Libnet 1.1 tutorial

2011-09-18

Index

Introduction

Before you try any of this, you need to have libnet installed and you need to know how to use `su` (or `sudo`). You can download the most up-to-date version of libnet I know of **here** (development goes on **here**).

You can download all the sample code you'll find throughout this tutorial **here**. No need to copy and paste.

How libnet works

Here's what you need to do to start injecting packets:

1) Fire up libnet with `libnet_init()`.
2) Build all headers, from the highest layer to the lowest. Say you'd like to build a UDP packet over IPv4 over Ethernet, with full control over all headers. You would need to call `libnet_build_udp()`, `libnet_build_ipv4()` and `libnet_build_ethernet()`, in this particular order.
3) Write the packet with `libnet_write()`.
4) Prepare for sending another packet by doing *one* of the following:
4a) Clear the packet with `libnet_clear_packet()`. Go back to 2) and write a different packet.
4b) Go back to 2) and update the packet using the same build functions, but feeding them the tags they returned on the last call.
5) When done sending all the packets you wanted to, clean up with `libnet_destroy()` and exit.

Understanding that logic is the hardest part. Now all you need to do is take a closer look at each function. `man libnet-functions.h` and `man libnet-headers.h` are going to be your best friends at this point.

The libnet context

Almost every libnet function receives a pointer of type `libnet_t*` to what is called the *libnet context* (called `l` in libnet's functions). This is merely all your stuff that libnet has stored in memory, like your headers and so on. It allows you to reuse and modify your previously created packets (see the section on **tags**), or even to have two or more stacks of headers in memory so you can alternate between them without having destroy and rebuild them every time.

Make sure to create the context at the beginning of your program and only destroy it at the end, when you're done sending packets. That is, *don't initialize and destroy a new context for every new packet you build*. Besides taking a lot more time to allocate and free all the memory, if you do it fast enough, it will even crash your program. So don't.

Receiving packets and checking your own

Libnet is only useful for *injecting* packets, not for *capturing* them. Libnet is actually incapable of receiving packets at all. If your program requires this functionality, please refer to **libpcap**.

To merely look at which replies you're receiving, or to look at your own packets which libnet is sending, you can use any sniffer. The best one is said to be **wireshark**, while**tcpdump** is a lighter command line alternative.

Tip on working with libnet and pcap: (If you don't understand anything I'm saying here, don't worry, come back after reading the rest of the tutorial, especially the part on the **build functions**.) When using information from captured packets in libnet functions, I find it is best to use the whole captured packet as a `u_char[]` (or `u_int8_t[]`, same thing), find the appropriate bytes, and then recast them to whatever type the libnet function is expecting. You can also cast the packet (or the relevant portions) as structs for a cleaner code, but then you should still use `u_char[]` for each field. This is in contrast with trying to use the structs recommended in the pcap tutorial above. Doing so might cause the program to select the wrong bytes (the first time I tried to do this, instead of getting the 4 bytes from the source address, I ended up with the last 2 bytes, and then the first 2 from the destination address; the struct was fine, but the program would shift 2 bytes when casting the address to a `u_int32_t`).

Note that addresses require no extra work at all, since they're certainly in network order in the packet, and that's what libnet is expecting as well.

So, for example, suppose you want to use the source IP address from a captured packet as the destination address in your own packet. So you have your callback function that `pcap_loop()` is calling when it sniffs a packet, which is passes along as a `u_char[]`. Let's say the packet you got is whatever over IP over ethernet, so you know the source IP address will start at the 27th byte of the packet (ethernet header is 14 bytes long, source address is the 13th byte in IP header). Since C counts from 0, this will be at `packet[26]`. So it'll look something like:

```
void build_reply(..., const uchar *packet) {
    u_char *ip_src = packet[26];
    ...
    ip_tag = libnet_build_ipv4(..., *(u_int_32t*)ip
_src, ...);
    ...
}
```

The bottom line is *work with pointers to bytes, cast when necessary*.

Compiling

To compile the following examples and also your own programs, you'll need to at least link with `-lnet`. For reference, here's what I use:

```
gcc -ggdb -Wall `libnet-config --defines`
    `libnet-config --libs` example.c -o example
```

Integer types

C standards allow different architectures to implement char, short int, int and long int types in **different amounts of bits**, according to some rules. That's usually fine. When you are dealing with networking, however, you can't have headers with different sizes because they were built on different architectures, right? That's why libnet uses**fixed length integer types**. The most important ones you'll encounter will be `u_int32_t`, which means a 32-bit unsigned integer, and `u_int8_t` which means an unsigned byte.

Here's what you need to know:
If there is a `u`, it's unsigned.
If there is no `u`, it's signed.
`int` means "integer".
8, 16, 32, 64 mean 8 bits (1 byte), 16 bits (2 bytes), 32 bits (4 bytes), 64 bits (8 bytes).
`t` means "type".

Note that `u_char`, `uchar`, `u_int8_t` and `uint8_t` should all mean the same thing on most architectures.

Errors

When calling `libnet_init()` (see **below**), you'll need a string (`errbuf`) to learn more about an error. After `libnet_init()` has been successfully called, though, you will get your error messages from a different source:

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

All it needs is the **context**. Call it when you want to know more about the last error you've got.

Enough talk, let’s start coding.

Initializing and closing libnet

`libnet-functions.h` gives us the prototype for `libnet_init()`:

```
libnet_t * libnet_init (int injection_type, char *d
evice,
    char *err_buf)
```

From last to first:

- `err_buf` is a string which will hold an error message if something goes wrong.
- `device` is the device’s name (as in `"eth0"`) or its IP address (as in `"10.0.0.1"`). If device is set to NULL, libnet will try to find a device for you.**Warning:**If you encounter any strange errors when creating headers or writing packets, try passing a device here (eth0, wlan0, lo) instead of `NULL`.
- `injection_type` is the injection type, as in "from the link layer up" or "from the network layer up". We’ll use `LIBNET_RAW4` (IPv4 and above) and `LIBNET_LINK` (link layer and above).

The function returns a pointer to the libnet context, which you’ll need for all header building functions.

Let’s try it out:

Example 1: init.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>

int main() {

  libnet_t *l;  /* the libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  libnet_destroy(l);
  return 0;
}
```

Or we can provide a device name or IP address as an argument:

Example 2: init_devname.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>

int main(int argc, char **argv) {

  libnet_t *l;  /* the libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];

  if ( argc == 1 ) {
    fprintf(stderr,"Usage: %s device\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  l = libnet_init(LIBNET_RAW4, argv[1], errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  libnet_destroy(l);
  return 0;
}
```

Addresses

If you are going to handle IPv4 and Ethernet addresses, you will be dealing with a `u_int32_t` (unsigned 32 bits [4 bytes] integer) for each IP address and a `u_int8_t[6]` array (unsigned 8 bits [1 byte] integer) for Ethernet. When you get one of these addresses from libnet, you will get them in network byte order. Even if your architecture stores integers in little-endian order (e.g., x86 and x86_64), you will get your addresses in memory with most significant bytes first (lower addresses) and least significant bytes last (higher addresses). Hopefully, the next example will illustrate that.

Warning: Make sure to differentiate between numerical addresses stored in byte arrays and string representations of addresses stored in char arrays. The first might look like `u_int8_t[] = {127, 0, 0, 1}`, while the latter would look like `char[] = "127.0.0.1"` which equals `char[] = {'1', '2', '7', '.', '0', '.', '0', '.', '1', '\0' }`. They are not the same thing. Build functions will always ask for numerical addresses, while name-resolving functions will ask for or return strings (and return or ask for the corresponding numerical address). Addresses extracted from packets read by `libpcap` will be numerical, not strings.

The functions you need for dealing with IPv4 address to string, string to IPv4 address and string to Ethernet address are:

```
char* libnet_addr2name4 (u_int32_t in, u_int8_t use
_name)

u_int32_t libnet_name2addr4 (libnet_t *l, char *hos
t_name,
    u_int8_t use_name)

u_int8_t* libnet_hex_aton (int8_t * s, int * len)
```

`libnet_addr2name4()` will take the 4 bytes address `in` and return a string with its dotted decimal representation (e.g., 192.168.0.1) if `use_name` is `LIBNET_DONT_RESOLVE`, or its DNS name (e.g., google.com) if `use_name` is `LIBNET_RESOLVE`.

`libnet_name2addr4()` will do the exact opposite of `libnet_addr2name4()`, and will also need the libnet context as its first argument.

`libnet_hex_aton()` will take a string (`int8_t* == char*`) of two digits hexadecimal numbers separated by colons (e.g., 00:30:0A:67:A6:5C) and return that address in a `u_int8_t` array. The array's length is stored in `len` (for Ethernet, it's usually 6). As we can see on `man libnet-functions.h`, `libnet_hex_aton()` implicitly calls `malloc()` and that memory needs to be freed after you are done with it. Remember this.

To accomplish the opposite effect of `libnet_hex_aton()` you can call printf with “%02X” for each byte, separating them with colons. We'll do that in the following example.

Example 3: addr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  libnet_t *l;  /* the libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];
  char ip_addr_str[16], mac_addr_str[18];
  u_int32_t ip_addr;
  u_int8_t *ip_addr_p, *mac_addr;
  int i, length;  /* for libnet_hex_aton() */

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* IP address */

  printf("IP address: ");
  scanf("%15s",ip_addr_str);

  ip_addr = libnet_name2addr4(l, ip_addr_str,\
                LIBNET_DONT_RESOLVE);

  if ( ip_addr != -1 ) {

    /* ip_addr is ready to be used in a build funct
ion.
     * We'll print its contents to stdout to check
if
     *  everything went fine. */

    /* libnet_name2addr4 returns the address in net
work
     * order (big endian). */

    ip_addr_p = (u_int8_t*)(&ip_addr);
    /* Check your system's endianess: */
    /*
    printf("ip_addr:   %08X\n", ip_addr);
    printf("ip_addr_p: %02X%02X%02X%02X\n", ip_addr
_p[0],\
        ip_addr_p[1], ip_addr_p[2], ip_addr_p[3]);
    */

    printf("Address read: %d.%d.%d.%d\n", ip_addr_p
[0],\
        ip_addr_p[1], ip_addr_p[2], ip_addr_p[3]);

    /* This would output the same thing, but I want
ed to
     * show you how the address is stored in memory
. */
    /*
    printf("Address read: %s\n", libnet_addr2name4(
ip_addr,\
        LIBNET_DONT_RESOLVE));
    */
  }
  else
    fprintf(stderr, "Error converting IP address.\n
");

  /* MAC address */

  printf("MAC address: ");
  scanf("%17s", mac_addr_str);

  mac_addr = libnet_hex_aton(mac_addr_str, &length)
;

  if (mac_addr != NULL) {

    /* mac_addr is ready to be used in a build func
tion.
     * We'll print its contents to stdout to check
if
     * everything went fine. */

    printf("Address read: ");
    for ( i=0; i < length; i++) {
      printf("%02X", mac_addr[i]);
      if ( i < length-1 )
        printf(":");
    }
    printf("\n");
    /* Remember to free the memory allocated by
     * libnet_hex_aton() */
    free(mac_addr);
  }
  else
    fprintf(stderr, "Error converting MAC address.\n
");

  libnet_destroy(l);
  return 0;
}
```

Here you can see one of many casting tricks you might need in the future. On line 42, we'll cast `&ip_addr` into a `u_int8_t*` and store it in `ip_addr_p`. We are turning what would be a pointer to a 4 byte integer into a pointer to an array of 4 single bytes. The reason we do that is so that even if your system uses little-endian integers, we'll be able to read the bytes in the correct order (most significant first). If you are on a PC, try uncommenting lines 44-46; you should get the same thing except that all bytes are swapped (2 hex digits == 1 byte). You should also note that we could have accomplished the same result with `libnet_addr2name4()`. Uncomment lines 53-54 to check.

It may also be very useful to get our own IP and MAC addresses from libnet. For that, we'll need:

```
u_int32_t libnet_get_ipaddr4 (libnet_t *l)

libnet_ether_addr * libnet_get_hwaddr (libnet_t *l)
```

No explanation needed, except for the libnet_ether_addr type:

```
struct libnet_ether_addr {
    u_char  ether_addr_octet[6]; /* Ethernet addres
s */
};
```

It's just a struct with an array of 6 unsigned bytes like the one we used above.

Warning: If you get the error `ioctl(): Can't assign requested address` when calling `libnet_get_ipaddr4()` , it probably means your network device doesn't have an IP address assigned to it. If you're sure that it does, then libnet is probably using the wrong device (e.g. you think it is using `wlan0`, but it's actually using `eth0`). If `libnet_get_hwaddr()` is working, try calling `ifconfig` (`ipconfig` if you're on windows) and compare the interface's

MAC address to what libnet is reporting. If it really is selecting the wrong interface, pass the right one instead of `NULL` when calling `libnet_init()`.

Here’s an example of their usage:

Example 4: get_own_addr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  libnet_t *l;  /* libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];
  u_int32_t ip_addr;
  struct libnet_ether_addr *mac_addr;

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  ip_addr = libnet_get_ipaddr4(l);
  if ( ip_addr != -1 )
    printf("IP address: %s\n", libnet_addr2name4(ip
_addr,\
                                    LIBNET_DONT_RESOLVE));
  else
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
                    libnet_geterror(l));

  mac_addr = libnet_get_hwaddr(l);
  if ( mac_addr != NULL )
    printf("MAC address: %02X:%02X:%02X:%02X:%02X:%
02X\n",\
        mac_addr->ether_addr_octet[0],\
        mac_addr->ether_addr_octet[1],\
        mac_addr->ether_addr_octet[2],\
        mac_addr->ether_addr_octet[3],\
        mac_addr->ether_addr_octet[4],\
        mac_addr->ether_addr_octet[5]);
```

```
   else
     fprintf(stderr, "Couldn't get own MAC address:
%s\n",\
                     libnet_geterror(l));

   libnet_destroy(l);
   return 0;
}
```

The build functions

Libnet makes available a build function for each type of header you may want to use in your packet. As you might notice, some of these functions have a smaller version of themselves, called autobuild. The build functions will let you control every piece of information the header will carry. Usually, however, you will only want to fill out the most important fields and let libnet handle the rest, therefore we have the autobuild functions.

In all subsequent examples, I'll use the autobuild funtions wherever possible. Do not hesitate to try out their build counterparts, though. You'll just need to know your headers (i.e., now is a good time to read those RFCs). For example, here's the regular IPv4 build function:

```
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 * payloa
d,
    u_int32_t payload_s, libnet_t * l, libnet_ptag_
t ptag)
```

`len` is total packet length (from the network layer POV, i.e. not including the link layer header);
`tos` is type of service (useless, set to 0);
`id` is the sequential id number (leave as 0 for the kernel to fill it in for you);
`frag` is fragmentation flags and offset (0 for no fragmentation (if do want it, checkout**advanced mode** below));
`prot` is the next header's protocol (useful macros are `IPPROTO_ICMP`, `IPPROTO_TCP`, `IPPROTO_UDP`);
`ttl` is time to live, the number of hops before a router discards your packet considering it entered a routing loop (usually set to 64 or 255, or incremented from 1 for tracerouting, for example);
`sum` is the checksum (leave as 0 for the kernel to fill it in);
`src` and `dst` are the source and destination addresses;

`payload` and `payload_s` are a pointer to and length of the payload (`NULL` and 0 if there's none);
`l` and `ptag` are libnet's context and the tag used to modify this header (see **next section**).

As you can see, the actual header stuff is straightforward. If in doubt, refer to your textbook or wikipedia. If still in doubt, set to 0. ;)

Warning: If you're building an application layer header on top of IPv4, *do not pass the payload to* `libnet_build_ipv4()` . Pass it to the TCP or UDP build functions, otherwise you'll get an IP header, the payload, and then the next header.

Alright, let's build and sent an ICMP echo request to an address read from stdin. That packet will carry a 10 byte payload that could be any size (up to (64K – 28) bytes, or the link layer protocol's MTU if unwilling to fragment) and anything. In this case, it's a string that goes "libnet :D". When we pass it to `libnet_build_icmpv4_echo()` , we cast it as `u_int8_t*` . If we were reading the packet on the other end or even reading its reply, all we would need to do is cast it back to `char*` . As mentioned above, when you do not wish to send any data, simply pass `NULL` as the payload and 0 as its length.

We'll need these:

```
libnet_ptag_t libnet_build_icmpv4_echo (u_int8_t ty
pe,
    u_int8_t code, u_int16_t sum, u_int16_t id,
    u_int16_t seq, u_int8_t * payload, u_int32_t pa
yload_s,
    libnet_t * l, libnet_ptag_t ptag)

libnet_ptag_t libnet_autobuild_ipv4 (u_int16_t len,
    u_int8_t prot, u_int32_t dst, libnet_t *l)
```

For `libnet_build_icmp4_echo()` , we'll be interested in `seq` (sequence number) and `id` (identification number), `payload` (the extra data we're sending), `payload_s` (the payload's size). `l` is the libnet context we initialized with `libnet_init()` . Don't worry about the `libnet_ptag_t` type and `ptag` , we will talk about them later **when we are sending multiple packets**.

`libnet_autobuild_ipv4()` is straightforward. Go take a look at both functions' descriptions in `man libnet-functions.h` .

We will use libnet's pseudo-random number generating abilities for the first time. It's pretty easy: seed the generator with `libnet_seed_prand()` , and then get as many numbers as you like with `libnet_get_prand()` . Go take a look at those functions' descriptions in `man libnet-functions.h` also.

Example 5: ping.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  libnet_t *l;  /* libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE], ip_addr_str[16];
  u_int32_t ip_addr;
  u_int16_t id, seq;
  char payload[] = "libnet :D";
  int bytes_written;

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Generating a random id */

  libnet_seed_prand (l);
  id = (u_int16_t)libnet_get_prand(LIBNET_PR16);

  /* Getting destination IP address */

  printf("Destination IP address: ");
  scanf("%15s",ip_addr_str);

  ip_addr = libnet_name2addr4(l, ip_addr_str,\
                  LIBNET_DONT_RESOLVE);

  if ( ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building ICMP header */

  seq = 1;
```

```
   if (libnet_build_icmpv4_echo(ICMP_ECHO, 0, 0, id,
  seq,\
          (u_int8_t*)payload,sizeof(payload), l, 0) =
= -1)
   {
     fprintf(stderr, "Error building ICMP header: %s
\n",\
         libnet_geterror(l));
     libnet_destroy(l);
     exit(EXIT_FAILURE);
   }

   /* Building IP header */

   if (libnet_autobuild_ipv4(LIBNET_IPV4_H +\
         LIBNET_ICMPV4_ECHO_H + sizeof(payload),\
         IPPROTO_ICMP, ip_addr, l) == -1 )
   {
     fprintf(stderr, "Error building IP header: %s\n
",\
         libnet_geterror(l));
     libnet_destroy(l);
     exit(EXIT_FAILURE);
   }

   /* Writing packet */

   bytes_written = libnet_write(l);
   if ( bytes_written != -1 )
     printf("%d bytes written.\n", bytes_written);
   else
     fprintf(stderr, "Error writing packet: %s\n",\
         libnet_geterror(l));

   libnet_destroy(l);
   return 0;
}
```

Note that we built `libnet_build_icmpv4_echo()` and `libnet_autobuild_ipv4()` in this particular order.

In both functions we needed some macros for the headers sizes (`LIBNET_IPV4_H` , `LIBNET_ICMPV4_ECHO_H`), upper layer (not really, but IP thinks it is) protocol (`IPPROTO_ICMP`) and ICMP packet type (`ICMP_ECHO`). All of these can be found in `man libnet-headers.h` . Actually, `IPPROTO_ICMP` is not there and I'm not really sure where it is, but you can just write it down.

Here’s what tcpdump sniffed when I (10.0.0.3) used that program to ping my adsl modem (10.0.0.1).

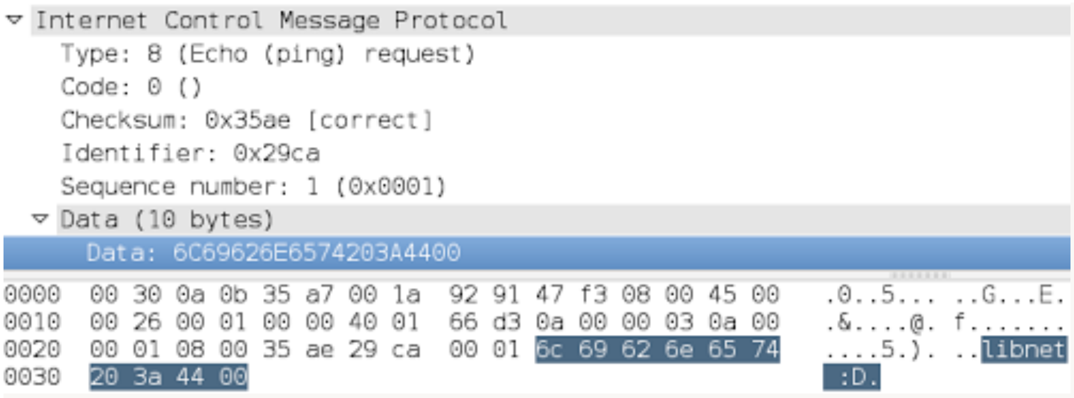
```
# tcpdump -n -ttt icmp

000000 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 3276,
    seq 1, length 18

000936 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 3276,
    seq 1, length 18
```

`length` refers to the data carried by IP which is 10 bytes from the payload and 8 bytes from the ICMP header. When I ran the program it outputted “38 bytes written.” That’s because we called `libnet_init` () with `LIBNET_RAW4`, so it is only telling us how many bytes were written to the link layer, not to the wire.

Here’s the payload in wireshark:



```
▽ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0 ()
    Checksum: 0x35ae [correct]
    Identifier: 0x29ca
    Sequence number: 1 (0x0001)
  ▽ Data (10 bytes)
      Data: 6C69626E6574203A4400
0000  00 30 0a 0b 35 a7 00 1a  92 91 47 f3 08 00 45 00   .0..5... ..G...E.
0010  00 26 00 01 00 00 40 01  66 d3 0a 00 00 03 0a 00   .&....@. f.......
0020  00 01 08 00 35 ae 29 ca  00 01 6c 69 62 6e 65 74   ....5.). ..libnet
0030  20 3a 44 00                                         :D.
```

Now let’s do something similar on the link layer. The following example sends an ARP request for the IP address read from stdin. We’ll need:

```
libnet_ptag_t libnet_autobuild_arp (u_int16_t op,
    u_int8_t * sha, u_int8_t * spa, u_int8_t * tha,
    u_int8_t * tpa, libnet_t * l)

libnet_ptag_t libnet_autobuild_ethernet (u_int8_t *
 dst,
    u_int16_t type, libnet_t * l)
```

Read their descriptions in `man libnet-functions.h`. Read the respective RFCs if you need more information.

Example 6: arp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  libnet_t *l;  /* the libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE], target_ip_addr_s
tr[16];
  u_int32_t target_ip_addr, src_ip_addr;
  u_int8_t mac_broadcast_addr[6] = {0xff, 0xff, 0xf
f, 0xff,\
          0xff, 0xff},
     mac_zero_addr[6] = {0x0, 0x0, 0x0, 0x0, 0x0, 0
x0};
  struct libnet_ether_addr *src_mac_addr;
  int bytes_written;

  l = libnet_init(LIBNET_LINK, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Getting our own MAC and IP addresses */

  src_ip_addr = libnet_get_ipaddr4(l);
  if ( src_ip_addr == -1 ) {
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
                    libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  src_mac_addr = libnet_get_hwaddr(l);
  if ( src_mac_addr == NULL ) {
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
                    libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Getting target IP address */

  printf("Target IP address: ");
  scanf("%15s",target_ip_addr_str);

  target_ip_addr = libnet_name2addr4(l, target_ip_a
ddr_str,\
      LIBNET_DONT_RESOLVE);

  if ( target_ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building ARP header */

  if ( libnet_autobuild_arp (ARPOP_REQUEST,\
      src_mac_addr->ether_addr_octet,\
      (u_int8_t*)(&src_ip_addr), mac_zero_addr,\
      (u_int8_t*)(&target_ip_addr), l) == -1)
  {
    fprintf(stderr, "Error building ARP header: %s\n
",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building Ethernet header */

  if ( libnet_autobuild_ethernet (mac_broadcast_add
r,\
                          ETHERTYPE_ARP, l) == -1 )
  {
    fprintf(stderr, "Error building Ethernet header
: %s\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }
  /* Writing packet */

  bytes_written = libnet_write(l);
  if ( bytes_written != -1 )
    printf("%d bytes written.\n", bytes_written);
  else
    fprintf(stderr, "Error writing packet: %s\n",\
        libnet_geterror(l));

  libnet_destroy(l);
  return 0;
}
```

As previously mentioned, we need to call `libnet_init()` with `LIBNET_LINK`. Note that `libnet_autobuild_arp()` (`libnet_build_arp()` too) expects IPv4 addresses as an array of 4 `u_int8_t`, instead of the `u_int32_t` that `libnet_autobuild_ipv4()` / `libnet_build_ipv4()` expected. You can see above that a simple cast solves the problem.

Here's what tcpdump sniffed when I used that example to request my modem's (10.0.0.1) MAC address:

```
# tcpdump -n -ttt arp

000000 arp who-has 10.0.0.1 tell 10.0.0.3

000456 arp reply 10.0.0.1 is-at 00:30:0a:67:a6:5c
```

The MAC address has been altered to protect my modem's secret identity.

The output I got when running it was "42 bytes written." This time, these are bytes written to the wire and not the link layer (the ARP header is always 28 bytes long; the Ethernet header, 14).

Sending multiple packets

So, now you can send a single packet successfully, but how do you send multiple packets in a row? Well, you can choose between using tags to modify already built headers, or calling `libnet_clear_packet()` and rebuilding all headers from scratch.

Tags are integers with a `libnet_ptag_t` type. When you call a build function, it will return a tag. That tag identifies the header built inside libnet's context. When you are calling a build function for the first time, you can pass 0 as the tag, or a tag initialized with `tag = LIBNET_PTAG_INITIALIZER`. That will make the function build a new header, and wrap it around what is already built. When calling a build function again, you should pass the tag it returned the previous time, so that the header will be modified instead of a new one being built.

If you do not wish to modify an existing header, you need to erase it with `libnet_clear_packet()`. Libnet suggests you only do this when you are about to build a completely different packet with completely different protocols, using the same context. Since I'm lazy, though, I'll do exactly what I'm telling you not to do below on my libnet_clear_packet() example. Please don't follow my example, use tags whenever possible.

Warning: Clearing packets and recreating them as fast as you can is known to crash programs, so if you're planning on firing big amounts of packets, you should be using tags.

Now, if you really are in a situation where it's better to clear the packet, go ahead and call `libnet_clear_packet()`, and then you can build new headers passing 0 as the tag. Remember that you will need to rebuild *all* headers, and that you will need to do it from the upper layers to the lower ones just like the first time.

Here's how you should use tags:

Example 7: reping_tags.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>
#include <unistd.h>

int main() {

   libnet_t *l;  /* libnet context */
   char errbuf[LIBNET_ERRBUF_SIZE], ip_addr_str[16];
   u_int32_t ip_addr;
   libnet_ptag_t icmp_tag, ip_tag;
   u_int16_t id, seq;
   int i;

   l = libnet_init(LIBNET_RAW4, NULL, errbuf);
   if ( l == NULL ) {
     fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
     exit(EXIT_FAILURE);
   }

   icmp_tag = ip_tag = LIBNET_PTAG_INITIALIZER;
```

```
  /* Generating a random id */

  libnet_seed_prand(l);
  id = (u_int16_t)libnet_get_prand(LIBNET_PR16);

  /* Getting destination IP address */

  printf("Destination IP address: ");
  scanf("%15s",ip_addr_str);

  ip_addr = libnet_name2addr4(l, ip_addr_str,\
      LIBNET_DONT_RESOLVE);

  if ( ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building ICMP header */

  seq = 1;

  icmp_tag = libnet_build_icmpv4_echo(ICMP_ECHO, 0,
 0, id,\
                  seq, NULL, 0, l, 0);

  if (icmp_tag == -1) {
    fprintf(stderr, "Error building ICMP header: %s
\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building IP header */

  ip_tag = libnet_autobuild_ipv4(LIBNET_IPV4_H +\
      LIBNET_ICMPV4_ECHO_H, IPPROTO_ICMP, ip_addr,
l);

  if (ip_tag == -1) {
    fprintf(stderr, "Error building IP header: %s\n
",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Writing 4 packets */

  for ( i = 0; i < 4; i++ ) {

    /* Updating the ICMP header */
    icmp_tag = libnet_build_icmpv4_echo(ICMP_ECHO,
0, 0,\
        id, (seq + i), NULL, 0, l, icmp_tag);

    if (icmp_tag == -1) {
      fprintf(stderr, "Error building ICMP header:
%s\n",\
          libnet_geterror(l));
      libnet_destroy(l);
      exit(EXIT_FAILURE);
    }

    if ( libnet_write(l) == -1 )
      fprintf(stderr, "Error writing packet: %s\n",
\
          libnet_geterror(l));

    /* Waiting 1 second between each packet */
    sleep(1);

  }

  libnet_destroy(l);
  return 0;

}
```

Just remember that when we are calling `libnet_build_icmpv4_echo()` for any time other than the first, we are not building it, just modifying the already built header.

Here's tcpdump's output:

```
# tcpdump -n -ttt icmp

000000 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 26873,
    seq 1, length 8
```

```
000486 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 26873,
    seq 1, length 8

999623 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 26873,
    seq 2, length 8

000479 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 26873,
    seq 2, length 8

999568 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 26873,
    seq 3, length 8

000533 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 26873,
    seq 3, length 8

999519 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 26873,
    seq 4, length 8

000805 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 26873,
    seq 4, length 8
```

Now let's try the same thing using `libnet_clear_packet()`. As stated above, this is a waste of CPU time, and I'm only doing it because I'm lazy.

Example 8: reping_clear.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>
#include <unistd.h>

int main() {

  libnet_t *l;  /* libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE], ip_addr_str[16];
  u_int32_t ip_addr;
  u_int16_t id, seq;
  int i;

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Generating a random id */

  libnet_seed_prand(l);
  id = (u_int16_t)libnet_get_prand(LIBNET_PR16);

  /* Getting destination IP address */

  printf("Destination IP address: ");
  scanf("%15s",ip_addr_str);

  ip_addr = libnet_name2addr4(l, ip_addr_str,\
      LIBNET_DONT_RESOLVE);

  if ( ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Writing 4 packets */

  seq = 1;

  for ( i = 0; i < 4; i++ ) {

    /* Building the ICMP header */
    if ( libnet_build_icmpv4_echo(ICMP_ECHO, 0, 0,
id,\
        (seq + i), NULL, 0, l, 0) == -1 ) {
      fprintf(stderr, "Error building ICMP header:
%s\n",\
          libnet_geterror(l));
      libnet_destroy(l);
      exit(EXIT_FAILURE);
    }

    /* Building the IP header */
    if ( libnet_autobuild_ipv4(LIBNET_IPV4_H + \
          LIBNET_ICMPV4_ECHO_H, IPPROTO_ICMP,\
          ip_addr, l) == -1 ) {
      fprintf(stderr, "Error building IP header: %s
\n",\
          libnet_geterror(l));
      libnet_destroy(l);
      exit(EXIT_FAILURE);
    }

    if ( libnet_write(l) == -1 )
      fprintf(stderr, "Error writing packet: %s\n",
\
          libnet_geterror(l));

    /* Clearing the packet */
    /* Comment this to see what happens when you re
build
     * headers without calling libnet_clear_packet(
) */
    libnet_clear_packet(l);

    /* Waiting 1 second between each packet */
    sleep(1);

  }

  libnet_destroy(l);
  return 0;
}
```

Note that you have to rebuild the whole packet again, and that you must do it in the correct order (upper layer to lower layer) again.

Well, that wasn’t interesting at all. Let’s try commenting out `libnet_clear_packet()` and see what happens:

```
# tcpdump -n -ttt icmp

000000 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 9701,
    seq 1, length 8

000486 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 9701,
    seq 1, length 8

999605 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 9701,
    seq 2, length 36

000483 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 9701,
    seq 2, length 36

999596 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 9701,
    seq 3, length 64

000500 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 9701,
    seq 3, length 64

999585 IP 10.0.0.3 > 10.0.0.1: ICMP echo request, i
d 9701,
    seq 4, length 92

000558 IP 10.0.0.1 > 10.0.0.3: ICMP echo reply,   i
d 9701,
    seq 4, length 92
```

You should remember from the first time we tried sending an ICMP echo request that it is 20 (IP header) + 8 (ICMP header) + 0 (payload size) bytes long when going to the link layer. You should also remember that the length outputted by tcpdump refers to the data carried by IP. Now let’s try this: 8 bytes (ICMP header for the second packet) + 28 bytes (previous packet as seen by the libnet context) == 36 bytes. Now add 20 bytes (IP header for the second packet) and 8 more bytes (ICMP header for the third packet), and we get 64. Do this again and you will get 92 for the last packet.

What we can see here is that when we forget to call `libnet_clear_packet()`, all headers we build again will go before the ones already built, and carry them as if they were payload we meant to send in that packet.

IP fragmentation and libnet

If you need to send a packet bigger than the MTU at hand using libnet, you’ll need to fragment it yourself. Not only will libnet not help you in doing it, it will actually get in the way, by calculating the upper-layer checksum using only the first fragment’s payload, causing your packet to get discarded at destination. To get the correct checksum, you can create one big packet, have libnet calculate the checksum, then retrieve it without writing the packet to the wire. To do this, we'll have to take a look at libnet's advanced mode.

Advanced mode

Libnet has a few advanced functions, which are only available when you open it in advanced mode. If you remember **when we looked into libnet_init()**, we could choose to deal with the link layer or not by using `LIBNET_LINK` and `LIBNET_RAW4` respectively as the injection type. For advanced mode, we would use `LIBNET_LINK_ADV` and `LIBNET_RAW4_ADV` in the same way. Except that we won't actually use `LIBNET_RAW4_ADV` since all advanced functions seem to require `LIBNET_LINK_ADV`.

Anyway, in advanced mode, you'll have access to:

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

This function will give you a pointer (in `u_int8_t ** header`) to the header referenced by `ptag`. `header_s` will point to a `u_int32_t` containing the size of the header.

Here's an example:

Example 9: cull_header.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  /* Builds an IP header and prints it */

  libnet_t *l;
  char errbuf[LIBNET_ERRBUF_SIZE];
  libnet_ptag_t ip_tag;
  u_int8_t *ip_header;
  u_int32_t ip_header_size;
  int i;

  l = libnet_init(LIBNET_LINK_ADV, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Building IP header, size = 20 bytes, dest addr
ess =
   * 0.0.0.0, upper layer protocol = 0 */
  ip_tag = libnet_autobuild_ipv4(20, 0, 0, l);
  if ( ip_tag == -1 ) {
    fprintf(stderr, "Error building IP header: %s\n
",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Getting a pointer to the header */
  if (libnet_adv_cull_header(l, ip_tag, &ip_header,
\
        &ip_header_size) == -1) {
    fprintf(stderr,"libnet_adv_cull_header() failed
: %s\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Printing the header */
  for (i=0; i < ip_header_size; i++) {
    printf("%02X ", ip_header[i]);
  }
  printf("\n");

  libnet_destroy(l);
  return 0;
}
```

```
# ./cull_header

45 00 00 14 00 00 00 00 40 00 00 00 0A 00 00 03 00
00 00 00
```

Similarly, there’s:

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

This one works the same way, but gives you a pointer to whole packet which you should have already built. If you call it, you’ll need to free the memory later with

```
void libnet_adv_free_packet (libnet_t * l,
    u_int8_t * packet)
```

If you modify the packet from `libnet_adv_cull_packet()` or build your own packet from scratch, you can send it with

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

To exemplify, I’ll rewrite the ARP example I used when talking about the build_functions, but I’ll build the ARP header without the target IP address and add it directly into the packet:

Example 10: cull_packet.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

int main() {

  /* Builds an ARP request, then pulls it from libn
et,
   * changes the target IP address and writes it. */


  libnet_t *l;
  char errbuf[LIBNET_ERRBUF_SIZE], target_ip_addr_s
tr[16];
  u_int32_t src_ip_addr, target_ip_addr = 0;
  u_int8_t mac_broadcast_addr[6] = {0xff, 0xff, 0xf
```

```
f, 0xff,\
      0xff, 0xff},
  mac_zero_addr[6] = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
;
  struct libnet_ether_addr *src_mac_addr;
  int i;

  u_int8_t *packet, *target_ip_addr_p;
  u_int32_t packet_size;


  l = libnet_init(LIBNET_LINK_ADV, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Getting our own MAC and IP addresses */
  src_ip_addr = libnet_get_ipaddr4(l);
  if ( src_ip_addr == -1 ) {
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  src_mac_addr = libnet_get_hwaddr(l);
  if ( src_mac_addr == NULL ) {
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building ARP header with target IP address = 0
.0.0.0 */
  if ( libnet_autobuild_arp (ARPOP_REQUEST,\
      src_mac_addr->ether_addr_octet,\
      (u_int8_t*)(&src_ip_addr), mac_zero_addr,\
      (u_int8_t*)(&target_ip_addr), l) == -1)
  {
    fprintf(stderr, "Error building ARP header: %s\n
",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building Ethernet header */
  if ( libnet_autobuild_ethernet (mac_broadcast_add
r,\
      ETHERTYPE_ARP, l) == -1 )
  {
    fprintf(stderr, "Error building Ethernet header
: %s\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Pulling the packet */
  if (libnet_adv_cull_packet(l, &packet, &packet_si
ze)\
       == -1) {
    fprintf(stderr,"libnet_adv_cull_packet() failed
: %s\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Getting target IP address */
  printf("Target IP address: ");
  scanf("%15s",target_ip_addr_str);

  target_ip_addr = libnet_name2addr4(l, target_ip_a
ddr_str,\
      LIBNET_DONT_RESOLVE);

  if ( target_ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Changing the target */
  /* We want to change the 39th, 40th, 41st and 42n
d bytes:
    * Ethernet header (14) + ARP's hw(2), proto(2),
hw
    * size(1), proto size(1), opcode(2), src hw addr
(6), src
```

```
    * ip add(4), target hw addr(6) = 38 */
   if (packet_size >= 42) {
     target_ip_addr_p = (u_int8_t *)&target_ip_addr;
     for (i=0; i < 4; i++) {
       packet[38+i] = target_ip_addr_p[i];
     }
   }

   /* Writing packet */
   if (libnet_adv_write_link(l, packet, packet_size)
  == -1) {
     fprintf(stderr, "Error writing packet: %s\n",\
         libnet_geterror(l));
   }

   /* Freeing up the memory */
   libnet_adv_free_packet(l, packet);

   libnet_destroy(l);
   return 0;
 }
```

```
# tcpdump -n -ttt arp

000000 arp who-has 10.0.0.1 tell 10.0.0.3

000459 arp reply 10.0.0.1 is-at 00:30:0a:67:a6:5c
```

Now, let’s move on to the useful stuff.

IP fragmentation with libnet

As stated above, the only way I was able to fragment packets with libnet was implementing the fragmentation myself. In this example, I’ll send an ICMP echo request with arbitrarily big random payload. The MTU I chose is Ethernet’s 1500 bytes, since that’s what I have here. I hope it will be simple to adapt the code to send a TCP or UDP packet with any kind of payload.

Example 11: frag_ping.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

#define MTU 1500

libnet_t *l;  /* libnet context */

void frag_and_send(u_int8_t *payload,\
    u_int32_t total_pload_size);

u_int16_t get_sum(u_int8_t *payload, \
    u_int32_t total_pload_size, u_int16_t id,\
    u_int16_t seq);

int main() {

  int i;
  char errbuf[LIBNET_ERRBUF_SIZE];
  /* It's a good idea to have the payload as an arr
ay of
   * bytes. If yours isn't, make a pointer to it an
d cast
   * it.*/
  u_int8_t payload[3000];

  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr,\
        "libnet_init() failed (raw4, 1st call): %s\n
",\
        errbuf);
    exit(EXIT_FAILURE);
  }

  /* Generating random payload */
  libnet_seed_prand (l);

  for (i = 0; i < sizeof(payload); i++) {
    payload[i] = libnet_get_prand(LIBNET_PR8);
  }

  /* Building and sending the fragments */
  frag_and_send(payload, sizeof(payload));

  libnet_destroy(l);
  return 0;
}

void frag_and_send(u_int8_t *payload,\
    u_int32_t total_pload_size) {

  /* Builds and sends the first packet, calling get
_sum() to
   * get the correct checksum for the ICMP packet (
with the
   * whole payload). Then builds and sends IP fragm
ents
   * until all the payload is sent. */

  char ip_addr_str[16];
  u_int32_t ip_addr, src_addr;
  u_int16_t id, seq, ip_id;
  /* hdr_offset = fragmentation flags + offset (in
bytes)
   * divided by 8 */
  int pload_offset, hdr_offset;
  int bytes_written, max_pload_size, packet_pload_s
ize;
  libnet_ptag_t ip_tag;

  /* Generating random IDs */

  id = (u_int16_t)libnet_get_prand(LIBNET_PR16);
  /* We need a non-zero id number for the IP header
s,
   * otherwise libnet will increase it after each
   * build_ipv4, breaking the fragments */
  ip_id = (u_int16_t)libnet_get_prand(LIBNET_PR16);

  seq = 1;

  /* Getting IP addresses */

  src_addr = libnet_get_ipaddr4(l);
  if ( src_addr == -1 ) {
    fprintf(stderr, "Couldn't get own IP address: %s
\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  printf("Destination IP address: ");
  scanf("%15s",ip_addr_str);

  ip_addr = libnet_name2addr4(l, ip_addr_str,\
      LIBNET_DONT_RESOLVE);

  if ( ip_addr == -1 ) {
    fprintf(stderr, "Error converting IP address.\n
");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Getting max payload size */

  max_pload_size = (MTU - LIBNET_IPV4_H);
  /* making it a multiple of 8 */
  max_pload_size -= (max_pload_size % 8);

  pload_offset = 0;

  /* Building the first packet, which carries the I
CMP
   * header */

  /* We're doing (payload size - icmp header size)
and not
   * checking if it's a multiple of 8 because we kn
ow the
   * header is 8 bytes long */
  if ( total_pload_size > (max_pload_size - \
          LIBNET_ICMPV4_ECHO_H) ) {
    hdr_offset = IP_MF;
    packet_pload_size = max_pload_size - \
        LIBNET_ICMPV4_ECHO_H;
  }
  else {
    hdr_offset = 0;
    packet_pload_size = total_pload_size;
  }

  /* ICMP header */
  if ( libnet_build_icmpv4_echo(ICMP_ECHO, 0, \
          get_sum(payload, total_pload_size, id, se
q),\
          id, seq, payload, packet_pload_size, l, 0
) == -1 )
  {
    fprintf(stderr, "Error building ICMP header: %s
\n", \
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* First IP header (no payload, offset == 0) */
  if ( libnet_build_ipv4((LIBNET_IPV4_H + \
          LIBNET_ICMPV4_ECHO_H + packet_pload_size)
, 0,\
        ip_id, hdr_offset, 255, IPPROTO_ICMP, 0, sr
c_addr,\
        ip_addr, NULL, 0, l, 0) == -1 )
  {
    fprintf(stderr, "Error building IP header: %s\n
",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Writing packet */

  bytes_written = libnet_write(l);

  if ( bytes_written != -1 )
    printf("%d bytes written.\n", bytes_written);
  else
    fprintf(stderr, "Error writing packet: %s\n", \
          libnet_geterror(l));

  /* Updating the offset */
  pload_offset += packet_pload_size;

  /* Clearing */
  /* We need to get rid of the ICMP header to build
 the
   * other fragments */
  libnet_clear_packet(l);

  ip_tag = LIBNET_PTAG_INITIALIZER;

  /* Looping until all the payload is sent */
  while ( total_pload_size > pload_offset ) {

    /* Building IP header */

    /* checking if there will be more fragments */
    if ((total_pload_size - pload_offset) > max_plo
ad_size)
    {
      /* In IP's eyes, the ICMP header in the first
 packet
       * needs to be in the offset, so we add its s
ize to
       * the payload offset here */
      hdr_offset = IP_MF + (pload_offset + \
          LIBNET_ICMPV4_ECHO_H)/8;
      packet_pload_size = max_pload_size;
    }
    else {
      /* See above */
      hdr_offset = (pload_offset + LIBNET_ICMPV4_EC
HO_H)/8;
      packet_pload_size = total_pload_size - pload_
offset;
    }

    ip_tag = libnet_build_ipv4( \
        (LIBNET_IPV4_H + max_pload_size), 0, ip_id,
\
        hdr_offset, 255, IPPROTO_ICMP, 0, src_addr,
\
        ip_addr, (payload + pload_offset),\
        packet_pload_size, l, ip_tag);

    if ( ip_tag == -1 ) {
      fprintf(stderr, "Error building IP header: %s
\n", \
          libnet_geterror(l));
      libnet_destroy(l);
      exit(EXIT_FAILURE);
    }


    /* Writing packet */

    bytes_written = libnet_write(l);

    if ( bytes_written != -1 )
      printf("%d bytes written.\n", bytes_written);
    else
      fprintf(stderr, "Error writing packet: %s\n",
 \
          libnet_geterror(l));
    /* Updating the offset */
    pload_offset += packet_pload_size;
  }
}

u_int16_t get_sum(u_int8_t *payload,\
    u_int32_t total_pload_size, u_int16_t id, u_int
16_t seq)
{

  /* Builds the ICMP header with the whole payload,
 gets
   * the checksum from it and returns it (in host o
rder). */

  char errbuf[LIBNET_ERRBUF_SIZE];
  libnet_ptag_t icmp_tag;
  u_int8_t *packet;
  u_int32_t packet_size;
  u_int16_t *sum_p, sum;
  u_int8_t dummy_dst[6] = {0, 0, 0, 0, 0, 0};

  icmp_tag = LIBNET_PTAG_INITIALIZER;

  /* Switching to advanced link mode */
  /* Nothing should be built yet and all random num
bers
   * should be already generated. */
  libnet_destroy(l);
  l = libnet_init(LIBNET_LINK_ADV, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr,"libnet_init() failed (link_adv)
: %s\n",\
        errbuf);
    exit(EXIT_FAILURE);
  }

  /* Building the header */
  icmp_tag = libnet_build_icmpv4_echo(ICMP_ECHO, 0,
 0, id,\
      seq, payload, total_pload_size, l, icmp_tag);

  if ( icmp_tag == -1 ) {

    fprintf(stderr, "Error building ICMP header: %s
\n", \
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building dummy IP header */
  if ( libnet_autobuild_ipv4((LIBNET_IPV4_H + \
          LIBNET_ICMPV4_ECHO_H +total_pload_size),
\
        IPPROTO_ICMP, 0, l) == -1 ) {
    fprintf(stderr, "Error building dummy IP header
: %s\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building dummy Ethernet header */
  if ( libnet_autobuild_ethernet (dummy_dst, ETHERT
YPE_IP,\
          l) == -1 ) {
    fprintf(stderr,\
        "Error building dummy Ethernet header: %s\n
",
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }


  /* Pulling the packet */
  if (libnet_adv_cull_packet(l, &packet, &packet_si
ze)\
      == -1) {
    fprintf(stderr, "Error pulling the packet: %s\n
", \
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Grabbing the checksum */
  /* We want the 37th and 38th bytes: eth header (1
4) + ip
   * header (20) + icmp type and code (2) = 36 */
  sum_p = (u_int16_t*)(packet + 36);
  sum = ntohs(*sum_p);

  /* Freeing memory */
  libnet_adv_free_packet(l, packet);
  /* Clearing the header */
  libnet_clear_packet(l);

  /* Switching back to IPv4 raw socket mode */
  libnet_destroy(l);
  l = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if ( l == NULL ) {
    fprintf(stderr,\
        "libnet_init() failed (raw4, 2nd call): %s\n
",\
        errbuf);
    exit(EXIT_FAILURE);
  }

  return sum;
}
```

I’ve trimmed and formated tcpdump’s output as much as possible for clarity.

```
# tcpdump -n -ttt -vv icmp

000000 IP (id 28896, offset 0, flags [+], length 15
00)
    10.0.0.3 > 10.0.0.1: ICMP echo request, id 3371
, seq 1

000034 IP (id 28896, offset 1480, flags [+], length
 1500)
    10.0.0.3 > 10.0.0.1

000012 IP (id 28896, offset 2960, flags [none], len
gth 68)
    10.0.0.3 > 10.0.0.1


001657 IP (id 41612, offset 0, flags [+], length 15
00)
    10.0.0.1 > 10.0.0.3: ICMP echo reply, id 3371,
seq 1

000215 IP (id 41612, offset 1480, flags [+], length
 1500)
    10.0.0.1 > 10.0.0.3
```

```
000007 IP (id 41612, offset 2960, flags [none], len
gth 68)
    10.0.0.1 > 10.0.0.3
```

IPv6

Note: Before we begin I'd like to point out that in the following examples I'll be using a few functions that are not available on `<libnet-1.1.6` (namely, `libnet_build_icmpv6_echo()` and `libnet_autobuild_ipv6()` (which implies `libnet_get_ipaddr6()`)). If your distribution only has a dated version of libnet, you can get the latest one from **github**. I'll be using these functions just because they'll make my life easier in the examples below, but you don't actually need them for building IPv6 packets (well, at least not TCP/UDP over IPv6).

IPv6 addresses

Libnet stores IPv6 addresses using the following structure, defined in libnet-headers.h:

```
/*
 * IPv6 address
 */
struct libnet_in6_addr
{
  union
  {
    uint8_t    __u6_addr8[16];
    uint16_t   __u6_addr16[8];
    uint32_t   __u6_addr32[4];
  } __u6_addr;   /* 128-bit IP6 address */
};
```

Which isn't very different from the uint8_t arrays used for ethernet addresses.

Note: To understand how these functions return errors and how I handle that in the following example, please read the **next section**.

For converting IPv6 addresses from text to `struct libnet_in6_addr` and vice-versa, we'll use functions analogous to those we used for IPv4 addresses:

```
void libnet_addr2name6_r(struct libnet_in6_addr add
r,
    uint8_t use_name, char *host_name, int host_nam
e_len)

struct libnet_in6_addr libnet_name2addr6 (libnet_t
*l,
    char *host_name, uint8_t use_name)
```

You might remember that `use_name` can be `LIBNET_RESOLVE`, if you want to resolve DNS names, or `LIBNET_DONT_RESOLVE` if you'll be using an actual address and want to avoid a DNS lookup. Also notice that `libnet_addr2name6_r()` does not allocate the string for you, as `libnet_addr2name4()` did, so you'll have to do it and pass it as an argument, together with its size.

Here you can see these guys in action:

Example 12: addr6.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

#define BYTES_IN_IPV6_ADDR 16
#define MAX_CHARS_IN_IPV6_ADDR 39

int libnet_in6_addr_cmp(struct libnet_in6_addr addr
1, \
    struct libnet_in6_addr addr2) {
  /* Returns != 0 if addresses are equal, 0 otherwi
se. */

  uint32_t *p1 = (uint32_t*)&addr1.__u6_addr, \
     *p2 = (uint32_t*)&addr2.__u6_addr;

  return ((p1[0] == p2[0]) && (p1[1] == p2[1]) && \
      (p1[2] == p2[2]) && (p1[3] == p2[3]));
}

int main() {

  libnet_t *l; /* the libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];
  int i;

  char ipv6_addr_str[MAX_CHARS_IN_IPV6_ADDR+1];
  struct libnet_in6_addr ipv6_addr;
  u_int8_t *ipv6_addr_p;

  l = libnet_init(LIBNET_RAW6, NULL, errbuf);
  if ( l == NULL ) {
  fprintf(stderr, "libnet_init() failed: %s\n", err
buf);
    exit(EXIT_FAILURE);
  }

  printf("Enter an IPv6 address: ");
  /* too lazy not to hardcode this: */
  scanf("%39s",ipv6_addr_str);

  ipv6_addr = libnet_name2addr6(l, ipv6_addr_str, \
                  LIBNET_DONT_RESOLVE);

  if (libnet_in6_addr_cmp(ipv6_addr, in6addr_error)
 != 0) {
    fprintf(stderr, "Error converting IPv6 address.
\n");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  ipv6_addr_p = (u_int8_t*)&ipv6_addr.__u6_addr;
  printf("libnet_name2addr6() returned: ");
  for ( i=0; i < BYTES_IN_IPV6_ADDR; i++) {
    printf("%02x", ipv6_addr_p[i]);
    if ((i % 2 == 1) && i < BYTES_IN_IPV6_ADDR -1)
      printf(":");
  }
  printf("\n");

  ipv6_addr_str[0] = '';
  libnet_addr2name6_r(ipv6_addr, LIBNET_DONT_RESOLV
E, \
                  ipv6_addr_str, MAX_CHARS_IN_IPV6_
ADDR);
  printf("libnet_addr2name6() says it's: %s\n", \
                  ipv6_addr_str);

  libnet_destroy(l);
  return 0;
}
```

You may also remember that we looked into `libnet_get_ipaddr4()`, which returns your own IPv4 address. The analagous `libnet_get_ipaddr6()` is one of the functions mentioned above which are available only on >=libnet-1.1.6. You don't really need it if you'll be spoofing your packets' source address or if you can hardcode your own address or something similar. If you do have it available, it is just as straightforward as `libnet_get_ipaddr4()`:

```
struct libnet_in6_addr
libnet_get_ipaddr6(libnet_t *l)
```

IPv6 address errors

IPv4 address functions reported errors by returning 255.255.255.255, which is stored in a `uint_32t` as thirty-two 1s, which we could neatly check by comparing to -1, as in

```
if (ipv4_addr == -1)
  /* error handling */
```

The IPv6 functions do the same thing, using the constant `in6addr_error`:

```
const struct libnet_in6_addr
    in6addr_error = IN6ADDR_ERROR_INIT;
```

defined in `libnet-macros.h` as

```
#define IN6ADDR_ERROR_INIT { { { 0xff, 0xff, 0xff,
0xff,
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
 0xff,
    0xff, 0xff, 0xff } } }
```

Unfortunately, if we try to check it the same way as with IPv4, it will come out a little bit less neat:

```
if (*(uint32_t*)&ipv6_addr.__u6_addr == -1 && \
      *((uint32_t*)&ipv6_addr.__u6_addr +1) == -1 &
& \
      *((uint32_t*)&ipv6_addr.__u6_addr +2) == -1 &
& \
      *((uint32_t*)&ipv6_addr.__u6_addr +3) == -1)
  /* error handling */
```

So, I actually use this:

```
int libnet_in6_addr_cmp(struct libnet_in6_addr addr
1,
                        struct libnet_in6_addr addr
2) {
  /* Returns != 0 if addresses are equal, 0 otherwi
se. */

  uint32_t *p1 = (uint32_t*)&addr1.__u6_addr, \
           *p2 = (uint32_t*)&addr2.__u6_addr;

  return ((p1[0] == p2[0]) && (p1[1] == p2[1]) && \
          (p1[2] == p2[2]) && (p1[3] == p2[3]));
}
```

and then just

```
if (libnet_in6_addr_cmp(ipv6_addr, in6addr_error) !
= 0)
  /* error handling */
```

IPv6 build functions

Earlier we used `libnet_autobuild_ipv4()`, which is really just a way to call `libnet_build_ipv4()` and have it fill in default values for pretty much everything. The analogous is `libnet_autobuild_ipv6()`, which, as I mentioned above, has only been implemented as of libnet-1.1.6. But again, you don't really need it, you can just use `libnet_build_ipv6()`:

```
libnet_ptag_t libnet_autobuild_ipv6 (uint16_t len,
    uint8_t nh, struct libnet_in6_addr dst, libnet_
t *l,
    libnet_ptag_t ptag)

libnet_ptag_t libnet_build_ipv6 (uint8_t tc, uint32
_t fl,
    uint16_t len, uint8_t nh, uint8_t hl,
    struct libnet_in6_addr src, struct libnet_in6_a
ddr dst,
    const uint8_t *payload, uint32_t payload_s, lib
net_t *l,
    libnet_ptag_t ptag)

libnet_ptag_t libnet_build_icmpv6_echo (uint8_t typ
e,
        uint8_t code, uint16_t sum, uint16_t id,
        uint16_t seq, uint8_t *payload, uint32_t pa
yload_s,
        libnet_t *l, libnet_ptag_t ptag)
```

As you can see, all you need, besides your own source address, is to set `tc` and `fl` to zero, and `hl` to 64 or 255 or whatever. In the example below I'll be using `libnet_autobuild_ipv6()` and `libnet_build_icmpv6_echo` (also >=libnet-1.1.6) to ping over IPv6, but hopefully you can easily adapt the code to inject your own TCP or UDP packets.

Example 13: ping6.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libnet.h>
#include <stdint.h>

#define MAX_CHARS_IN_IPV6_ADDR 39

int libnet_in6_addr_cmp(struct libnet_in6_addr addr
1, \
    struct libnet_in6_addr addr2) {
  /* Returns != 0 if addresses are equal, 0 otherwi
se. */

  uint32_t *p1 = (uint32_t*)&addr1.__u6_addr, \
     *p2 = (uint32_t*)&addr2.__u6_addr;

  return ((p1[0] == p2[0]) && (p1[1] == p2[1]) && \
      (p1[2] == p2[2]) && (p1[3] == p2[3]));
```

```
}

int main() {

  libnet_t *l;  /* libnet context */
  char errbuf[LIBNET_ERRBUF_SIZE];

  char ip_addr_str[MAX_CHARS_IN_IPV6_ADDR+1];
  struct libnet_in6_addr ip_dst_addr;
  char payload[] = "libnet :D";
  u_int16_t id, seq;
  int bytes_written;

  l = libnet_init(LIBNET_RAW6, "eth0", errbuf);
  if ( l == NULL ) {
    fprintf(stderr, "libnet_init() failed: %s\n", e
rrbuf);
    exit(EXIT_FAILURE);
  }

  /* Generating a random id */

  libnet_seed_prand (l);
  id = (u_int16_t)libnet_get_prand(LIBNET_PR16);

  /* Getting destination IP address */

  ip_addr_str[0] = '';

  printf("Destination IPv6 address: ");
  /* too lazy not to hardcode this: */
  scanf("%39s",ip_addr_str);

  ip_dst_addr = libnet_name2addr6(l, ip_addr_str, \
                        LIBNET_DONT_RESOLVE);

  if (libnet_in6_addr_cmp(ip_dst_addr, in6addr_erro
r)) {
    fprintf(stderr, "Error converting IPv6 address.
\n");
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building ICMP header */

  seq = 1;
  if (libnet_build_icmpv6_echo(ICMP6_ECHO, 0, 0, id
, seq, \
        (u_int8_t*)payload, sizeof(payload), l, 0)
== -1)
  {
    fprintf(stderr, "Error building ICMPv6 header:
%s\n", \
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Building IP header */

  if (libnet_autobuild_ipv6(LIBNET_ICMPV6_ECHO_H +
\
        sizeof(payload), IPPROTO_ICMP6, ip_dst_addr
, \
        l, 0) == -1)
  {
    fprintf(stderr, "Error building IPv6 header: %s
\n",\
        libnet_geterror(l));
    libnet_destroy(l);
    exit(EXIT_FAILURE);
  }

  /* Writing packet */

  bytes_written = libnet_write(l);
  if ( bytes_written != -1 )
    printf("%d bytes written.\n", bytes_written);
  else
    fprintf(stderr, "Error writing packet: %s\n",\
        libnet_geterror(l));

  libnet_destroy(l);
  return 0;
}
```

Here are the packets on tcpdump:

```
000000 IP6 fe80::82ee:73ff:ac9d:b582 >
        fe80::21e:bff:ba26:f210: ICMP6, echo reques
t,
        seq 1, length 18

000289 IP6 fe80::21e:bff:ba26:f210 >
        fe80::82ee:73ff:ac9d:b582: ICMP6, echo repl
y,
        seq 1, length 18
```

And that's about it. Just replace your IPv4 functions with the IPv6 ones and you're good to go. You can find a few functions for doing more complex IPv6 header stuff at the `libnet-functions.h` man page, but, as was the case with IPv4 options, I think they are beyond the scope of this tutorial (and my own mastery of TCP/IP), so I'll just stop here.

Happy injecting.