An overview of raw sockets programming with FreeBSD
Chapter I: Transport Control Protocol

1. What is a raw socket?

Raw sockets are what the name sais: sockets which offer the programmer the possibility to have absolute control over the data which is being sent or received through the network.

They are very usefull when someone needs to create their own protocol, using the current system's stack. Actually, with raw sockets, the programmer has control of every single bit which is sent via the network. This is amazing, and provides an overwhelming power. Anything which goes over the network is nothing more but a linear field of bits, 1 or 0, incoming and outgoing. Raw sockets give the programmer full control over every bit.

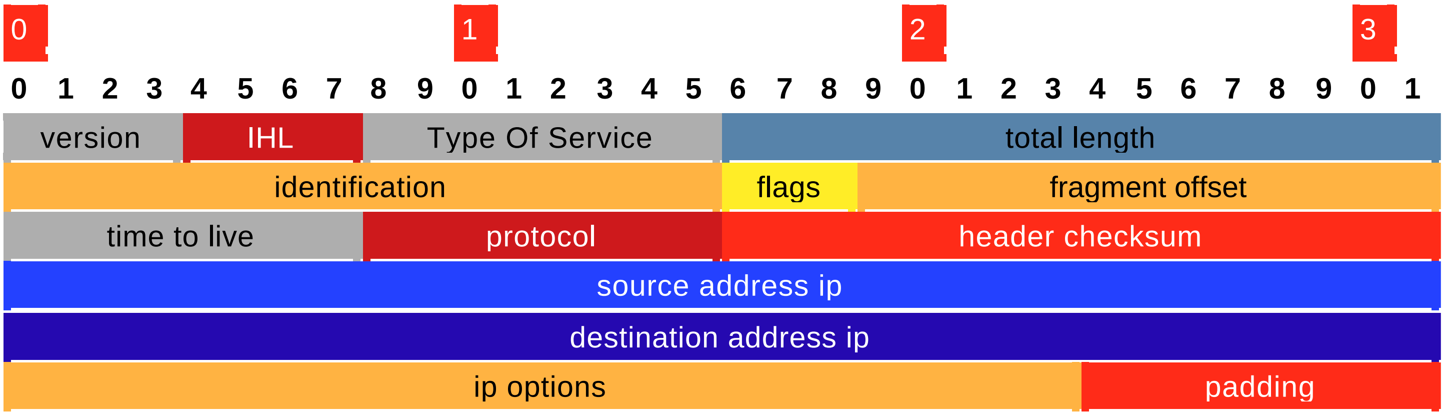
Creating your own protocol for sending and receiving data is not a joke. It is a difficult serious task, but has it's advantages. No matter what the reasons for creating a special protocol are, these are some obvious examples: encrypted traffic tunnels, with pseudo-random protocol (before somebody attacks the crypto-system, they must first completely understand the new weird protocol),optimized voice and video conference protocols, which will really increase the quality and the performance of the sessions.

Raw sockets are goldish for hacking and network probing. Having raw sockets programming as a skill, is an inestimable advantage for a network tester. You may send packets which the remote kernel is not expecting, study it's reactions, firewalk over firewalls, and who could spell them all!

Knowing raw sockets, is exactly what knowing to code in C and assembly is for Unix.

2. Theory of the packet

Everything which goes out and in over the internet is just a linear set of bits, which can take the value of 1 or 0. In order to understand this, data has been split in packets, and packets were synthetized and organized in logical structure of bits. Let's take a look over the IP packet structure:



version	4 bits, indicates the ip version. Can be 4 or 6.
internet header length	The length of the internet header measured in 32 bit words. It points to the beginning of the data (which follows right after the padding). The minimum IHL is 5 words, which equal with 20 bytes.
type of service	8 bits, abstract field, sometimes taken into account on routing, sometimes not
total length	16 bits, measures datagram length in octets, including internet header and data field
identification	16 bit number used to reassembly fragmented datagrams
flags	3 bits and first always zero, used for fragmentation signalling
fragment offset	13 bits, indicates where in the datagram the current fragment belongs. relevant in fragmentation only.
time to live	8 bits, decreases at every hop, makes sure a packet will not travel forever lost in space
protocol	8 bits
header checksum	16 bit checksum of the ip header, changes at every hop because of **TTL**
source address	32 bits
destination address	32 bits
ip options	This is optional,a list of options terminated by a null byte. They can miss.
padding	Null bytes assigned after options, in order to fill the adjacent space in such a way that the options field + padding field = multiple of 32 bits. If there is no options field, there is no need for padding.

The reason why the header must be a multiple of 32 bits is because it is easier for a binary CPU to process the data. Also, this is why the IHL is measured in 32 bit words, always being a multiple of 4 bytes.

It is easy to notice that the smallest header is the one without options (and so without padding), and has the length of 5 x 4 bytes.

This structure is defined by the system headers, in */usr/include/netinet/ip.h*, as follows:

```
/*
 * Structure of an internet header, naked of options.
 */
struct ip {
#ifdef _IP_VHL
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
#else
#if BYTE_ORDER == LITTLE_ENDIAN
        u_int   ip_hl:4,                /* header length */
                ip_v:4;                 /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
        u_int   ip_v:4,                 /* version */
                ip_hl:4;                /* header length */
#endif
#endif /* not _IP_VHL */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
#define IP_RF 0x8000                    /* reserved fragment flag */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
```

I tried to chose the colors in such a way to underline the most critical sections with warmer colours, where being critical means oftenly being mistaken. Also this should have a psychologycal impact for the reader (yah right).

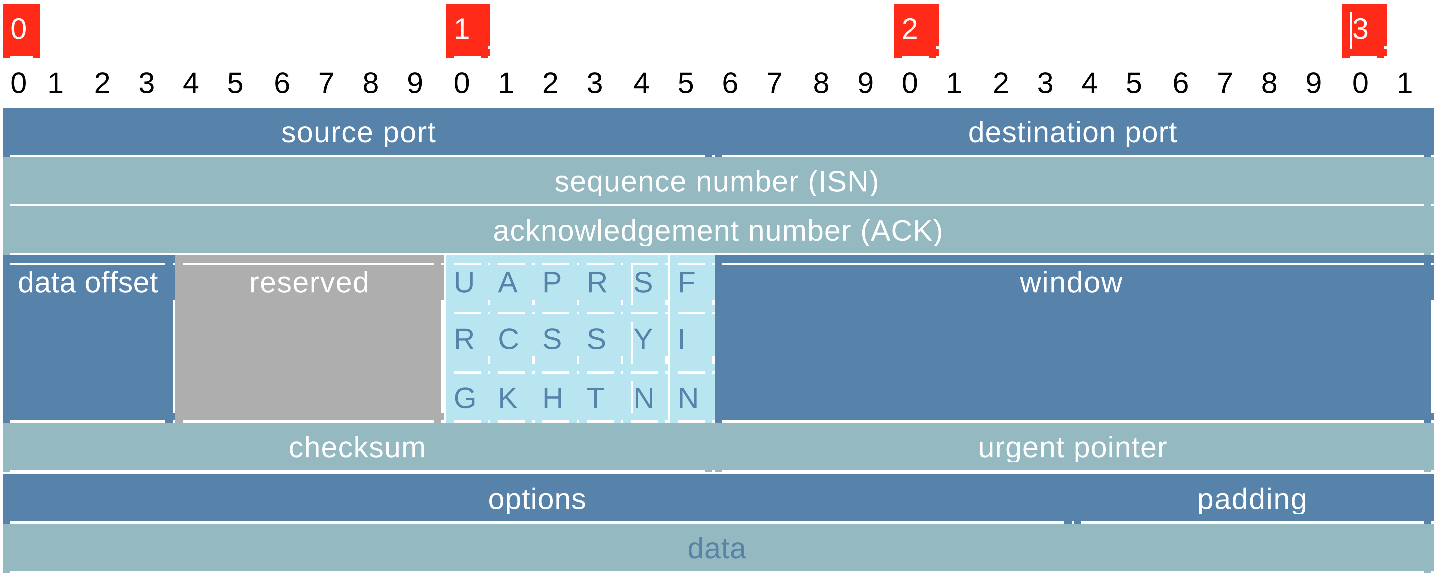
The *checksum* is computed *universally* with the *internet checksum algorithm* which is implemented in the snippet below, ripped from the FreeBSD src/sys files:

```
unsigned short in_cksum(unsigned short *addr, int len)
{
    register int sum = 0;
    u_short answer = 0;
    register u_short *w = addr;
    register int nleft = len;
    while (nleft > 1)
      {
              sum += *w++;
              nleft -= 2;
      }

    if (nleft == 1)
      {
              *(u_char *) (&answer) = *(u_char *) w;
              sum += answer;
      }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return (answer);
}
```

It is easy to notice that the IP protocol cannot transport data by itself. However, it can encapsulate other protocols, like the **Transport Control Protocol**, which are able to carry data among their body.

The schematics for the TCP are presented below:



source port	16 bit value holding the source port
destination port	16 bit value holding the destination port
sequence number	32 bit value holding the *Internet Sequence Number*, used for synchronizing packets
acknowledgement number	32 bit value holding the acknowledgement number, which represents the next ISN the sender of the packet is expecting to receive.
data offset	4 bits, representing the number of 32 bit words in the TCP header, and pointing to where the data begins.
reserved	6 bits reserved for future use, must be zero
control bits	6 bits, from left to right: URG,ACK,PSH,RST,SYN,FIN
window	16 bits, representing the number of data bytes starting with the one indicated in the acknowledgement field, which the sender of this packet is willing to accept
checksum	16 bits, checksum of the TCP ***pseudo****header* (will be discussed in great detail later)
urgent pointer	16 bits, holds the current value of the urgent pointer, as a positive offset from the current ISN. It points to the sequence number of the octet following the urgent data (only relevant if URG flag-bit is set)
options	optional field of variable length, stored in bytes (multiple of 8 bits). The list of options is terminated by a null byte
padding	null bytes padded to the end of the options field in order to fill the 32 bit space. only relevant if the options field is present

This structure is defined in the system header files as show below, in */usr/include/netinet/tcp.h*:

```
/*
 * TCP header.
 * Per RFC 793, September, 1981.
 */
struct tcphdr {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
#if BYTE_ORDER == LITTLE_ENDIAN
        u_int   th_x2:4,                /* (unused) */
                th_off:4;               /* data offset */
#endif
#if BYTE_ORDER == BIG_ENDIAN
        u_int   th_off:4,               /* data offset */
                th_x2:4;                /* (unused) */
#endif
        u_char  th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
#define TH_ECE  0x40
#define TH_CWR  0x80
#define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)

        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};
```

The TCP **pseudo**header function is computed using the universal internet checksum function, which was presented earlier. The trick is, in the case of the Transport Control Protocol, the checksum is ***not the checksum of the TCP header,*** as opposite to the IP header checksum, which is, actually, the checksum of the IP header (as the name sais).

The TCP checksum is being computed over the following **pseudoheader** which needs to be **created** based on the information **from** the real **TCP and IP** headers:

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

source address (32 bits)		
destination address (32 bits)		
8 zero bits (1 null byte)	protocol (8 bits)	tcp length (16 bits)

The *tcp length* measures in bytes the length of the TCP header + the length of the data field. In case the data field is missing, the tcp length is equal with the tcp header length.

3. How to read from a raw socket?

Sockets were originally introduced by BSD. On a *BSD system, like FreeBSD, it is not possible to read TCP or UDP from a raw socket. It is, however, possible, to read any of the other protocols from a raw socket, like ICMP.

In order to "read" raw TCP and UDP packets, on the *BSD systems, the programmer needs to "sniff" the network. This is done by accessing a raw interface to the data link layers, like the *Berkeley Packet Filter*. This provides access on all the traffic, including the one for other clients in the local network, or external clients in the case of a gateway but we are only interested to intercept specific traffic, usually responses to the raw packets that were generated using raw sockets.

Reading and understanding the *bpf(4)* manual pages is an imperative. However, we will present here a simple example of working with bpf.

The first step, is to open /dev/bpf0. If bpf0 cannot be accessed, then try to open /dev/bpf1. And so on, until one of them is free. Actually, every time a process opens /dev/bpfN, a new device, /dev/bpf(N+1) will become accesible. Bpf supports a huge number of opened devices.

Secondly, the opened device needs an interface to be attached to it, from where to sniff the packets. Bpf can also be used for writing packets (a way of seding raw packets without using raw sockets). In order to do that, an *ifreq* structure needs to be defined, where ifreq.ifr_name must be set to the name of the interface. For example, "ed0" or "rl0". After that,

```
ioctl (bpf_file_descriptor,BIOCSETIF,&ifreq)
```

the attachement is done using a special ioctl:

In order to achieve the "fast sniffing mode", and receive packets immediately, we need to use a buffer of the same size with the one which bpf uses. For this, we either set the standard bpf buffer value, either dynamically allocate buffer space coresponding to the right

```
ioctl(bpf_file_descriptor,BIOCGBLEN,&buflen)
```

size. In this example, we ask for the bpf default buffer size:

```
int true=1;
ioctl(bpf_file_descriptor,BIOCIMMEDIATE,&true)
```

Then we can tell bpf that we are going to use the immediate mode:

At this point, we can read and write from the opened bpf device. But if we need to intercept specific packets, then we must set up some bpf filters to exclude unnecessary traffic. This is done something similar to machine code, working inside an internal register of bpf:

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
/*  load halfword at position 12 from packet into register  */
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x0800, 0, 11),
/*  is it 0x800? if no, jump over 11 instructions, else jump over 0  */
        BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
/*  load from position 23  */
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 6, 0, 9),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, sip, 0, 7),
/*  is it our sip source ip ? if no, jump  */
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, dip, 0, 5),
/*  is it our dip destination ip? if no, jump  */
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 34),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, sport, 0, 3),
/*  check the source port sport  */
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 36),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, dport, 0, 1),
/*  check the destination port dport  */
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
/*  if we reach here, return -1 which will allow the packet to be read  */
        BPF_STMT(BPF_RET+BPF_K, 0),
/*  if we reach here, return 0 which will ignore the packet */
};
```

Note that we are using ethernet interfaces, and we can see that at position 12 in the ethernet header, the protocol is stored. 0x800 stands for IP. Further, we check for TCP, then for source/destination ips and ports.

Then we tell bpf about our filter:

```
struct bpf_program bpf_program = {
        14,
        (struct bpf_insn *) &insns
};
ioctl(fd, BIOCSETF, (struct bpf_program *) &bpf_program)
```

And reading is now trivial:

```
buf = (struct bpf_hdr *) malloc(buflen);
bzero(buf, buflen);
read(fd, buf, buflen);
```

4. Appendix

The following code will send 10 SYN requests to the target and display the remote server's ISN from the SYN_ACK packets. It can be used to test the remote ip stack (very interesting with windows hehe).

The sources for the coude can be found in seq.c file attached to this paper.

The output looks similar to this:

```
beast# ./seq
        DIF             ISN                     RRT(usec)
      ----------     177707348                  147404
      88868          177796216                  137504
      97809          177894025                  137664
      108487         178002512                  140807
      109471         178111983                  138693
      88804          178200787                  139904
      88736          178289523                  137582
      90068          178379591                  138711
      123313         178502904                  141747
      108665         178611569                  137737
beast#
```

The source, seq.c:

```
/*
* Example of using TCP raw sockets on FreeBSD 4.x and 5.x maybe other too.
* Written for educational purposes by Clau & Burebista
*
*                    www.reversedhell.net
* clau:      clau@reversedhell.net, dr.clau@xnet.ro
* burebista: aanton@reversedhell.net, uber@rdslink.ro
*
* We both worked equally hard studying the headers, the sources of the
* system and of other programs, thus the order of precedence in the
* credit field is completely random.
*
* I, burebista, would like to thank the freebsd-hackers mailing list for
* their help regarding header includes.
*
*/


#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/ioctl.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <net/if.h>
#include <net/bpf.h>
#include <net/ethernet.h>
```

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>



#define PORT 80
#define INTERFACE "ed0"

/* #define debug */

/*
*	bpf_open()
*
*	opens the first available /dev/bpf device
*	and returns the file descriptor
*
*/
int bpf_open() {
	char dev[] = "/dev/bpf";
	char num[2], buf[11];
	int i, fd;

	fd = -1;
	i = 0;
	do {
		sprintf((char *) &buf, "%s%u", dev, i);
		fd = open((char *) &buf, O_RDWR);
		i++;
	} while(fd < 0 && i < 10);

#ifdef debug
	printf("bpf_open:\t%s\n", buf);
#endif

	return fd;
}


/*
*	seq_read
*
*	sip - source IP for filter
*	dip - destination IP for filter
*	sport - source port for filter
*	dport - destination port for filter
*	(all in host byteorder)
*/
int seq_read(int fd, int sip, int dip, short sport, short dport) {
	int true = 1;
	int buflen, r;
	struct bpf_hdr *buf;
	struct ifreq ifreq;
	struct bpf_insn insns[] = {
		BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x0800, 0, 11),
```

```
		BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 6, 0, 9),
		BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, sip, 0, 7),
		BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, dip, 0, 5),
		BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 34),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, sport, 0, 3),
		BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 36),
		BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, dport, 0, 1),
		BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
		BPF_STMT(BPF_RET+BPF_K, 0),
	};
	struct bpf_program bpf_program = {
		14,
		(struct bpf_insn *) &insns
	};
	struct timeval timeval;
	struct ip *iph;
	struct tcphdr *tcph;

	strcpy((char *) ifreq.ifr_name, INTERFACE);
	if (ioctl(fd, BIOCSETIF, &ifreq) < 0) {
		perror("ioctl BIOCSETIF");
		return -1;
	}

	ioctl(fd, BIOCGBLEN, &buflen);

	if (ioctl(fd, BIOCIMMEDIATE, (u_int) &true) < 0) {
		perror("BIOCIMMEDIATE");
		return -1;
	}

	timeval.tv_sec = 5;
	timeval.tv_usec = 0;
	if (ioctl(fd, BIOCSRTIMEOUT, (struct timeval *) &timeval) < 0) {
		perror("set timeout");
		return -1;
	}

	if (ioctl(fd, BIOCSETF, (struct bpf_program *) &bpf_program) < 0) {
		perror("set filter");
		return -1;
	}

	buf = (struct bpf_hdr *) malloc(buflen);
	bzero(buf, buflen);

	r = read(fd, buf, buflen);

	iph = (struct ip *) ((char *) buf + buf->bh_hdrlen + sizeof(struct ether_header));
	tcph = (struct tcphdr *) ((char *) iph + sizeof(struct ip));
```

```
#ifdef debug
		printf("IP SRC:\t\t%s\n", inet_ntoa(iph->ip_src));
		printf("IP DST:\t\t%s\n", inet_ntoa(iph->ip_dst));
		printf("TCP SRC:\t%u\n", ntohs(tcph->th_sport));
		printf("TCP DST:\t%u\n", ntohs(tcph->th_dport));
		printf("SEQ #:\t\t%u\n", ntohl(tcph->th_seq));
#endif

		if (r > 0)
			return ntohl(tcph->th_seq);

		return 0;
}


unsigned short in_cksum(unsigned short *addr, int len)
{
		register int sum = 0;
		u_short answer = 0;
		register u_short *w = addr;
		register int nleft = len;
		while (nleft > 1)
		{
			sum += *w++;
			nleft -= 2;
		}
		if (nleft == 1)
		{
			*(u_char *) (&answer) = *(u_char *) w;
			sum += answer;
		}
		sum = (sum >> 16) + (sum & 0xffff);
		sum += (sum >> 16);
		answer = ~sum;
		return (answer);
}

int main (void) {
		int bpf = bpf_open();						/* open /dev/bpfX */
		int s = socket (PF_INET, SOCK_RAW, IPPROTO_IP);		/* open raw socket */
		int addr_len,i,oldisn=0,newisn,one=1;
		const int *val = &one;
		struct timeval tv1,tv2;
		struct timezone tz1,tz2;

		char datagram[4096];		/* datagram buffer */
		char pseudohdr[1024];	/* pseudoheader buffer for computing tcp checksum */
		struct ip *iph = (struct ip *) datagram;
		struct tcphdr *tcph = (struct tcphdr *) (datagram + sizeof (struct ip));
		struct sockaddr_in sin;
		struct sockaddr_in sout;
		int tcphdr_size = sizeof(struct tcphdr);
```

```
        sin.sin_family = AF_INET;
        sin.sin_port = htons (PORT);
        sin.sin_addr.s_addr = inet_addr ("12.130.5.36"); /* destination ip */

        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);      /* data size = 0 */
        iph->ip_id = htons (31337);
        iph->ip_off = 0;
        iph->ip_ttl = 250;
        iph->ip_p = 6;
        iph->ip_sum = 0;
        iph->ip_src.s_addr = inet_addr ("81.196.32.25");/* source ip (me!) */
        iph->ip_dst.s_addr = sin.sin_addr.s_addr;
        tcph->th_sport = htons (1234);   /* source port */
        tcph->th_dport = htons (PORT);  /* destination port */
        tcph->th_seq = htonl(31337);
        tcph->th_ack = 0;/* in first SYN packet, ACK is not present */
        tcph->th_x2 = 0;
        tcph->th_off = sizeof(struct tcphdr)/4; /* data position in the packet */
        tcph->th_flags = TH_SYN;        /* initial connection request */
        tcph->th_win = htons (57344);   /* FreeBSD uses this value too */
        tcph->th_sum = 0; /* we will compute it later */
        tcph->th_urp = 0;

        if (tcphdr_size % 4 != 0)                       /* takes care of padding to 32 bits */
                tcphdr_size = ((tcphdr_size % 4) + 1) * 4;
        #ifdef debug
                printf("packet size:\t%u\n", tcphdr_size);
        #endif

/* create the pseudo header
*
*                   +--------+--------+--------+--------+
*                   |           Source Address          |
*                   +--------+--------+--------+--------+
*                   |         Destination Address       |
*                   +--------+--------+--------+--------+
*                   |  zero  |  PTCL  |    TCP Length   |
*                   +--------+--------+--------+--------+
*
*    The TCP Length is the TCP header length plus the data length in
*    octets (this is not an explicitly transmitted quantity, but is
*    computed), and it does not count the 12 octets of the pseudo
*    header.
*/
```

```
	memset(pseudohdr,0x0,sizeof(pseudohdr));
	memcpy(&pseudohdr,&(iph->ip_src.s_addr),4);
	memcpy(&pseudohdr[4],&(iph->ip_dst.s_addr),4);
	pseudohdr[8]=0; // just to underline this zero byte specified by rfc
	pseudohdr[9]=(u_int16_t)iph->ip_p;
	pseudohdr[10]=(u_int16_t)(tcphdr_size&0xFF00)>>8;
	pseudohdr[11]=(u_int16_t)(tcphdr_size&0x00FF);
	memcpy(&pseudohdr[12], tcph, sizeof(struct tcphdr));

/*
*end of pseudo header part
*/

	tcph->th_sum = in_cksum ((unsigned short *) (pseudohdr),tcphdr_size+12);
	#ifdef debug
		printf ("IP checksum set to  : %p\n",ntohs(iph->ip_sum));
		printf ("TCP checksum set to : %p\n",ntohs(tcph->th_sum));
	#endif



	if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)
		printf ("Warning: Cannot set HDRINCL!\n");

	fprintf(stderr,"     DIF                          ISN                     RRT(usec)\n");
	for (i=0;i<=9;i++)
	{
		if (sendto (s,datagram,iph->ip_len,0,(struct sockaddr *) &sin, sizeof (sin)) < 0)
		{
			perror ("sendto");
			exit(1);
		}
		gettimeofday(&tv1,&tz1);
		newisn=seq_read(bpf,
			ntohl(iph->ip_dst.s_addr),
			ntohl(iph->ip_src.s_addr),
			ntohs(tcph->th_dport),
			ntohs(tcph->th_sport));
		if (newisn==0) {
					fprintf(stderr,"\nOperation timed out!\n\n");
					close(s);
					close(bpf);
					exit(1);
				}
		if (i==0) fprintf(stderr,"----------                   %10u                  ",newisn);
		else fprintf(stderr,"%10u                    %10u
                                                   ",newisn-oldisn,newisn);
		gettimeofday(&tv2,&tz2);
		fprintf(stderr,"%d\n",((tv2.tv_sec - tv1.tv_sec) * 1000000 + (tv2.tv_usec -
                                                        tv1.tv_usec)));
		oldisn=newisn;
		/* time is measured in microseconds */
	}
	close(s);
	close(bpf);
	return 0;
}
```

5. Post Scriptum

Clau dedicates this to his beloved wife, Manuela. He also dedicates it to himself.

Burebista dedicates this to his two sisters, Ana and Maria, in no particular order. I am lost without you.

He also greets and thanks Undertaker (Yo!) and the Undernet #cracking Channel.

Greetings and thankings to smfcs (#asm undernet channel), for everything he has contributed to the community.

And a lot of thanks to the FreeBSD guys for making such a nice OS.

I also greet Mr. Dev Mazumdar, president of the 4 Front Technologies, for donating me a freebsd opensound license. (see www.opensound.com).

Thanks and greetings to Arthur (you know me, I know you), I wish you are well.

Thanks to everyone who left something behind and contributed in return.

Profound thanks to all the OpenSource community. Guys, I only use your stuff too:P!

Feel free to send your comments/questions/bugs to:

clau@reversedhell.net or dr.clau@xnet.ro
aanton@reversedhell.net or uber@rdslink.ro

6. References

The best reference are the system header files. Also, sources from other programs which use raw sockets are goldish. The first place to look for anything, are the well known manual pages.

At this time, I do not know of any references talking about raw socket programming on the net. Hopefully, there will be more.

We all know that people are divided in two categories: those who actually do something and leave something behind, and those who take the merits.