vs. IP vs. TCP vs. UDP etc. If these concepts are foreign I highly suggest you invest in a **good** (e.g. probably can't find at Best Buy) networking book. My favorites are:

- Computer Networking : A Top-Down Approach Featuring the Internet (3rd Edition) by James F. Kurose, Keith W. Ross
- UNIX Network Programming by W. Richard Stevens
- The Protocols (TCP/IP Illustrated, Volume 1) by W. Richard Stevens

This tutorial does not assume any previous knowledge in network programming, just a basic familiarity with c. If you already are a c/c++ master, then you might as well just **man 3 pcap**. You should have a working c compiler on your system and libpcap installed. All source in this section was written and tested on linux, kernel 2.2.14, while it should be mostly portable (hehe) I can't guarantee that it will compile or run on other operating systems. You are going to want to run as root so be careful and be sure not to break your box in the meantime. Oh, and though I have tested and run all the code presented in this tutorial with no problems, I am NOT responsible if your shit breaks and has to be quarantined by the health department... aka play at your own risk....

Intro: Finally, you've made it (either by reading, skimming or skipping) to the start of the tutorial. We'll start at the verryyy begining and define a few thing before getting into the nity-grity -- howver if you are eager to get moving, scroll to the bottom of this page, cut, paste, compile and enjoy. For the rest of you, the following two definition may give you a clue about what we are doing, what the tools we will be using.

- **Packet Capture** Roughly means, *to grab a copy of packets off of the wire before they are processed by the operating system*. Why would one want to do this? Well, its cool. More practically, packet capture is widely used in network security tools to analyze raw traffic for detecting malicious behaviour (scans and attacks), sniffing, fingerprinting and many other (often devious) uses.
- **libpcap** "provides implementation-independent access to the underlying packet capture facility provided by the operating system" (Stevens, UNP page. 707). So pretty much, libpcap is the library we are going to use to grab packets right as they come off of the network card.

Getting Started Well there is an awful lot to cover.. so lets just get familiar with **libpcap**. All the examples in this tutorial assume that you are sitting on an Ethernet. If this is not the case, then the basics are still relevant, but the code presented later on involving decoding the Ethernet header obviously isn't :-(*sorry*. Allright... crack your knuckles *crunch* and lets get ready to code our **FIRST LIBPCAP PROGRAM :)**. Go ahead and copy the following program into your favorite editor (which should be **vim** if you have any sense :-) save, and compile with...

%>gcc ldev.c -lpcap

```
/* ldev.c
   Martin Casado

   To compile:
   >gcc ldev.c -lpcap

   Looks for an interface, and lists the network ip
   and mask associated with that interface.
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>  /* GIMME a libpcap plz! */
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
  char *dev; /* name of the device to use */
  char *net; /* dot notation of the network address */
  char *mask;/* dot notation of the network mask    */
  int ret;   /* return code */
  char errbuf[PCAP_ERRBUF_SIZE];
  bpf_u_int32 netp; /* ip          */
  bpf_u_int32 maskp;/* subnet mask */
  struct in_addr addr;

  /* ask pcap to find a valid device for use to sniff on */
  dev = pcap_lookupdev(errbuf);

  /* error checking */
  if(dev == NULL)
  {
   printf("%s\n",errbuf);
   exit(1);
  }

  /* print out device name */
  printf("DEV: %s\n",dev);

  /* ask pcap for the network address and mask of the device */
  ret = pcap_lookupnet(dev,&netp,&maskp,errbuf);

  if(ret == -1)
  {
   printf("%s\n",errbuf);
   exit(1);
  }

  /* get the network address in a human readable form */
  addr.s_addr = netp;
  net = inet_ntoa(addr);

  if(net == NULL)/* thanks Scott :-P */
  {
    perror("inet_ntoa");
    exit(1);
  }

  printf("NET: %s\n",net);

  /* do the same as above for the device's mask */
  addr.s_addr = maskp;
  mask = inet_ntoa(addr);

  if(mask == NULL)
  {
    perror("inet_ntoa");
    exit(1);
  }

  printf("MASK: %s\n",mask);

  return 0;
}
```

Did you run the program? If not, run it :-) Assuming it compiled, and ran correctly your output should be something like...

```
DEV: eth0
NET: 192.168.12.0
MASK: 255.255.255.0
```

The value for DEV is your default interface name (likely eth0 on linux, could be eri0 on solaris). The NET and MASK values are your primary interface's subnet and subnet mask. Don't know what those are? Might want to read this
.

"So what did we just do?", you ask. Well, we just asked libpcap to give us some specs on an interface to listen on.
"Whats an interface?"
Just think of an interface as your computers hardware connection to whatever network your computer is connected to. On Linux, eth0 denotes the first Ethernet card in your computer. (btw you can list all of your interfaces using the **ifconfig** command).

OK at this point we can compile a pcap program that essentially does nothing. On to grabbing our first packet ...

[Next]

Capturing Our First Packet

Well now we sort of know the nature of packet capture, we have identified that we do in fact have an interface to pull things from, how about we go ahead and grab a packet!
"Just give me the damn example and let me hack...", you cry
Very well..... Here you go.. download from here.. testpcap1.c or just cut and paste below.

```
/***************************************************
* file:     testpcap1.c
* Date:     Thu Mar 08 17:14:36 MST 2001
* Author:   Martin Casado
* Location: LAX Airport (hehe)
*
* Simple single packet capture program
*****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h> /* if this gives you an error try pcap/pcap.h */
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h> /* includes net/ethernet.h */

int main(int argc, char **argv)
{
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;     /* pcap.h */
    struct ether_header *eptr;  /* net/ethernet.h */

    u_char *ptr; /* printing out hardware header info */

    /* grab a device to peak into... */
    dev = pcap_lookupdev(errbuf);

    if(dev == NULL)
    {
        printf("%s\n",errbuf);
        exit(1);
    }

    printf("DEV: %s\n",dev);

    /* open the device for sniffing.

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

       snaplen - maximum size of packets to capture in bytes
       promisc - set card in promiscuous mode?
       to_ms   - time to wait for packets in miliseconds before read
       times out
       errbuf  - if something happens, place error string here

       Note if you change "prmisc" param to anything other than zero, you will
       get all packets your device sees, whether they are intendeed for you or
       not!! Be sure you know the rules of the network you are running on
       before you set your card in promiscuous mode!!     */

    descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf);

    if(descr == NULL)
    {
        printf("pcap_open_live(): %s\n",errbuf);
        exit(1);
    }



    /*
       grab a packet from descr (yay!)
       u_char *pcap_next(pcap_t *p,struct pcap_pkthdr *h)
       so just pass in the descriptor we got from
       our call to pcap_open_live and an allocated
       struct pcap_pkthdr                                 */

    packet = pcap_next(descr,&hdr);

    if(packet == NULL)
    {/* dinna work *sob* */
        printf("Didn't grab packet\n");
        exit(1);
    }

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

    printf("Grabbed packet of length %d\n",hdr.len);
    printf("Recieved at ..... %s\n",ctime((const time_t*)&hdr.ts.tv_sec));
    printf("Ethernet address length is %d\n",ETHER_HDR_LEN);

    /* lets start with the ether header... */
    eptr = (struct ether_header *) packet;

    /* Do a couple of checks to see what packet type we have..*/
    if (ntohs (eptr->ether_type) == ETHERTYPE_IP)
    {
        printf("Ethernet type hex:%x dec:%d is an IP packet\n",
                ntohs(eptr->ether_type),
                ntohs(eptr->ether_type));
    }else  if (ntohs (eptr->ether_type) == ETHERTYPE_ARP)
    {
        printf("Ethernet type hex:%x dec:%d is an ARP packet\n",
                ntohs(eptr->ether_type),
                ntohs(eptr->ether_type));
    }else {
        printf("Ethernet type %x not IP", ntohs(eptr->ether_type));
        exit(1);
    }

    /* copied from Steven's UNP */
    ptr = eptr->ether_dhost;
    i = ETHER_ADDR_LEN;
    printf(" Destination Address:  ");
    do{
        printf("%s%x",(i == ETHER_ADDR_LEN) ? " " : ":",*ptr++);
    }while(--i>0);
    printf("\n");

    ptr = eptr->ether_shost;
    i = ETHER_ADDR_LEN;
    printf(" Source Address:  ");
    do{
        printf("%s%x",(i == ETHER_ADDR_LEN) ? " " : ":",*ptr++);
    }while(--i>0);
    printf("\n");

    return 0;
}
```

Well, that wasn't too bad was it?! Lets give her a test run ..

```
[root@pepe libpcap]# ./a.out
DEV: eth0
Grabbed packet of length 76
Recieved at time..... Mon Mar 12 22:23:29 2001

Ethernet address length is 14
Ethernet type hex:800 dec:2048 is an IP packet
 Destination Address:   0:20:78:d1:e8:1
 Source Address:   0:a0:cc:56:c2:91
[root@pepe libpcap]#
```

After typing **a.out** I jumped into another terminal and tried to ping www.google.com. The output captured the ICMP packet used to ping www.google.com. If you don't know exactly what goes on under the covers of a network you may be curios how the computer obtained the destination ethernet address. Aha! You don't actually think that the destination address of the ethernet packet is the same as the machine at www.google.com do you!? The destination address is the next hop address of the packet, most likely your network gateway ... aka the computer that ties your network to the internet. The packet must first find its way to your gateway which will then forward it to the next hop based on ist routing table. Lets do a quick sanity check to see if we in fact are sending to the gateway .... You can use the **route** command to look at your local computer's routing table. The routing table will tell you the next hop for each destination. The last entry (default) is for all packets not sent locally (127 subnet) or to the 192.16.1 subnet. These packets are forwarded to 192.168.1.1.

```
[root@pepe libpcap]# /sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
```

```
192.168.1.0     *               255.255.255.0   U     0      0        0 eth0
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
default         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
```

we can then use the **arp**command determine the hardware address for 192.168.1.1.

```
[root@pepe libpcap]# /sbin/arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:20:78:D1:E8:01   C                     eth0
```

If your gateway is not in your arp cache, try and ping it, and then retry the arp command. The point is this, in order for your computer to send the packet it must first get the MAC address of the next hop (00:20:78:D1:E8:01 for my network).

An obvious follow-up question is, "how did my computer know the gateway hardware address"? Let me then digress for a moment. My computer knows the IP address of the gateway. As you can see from the handy-dandy **arp** command there is an internal table (the arp cache) which maps IP addresses to hardware addresses.

Hardware addresses on ethernet are obtained using the Address Resolution Protocol or **ARP**. ARP is is described in RFC826 which can be found... Here! It works as follows. If my computer wants to know the hardware address for the computer with IP 1.2.3.4, it sends and ARP request packet to Ethernet broadcast out of the Interface which 1.2.3.4. as attached. All computers connected to this interface (including 1.2.3.4) should recevie the packet and process the requests. However, only 1.2.3.4 should issue a reply which will contain its Ethernet address. On receipt of the reply, my computer will "cache" out the hardware address for all subsequent packets sent to 1.2.3.4 (until the cache entry times out). ARP packets are of Thernet type... **ETHERTYPE_ARP** which is defined in net/ethernet.h as follows.

```
#define ETHERTYPE_ARP           0x0806          /* Address resolution */
```

You can force an Ethernet ARP request by clearing your computer's ARP cache. Below I do this, and then run the above program again to grab the outgoing ARP request.

```
[root@pepe libpcap]# /sbin/arp -n    # look at arp cache
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:20:78:D1:E8:01   C                     eth0

[root@pepe libpcap]# /sbin/arp -n -d 192.168.1.1  #delete gateqay entrance
[root@pepe libpcap]# /sbin/arp -n   #make sure gateway hardware addy is empty
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1                      (incomplete)                              eth0
[root@pepe libpcap]# ./a.out
DEV: eth0
Grabbed packet of length 42
Recieved at time..... Tue Mar 13 00:36:49 2001

Ethernet address length is 14
Ethernet type hex:806 dec:2054 is an ARP packet
```

```
 Destination Address:   ff:ff:ff:ff:ff:ff
 Source Address:   0:a0:cc:56:c2:91
[root@pepe libpcap]
```

So as you can see, once the hardware address was removed the the cache, my computer needed to send an arp request to broadcast (i.e. ff:ff:ff:ff:ff:ff) looking for the owner of the higher level address, in this case IP 192.168.1.1. What do you think would happen if you cleared your arp cache and modified testpcap1.c to capture 2 packets?! Hey I know why don't you try it :-P~~~~

Lets now disect the packet by checking out <net/ethernet.h> right now we are not concerned with the network or transport protocol, we just want to peer into the ethernet headers.... Lets say that we are runnig at 10Mb/s...

```
/* 10Mb/s ethernet header */
struct ether_header
{
  u_int8_t  ether_dhost[ETH_ALEN];      /* destination eth addr */
  u_int8_t  ether_shost[ETH_ALEN];      /* source ether addr    */
  u_int16_t ether_type;                 /* packet type ID field */
} __attribute__ ((__packed__));
```

So it looks like the first ETH_ALEN bytes are the destination ethernet address (look at linux/if_ether.h for the definition of ETH_ALEN :-) of the packet (presumedly your machine). The next ETH_ALEN bytes are the source. Finally, the last word is the packet type. Here are the protocol ID's on my machine from net/ethernet.h

```
/* Ethernet protocol ID's */
#define ETHERTYPE_PUP           0x0200      /* Xerox PUP */
#define ETHERTYPE_IP            0x0800          /* IP */
#define ETHERTYPE_ARP           0x0806          /* Address resolution */
#define ETHERTYPE_REVARP        0x8035          /* Reverse ARP */
```

For the purpose of this tutorial I will be focusing on IP and perhaps a little bit on ARP... the truth is I have no idea what the hell Xerox PUP is.

Allright so where are we now? We know the most basic of methods for grabbing a packet. We covered how hardware addresses are resolved and what a basic ethernet packet looks like. Still we are using a ver small subset of the functionality of libpcap, and we haven't even begun to peer into the packets themselves (other than the hardware headers) so much to do and so little time :-) As you can probably tell by now, it would be near impossible to do any real protocol analysis with a program that simply captures one packet at a time. What we really want to do is write a simple packet capturing engine that will nab as many packets as possible while filtering out those we dont want. In the next section we will construct a simple packet capturing engine which will aid us in packet dissection later on.

[prev] [next]

Writing a Basic Packet Capture Engine

Hi :-), this section consists of a discussion on how to write a simple packet capture engine. The goal is to demonstrate methods of capturing and filtering multiple packets to aid in packet analysis. All the juicy info on disecting IP packets and forging new ones are reserved for later sections.. Yes I can see your dissapointment, but you must admit that a program that captures a single packet at a time is pretty much useless.

We'll start by looking at:

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

When **pcap_loop(..)** is called it will grab *cnt* packets (it will loop infinitely when cnt is -1) and pass them to the callback function which is of type **pcap_handler**. and what is pcap handler?? well lets go to the handy dandy header file..

```
typedef void (*pcap_handler)(u_char *arg, const struct pcap_pkthdr *, const u_char *);
```

We are interested in arguments 2 and 3, the pcap packet header and a const u_char consisting of the packet. The first argument (*arg*) is a pointer to data that you passed to *pcap_loop* initially as the last argument. This is used to pass data to the packet processing routine without having to resort to globals.

As a primer, lets write a q&d program that will loop and get *n* packets, then exit.

Download testpcap2.c Here or just cut and paste from below

```
/**********************************************************************
* file:   testpcap2.c
* date:   2001-Mar-14 12:14:19 AM
* Author: Martin Casado
* Last Modified:2001-Mar-14 12:14:11 AM
*
* Description: Q&D proggy to demonstrate the use of pcap_loop
*
**********************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

/* callback function that is passed to pcap_loop(..) and called each time
 * a packet is recieved                                                    */
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    static int count = 1;
    fprintf(stdout,"%d, ",count);
    if(count == 4)
        fprintf(stdout,"Come on baby sayyy you love me!!! ");
    if(count == 7)
        fprintf(stdout,"Tiiimmmeesss!! ");
    fflush(stdout);
    count++;
}

int main(int argc,char **argv)
{
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;     /* pcap.h */
    struct ether_header *eptr;  /* net/ethernet.h */

    if(argc != 2){ fprintf(stdout,"Usage: %s numpackets\n",argv[0]);return 0;}

    /* grab a device to peak into... */
    dev = pcap_lookupdev(errbuf);
    if(dev == NULL)
    { printf("%s\n",errbuf); exit(1); }
    /* open device for reading */
    descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf);
    if(descr == NULL)
    { printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    /* allright here we call pcap_loop(..) and pass in our callback function */
    /* int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)*/
    /* If you are wondering what the user argument is all about, so am I!!   */
    pcap_loop(descr,atoi(argv[1]),my_callback,NULL);

    fprintf(stdout,"\nDone processing packets... wheew!\n");
    return 0;
}
```

Allright then, lets give her a whirl!

```
[root@pepe libpcap]# gcc testpcap2.c -lpcap
[root@pepe libpcap]# ./a.out 7
 1, 2, 3, 4, Come on baby sayyy you love me!!! 5, 6, 7, Tiiimmmeesss!!
```

```
Done processing packets... wheew!
[root@pepe libpcap]#
```

So as you can see, **my_callback(...)** was actually called 7 times before exiting. If you are testing your program by pinging an external machine the packets come slow enough to see them arrive in real time.. We could certainly put all of our packet analysis code in my_callback(..) and call it is done deal. But as good little coders we certainly aren't satisfied with such an easy and straightforward solution! The first problem is that pcap_loop(..) blocks indefinatly if no packet can be read. While this may be the desired behaviour it would be nice to timeout on the reads. Remember way back when we talked about **pcap_open_live(..)**? One of the arguments you can specify is a timeout value in miliseconds. **pcap_loop** actually ignores this argument, but **pcap_dispatch(..)** doesn't! So if we want our main looping mechanism to time-out replace pcap_loop() with pcap_dispatch(). Here is a description of pcap_dispatch(..) shamelessly stripped from the man page

****
pcap_dispatch() is used to collect and process packets. cnt specifies the maximum number of packets to process before returning. A cnt of -1 processes all the packets received in one buffer. A cnt of 0 processes all packets until an error occurs, EOF is reached, or the read times out (when doing live reads and a non-zero read timeout is specified). callback specifies a routine to be called with three arguments: a u_char pointer which is passed in from pcap_dispatch(), a pointer to the pcap_pkthdr struct (which precede the actual network headers and data), and a u_char pointer to the packet data. The number of packets read is returned. Zero is returned when EOF is reached in a ``savefile.'' A return of -1 indicates an error in which case pcap_perror() or pcap_geterr() may be used to display the error text.
****

In many applications using packet capture, you are not going to be interested in every packet on your network. Take the following scenario. Little Johny just bought the coolest new internet game to hit the markets. Little Johny wants to be the first kid to hack up a bot for the game, but unlike all other little kiddies, Johny is going to write his own packet capture engine instead of using something canned. Little Johnny uses tcpdump and notices that when the game starts up and he connects to the server.. it is connecting to 216.148.0.87 on port 26112. What should little Johnny do to only capture packets to or from 216.148.0.87 port 26112? Enter... **pcap_compile(..)** and **pcap_setfilter(...)** !!!

Note that we could read in all packets, sort through them one by one to pick out the subset we are interested in. However, each callback into user space is (of course) a kernel crossing and could become quite expenise. Luckily libpcap provides an interface where you can specify exactly which packets you are interested as bpf (berkeley packet filter) programs. In brief, to do this you need to pass a filter program as a string to pcap_compile() and then set it as a filter.... the problem is that the pcap man page doesn't provide any detail of what the filter program should look like (at least mine doesn't). Is all lost!? No! because we have the handy dandy program **tcpdump** and its man page. You should have tcpdump already installed on your machine (which tcpdump) but if you don't I highly suggest you put it on. Tcpdump is pretty much a wrapper of libpcap. What is useful to us at the moment is that it accepts filter programs from the command line! Aha! a reference. The tcpdump man page explicitly describes the syntax and semantics of the filter language, which is (of course) pretty straight forward. Here are the pertinent sections from my man pages..

```
The program consists of one or more primitives.  Primitives usu�
ally  consist  of  an  id  (name or number) preceded by one or more
qualifiers.  There are three different kinds of qualifier:

type   qualifiers say what kind of thing  the  id  name  or  number
       refers  to.   Possible  types are host, net and port.  E.g.,
       `host foo', `net 128.3', `port 20'.  If  there  is  no  type
       qualifier, host is assumed.

dir    qualifiers specify a particular transfer direction to and/or
       from id.  Possible directions are src, dst, src or  dst  and
       src  and dst.  E.g., `src foo', `dst net 128.3', `src or dst
       port ftp-data'.  If there is no dir qualifier, src or dst is
       assumed.  For `null' link layers (i.e. point to point proto�
       cols such as slip) the inbound and outbound  qualifiers  can
       be used to specify a desired direction.

proto  qualifiers  restrict  the  match  to  a particular protocol.
       Possible protos are: ether, fddi,  ip,  arp,  rarp,  decnet,
       lat, sca, moprc, mopdl, tcp and udp.  E.g., `ether src foo',
       `arp net 128.3', `tcp port 21'.  If there is no proto quali�
       fier,  all  protocols  consistent with the type are assumed.
       E.g., `src foo' means `(ip or arp or rarp) src foo'  (except
       the latter is not legal syntax), `net bar' means `(ip or arp
       or rarp) net bar' and `port 53' means  `(tcp  or  udp)  port
       53'.

In  addition  to the above, there are some special `primitive' key�
words that don't follow  the  pattern:  gateway,  broadcast,  less,
greater  and  arithmetic  expressions.   All of these are described
below.

More complex filter expressions are built up  by  using  the  words
and,  or  and  not  to combine primitives.  E.g., `host foo and not
port ftp and not port ftp-data'.  To save typing, identical  quali�
fier  lists can be omitted.  E.g., `tcp dst port ftp or ftp-data or
domain' is exactly the same as `tcp dst port ftp or  tcp  dst  port
ftp-data or tcp dst port domain'.

Allowable primitives are:

dst host host
       True  if  the  IP  destination  field of the packet is host,
       which may be either an address or a name.

src host host
       True if the IP source field of the packet is host.

host host
       True if either the IP source or destination of the packet is
       host.   Any  of  the above host expressions can be prepended
       with the keywords, ip, arp, or rarp as in:
```

```
                   ip host host
              which is equivalent to:
                   ether proto \ip and host host
              If host is a name with multiple IP addresses,  each  address
              will be checked for a match.

       ether dst ehost
              True  if  the  ethernet destination address is ehost.  Ehost
              may be either a name  from  /etc/ethers  or  a  number  (see
              ethers(3N) for numeric format).

       ether src ehost
              True if the ethernet source address is ehost.

       ether host ehost
              True if either the ethernet source or destination address is
              ehost.

       gateway host
              True if the packet used host as a gateway.  I.e., the ether�
              net  source  or destination address was host but neither the
              IP source nor the IP destination was host.  Host must  be  a
              name  and  must be found in both /etc/hosts and /etc/ethers.
              (An equivalent expression is
                   ether host ehost and not host host
              which can be used with either names or numbers  for  host  /
              ehost.)

       dst net net
              True  if the IP destination address of the packet has a net�
              work number of net. Net may be either a name from  /etc/net�
              works or a network number (see networks(4) for details).

       src net net
              True  if  the  IP source address of the packet has a network
              number of net.

       net net
              True if either the IP source or destination address  of  the
              packet has a network number of net.

       net net mask mask
              True  if  the  IP address matches net with the specific net�
              mask.  May be qualified with src or dst.

       net net/len
              True if the IP address matches net a netmask len bits  wide.
              May be qualified with src or dst.

       dst port port
              True if the packet is ip/tcp or ip/udp and has a destination
              port value of port.  The port can be a number or a name used
              in  /etc/services  (see  tcp(4P) and udp(4P)).  If a name is
              used, both the port number and protocol are checked.   If  a
              number  or  ambiguous  name is used, only the port number is
              checked (e.g., dst port 513 will print both tcp/login  traf�
              fic  and  udp/who  traffic,  and port domain will print both
              tcp/domain and udp/domain traffic).

       src port port
              True if the packet has a source port value of port.

       port port
              True if either the source or destination port of the  packet
              is port.  Any of the above port expressions can be prepended
              with the keywords, tcp or udp, as in:
                   tcp src port port
              which matches only tcp packets whose source port is port.

       less length
              True if the packet has  a  length  less  than  or  equal  to
              length.  This is equivalent to:
                   len <= length.

       greater length
              True  if  the  packet  has a length greater than or equal to
              length.  This is equivalent to:
                   len >= length.

       ip proto protocol
              True if the packet is an ip packet (see ip(4P)) of  protocol
              type protocol.  Protocol can be a number or one of the names
              icmp, igrp, udp, nd, or tcp.  Note that the identifiers tcp,
              udp,  and  icmp  are  also  keywords and must be escaped via
              backslash (\), which is \\ in the C-shell.

       ether broadcast
              True if the packet is an  ethernet  broadcast  packet.   The
              ether keyword is optional.

       ip broadcast
              True if the packet is an IP broadcast packet.  It checks for
              both the all-zeroes and all-ones broadcast conventions,  and
              looks up the local subnet mask.

       ether multicast
              True  if  the  packet  is an ethernet multicast packet.  The
              ether keyword is optional.  This is shorthand for  `ether[0]
              & 1 != 0'.
              ip multicast
                     True if the packet is an IP multicast packet.

              ether proto protocol
                     True  if the packet is of ether type protocol.  Protocol can
                     be a number or a name like ip, arp,  or  rarp.   Note  these
                     identifiers  are also keywords and must be escaped via back�
                     slash (\).  [In the  case  of  FDDI  (e.g.,  `fddi  protocol
                     arp'), the protocol identification comes from the 802.2 Log�
                     ical Link Control (LLC) header, which is usually layered  on
                     top  of the FDDI header.  Tcpdump assumes, when filtering on
                     the protocol identifier, that all FDDI  packets  include  an
                     LLC  header,  and  that  the LLC header is in so-called SNAP
                     format.]

              ip, arp, rarp, decnet
                     Abbreviations for:
                          ether proto p where p is one of the above protocols.

              tcp, udp, icmp
                     Abbreviations for:
                          ip proto p
                     where p is one of the above protocols.

              expr relop expr
                     True  if the relation holds, where relop is one of >,
                     <, >=, <=, =, !=, and expr is an arithmetic
                     expression composed  of integer constants (expressed in
                     standard C syntax), the nor� mal binary operators [+, -,
                     *, /, &, |], a length  operator, and  special  packet
                     data accessors.  To access data inside the packet, use the
                     following syntax: proto [ expr : size ] Proto is one of
                     ether, fddi, ip, arp,  rarp,  tcp,  udp,  or icmp,  and
                     indicates the protocol layer for the index opera� tion.
                     The byte offset, relative to the  indicated  protocol
                     layer, is given by expr.  Size is optional and indicates
                     the number of bytes in the field of interest; it can  be
                     either one,  two,  or four, and defaults to one.  The
                     length opera� tor, indicated by the keyword len, gives the
                     length  of  the packet.

                     For example, `ether[0] & 1 != 0' catches all multicast traf�
                     fic.  The expression `ip[0] & 0xf != 5' catches all IP pack�
                     ets  with  options.  The  expression  `ip[6:2] & 0x1fff = 0'
                     catches only unfragmented datagrams and frag zero  of  frag�
                     mented  datagrams.   This check is implicitly applied to the
                     tcp and udp index operations.  For instance,  tcp[0]  always
                     means  the first byte of the TCP header, and never means the
                     first byte of an intervening fragment.

       Primitives may be combined using:

                     A parenthesized group of primitives and operators (parenthe�
                     ses are special to the Shell and must be escaped).

                     Negation (`!' or `not').

                     Concatenation (`&&' or `and').

                     Alternation (`||' or `or').

              Negation  has  highest  precedence.   Alternation and concatenation
              have equal precedence and  associate  left  to  right.   Note  that
              explicit  and  tokens, not juxtaposition, are now required for con�
              catenation.

              If an identifier is given without a keyword, the most  recent  key�
              word is assumed.  For example,
                   not host vs and ace
              is short for
                   not host vs and host ace
              which should not be confused with
                   not ( host vs or ace )

              Expression  arguments  can  be passed to tcpdump as either a single
              argument or as multiple arguments, whichever  is  more  convenient.
              Generally,  if  the expression contains Shell metacharacters, it is
              easier to pass it as a single, quoted argument.  Multiple arguments
              are concatenated with spaces before being parsed.

    EXAMPLES
           To print all packets arriving at or departing from sundown:
                  tcpdump host sundown

           To print traffic between helios and either hot or ace:
                  tcpdump host helios and \( hot or ace \)

           To print all IP packets between ace and any host except helios:
                  tcpdump ip host ace and not helios

           To print all traffic between local hosts and hosts at Berkeley:
                  tcpdump net ucb-ether

           To  print  all  ftp  traffic through internet gateway snup: (note that the
           expression is quoted to prevent  the  shell  from  (mis-)interpreting  the
           parentheses):
                  tcpdump 'gateway snup and (port ftp or ftp-data)'

           To print traffic neither sourced from nor destined for local hosts (if you
           gateway to one other net, this stuff should never make it onto your  local
           net).
                  tcpdump ip and not net localnet

           To  print  the start and end packets (the SYN and FIN packets) of each TCP
           conversation that involves a non-local host.
                  tcpdump 'tcp[13] & 3 != 0 and not src and dst net localnet'

           To print IP packets longer than 576 bytes sent through gateway snup:
                  tcpdump 'gateway snup and ip[2:2] > 576'

           To print IP broadcast or multicast packets that were not sent via ethernet
           broadcast or multicast:
                  tcpdump 'ether[0] & 1 = 0 and ip[16] >= 224'

           To  print  all  ICMP packets that are not echo requests/replies (i.e., not
           ping packets):
                  tcpdump 'icmp[0] != 8 and icmp[0] != 0"
```

Ok, so that is a lot of info (and probably more than we need) but it gives us a starting point. Lets give it a shot... on my network I have a linux box and a windoze machine connected to a non switched hub. Therefore, if I place my ethernet card in promiscuous mode on my linux machine I should be able to see all traffic going to (and coming from) my windows machine. Lets see if the examples from the man page will work if directly fed to pcap_compile..

Consider the following program... (download here)

```
/**********************************************************************
* file:   testpcap3.c
* date:   Sat Apr 07 23:23:02 PDT 2001
* Author: Martin Casado
* Last Modified:2001-Apr-07 11:23:05 PM
*
* Investigate using filter programs with pcap_compile() and
* pcap_setfilter()
*
**********************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

/* just print a count every time we have a packet...                        */
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    static int count = 1;
    fprintf(stdout,"%d, ",count);
    fflush(stdout);
    count++;
}

int main(int argc,char **argv)
{
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;     /* pcap.h                    */
    struct ether_header *eptr;  /* net/ethernet.h            */
    struct bpf_program fp;      /* hold compiled program     */
    bpf_u_int32 maskp;          /* subnet mask               */
    bpf_u_int32 netp;           /* ip                        */


    if(argc != 2){ fprintf(stdout,"Usage: %s \"filter program\"\n"
            ,argv[0]);return 0;}

    /* grab a device to peak into... */
    dev = pcap_lookupdev(errbuf);
    if(dev == NULL)
    { fprintf(stderr,"%s\n",errbuf); exit(1); }

    /* ask pcap for the network address and mask of the device */
    pcap_lookupnet(dev,&netp,&maskp,errbuf);

    /* open device for reading this time lets set it in promiscuous
     * mode so we can monitor traffic to another machine             */
    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
    if(descr == NULL)
    { printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    /* Lets try and compile the program.. non-optimized */
    if(pcap_compile(descr,&fp,argv[1],0,netp) == -1)
    { fprintf(stderr,"Error calling pcap_compile\n"); exit(1); }

    /* set the compiled program as the filter */
    if(pcap_setfilter(descr,&fp) == -1)
    { fprintf(stderr,"Error setting filter\n"); exit(1); }

    /* ... and loop */
    pcap_loop(descr,-1,my_callback,NULL);

    return 0;
}
```

This program accepts a string from the user, (similar to tcpdump) compiles it and sets it as a filter. Lets go ahead and try it with an example similar to the one in the tcpdump examples..

```
[root@localhost libpcap]# gcc testpcap3.c -lpcap
[root@localhost libpcap]# ./a.out "host www.google.com"
(** try and ping www.slashdot.org ... nothing **)
(** try and ping www.google.com **)
1, 2, 3, 4, 5, 6,
(** hurray! **)
```

It looks like our filter program worked. Lets try to see if we can capture packets from a different machine on the same network.... how about my windows machine when it connects to battle.net..

```
[root@localhost libpcap]# ./a.out "src 192.168.1.104"
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
```

Yes yes!!! we are getting very close to actually having some power, but first we need to find out how to look inside the packets and pull out information. At last! the next section will delve into disecting packets so we can really analyze what is going on in our networks!!

[prev] [next]

Packet Analysis

This section will focus on peaking into the packets to extract the information (which is what we wanted to begin with). First off we must arm ourselves! Go ahead and get all the relevent RFC's. Lets start off with RFC 791 (IP) RFC 768 (UDP) RFC 826 (ARP) RFC 792 (ICMPv4) and of course RFC 793 (TCPv4) The truth is, once you have these files you dont really need me *sigh* but then again... why right your own code when you can just copy mine! hehe

I would highly recommend you use another packet sniffer to double check your programs... tcpdump will do just fine, and ethereal just kicks ass, you can get either (and more!!) at http://www.tcpdump.org/related.html. Both of these programs are capable of analyzing all fields of a packet, plus the data. Sure we could use them instead of creating our own... but what fun would that be?

I would prefer not to have to rewrite the main body of the program for each new example like I have done previously. Instead I am going to use the same main program and only post the callback function which gets passed to the pcap_loop() or pcap_dispatch() function. Below is a copy of the main program I intend on using (nothing special), go ahead and cut and paste it or download it here.

```
/**********************************************************************
* file:   pcap_main.c
* date:   Tue Jun 19 20:07:49 PDT 2001
* Author: Martin Casado
* Last Modified:2001-Jun-23 12:55:45 PM
*
* Description:
* main program to test different call back functions
* to pcap_loop();
*
* Compile with:
* gcc -Wall -pedantic pcap_main.c -lpcap (-o foo_err_something)
*
* Usage:
* a.out (# of packets) "filter string"
*
**********************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
```

```
#include <net/ethernet.h>
#include <netinet/ether.h>

/*
 * workhorse function, we will be modifying this function
 */
void my_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
}


int main(int argc,char **argv)
{
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    struct bpf_program fp;      /* hold compiled program     */
    bpf_u_int32 maskp;          /* subnet mask               */
    bpf_u_int32 netp;           /* ip                        */
    u_char* args = NULL;

    /* Options must be passed in as a string because I am lazy */
    if(argc < 2){
        fprintf(stdout,"Usage: %s numpackets \"options\"\n",argv[0]);
        return 0;
    }

    /* grab a device to peak into... */
    dev = pcap_lookupdev(errbuf);
    if(dev == NULL)
    { printf("%s\n",errbuf); exit(1); }

    /* ask pcap for the network address and mask of the device */
    pcap_lookupnet(dev,&netp,&maskp,errbuf);

    /* open device for reading. NOTE: defaulting to
     * promiscuous mode*/
    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
    if(descr == NULL)
    { printf("pcap_open_live(): %s\n",errbuf); exit(1); }


    if(argc > 2)
    {
        /* Lets try and compile the program.. non-optimized */
        if(pcap_compile(descr,&fp,argv[2],0,netp) == -1)
        { fprintf(stderr,"Error calling pcap_compile\n"); exit(1); }

        /* set the compiled program as the filter */
        if(pcap_setfilter(descr,&fp) == -1)
        { fprintf(stderr,"Error setting filter\n"); exit(1); }
    }

    /* ... and loop */
    pcap_loop(descr,atoi(argv[1]),my_callback,args);

    fprintf(stdout,"\nfinished\n");
    return 0;
}
```

I will be using the above program and merely replacing the callback function **my_callback** for demo programs in this section.

Lets start by looking at the datalink headers. "Didn't we already do this", you ask. Sure... sort of, but we didn't spend much time on it so lets just get this out of the way. Looking at the datalink header isn't all too exciting, but it certainly is something we want to stick in our toolkit so we will gloss over the important stuff and continue on. The most important element of the ether header to us is the ether type. Remember **struct ether_header** from **net/ethernet.h**? just so you don't have to click back, here it is again whith the definition of an ether_addr.

```
/* This is a name for the 48 bit ethernet address available on many
   systems.  */
struct ether_addr
{
  u_int8_t ether_addr_octet[ETH_ALEN];
} __attribute__ ((__packed__));

/* 10Mb/s ethernet header */
struct ether_header
{
  u_int8_t  ether_dhost[ETH_ALEN];      /* destination eth addr */
  u_int8_t  ether_shost[ETH_ALEN];      /* source ether addr    */
  u_int16_t ether_type;                 /* packet type ID field */
} __attribute__ ((__packed__));
```

Fortunatly (at least in Linux) **netinet/ether.h** provides us with some fuzzy routines to convert ethernet headers to readable ascii and back..

```
/* Convert 48 bit Ethernet ADDRess to ASCII.  */
extern char *ether_ntoa (__const struct ether_addr *__addr) __THROW;
extern char *ether_ntoa_r (__const struct ether_addr *__addr, char *__buf)
     __THROW;

/* Convert ASCII string S to 48 bit Ethernet address.  */
extern struct ether_addr *ether_aton (__const char *__asc) __THROW;
extern struct ether_addr *ether_aton_r (__const char *__asc,
                                        struct ether_addr *__addr) __THROW;
```

as well as ethernet address to HOSTNAME resolution (that should ring a bell.. :-)

```
/* Map HOSTNAME to 48 bit Ethernet address.  */
extern int ether_hostton (__const char *__hostname, struct ether_addr *__addr)
     __THROW;
```

Previously I pasted some code shamelessly stolen from Steven's Unix Network PRogramming to print out the ethernet header, from now on we take the easy route. Here is a straightforward callback function to handle ethernet headers, print out the source and destination addresses and handle the type.

```
u_int16_t handle_ethernet
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet);

/* looking at ethernet headers */

void my_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    u_int16_t type = handle_ethernet(args,pkthdr,packet);

    if(type == ETHERTYPE_IP)
    {/* handle IP packet */
    }else if(type == ETHERTYPE_ARP)
    {/* handle arp packet */
    }
    else if(type == ETHERTYPE_REVARP)
    {/* handle reverse arp packet */
    }/* ignorw */
}

u_int16_t handle_ethernet
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    struct ether_header *eptr;  /* net/ethernet.h */

    /* lets start with the ether header... */
    eptr = (struct ether_header *) packet;

    fprintf(stdout,"ethernet header source: %s"
            ,ether_ntoa((const struct ether_addr *)&eptr->ether_shost));
    fprintf(stdout," destination: %s "
            ,ether_ntoa((const struct ether_addr *)&eptr->ether_dhost));

    /* check to see if we have an ip packet */
    if (ntohs (eptr->ether_type) == ETHERTYPE_IP)
    {
        fprintf(stdout,"(IP)");
    }else  if (ntohs (eptr->ether_type) == ETHERTYPE_ARP)
    {
        fprintf(stdout,"(ARP)");
    }else  if (ntohs (eptr->ether_type) == ETHERTYPE_REVARP)
    {
        fprintf(stdout,"(RARP)");
    }else {
        fprintf(stdout,"(?)");
        exit(1);
    }
    fprintf(stdout,"\n");

    return eptr->ether_type;
}
```

You can download the full code here.

Whew! Ok got that out of the way, currently we have a relatively simple framework to print out an ethernet header (if we want) and then handle the type. Lets start by looking at the IP header.

IP:
We'll need to wip out our handy dandy RFC's (791 in this case) and take a look at what it has to say about IP headers... here is a copy of the section which decsribes the header.

```
3.1 Internet Header Format

       A summary of the contents of the internet header follows:

          0                   1                   2                   3
          0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |Version|  IHL  |Type of Service|          Total Length         |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |         Identification        |Flags|      Fragment Offset    |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |  Time to Live |    Protocol   |         Header Checksum       |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |                       Source Address                          |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |                    Destination Address                        |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         |                    Options                    |    Padding    |
         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                           Example Internet Datagram Header

                                      Figure 4.

         Note that each tick mark represents one bit position.
```

Now lets peak at **netinet/ip.h**

```
struct ip
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;               /* header length */
    unsigned int ip_v:4;                /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;                /* version */
    unsigned int ip_hl:4;               /* header length */
#endif
    u_int8_t 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_int8_t ip_ttl;                    /* time to live */
    u_int8_t ip_p;                      /* protocol */
    u_short ip_sum;                     /* checksum */
    struct in_addr ip_src, ip_dst;      /* source and dest address */
  };
```

Cool, they seem to match up perfectly.... this of course would be fine to use, but I prefer to follow the tcpdump method of handling the version and header length.

```
struct my_ip
        u_int8_t        ip_vhl;         /* header length, version */
#define IP_V(ip)        (((ip)->ip_vhl & 0xf0) >> 4)
#define IP_HL(ip)       ((ip)->ip_vhl & 0x0f)
        u_int8_t        ip_tos;         /* type of service */
        u_int16_t       ip_len;         /* total length */
        u_int16_t       ip_id;          /* identification */
        u_int16_t       ip_off;         /* fragment offset field */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        u_int8_t        ip_ttl;         /* time to live */
        u_int8_t        ip_p;           /* protocol */
        u_int16_t       ip_sum;         /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
```

Lets take a first stab at peaking into the IP header... consider the following function (full source here).

```
u_char* handle_IP
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    const struct my_ip* ip;
    u_int length = pkthdr-&len;
    u_int hlen,off,version;
    int i;

    int len;

    /* jump pass the ethernet header */
    ip = (struct my_ip*)(packet + sizeof(struct ether_header));
    length -= sizeof(struct ether_header);

    /* check to see we have a packet of valid length */
    if (length < sizeof(struct my_ip))
    {
        printf("truncated ip %d",length);
        return NULL;
    }

    len     = ntohs(ip->ip_len);
    hlen    = IP_HL(ip); /* header length */
    version = IP_V(ip);/* ip version */

    /* check version */
    if(version != 4)
    {
      fprintf(stdout,"Unknown version %d\n",version);
      return NULL;
    }
    /* check header length */
    if(hlen < 5 )
    {
        fprintf(stdout,"bad-hlen %d \n",hlen);
    }

    /* see if we have as much packet as we should */
    if(length < len)
        printf("\ntruncated IP - %d bytes missing\n",len - length);

    /* Check to see if we have the first fragment */
    off = ntohs(ip->ip_off);
    if((off &apm; 0x1fff) == 0 )/* aka no 1's in first 13 bits */
    {/* print SOURCE DESTINATION hlen version len offset */
        fprintf(stdout,"IP: ");
        fprintf(stdout,"%s ",
                inet_ntoa(ip->ip_src));
        fprintf(stdout,"%s %d %d %d %d\n",
                inet_ntoa(ip->ip_dst),
                hlen,version,len,off);
    }

    return NULL;
}
```

Given a clean arp cache this is what the output looks like on my machine, when I try to telnet to 134.114.90.1...

```
[root@localhost libpcap]# ./a.out 5
ETH: 0:10:a4:8b:d3:b4 ff:ff:ff:ff:ff:ff (ARP) 42
ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (ARP) 60
ETH: 0:10:a4:8b:d3:b4 0:20:78:d1:e8:1 (IP) 74
IP: 192.168.1.100 134.114.90.1 5 4 60 16384
ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (IP) 60
IP: 134.114.90.1 192.168.1.100 5 4 40 0
```

Lets try and reconstruct the conversation shall we?

- **my computer:** Who has the gateways IP (192.168.1.100)?
  ETH: 0:10:a4:8b:d3:b4 ff:ff:ff:ff:ff:ff (ARP) 42
- **gateway:** I do!!
  ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (ARP) 60
- **my computer(through gateway):** Hello Mr. 134.114.90.1 can we talk?
  ETH: 0:10:a4:8b:d3:b4 0:20:78:d1:e8:1 (IP) 74 IP: 192.168.1.100 134.114.90.1 5 4 60 16384
- **134.114.90.1:** Nope, I'm not listening
  ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (IP) 60 IP: 134.114.90.1 192.168.1.100 5 4 40 0

I have admittedly skipped TONS of information in a rush to provide you with code to display the IP header (thats all you really wanted anyways wasn't it :-). That said, if you are lost don't worry, I will slow down and attempt to describe what exactly is going on. All that you really need to know up to this point is..

- All packets are sent via ethernet
- The ethernet header defines the protocol type of the packet it is carrying
- IP is one of these types (as well as ARP and RARP)
- The IP header is confusing ...

So before getting too far into packet dissection it would probably benefit us to regress a bit and talk about IP...
"awww but.... that sounds boring!",you say. Well if you are really anxious I would suggest you grab the tcpdump source and take a look at the following methods ... :-)

- ether_if_print (print-ether.c)
- ip_print (print-ip.c)
- tcp_print (print-tcp.c)
- udp_print (print-udp.c)

I've also found the sniffit source to be a great read.

[prev] [next]