

- EN KIOSQUE ACTUELLEMENT ! -

Building packets for dummies and others with libnet

[Intro](#)

[Easy building](#)

[Advanced features](#)

[Routing](#)

[References](#)

[Changelog](#)

Introduction

Libnet [1] is a C portable library for packet creation and injection. It was made available in 1998 in its early version. This was the first time an *open source* and *portable* library allowed programmers to build packets, and send them to the wire. This first version have been made by Mike Schiffman and already provides ways to build tons of packets.

Unfortunately, the version 1.0 lacks some features, and was even counter-intuitive in some situations: developer still had to manage memory for its packets through `malloc()/free()`, checksumming was a real pain, packets were built from layer 1 to layer 7, ...

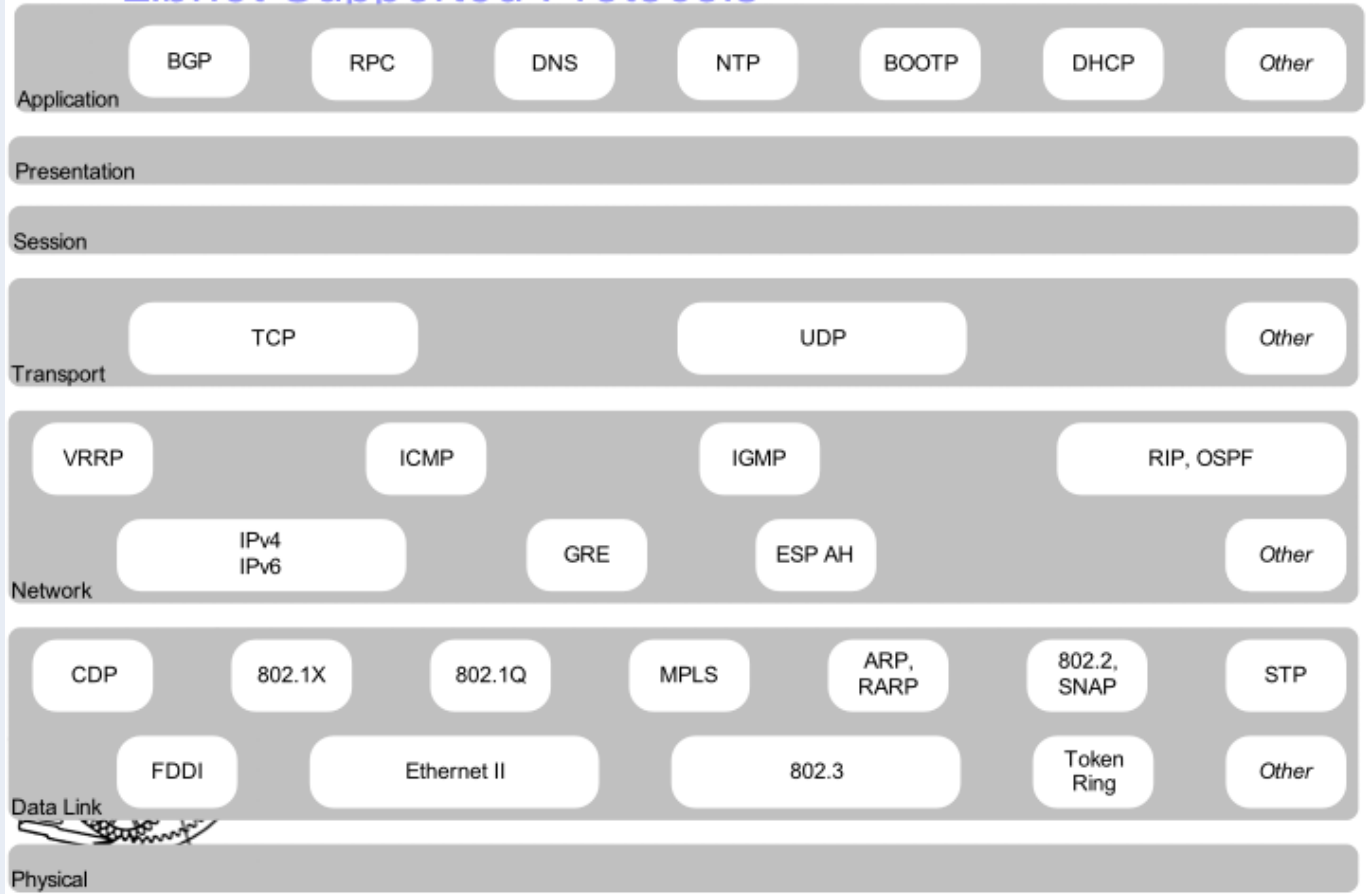
So, one day in a not so far away past, Mike decides to change the libnet's API and fix lost of the first release problems. Here comes the version 1.1.x, which is the subject of this article.

With this new API comes a lot of cool features, making life really easier:

- always more new protocols are supported.
- packets are built top-down, i.e. in the way they are built when you do it through any application.
- you dont have to manage memory: all that is now fully transparent to the user.
- unless you want to provide your own (and maybe wrong) checksum, all the required computations are made automatically.
- it is greatly configurable as you can chose to have control on every parameter or, on the contrary, let libnet decides for you.

These are only examples, and this article will focus on lots of libnet's features.

Libnet Supported Protocols



Last but not least, one may wonder when he should use libnet. The answer is easy: as soon as you have an application that built specific packets, libnet is the required library. Of course, you usually do not write applications that only send packets, they also capture some, and compare them, ... That makes other libraries like libpcap [3], libnids [4], or libdnet [5] very complementary: libnet is a building block used to construct **bigger things**.

Building packets made easy

Building packets with libnet-1.1.x, there are only four steps to send a packet:

1. initialize the *context* of the packet, i.e. chose interface and the level of control you want to have on the packet.
2. build each *protocol block* (referenced as *pblock* in the following), so for instance, to send a ping, build the echo-request pblock, then IP pblock (and then optionally the Ethernet pblock).
3. write the packet to wire: if checksums are needed, they will automatically be computed when pblocks are coalesce in the packet.
4. clean your context once you have done with it.

In this section, we will explore each of these steps. But before doing that, we will shortly introduce the 2 main structures now used in libnet: the context and the pblocks.

Context and pblocks

A *context* is a C structure used in libnet to define where and how the packet is going to be built. Thus, it contains some internals but also data provided by the user along the existence of the packet that is going to be built:

```
typedef struct
{
    int fd;                /* general parameters */
    int protocol;
    int injection_type;
```

```

libnet_pblock_t *protocol_blocks; /* protocol headers / data */
libnet_pblock_t *pblock_end;

int link_type; /* write parameters */
int link_offset;
int aligner;
u_char *device;

struct libnet_stats stats; /* misc. */
libnet_ptag_t ptag_state;
char label[LIBNET_LABEL_SIZE];
char err_buf[LIBNET_ERRBUF_SIZE];
} libnet_t;

```

Note that **a user will never manipulate directly a context**. The only thing a user will see is a pointer of such a structure, that he will have to provide to each function he'll call.

But the core element of libnet is *pblock*. First of all, as with the context, **a user will never manipulate directly a pblock**.

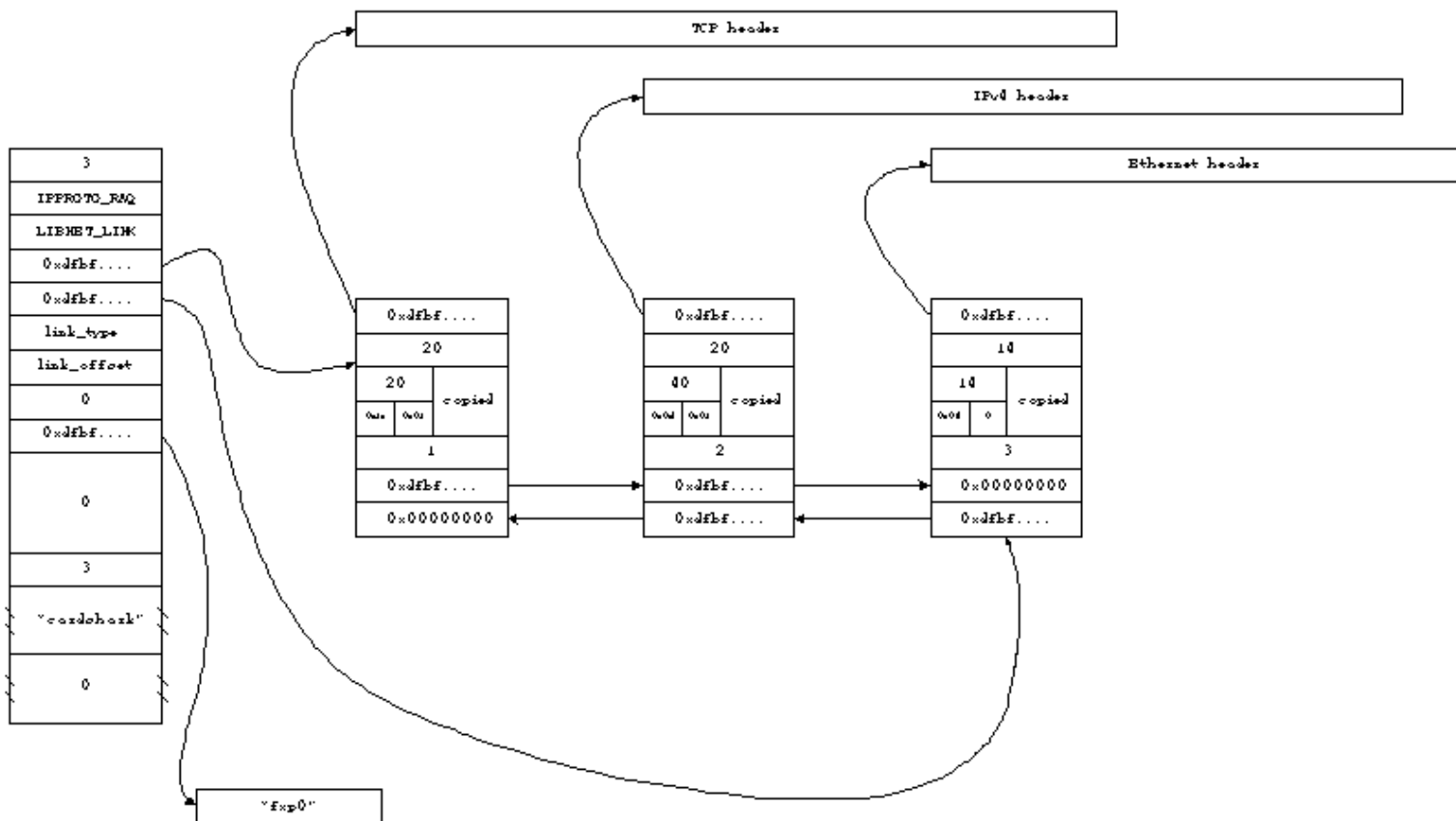
With so many protocols supported, you may wonder how libnet is doing! In fact, it will use the same trick as the one used with sockets or file descriptors on most of the system: each time you build a protocol block, you will receive back an integer, called a *protocol tag (ptag)* that will refer to that specific pblock. You can see it as a descriptor on a protocol block. So, the only thing manipulated by the user is an integer. All the "mess" is hidden in the C structure:

```

struct libnet_protocol_block
{
    u_char *buf; /* protocol buffer */
    u_long b_len; /* length of buf */
    u_long h_len; /* header length (for checksumming) */
    u_long copied; /* bytes copied */
    u_char type; /* type of pblock */
    u_char flags; /* control flags */
    libnet_ptag_t ptag; /* protocol block tag */
    struct libnet_protocol_block *next; /* next pblock */
    struct libnet_protocol_block *prev; /* prev pblock */
};
typedef struct libnet_protocol_block libnet_pblock_t;

```

The last point is how context and pblocks are related ? If you look in the above definition of a pblock, it contains all that is needed to create a double linked list (pointers *prev* and *next*). And in the definition of a context, one can find the head (*protocol_blocks*) and the tail (*pblock_end*) of this list.



It is time now to build packets :)

Initializing the context

As we have already seen, a context is the first step required to build packets. This is done with the following function:

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

- *injection_type*: this decides the way you will use libnet, depending on the control you need of the packets.
 - LIBNET_RAW4, LIBNET_RAW6: build a raw packet, thus you control anything from IP layer to above, respectively in IPv4 or IPv6
 - LIBNET_LINK, LIBNET_LINK6: build a packet from link-layer to upper levels.

The decision whether you should use the RAW or LINK interface depends mainly of your application: if it is LAN oriented, you may need the link-layer interface. Otherwise, RAW interface will be the one you need as all the ARP resolution will then be handled by your system.

Lastly, all these interfaces are also in an advanced mode (add the prefix `_ADV` for users requiring additional control over the packet creation or injection process.

- *device*: specify the device you will use for the injection, either by giving its name (`eth0`), or an IP address.

Warning

in RAW mode, routing is done by the kernel on your packet. It may be injected on another device than the one you specify here.

FRED: check what you wrote above ... I need to look in my mails and perform some tests.

- *err_buf*: as `libnet_init()` is the first function you will call, no structure is yet prepared. Thus, if there is a failure during initialization it needs a buffer to write the error message. This buffer is assumed to be `LIBNET_ERRBUF_SIZE` bytes long.

In case of failure, *err_buf* is filled and NULL is returned. In case of success, a pointer on a context (`libnet_t`) is returned. It will have to be provided to any further calls of libnet's functions.

Example 1: initialization

```
libnet_t *l;
char errbuf[LIBNET_ERRBUF_SIZE];
l = libnet_init(LIBNET_RAW4, /* Injection type */
               NULL, /* Network interface */
               errbuf); /* errbuf */
```

We build a context for RAW IPv4 packets. We don't specify a device and libnet will resolve it by taking the first up device.

Building pblocks

libnet lets you build lots of protocol blocks. All the provided functions are on the same model:

- protocol specific options;
- a *payload* and its size: that is pointer on `u_char*` that describe whatever you want. For instance, it can be some parts of a packet in an ICMP error message, or a DNS query/answer;
- the context;
- a protocol tag: if it is 0, it means a new protocol block will be build. Otherwise, libnet will look for the corresponding protocol block, and change it according to the new provided values.

Example 2: building protocol blocks

We want to build an application sending a DNS query. So, we will need a DNS pblock, an UDP pblock and an IPv4 pblock. Let's define some argument of our program we will use later:

- `char *query`: what we want to resolve.
- `u_long src_ip = 0, dst_ip = 0`: addresses for the client and the DNS server.

Even if libnet does a lot of things, we still need to build the query by ourself:

```
/* build the query that is going to be send to the dns server */
char payload[1024];
u_short payload_s;
payload_s = sprintf(payload, sizeof payload, "%s%c%c%c%c",
                   (char)(strlen(query)&0xff), query, 0x00, 0x00, 0x01, 0x00, 0x01);
```

Now, we can build our protocol blocks:

```
/* declare all ptag I need */
libnet_ptag_t ip = 0, udp = 0, dns = 0;
dns = libnet_build_dnsv4(
    LIBNET_UDP_DNSV4_H, /* TCP or UDP */
    0x7777, /* id */
    0x0100, /* this is a request */
    1, /* num_q */
    0, /* num_anws_rr */
    0, /* num_auth_rr */
    0, /* num_addi_rr */
    payload,
    payload_s,
    1,
    0
);
if (dns == -1)
{
    fprintf(stderr, "Can't build DNS packet: %s\n", libnet_geterror(1));
    goto bad;
}
```

We use the payload to give the query to the DNS server. The nature of the payload depends on the protocol, and here, of the 3rd argument (`flags`) which describe what kind of packet it is.

The return value of this builder is a `ptag` (a protocol tag). As we set it to 0 in the argument list, a new protocol block will be built. We store the dns `ptag` to test the return value, and for a (potential) next use.

As we did with DNSv4 protocol block, we also need to build UDP and IPv4 protocol blocks from scratch:

```
udp = libnet_build_udp(
    0x6666,          /* source port */
    53,             /* destination port */
    LIBNET_UDP_H + LIBNET_UDP_DNSV4_H + payload_s, /* packet length */
    0,             /* checksum */
    NULL,         /* payload */
    0,           /* payload size */
    1,          /* libnet handle */
    0);        /* libnet id */
if (udp == -1)
{
    fprintf(stderr, "Can't build UDP header: %s\n", libnet_geterror(1));
    goto bad;
}
```

The important argument here is the packet length. To compute it, we need to know what will be placed "above", in the upper layer of the DNS packet: you just have to know about the encapsulation process used in TCP/IP model to be able to compute this length.

Here, we have an UDP packet (`LIBNET_UDP_H`) that will contain a DNS header (`LIBNET_UDP_DNSV4_H`) for a query (`payload_s`). Thus, our packet's length is the sum of all these elements.

Next step is to build the IPv4 header:

```
ip = libnet_build_ipv4(
    LIBNET_IPV4_H + LIBNET_UDP_H + LIBNET_UDP_DNSV4_H + payload_s, /* length */
    0,                    /* TOS */
    242,                 /* IP ID */
    0,                   /* IP Frag */
    64,                  /* TTL */
    IPPROTO_UDP,        /* protocol */
    0,                   /* checksum */
    src_ip,              /* source IP */
    dst_ip,              /* destination IP */
    NULL,                /* payload */
    0,                   /* payload size */
    1,                   /* libnet handle */
    0);                  /* libnet id */
if (ip == -1)
{
    fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(1));
    exit(EXIT_FAILURE);
}
```

Now, the length of the packet has changed: it is the same as the UDP length (`LIBNET_UDP_H + LIBNET_UDP_DNSV4_H + payload_s`), but we also need to add the IPv4 header length itself (`LIBNET_IPV4_H`).

It is over as we have all our protocol blocks. They are stored in the context `l`.

Writing to the wire

this step is really easy as there is only one function to call, which wraps the real writing, depending on the kind of injection you requested.

```
int      ( libnet_t * ,
libnet_write
      )
```

This function returns the amount of bytes written to the wire. So you can check the expected packet size with what have been written. Anyway, if there is a failure, -1 is returned.x

Example 3: writing to the wire

```
c = libnet_write(l);
if (c == -1)
{
    fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
    goto bad;
}
else
{
    fprintf(stderr, "Wrote %d byte DNS packet; check the wire.\n", c);
}
```

Clean up

Once it is over, and if you are a "clean programmer" you will want to free memory you used in libnet. That is done calling a single function.

```
void      ( libnet_t * ,
libnet_destroy
      )
```

In fact, this function does a bit more than only freeing memory used for protocol blocks. It also closes the network interfaces, and some internals used in libnet.

Example 4: clean up

```
libnet_destroy(l);
return (EXIT_SUCCESS);
bad:
libnet_destroy(l);
return (EXIT_FAILURE);
```

Last but not least

In the above examples, we set the checksum in IPv4 and UPd protocol blocks to 0 so that libnet does the computation itself. If we had set any other value, this computation would have been skipped. This behavior is controlled by the function `libnet_toggle_checksum()`.

Nevertheless, remember that checksum of IPv4 header is always computed by the system with RAW interface.

FRED: check what you wrote above ...

Another cool and efficient feature of libnet is its capacity to set local parameters by itself if you dont do it. For instance, if you give a 0 IP address in an IP protocol block, it will be automatically replaced by the address of the device you are using.

But there is an even better solution for the most laziest guys. Some builders are in fact auto-builders and you just have to provide very few parameters: `libnet_autobuild_ethernet()`, `libnet_autobuild_fddi()`, `libnet_autobuild_token_ring()`, `libnet_autobuild_link()`, `libnet_autobuild_arp()`, `libnet_autobuild_ipv4()`, `libnet_autobuild_ipv6()`.

Understanding routing with libnet

It is not because you set `device="eth0"` that your packet will be written to `eth0` :(This section explains what and how routing is done when writing a packet.

Here is my routing configuration:

```
$ netstat -nr
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window irtt Iface
192.168.0.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0
10.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth1
0.0.0.0 192.168.0.1 0.0.0.0 UG 0 0 0 eth0
```

Let's see what happens depending on the destination IP address and the initialization of libnet.

Raw packets - act I

- default route

```
[root@batman sample]# ./ip_raw -i eth0 -s 1.1.1.1 -d 2.2.2.2
libnet 1.1 packet shaping: IP + payload[raw]
Wrote 28 byte TCP packet; check the wire.
```

Packet is sent through eth0 as expected.

- route 192.168.0.0

```
[root@batman sample]# ./ip_raw -i eth0 -s 1.1.1.1 -d 192.168.0.1
libnet 1.1 packet shaping: IP + payload[raw]
Wrote 28 byte TCP packet; check the wire.
```

Packet is sent through eth0 as expected.

- route 10.0.0.0

```
[root@batman sample]# ./ip_raw -i eth0 -s 1.1.1.1 -d 10.0.0.3
libnet 1.1 packet shaping: IP + payload[raw]
Wrote 28 byte TCP packet; check the wire.
```

Here, with tcpdump, we get:

```
[root@batman sample]# tcpdump -n -i eth1 -s 0 -X -vvv
tcpdump: listening on eth1
11:53:57.674190 arp who-has 10.0.0.3 tell 10.0.0.1
0x0000 0001 0800 0604 0001 5254 05fd cc20 0a00 .....RT.....
0x0010 0001 0000 0000 0000 0a00 0003 .....
...
```

As this is a LIBNET_RAW4 use, the ARP resolution is made by the kernel, and it tries to lookup 10.0.0.3 which does not exist on my network. But anyway, device is eth0 and packet goes through eth1.

Raw packets - act II

Now, we force the device to eth1.

- default route

```
[root@batman sample]# ./ip_raw -i eth1 -s 1.1.1.1 -d 2.2.2.2
libnet 1.1 packet shaping: IP + payload[raw]
Wrote 28 byte TCP packet; check the wire.
```

The packet is captured on **eth0**:

```
[root@batman samples]# tcpdump -n -i eth0 -s 0 -X -vvv host 1.1.1.1
tcpdump: listening on eth0
12:05:05.339474 1.1.1.1.4369 > 2.2.2.2.8738: [udp sum ok] udp 0 (ttl 64, id 242, len 28)
0x0000      4500 001c 00f2 0000 4011 73da 0101 0101   E.....@.s.....
0x0010      0202 0202 1111 2222 0008 c6a5                ....."".....
```

No need to perform more experiments as the default route is used. It will be the same for a packet destined to 192.168.0.0/24.

Conclusion on routing and LIBNET_RAW

Even if this behavior can be surprising, it is not a bug: it is the normal behavior! With the raw interface, packet is provided to the kernel that perform some more operations (IP checksum for instance) before `_routing_` the packet.

Link layer

Let's see now what happens with LIBNET_LINK interface.

- default route

```
[root@batman sample]# ./ip_link -i eth1 -d 1.2.3.4
libnet 1.1 packet shaping: IP + payload[link]
Using device eth1
Wrote 42 byte IP packet; check the wire.
```

If routing were made by the kernel, this packet would go through the default route, aka eth0. But it is not ;)

```
[root@batman raynal]# tcpdump -n -i eth1 -s 0 -X -vvv
tcpdump: listening on eth1
17:47:45.125842 1.1.1.1.4369 > 1.2.3.4.8738: [bad udp cksum fdff!] udp 0 (ttl 64, id 242, len
28)
0x0000      4500 001c 00f2 0000 4011 73d8 0101 0101   E.....@.s.....
0x0010      0102 0304 1111 2222 0008 c6a5                ....."".....
```

So, using the LIBNET_LINK interface let you chose where you packet will really be written.

Conclusion

If you dont want your packet being routed by the system, use the link layer interface, or set the appropriate routes on your host ;)

Advanced features

Accessing raw data

Sometimes, you may want to access the raw buffers used for the protocol blocks, or for the whole packet. That is available iff you have initialized libnet in advanced mode (one of the above mode, suffixed with `_ADV`).

There are 2 functions that provides a pointer on internal data. Thus any modification made on what is pointed is also reported in the libnet's internals.

```
int
libnet_adv_cull_packet ( libnet_t * l,
                        u_int8_t ** packet,
                        u_int32_t * packet_s
                      )
```

The function `libnet_adv_cull_packet()` coalesces all the protocol blocks in a single packet, and put it in `packet`. Nevertheless, you sometimes simply need to get a protocol block. Then you can use `libnet_adv_cull_header()`.

```

int
libnet_adv_cull_header ( libnet_t * l,
                        libnet_ptag_t ptag,
                        u_int8_t ** header,
                        u_int32_t * header_s
                      )

```

Building multiple packets

In the version 1.1.0, libnet lets you build as many packets as you want, but you have to handle each context (the `libnet_t *`) by yourself. So, we provide a new data structure, a context queue (cq) to store/manage all the context you want.

Hence, you can describe several packets built on different protocols and injected through different interfaces. The only necessity is to use a new `libnet_t *` context for each of them. Then, you feed the cq with these contexts and use a built-in loop to go across the contexts.

A cq is in fact a double linked list, but the user just has to use some functions to manipulate it: ul>

- *Initialization*: there is no initialization. You just have to push a context in the queue with `libnet_cq_add()`.

```
int libnet_cq_add(libnet_t * l, char *label)
```

The label has to be unique for each newly added context. Each context must be unique, i.e. different labels and different addresses for all the contexts (by "address" here, we mean the pointer allocated somewhere in memory, not the IP ones).

- *Manipulating contexts*:

there 2 ways to retrieve context from the queue, depending on what you want to perform:

1. Running through all the contexts: either use the macro

```
for_each_context_in_cq()
```

or call the 3 functions simulating an iterator:

```
libnet_t * libnet_cq_head();
int libnet_cq_last();
libnet_t * libnet_cq_next();
```

2. Finding a context: then, just call

```
libnet_t *libnet_cq_find_by_label(char *label);
```

You will get a pointer on the proper context if it exists, null otherwise.

Lastly, you can remove a context from the queue:

```
libnet_t * libnet_cq_remove(libnet_t *l);
libnet_t * libnet_cq_remove_by_label(char *label);
```

In both cases, the return value is the pointer on the context that have been removed from the queue.

- *destroy the queue*: when you dont need it anymore, free its memory

```
void libnet_cq_destroy()
```

Example 5: smurf with context queue.

First step is to build all the needed context. The context are built in the usual way, so we wont put all the code. Once a context is fully described, it is added to the queue.

```

/* Create a context for each amplifier */
for (i = 0; optind < argc; optind++) {
    l = libnet_init(...);
    t = libnet_build_icmpv4_echo(..., 0);
    t = libnet_build_ipv4(..., 0);

    /* and finally, put it in the arena */
    snprintf(label, sizeof(label)-1, "echo%d", i);
    if (libnet_cq_add(l, label) == -1)
    {
        fprintf(stderr, "add error: %s\n", libnet_geterror(l));
        goto bad;
    }
}

```

All build functions take a ptag of 0 as argument. This is because all pblocks in each context must be independent. Thus you need to re-build each pblock for each context, even if they contain exactly the same bytes (that is on my TODO list ;-).

Now that the cq is filled, we can retrieve all contexts to use them.

```

for_each_context_in_cq(l) {
    for (i = 0; i < count; i++) {
        c = libnet_write(l);
        if (c == -1)
        {
            fprintf(stderr, "Write error (pkt #%d): %s\n", i, libnet_geterror(l));
            goto bad;
        }
        else
        {
            fprintf(stderr, "Wrote %d byte ICMP packet; check the wire.\n", c);
        }
    }
}

```

It's over :)

Playing with payload

We have already seen an example on usage of the payload. Another situation where it is required is with ICMP error messages. But the most interesting feature provided by the payload is that **it allows libnet to support any kind of packet, even those not yet supported by an official builder!** I told you it was cool ;)

Example 6: building a TFTP scanner.

The first step is to build the TFTP part, which will be a payload for the UDP protocol block.

```

/*
 * build payload
 *
 *      2 bytes      string      1 byte      string      1 byte
 *      -----
 *      | Opcode |  Filename |    0 |    Mode   |    0 |
 *      -----
 */
payload_s = 2 + strlen(filename) + 1 + strlen(mode) + 1;
payload = malloc(sizeof(char)*payload_s);
if (!payload)
{
    fprintf(stderr, "malloc error for payload\n");
    goto bad;
}
memset(payload, 0, payload_s);
payload[1] = 1; /* opcode - GET */
memcpy(payload + 2, filename, strlen(filename));
memcpy(payload + 2 + strlen(filename) + 1, mode, strlen(mode));

```

The payload is only a buffer that matches the description of the protocol block we want. So, we just have to look at the proper RFC to find the required structure, and fill our buffer with the corresponding data. Next step is to build the UDP block, with this buffer as payload.

```

udp = libnet_build_udp(
    0x1234, /* source port */
    69, /* destination port */
    LIBNET_UDP_H + payload_s, /* packet length */
    0, /* checksum */
    payload, /* payload */
    payload_s, /* payload size */
    1, /* libnet handle */
    0); /* libnet id */
if (udp == -1)
{
    fprintf(stderr, "Can't build UDP header: %s\n", libnet_geterror(1));
    goto bad;
}

```

We just have to take care at the length of this protocol block. Of course, as this is a UDP pblock, the base length is `LIBNET_UDP_H`. But as there is a payload, we must also mention it (see the red line above). Time for IP pblock now.

```

ip = libnet_build_ipv4(
    LIBNET_IPV4_H + LIBNET_UDP_H + payload_s, /* length - dont forget the
UDP's payload */
    0, /* TOS */
    0x4242, /* IP ID */
    0, /* IP Frag */
    0x42, /* TTL */
    IPPROTO_UDP, /* protocol */
    0, /* checksum */
    src_ip, /* source IP */
    dst_ip, /* destination IP */
    NULL, /* payload (already in UDP) */
    0, /* payload size */
    1, /* libnet handle */
    0); /* libnet id */
if (ip == -1)
{

```

```

        fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(1));
        goto bad;
    }

```

Mixing libpcap and libnet

TODO ...

What you really control with RAW packets

When manipulating packets through the RAW interface, the socket is located at IP layer. A buffer is built in your program (in *user space*). As soon as `write()` is called with your buffer, it is transferred into the *kernel space* to the *IP stack*. Then, some computations are made on your packet before it is passed to the driver of the network interface, which finally write the packet to the wire.

When the packet is in the IP stack, some parameters are adjusted, depending on the Operating System. Those parameters are marked by a **K** in the table right below. That means you can put whatever value you want, it will be overwritten by the kernel.

Help :)

	Length	ToS	IP ID	IP Frag	TTL	Proto	Checksum	IP src	IP dst
Linux 2.4	K						K		

References

- [1] Libnet
<http://www.packetfactory.net/libnet>
- [2] The Libnet Reference Manual v.01
<http://www.phrack.org/phrack/55/P55-06>
Mike Schiffman
- [3] libpcap
<http://www.tcpdump.org>
- [4] libnids
<http://www.packetfactory.net/libnids>
- [5] libdnet
<http://libdnet.sourceforge.net/>
- [3] Libnet for dummies - Libre en fête 2003
<http://www.security-labs.org/Libnet/libnet-1024x768/>
Frédéric Raynal

Changelog

Mon Dec 29 17:55:19 CET 2003

- add routing section
- add changelog

Frédéric Raynal - f.raynal@miscmag.com

Last modified: Mon Dec 29 18:17:21 CET 2003

frederic.raynal@security-labs.org

Last modified : Monday December 29 2003 -- 18:18

🇫🇷 Tous les documents présents sur ce site sont soumis à la [GNU Free Documentation License](#).

🇬🇧 All available documents on these pages are under the [GNU Free Documentation License](#)