Libpcap and Libnet

Why Libnet & Libpcap?

- Allow manipulation/interception of link layer packets
  - Using socket programming, kernel will fill in the source IP address, checksum etc.
    - Raw socket is one way to write IP packets directly but not everything is in IP
  - Allow testing of new protocols

libpcap

- `char*pcap_lookupdev(char * errbuf)`
  - returns a pointer to a network device suitable for use

```
char errbuf[PCAP_ERRBUF_SIZE];
dev = pcap_lookupdev(errbuf);
```

- `pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)`
  - to obtain a packet capture descriptor to look at packets on the network
  - snaplen – maximum bytes to capture
  - promisc – whether set to promiscuous mode
  - to_ms – timeout
  - ebuf – error message

libpcap

```
int pcap_datalink (pcap_t *p)
```

- Returns the link layer of an adapter.
  - DLT_EN10MB Ethernet (10Mb, 100Mb, 1000Mb, and up)
  - DLT_PPP
  - DLT_SLIP
  - …

Libpcap

- `int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)`
  - to compile the string str into a filter program
  - -1 upon error

- `int pcap_setfilter(pcap_t *p, struct bpf_program *fp)`
  - to specify a filter program. fp is a pointer to a bpf_program struct, usually the result of a call to pcap_com- pile().
  - -1 upon failure

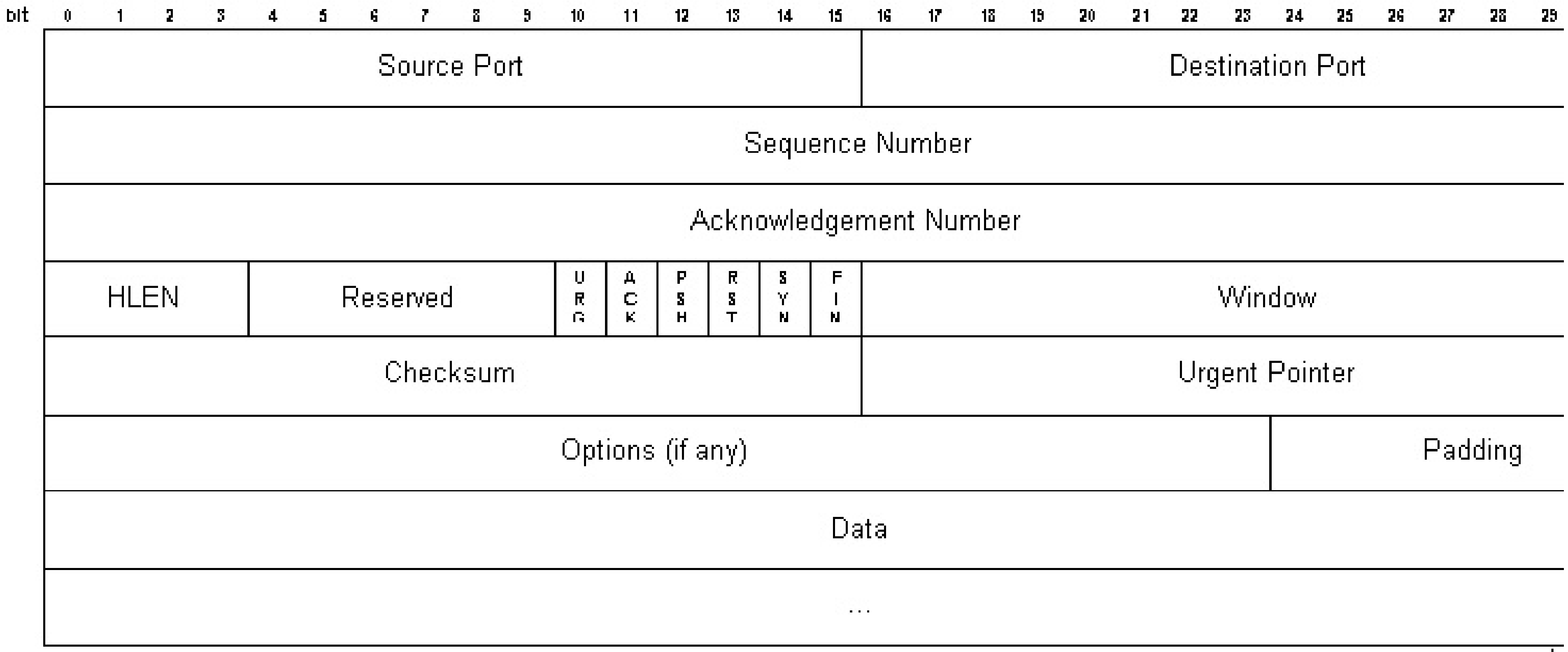
Filter Expression

Filter expression consists of an *id* preceded by one or more qualifier

- Type – host, net and port
  - E.g. net 128.3', `port 20'
- Dir – direction of transfer
  - src, dst, src or dst, src and dst
- Proto – ip, tcp, arp…
  - `tcp dst port ftp-data'
- *expr relop expr* (relop – relational operations >, <, >=, <=, ==
  - *proto* **[** *expr* **:** *size* **]** (expr gives the offset, size gives the length of data)
  - 'ip[6:2] & 0x1fff = 0'
  - `tcp[13] & 3 != 0'

only unfragmented datagrams and frag zero of fragmented datagrams

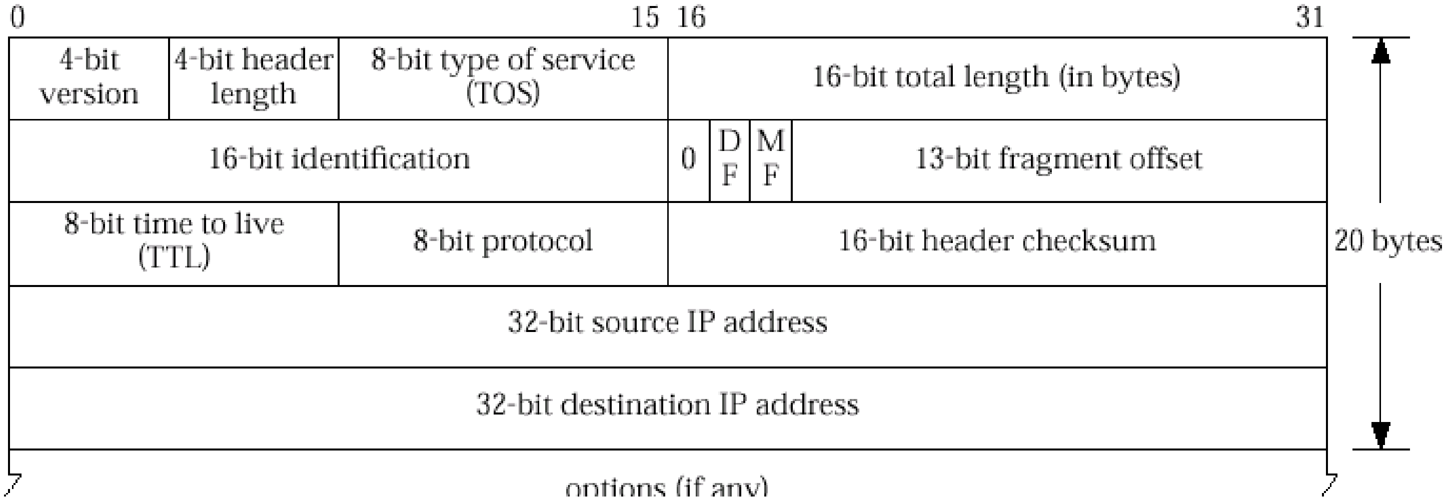
Fin or sync



bit 0 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

<table>
<tr><td colspan="8">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="10">Sequence Number</td></tr>
<tr><td colspan="10">Acknowledgement Number</td></tr>
<tr><td>HLEN</td><td>Reserved</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="2">Window</td></tr>
<tr><td colspan="8">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="9">Options (if any)</td><td>Padding</td></tr>
<tr><td colspan="10">Data</td></tr>
<tr><td colspan="10">...</td></tr>
</table>

IP Header



0 15 16 31

<table>
<tr><td>4-bit version</td><td>4-bit header length</td><td>8-bit type of service (TOS)</td><td colspan="4">16-bit total length (in bytes)</td></tr>
<tr><td colspan="3">16-bit identification</td><td>0</td><td>DF</td><td>MF</td><td>13-bit fragment offset</td></tr>
<tr><td colspan="2">8-bit time to live (TTL)</td><td>8-bit protocol</td><td colspan="4">16-bit header checksum</td></tr>
<tr><td colspan="7">32-bit source IP address</td></tr>
<tr><td colspan="7">32-bit destination IP address</td></tr>
<tr><td colspan="7">options (if any)</td></tr>
</table>

20 bytes

libpcap

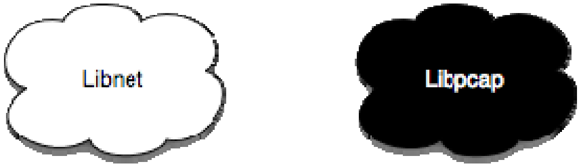
- **`u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)`**
  - reads the next packet and returns a u_char pointer to the *data* in that packet.

- **`int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)`**
  - keeps reading packets until cnt packets are processed or an error occurs.
  - callback – a function handler
  - user – optional arguments

References

- Unix Network Programming – Vol 1
- http://www.caida.org/outreach/resources/
- http://www.cet.nau.edu/~mc8/Socket/Tutorials/section1.html

What is libnet?



- A C Programming library for packet construction and injection
- The Yin to the Yang of libpcap
- Libnet's Primary Role in Life:
  - A simple interface for packet construction and injection
- Libnet IS good for:
  - Tools requiring meticulous control over every field of every header of every packet
- Libnet IS not well suited for:
  - Building client-server programs where the operating system should be doing most of the work

What’s inside of libnet?

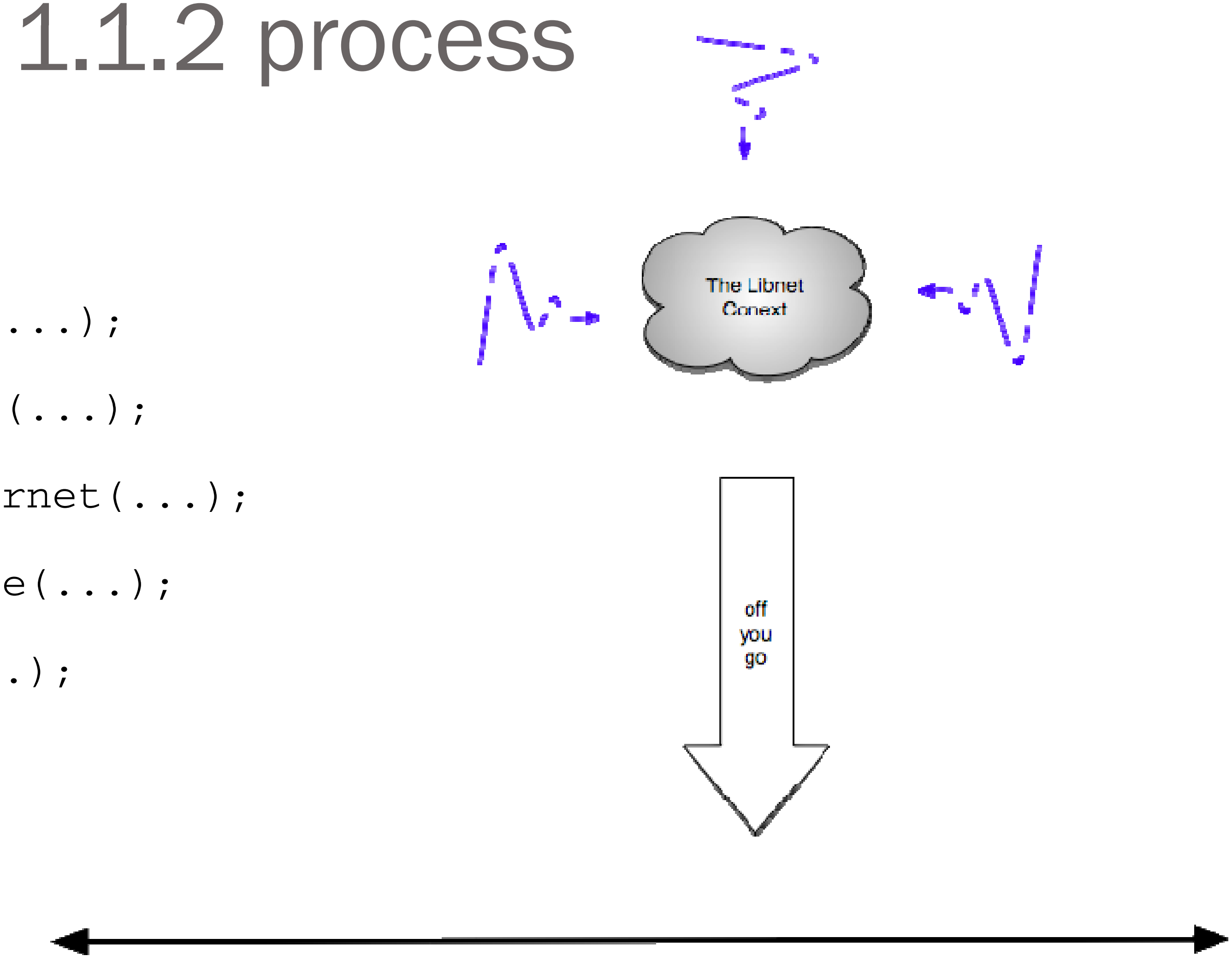
- As of libnet 1.1.2:
  - About 18,000 lines of C source code
  - 109 exported functions, 67 packet builder functions
  - Portable to all of today’s hottest operating systems:
    - Windows, OS X, BSD, Linux, Solaris, HPUX

Why use libnet?

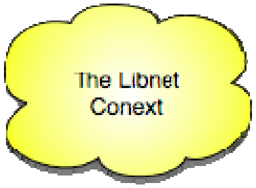
- Portability
  - Libnet is portable to all of our favorite and exquisitely cherished operating systems
- Ease of Use
  - As we will see, Libnet 1.1.x exports a braindead simple interface to building and injecting packets (4 easy steps)
- Robustness
  - Libnet supports all of today's in-demand protocols with more added all the time
    - More than 30 supported in Libnet 1.1.2 (see next slide)
    - Several link layers: Ethernet, Token Ring, FDDI, *802.11 planned*
- Open Source
  - Licensing
    - Libnet is released under a BSD license meaning it is basically free to use
  - Response-time in bug fixes
    - Large user-base; bugs are fixed quickly

Libnet 1.1.2 process

```
libnet_init(...);

libnet_build_tcp(...);

libnet_build_ipv4(...);

libnet_build_ethernet(...);

libnet_build_write(...);

libnet_destroy(...);
```



The libnet context

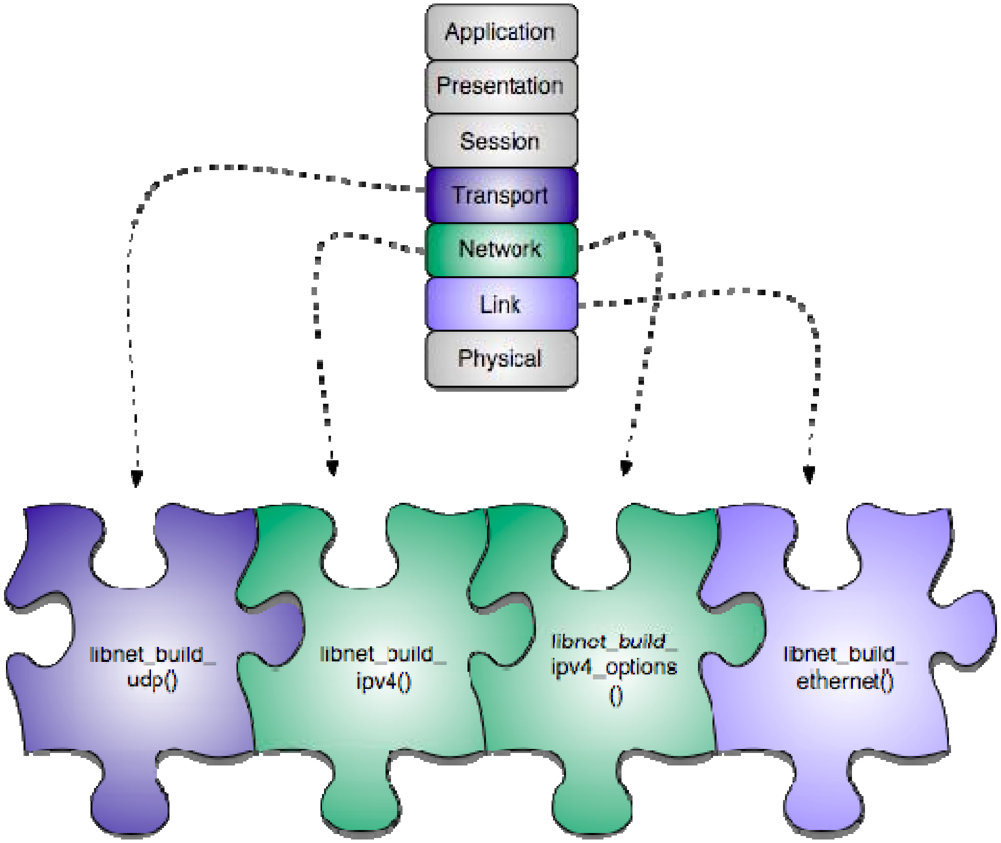


I'm super important

- Opaque monolithic data structure that is returned from `libnet_init();`
  - "`l`"
- Maintains state for the entire session
  - Tracks all memory usage and packet construction
  - Defines and describes a libnet session
- Used in almost every function
- (More detail later)

Packet construction

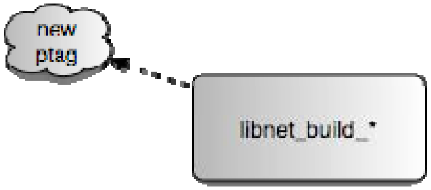
- The core of Libnet's functionality
- Packets are built in pieces
  - Each protocol layer is usually a separate function call
    - Generally two - four function calls to build an entire packet
- Packet builders take arguments corresponding to header values
- Approximates an IP stack; must be called in order
  - From the highest on the OSI model to the lowest
- A successful call to a builder function returns a ptag



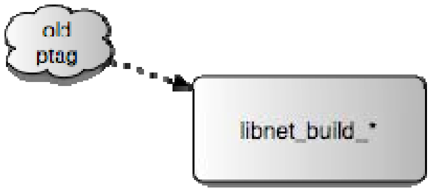
Packet construction

```
tcp = libnet_build_tcp(
      src_prt,                                    /* source port */
      dst_prt,                                    /* destination port
*/
      0x01010101,                                 /* sequence number
*/
      0x02020202,                                 /* acknowledgement
num */
      TH_SYN,                                     /* control flags */
      32767,                                      /* window size */
      0,                                          /* checksum */
      0,                                          /* urgent pointer */
      LIBNET_TCP_H + payload_s,                   /* TCP packet size
*/
      payload,                                    /* payload */
      payload_s,                                  /* payload size */
      l,                                          /* context */
      0);                                         /* ptag */
```

Ptags and Pblocks



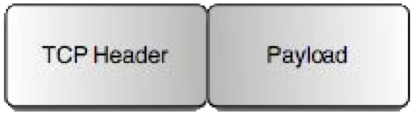
Creating a new protocol block; a new ptag is returned



Modifying an existing protocol block; an old ptag is passed in

- Protocol Tag == ptag
- Protocol Block == pblock
- Protocol Tags (ptags) used to track Protocol Blocks (pblocks)
  - Whenever a new packet piece is built it is stored in a pblock and a new ptag is returned
  - Whenever an existing packet piece is modified, an old ptag is used
    - Looped packet updating
- Ptags are handled directly by the user, pblocks are not

The payload interface



- A simple interface to append arbitrary payloads to packets
  - TCP, UDP, ICMP, IP
- All packet builder functions support this interface
- Use is optional

```
tcp = libnet_build_tcp(
      src_prt,                                      /* source port */
      dst_prt,                                      /* destination
port */
      0x01010101,                                   /* sequence number
*/
      0x02020202,                                   /* acknowledgement
num */
      TH_SYN,                                       /* control flags
*/
      32767,                                        /* window size */
      0,                                            /* checksum */
      0,                                            /* urgent pointer
*/
      LIBNET_TCP_H + payload_s,                /* TCP packet size */
```

Wire injection methods

- Raw socket interface (less complex)
  - Mid-level interface, packets built at the IP layer and above
    - No link header needs to be built
  - Removes all routing and interface decisions
  - Useful for "legitimate" packet tools that do not need to spoof address information
  - Packet passes through kernel's IP stack
    - Routing, checksums, firewalls all an issue
  - Less than granular level of control (next slide)
- Link layer interface (more complex)
  - Low-level interface, packets built at the link layer
  - Packet does not pass through the kernel's IP stack
    - Sovereign control of every field of the packet
    - All address and routing information needs to be provided
    - Some operating systems stamp outgoing MAC address of the Ethernet header (this is bypassable)

Raw Socket

	IP Fragmentation	IP Total Length	IP Checksum	IP ID	IP Source	Max size before kernel complains
Linux 2.2+	Performed if packet is larger than MTU	Always filled in	Always filled in	Filled in if left 0	Filled in if left 0	1500 bytes
Solaris 2.6+	Performed if packet is larger than MTU; Sets DF bit		Always filled in			
OpenBSD 2.8+	Performed if packet is larger than MTU		Always filled in			

Packet checksums

- Programmer no longer has to worry about checksum computation
- Common usage: programmer specifies a “0”; libnet autocomputes
  - Can be toggled off to use checksum of “0”
- Alternative usage: programmer specifies value, libnet uses that
  - Useful for fuzzing, using pre-computed checksums

```
ip = libnet_build_ipv4(
     LIBNET_IPV4_H + LIBNET_TCP_H + payload_s,   /* length */
     0,                                          /* TOS */
     242,                                        /* IP ID */
     0,                                          /* IP frag
*/
     64,                                         /* TTL */
     IPPROTO_TCP,                                /* protocol
*/
     0,                                          /* checksum
*/
     src_ip,                                     /* source IP
*/
     dst_ip,                                     /*
destination IP */
     NULL,                                       /* payload
*/
     0,                                          /* payload
size */
```

Initialization

```
libnet_t *
libnet_init(int injection_type, char *device, char *err_buf);
```

Initializes the libnet library and create the environment	
SUCCESS	A libnet context suitable for use
FAILURE	NULL, err_buf will contain the reason
injection_type	LIBNET_LINK, LIBNET_RAW4
device	“fxp0”, “192.168.0.1”, NULL
err_buf	Error message if function fails

```
l = libnet_init(LIBNET_LINK, “fxp0”, err_buf);
if (l == NULL)
{
    fprintf(stderr, “libnet_init(): %s”, errbuf);
}
```

Device (interface) selection

- Happens during initialization
- `libnet_init(LIBNET_LINK, “fxp0”, errbuf);`
  - Will initialize libnet’s link interface using the `fxp0` device
- `libnet_init(LIBNET_LINK, “192.168.0.1”, errbuf);`
  - Will initialize libnet’s link interface using the device with the IP address `192.168.0.1`
- `libnet_init(LIBNET_LINK, NULL, errbuf);`
  - Will initialize libnet’s link interface using the first “up” device it can find
  - `libnet_getdevice(l);`
- `libnet_init(LIBNET_RAW4, NULL, errbuf);`
  - Under the Raw socket interface no device is selected
    - Exception: Win32 does this internally since it is built on top of Winpcap
- New: devices with no IP address can be specified for use (stealth)

Error handling

```
char *
libnet_geterror(libnet_t *l);
```

Returns the last error message generated by libnet	
SUCCESS	An error string, `NULL` if none occurred
FAILURE	This function cannot fail
`l`	The libnet context pointer

```
l = libnet_autobuild_ipv4(len, IPPROTO_TCP, dst, l);
if (l == NULL)
{
    fprintf(stderr, "libnet_autobuild_ipv4(): %s",
        libnet_geterror(l));
}
```

Address resolution

```
u_int32_t
libnet_name2addr4(libnet_t *l, char *host_name, u_int8_t use_name);
```

Converts a IPv4 presentation format hostname into a big endian ordered IP number	
SUCCESS	An IP number suitable for use with libnet_build_*
FAILURE	-1, which is technically “255.255.255.255”
l	The libnet context pointer
host_name	The presentation format address
use_name	LIBNET_REOLVE, LIBNET_DONT_RESOLVE

```
dst = libnet_name2addr4(l, argv[optind], LIBNET_DONT_RESOLVE);
if (dst == -1)
{
    fprintf(stderr, “libnet_name2addr4(): %s”, libnet_geterror(l));
}
```

Address resolution

```
char *
libnet_addr2name4(u_int32_t address, u_int8_t use_name);
```

Converts a big endian ordered IPv4 address into a presentation format address	
SUCCESS	A string of dots and decimals or a hostname
FAILURE	This function cannot fail
`address`	The IPv4 address
`use_name`	`LIBNET_REOLVE`, `LIBNET_DONT_RESOLVE`

```
printf("%s\n", libnet_addr2name4(i, LIBNET_DONT_RESOLVE));
```

Packet construction: UDP

```
libnet_ptag_t
libnet_build_udp(u_int16_t sp, u_int16_t dp, u_int16_t len,
u_int16_t sum, u_int8_t *payload, u_int32_t payload_s, libnet_t
*l, libnet_ptag_t ptag);
```

Builds a UDP header	
SUCCESS	A ptag referring to the UDP packet
FAILURE	`-1`, and `libnet_get_error()` can tell you why
`sp`	The source UDP port
`dp`	The destination UDP port
`len`	Length of the UDP packet (including payload)
`sum`	Checksum, 0 for libnet to autofill
`payload`	Optional payload
`payload_s`	Payload size
`l`	The libnet context pointer
`ptag`	Protocol tag

Packet construction: IPv4

```
libnet_ptag_t
libnet_build_ipv4(u_int16_t len, u_int8_t tos, u_int16_t id,
u_int16_t frag, u_int8_t ttl, u_int8_t prot, u_int16_t sum,
u_int32_t src, u_int32_t dst, u_int8_t *payload,
u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);
```

Builds an IPv4 header	
SUCCESS	A ptag referring to the IPv4 packet
FAILURE	`-1`, and `libnet_get_error()` can tell you why
`len`	Length of the IPv4 packet (including payload)
`tos`	Type of service bits
`id`	IP identification
`frag`	Fragmentation bits
`ttl`	Time to live
`prot`	Upper layer protocol
`sum`	Checksum, 0 for libnet to autofill
`src`	Source IP address

Packet construction: IPv4

```
libnet_ptag_t
libnet_build_ipv4(u_int16_t len, u_int8_t tos, u_int16_t id,
u_int16_t frag, u_int8_t ttl, u_int8_t prot, u_int16_t sum,
u_int32_t src, u_int32_t dst, u_int8_t *payload,
u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);
```

Builds an IPv4 header	
SUCCESS	A ptag referring to the UDP packet
FAILURE	-1, and libnet_get_error() can tell you why
dst	Destination IP address
payload	Optional payload
payload_s	Payload size
l	The libnet context pointer
ptag	Protocol tag

Packet construction: Ethernet

```
libnet_ptag_t
libnet_build_ethernet(u_int8_t *dst, u_int8_t *src,
u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,
libnet_ptag_t ptag);
```

Builds an Ethernet header	
SUCCESS	A ptag referring to the Ethernet frame
FAILURE	`-1`, and `libnet_get_error()` can tell you why
`dst`	Destination ethernet address
`src`	Source ethernet address
`type`	Upper layer protocol type
`payload`	Optional payload
`payload_s`	Payload size
`l`	The libnet context pointer
`ptag`	Protocol tag

Shutdown

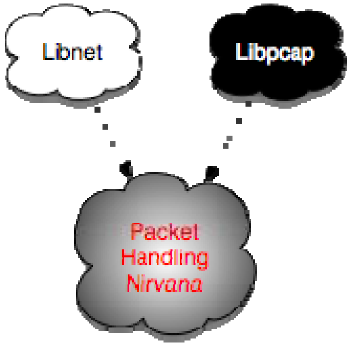
```
void
libnet_destroy(libnet_t *l);
```

Shuts down the libnet environment	
l	The libnet context pointer

```
libnet_destroy(l);
```

Libnet with other components

GNIP: *A poor man's ping*



A simple application

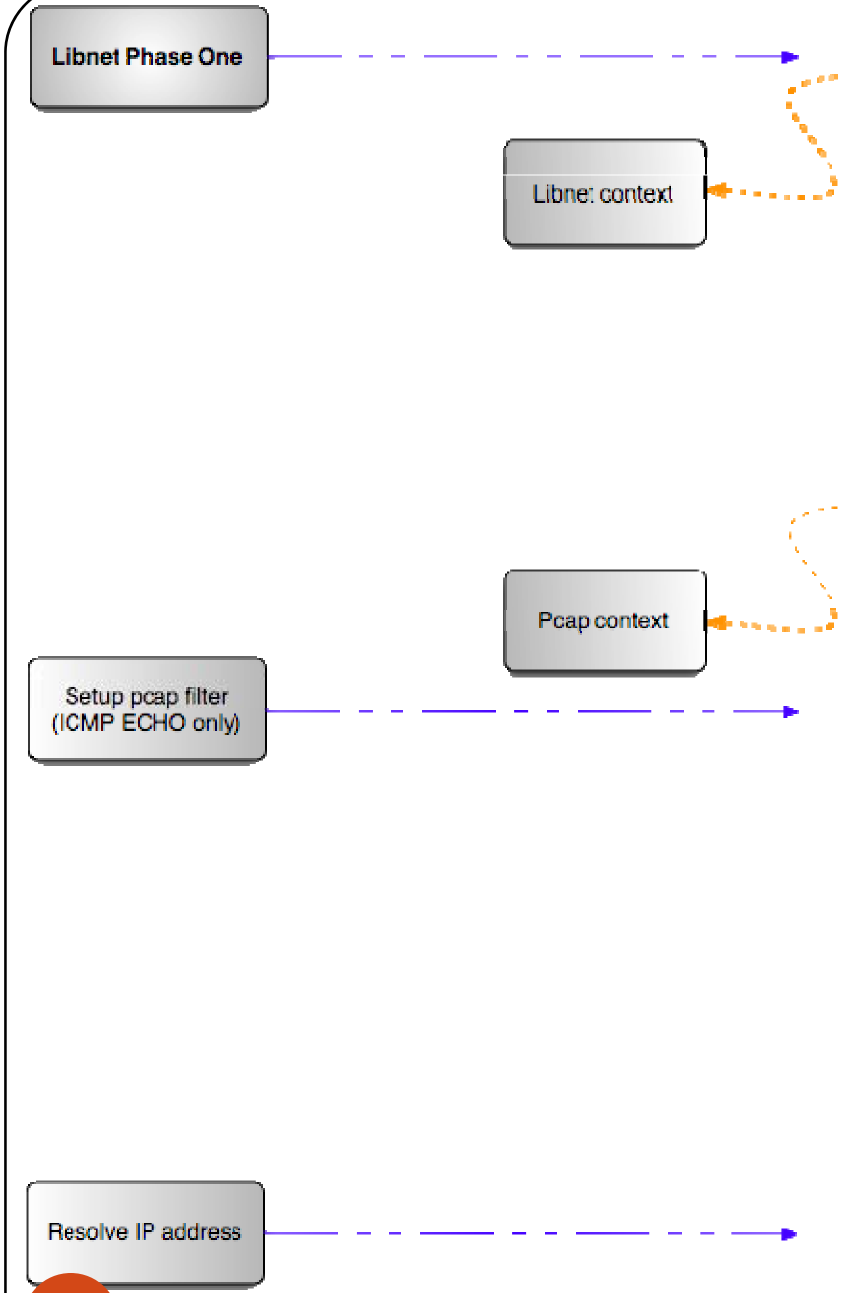
- Simple ping client
- 250 lines of source
- Illustrates some of libnet's (and libpcap's) core concepts
  - IPv4 packet construction
  - ICMP packet construction
  - Looped packet updating
  - Packet filtering, capturing and dissection

Libpcap packet filter (same as tcpdump)

Monolithic context variables

Side effect of closed interface

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

#define GNIP_FILTER "icmp[0] = 0"

void usage(char *);

int
main(int argc, char **argv)
{
    libnet_t *l = NULL;
    pcap_t *p = NULL;
    u_int8_t *packet;
    u_int32_t dst_ip, src_ip;
    u_int16_t id, seq, count;
    int c, interval = 0, pcap_fd, timed_out;
    u_int8_t loop, *payload = NULL;
    u_int32_t payload_s = 0;
    libnet_ptag_t icmp = 0, ip = 0;
    char *device = NULL;
    fd_set read_set;
    struct pcap_pkthdr pc_hdr;
    struct timeval timeout;
    struct bpf_program filter_code;
    bpf_u_int32 local_net, netmask;
    struct libnet_ipv4_hdr *ip_hdr;
    struct libnet_icmpv4_hdr *icmp_hdr;
    char errbuf[LIBNET_ERRBUF_SIZE];

    while((c = getopt(argc, argv, "I:i:c:")) !=
EOF)
    {
        switch (c)
        {
            case 'I':
                device = optarg;
                break;
            case 'i':
                interval = atoi(optarg);
                break;
            case 'c':
                count = atoi(optarg);
                break;
        }
    }

    c = argc - optind;
    if (c != 1)
    {
        usage(argv[0]);
```



Libnet Phase One

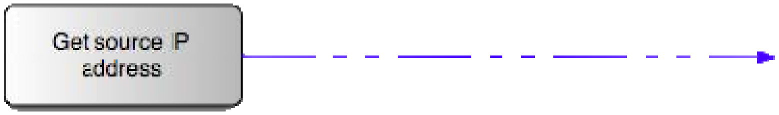
Libne: context

Pcap context

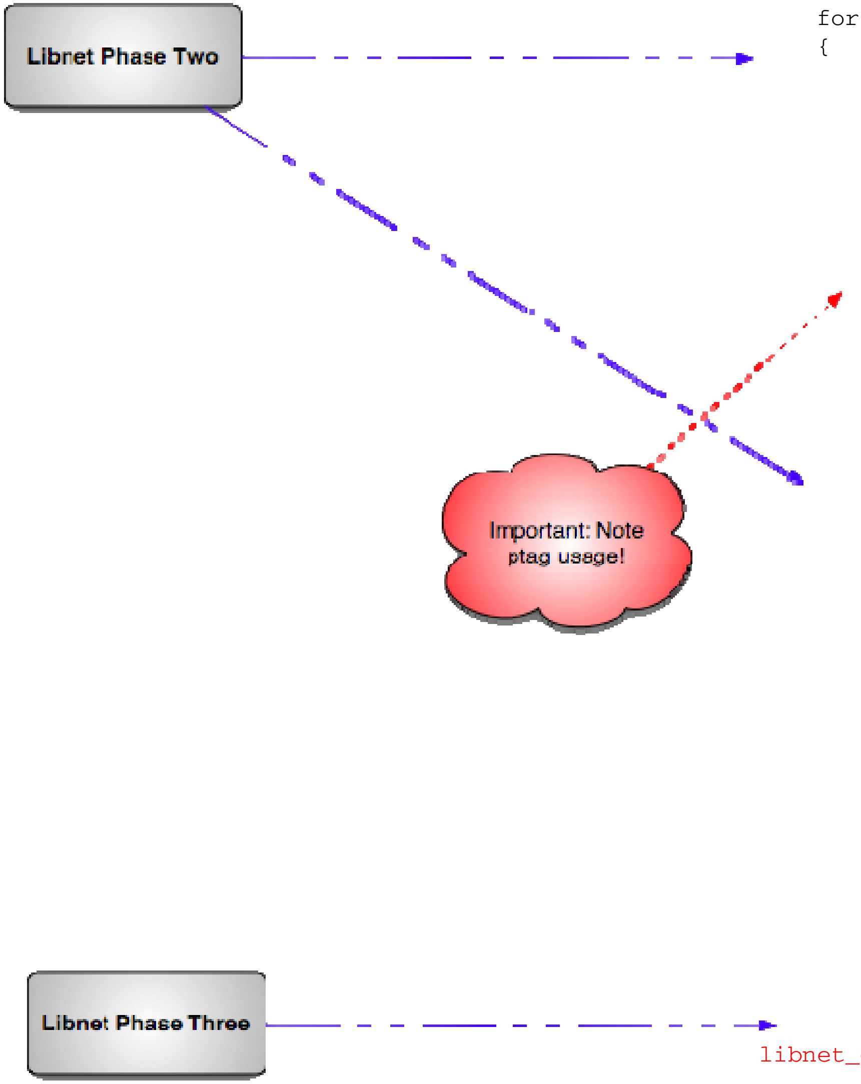
Setup pcap filter
(ICMP ECHO only)

Resolve IP address

```
/* initialize the libnet library */
l = libnet_init(LIBNET_RAW4, device, errbuf);
if (l == NULL)
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}

if (device == NULL)
{
    device = pcap_lookupdev(errbuf);
    if (device == NULL)
    {
        fprintf(stderr, "pcap_lookupdev() failed: %s\n", errbuf);
        goto bad;
   }
}

/* handcrank pcap */
p = pcap_open_live(device, 256, 0, 0, errbuf);
if (p == NULL)
{
    fprintf(stderr, "pcap_open_live() failed: %s", errbuf);
    goto bad;
}

/* get the subnet mask of the interface */
if (pcap_lookupnet(device, &local_net, &netmask, errbuf) == -1)
{
    fprintf(stderr, "pcap_lookupnet(): %s", errbuf);
    goto bad;
}

/* compile the BPF filter code */
if (pcap_compile(p, &filter_code, GNIP_FILTER, 1, netmask) == -1)
{
    fprintf(stderr, "pcap_compile(): %s", pcap_geterr(p));
    goto bad;
}

/* apply the filter to the interface */
if (pcap_setfilter(p, &filter_code) == -1)
{
    fprintf(stderr, "pcap_setfilter(): %s", pcap_geterr(p));
    goto bad;
}

dst_ip = libnet_name2addr4(l, argv[optind], LIBNET_RESOLVE);
if (dst_ip == -1)
{
    fprintf(stderr, "Bad destination IP address (%s).\n",
        libnet_geterror(l));
    goto bad;
}
```


Get source IP address

```
    src_ip = libnet_get_ipaddr4(l);
    if (src_ip == -1)
    {
        fprintf(stderr, "Can't determine source IP address
(%s).\n",
                libnet_geterror(l));
        goto bad;
    }

    interval ? interval : interval = 1;
    timeout.tv_sec = interval;
    timeout.tv_usec = 0;
    pcap_fd = pcap_fileno(p);

    fprintf(stderr, "GNIP %s (%s): %d data bytes\n",
            libnet_addr2name4(dst_ip, 1),
libnet_addr2name4(dst_ip, 0),
            LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_s);
```



```
    loop = 1;
    for (id = getpid(), seq = 0, icmp = LIBNET_PTAG_INITIALIZER; loop; seq++)
    {
        icmp = libnet_build_icmpv4_echo(
            ICMP_ECHO,                            /* type */
            0,                                    /* code */
            0,                                    /* checksum */
            id,                                   /* id */
            seq,                                  /* sequence number */
            payload,                              /* payload */
            payload_s,                            /* payload size */
            l,                                    /* libnet context */
            icmp);                                /* ptag */
        if (icmp == -1)
        {
            fprintf(stderr, "Can't build ICMP header: %s\n",
                libnet_geterror(l));
            goto bad;
        }

        ip = libnet_build_ipv4(
            LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_s, /* length */
            0,                                    /* TOS */
            id,                                   /* IP ID */
            0,                                    /* IP Frag */
            64,                                   /* TTL */
            IPPROTO_ICMP,                         /* protocol */
            0,                                    /* checksum */
            src_ip,                               /* source IP */
            dst_ip,                               /* destination IP */
            NULL,                                 /* payload */
            0,                                    /* payload size */
            l,                                    /* libnet context */
            ip);                                   /* ptag */
        if (ip == -1)
        {
            fprintf(stderr, "Can't build IP header: %s\n",
libnet_geterror(l));
            goto bad;
        }

        c = libnet_write(l);
        if (c == -1)
        {
            fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
            goto bad;
        }
```

Interface multiplexing

"Is this a response" logic

Libnet Phase Four

```
    FD_ZERO(&read_set);
    FD_SET(pcap_fd, &read_set);

    for (timed_out = 0; !timed_out && loop; )
    {
        c = select(pcap_fd + 1, &read_set, 0, 0, &timeout);
        switch (c)
        {
            case -1:
                fprintf(stderr, "select() %s\n", strerror(errno));
                goto bad;
            case 0:
                timed_out = 1;
                continue;
            default:
                if (FD_ISSET(pcap_fd, &read_set) == 0)
                {
                    timed_out = 1;
                    continue;
                }
                /* fall through to read the packet */
        }
        packet = (u_int8_t *)pcap_next(p, &pc_hdr);
        if (packet == NULL)
        {
            continue;
        }

        ip_hdr = (struct libnet_ipv4_hdr *)(packet + 14);
        icmp_hdr = (struct libnet_icmpv4_hdr *)(packet + 14 +
                (ip_hdr->ip_hl << 2));
        if (ip_hdr->ip_src.s_addr != dst_ip)
        {
            continue;
        }
        if (icmp_hdr->icmp_id == id)
        {
            fprintf(stderr, "%d bytes from %s: icmp_seq=%d ttl=%d\n",
                ntohs(ip_hdr->ip_len),
                libnet_addr2name4(ip_hdr->ip_src.s_addr, 0),
                icmp_hdr->icmp_seq, ip_hdr->ip_ttl);
        }
    }
}

libnet_destroy(l);
pcap_close(p);
return (EXIT_SUCCESS);
```

GNIP output

```
[rounder:Projects/misc/] root# ./gnip 4.2.2.2
GNIP vnsc-bak.sys.gtei.net (4.2.2.2): 28 data bytes
28 bytes from 4.2.2.2: icmp_seq=0 ttl=247
28 bytes from 4.2.2.2: icmp_seq=1 ttl=247
28 bytes from 4.2.2.2: icmp_seq=2 ttl=247
28 bytes from 4.2.2.2: icmp_seq=3 ttl=247
28 bytes from 4.2.2.2: icmp_seq=4 ttl=247
^C
```