Libnet 101, Part 1: The Primer

WHITE PAPER

Michael D. Schiffman
Director, Research and Development
Guardent, Inc.
June19, 2000

INTRODUCTION

Libnet is a reasonably small programming library, written mainly in C, providing a high-level, standard portable interface to low-level network packet shaping, handling and injection primitives. To further understand this rather wordy definition, let's go through it and expand on the major points.

- high-level interface: Libnet is written mostly in C and was designed to abstract out the pedantic architecture-specific details of low-level packet shifting.
- low-level packet shaping: One of libnet's major value-adds is the complete control it offers over every packet header field.
- portable interface: Libnet's interface is standard whether you're working on Linux, FreeBSD, Solaris or Windows NT.
- packet shaping: Libnet offers a wide array of packet constructors for several protocols (inside of TCP/IP).
- packet handling: Libnet also has a suite of supplementary functions to automate many of the tedious tasks inherent in low-level network programming.
- packet injection: Libnet allows the user to choose between two different packet injection methods.

Libnet was developed for two major reasons:

- To establish a simple interface by which network programmers could ignore the subtleties and nuances of low-level network programming (and therefore concentrate on writing their programs and solving their problems).
- To mitigate the irritation many network programmers experienced due to the lack of standards.

Prior to libnet, programmers had to struggle with confusing, obscure, and poorly documented interfaces to construct and write network packets. Libnet alleviates these problems and provides a well documented, simple API to quickly build portable programs that write network packets.

This short tutorial is designed to be an overall introduction to libnet and it will describe the library in moderate detail. This includes internals, design decisions and futures, as well as a step by step coding example.

WHY USE LIBNET?

Libnet is a very robust enabler in building useful and complex network applications (in fact, in June of 2000, a survey of 1200 nmap users found libnet and several libnet-based applications to be among their top 5 favorite security tools. See http://www.insecure.org/tools.html).

While some people may see libnet as a tool for hackers because it can be used to quickly craft network-based exploits, this is a very myopic point of view. Libnet can be used (and indeed has been used) to build vastly powerful network and network security based applications. It allows the implementer absolute control of packets on a discrete level, something not offered by traditional network programming interfaces. This control combined with libnet's easy to use interface, makes it a very effectual value-add to any network programmer.

LIBRARY OVERVIEW

Libnet is a simple C library. It is designed to be small, efficient and easy to use. Libnet's main goal is portable packet creation and injection. As of version 1.0.2, the core libnet tree consists of about 8000 lines of code in 43 files. There are 63 user-accessible functions, 26 of which are packet constructor routines. Libnet currently has support for 10 protocols, with several in development.

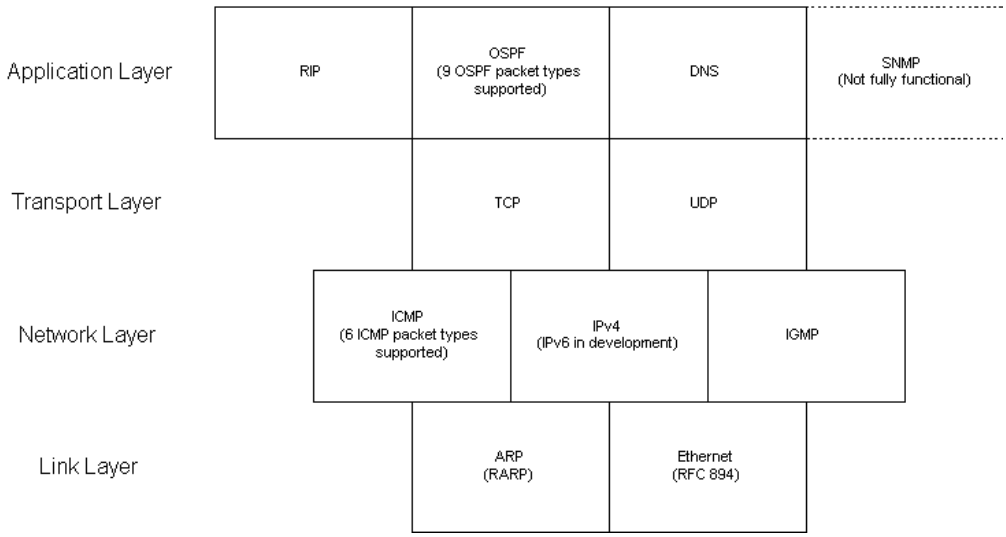

Figure 1. Libnet Supported Protocols

ARCHITECTURAL SUPPORT

Libnet has been ported to several of today's popular architectures. The following is a list of confirmed ports.

- OpenBSD 2.2 - 2.7 (i386)
- FreeBSD 2.2 - 4.0 (i386)
- NetBSD 1.3.x (i386)
- BSD/OS 3.x (i386)
- BSDi 3.0 (i386)
- Linux 2.2.x, 2.0.3x, 2.1.124 (i386, alpha); (libc: 2.4.x, glibc: 2.0.x)
- Solaris 7 (SPARC, gcc 2.7.2[13], 2.8.2), 2.6 (SPARC, gcc 2.8.2), 2.5.x (SPARC, gcc 2.7.2[13])
- IRIX; 6.x
- MacOS 5.3rhapsody (powerpc)
- Windows NT 4.0 and 5.0 (i386)

GETTING STARTED

To continue with this tutorial the programmer should download and install the library (version 1.0.2 is recommended): http://www.packetfactory.net/libnet/libnet.tar.gz. Even if your operating system includes libnet (many open-sourced OSs do), you might want to check and make sure you have a recent version of libnet:

```
strings $PATH_TO_LIBNET/libnet.a | grep version
```

If you've got 1.0.x, you're good to go.

Libnet uses GNU autoconf for precompilation configuration, so to start it on its way, simply type:

```
./configure.
```

Then to build and install the library:

```
make; make install
```

And optionally to build the test and example modules:

```
make test; make example
```

LIBPCAP

It is recommended that the programmer also install libpcap from http://www.tcpdump.org (it will be required for the coding example below). Incredibly useful and powerful programs can be written with libnet in conjunction with libpcap.

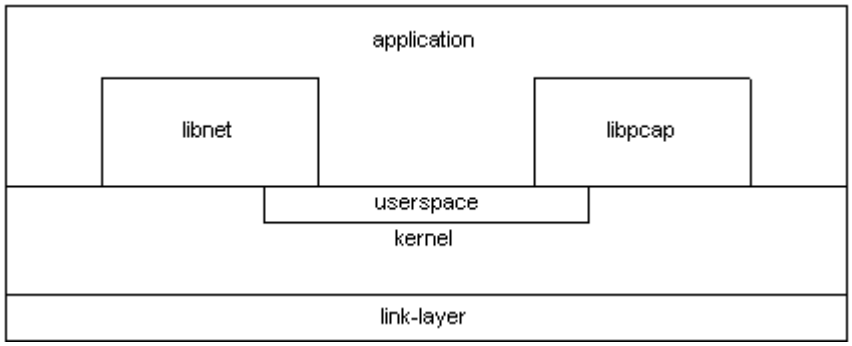

Figure 2. Application layering with libnet and libpcap

Now you're ready to start writing code.

DESIGN DECISIONS

Modularity

Big programs are made up of many smaller modules[1]. These modules provide the user with functions and data structures that are to be used in a program. A module comes in two parts: its interface and its implementation. The interface specifies what a module does, while the implementation specifies how the module does it. The interface declares all of the data types, function prototypes, global information, macros, or whatever is required by the module. The implementation adheres to the specifications set forth by the interface. Libnet is designed off of this paradigm. Each implementation, you'll find, has a corresponding interface.

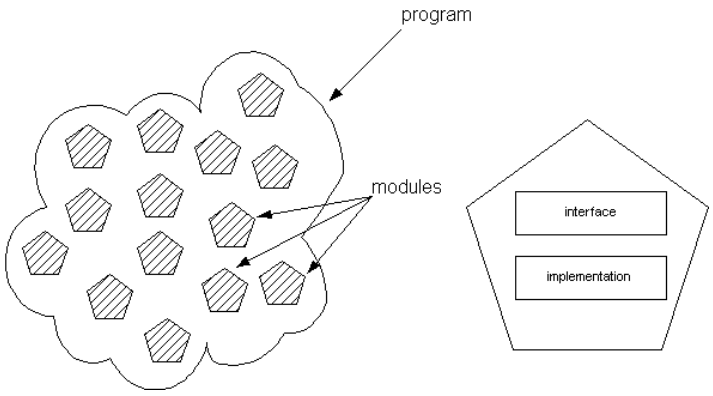

Figure 2. Interfaces and Implementations

There is a third piece to this architecture: the client. The client piece is code that imports and employs the interface, without having to even see the implementation. The programs developed on top of libnet are `client` code.

The Configure Script

Early on in the development of libnet, it became clear that there was a great deal of OS-and architecture-dependent code that had to be conditionally included and compiled. The autoconf configuration stuff (circa version 0.7) worked great to determine what needed to be included and excluded in order to build the library, but did nothing for post-install support. Many of these CPP macros were needed to conditionally include header information for user-based code. This was initially handled by relying on the user to define the proper macros, but this quickly proved inefficient.

Libnet now employs a simple configure script. This script is created during the autoconf configuration and is installed when the library is installed. It handles all of the OS and architecture dependencies automatically. It is mandatory to use the script. You will not be able to compile libnet-based code without it. The script is very simple to use. The following examples should dispel any confusion:

At the command line you can run the script to see what defines are used for that system:

```
echelon:~> libnet-config --defines -D_BSD_SOURCE -D__BSD_SOURCE -
D__FAVOR_BSD -DHAVE_NET_ETHERNET_H -DLIBNET_LIL_ENDIAN</tt></p>
```

At the command line to compile a simple program:

```
echelon:~> gcc -Wall `libnet-config --defines` foo.c -o foo `libnet-
config --libs`
```

In a Makefile:

```
DEFINES = `libnet-config --defines`
```

In a Makefile.in (employing autoheader):

```
DEFINES = `libnet-config --defines` @DEFS@
```

Injection Methods

Libnet offers two interfaces to inject packets on to the network; the raw socket interface and link-layer interface . People often wonder when to use the link-layer interface in place of the raw socket interface. It's mainly a trade of power and complexity for ease of use. The link-layer interface is slightly more complex and requires a bit more coding. The raw socket interface is simpler and quicker to employ. Both require some forethought as to how much memory to allocate (see the section on packet construction).

A standard invocation of either interface might include the following:

raw socket interface

```
libnet_init_packet();
libnet_open_raw_sock();

libnet_build_ip();
libnet_build_tcp();
libnet_do_checksum();
libnet_write_ip();
```

link-layer interface

```
libnet_init_packet();
libnet_open_link_interface()
                    ;
libnet_build_ethernet();
libnet_build_ip();
libnet_build_tcp();
libnet_do_checksum(); (IP)
libnet_do_checksum(); (TCP)
libnet_write_link_layer();
```

PACKET PUSHING IN FIVE EASY STEPS

In order to build and inject an arbitrary network packet, there is a standard order of operations to be followed. There are five easy steps to packet injection happiness:

1.Network Initialization

2.Memory Initialization

3.Packet Construction

4.Packet Checksums

5.Packet Injection

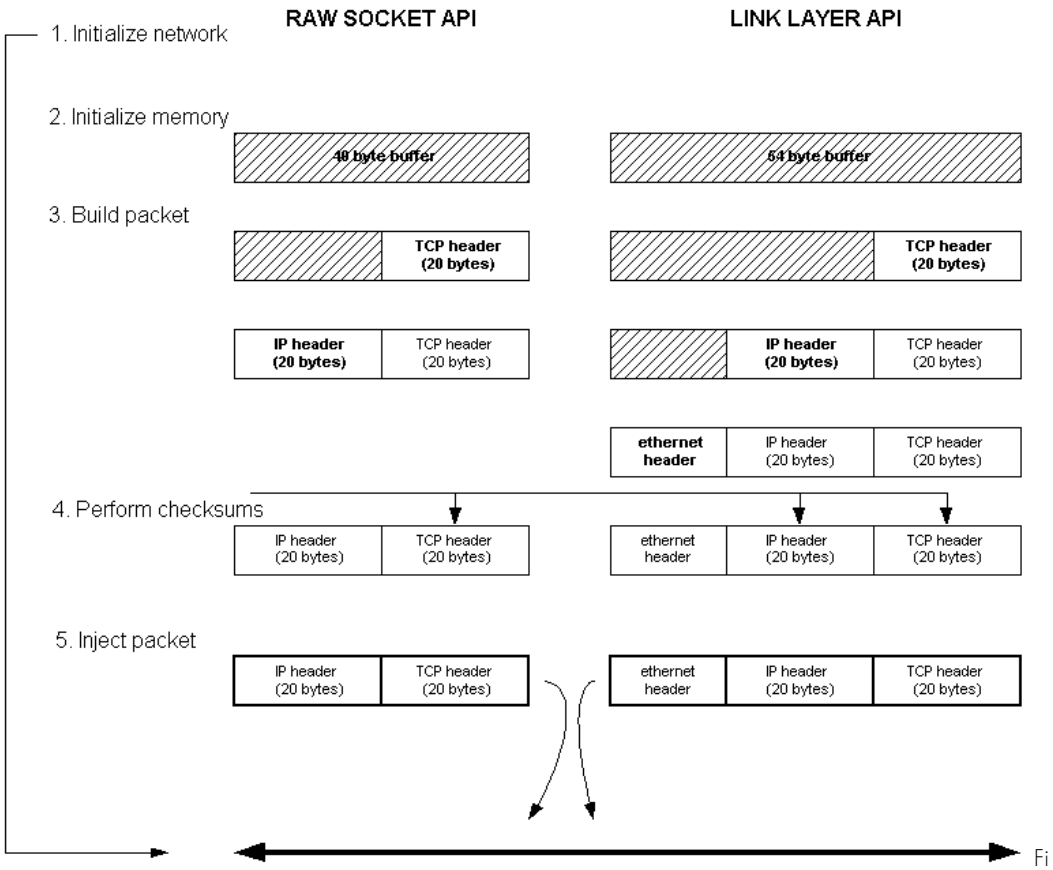

Figure 3. 5 Steps to packet injection

Network Initialization

The first step in using a libnet-enabled application is to bring up the network injection interface. With the raw socket interface , this is with a call to l**ibnet_open_raw_sock()** with the appropriate protocol (usually **PPROTO_RAW**). This call will return a raw socket with **IP_HDRINCL** set on the socket telling the kernel libnet will build the IP header.

The link-layer interface is brought up with a call to **libnet_open_link_interface()** with the proper device argument. This will return a pointer to a ready to go link-interface structure.

Memory Allocation

The next step is to allocate memory for a packet. The conventional way to do this is via a call to **libnet_init_packet()**. The programmer must be certain to allocate enough memory for the entire packet, taking into account all header fields as well as possible payload. Determining exactly how much memory to allocate will require some additional thought depending on which injection method is employed (see below for more information). For a simple TCP packet (sans options) with a 30 byte payload using the raw socket interface , 70 bytes would be required (IP header + TCP header+ payload). The same packet using the link-layer interface , requires 84 bytes (ethernet header + IP header + TCP header + payload). To be safe, the programmer could just allocate **IP_MAXPACKET + ETH_H** bytes (65549) and not worry about overwriting buffer boundaries.

When finished with the memory, it should be released with a call to **libnet_destroy_packet()** (this can either be in a garbage collection function or at the end of the program).

Another method of memory allocation is via the arena interface. Arenas are memory pools that allocate large chunks of memory in one call, divvy out chunks as needed, then de-allocate the whole pool when done. The libnet arena interface is quite useful when preloading different kinds of packets for rapid successive injection. It is initialized with a call to **libnet_init_packet_arena()** and chunks are retrieved with **libnet_next_packet_from_arena().** When finished with the memory it should be released with a call to **libnet_destroy_packet_arena()** (this can either be in a garbage collection function or at the end of the program).

Caveat Emptor: if the programmer does not allocate enough memory for the target packet(s), the program will probably segfault. Libnet can detect when a NULL pointer is passed to a packet constructor, but not when an insufficient amount of memory is passed.

Packet Construction

Packets are constructed modularly. For each protocol layer, there will be a corresponding call to a libnet build function. Depending on the end goal, different things may happen here. For the above raw socket example, calls to **libnet_build_ip()** and **libnet_build_tcp()** will be made. For the link-layer example, an additional call to **libnet_build_ethernet()** will be made. It is

important to note that the ordering of the packet constructor function calls is not significant, it is only necessary that the correct memory offsets be passed to these functions. The functions need to build the packet headers inside the packet buffer as they would appear on the wire and be de-multiplexed by the recipient, and this can be done in any order.

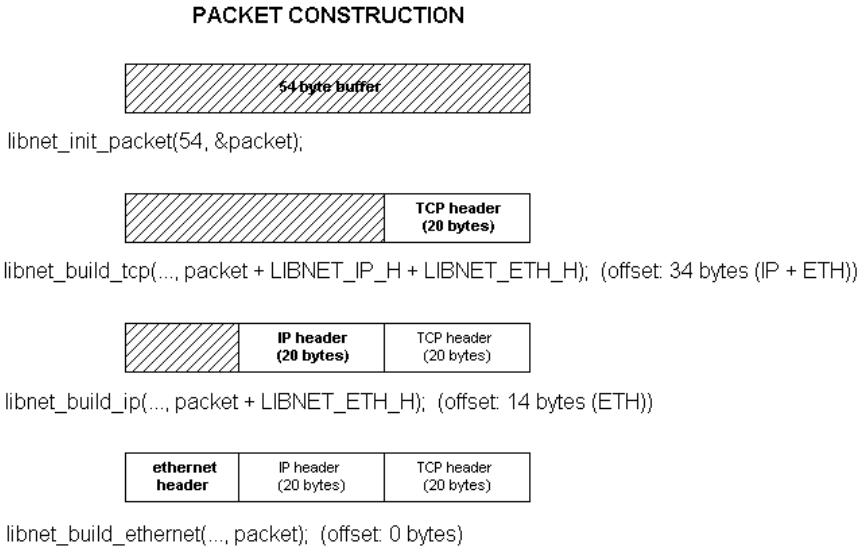

Figure 4. Modular Packet Construction

The packet buffer is initialized with a call to **`libnet_init_packet()`**. **`libnet_build_ethernet()`** is passed the beginning of the buffer with an offset of 0 (as it needs to build an ethernet header at the front of the packet). **`libnet_build_ip()`** would get the buffer at a 14 byte (**`ETH_H`**) offset to construct the IP header in the correct location, while **`libnet_build_tcp()`** gets the buffer 20 bytes beyond this (or at a 34 bytes offset from the beginning (**`ETH_H + IP_H`**)).

Packet Checksums

The next-to-last step is to compute the packet checksums (assuming the packet is an IP packet of some sort). For the raw socket interface, the programmer need only compute a transport layer checksum (assuming the packet has a transport layer protocol) as the kernel will handle the IP checksum. For the link-layer interface, the IP checksum must be explicitly computed. Checksums are calculated via **`libnet_do_checksum()`**, which expects a pointer to the IP header of the packet.

Packet Injection

The final step is to write the packet to the network. Using the raw socket interface this is accomplished with **`libnet_write_ip()`**, and with the link-layer interface it is accomplished with **`libnet_write_link_layer()`**. Both functions return the number of bytes written (which should jibe with the size of the packet in question) or a -1 on error.

ADDITIONAL CONSIDERATIONS

Spoofing Ethernet Addresses

Certain operating systems using the Berkeley Packet Filter (bpf) for link-layer access do not allow arbitrary specification of source ethernet addresses. This is not so much a bug as it is an oversight in the protocol. The way around this is to patch the kernel. There are two ways to patch a kernel, either statically, with kernel diffs (which requires the individual to have the kernel sources, and know how to rebuild and install a new kernel) or dynamically, with loadable kernel modules (lkms). Libnet includes code to do both. Since it is a bit overzealous to assume people will want to patch their kernel for a library, included with the libnet distribution is lkm code to seamlessly bypass the bpf restriction.

In order to spoof ethernet packets on bpf-based systems (currently supported are FreeBSD and OpenBSD) change the corresponding support/bpf directory and build the module, and modload it.

The module works as per the following description:

The 4.4BSD machine-independent ethernet driver does not allow upper layers to forge the ethernet source address; all ethernet outputs cause the output routine to build a new ethernet header, and the process that does this explicitly copies the MAC address registered to the interface into this header.

This is odd, because the bpf writing convention asserts that writes to bpf must include a link-layer header; it's intuitive to assume that this header is, along with the rest of the packet data, written to the wire.

This is not the case. The link-layer header is used solely by the bpf code in order to build a sockaddr structure that is passed to the generic ethernet output routine; the header is then effectively stripped off the packet. The ethernet output routine consults this sockaddr to obtain the ethernet type and destination address, but not the source address.

The Libnet lkm simply replaces the standard ethernet output routine with a slightly modified one. This modified version retrieves the source ethernet address from the sockaddr and uses it as the source address for the header written the wire. This allows bpf to be used to seamlessly forge ethernet packets in their entirety, which has applications in address management.

The modload glue provided traverses the global list of system interfaces, and replaces any pointer to the original ethernet output routine with the new one we've provided. The unload glue undoes this. The effect of loading this module will be that all ethernet interfaces on the system will support source address forging.

Raw Sockets Limitations

Raw sockets are horribly non-standard across different platforms.

- Under some x86 BSD implementations the IP header length and fragmentation bits need to be in host byte order, and under others, network byte order.
- Solaris does not allow you to set many IP header-related bits including the length, fragmentation flags, or IP options.
- Linux requires SO_BROADCAST to be set on the raw socket for the injection of broadcast IP datagrams (which libnet now does).

Because of these quirks, it is often recommended that the programmer employ libnet's link-layer interface over the raw socket interface.

CONCLUSION

Libnet URLs

- Libnet homepage: http://www.packetfactory.net/Projects/libnet
- Libnet mailing list: http://www.packetfactory.net/libnet/mailing-list.html
- Libnet-based projects: http://www.packetfactory.net, http://www.monkey.org/~dugsong/dsniff/

REFERENCES

[1] Hanson, David R., "C Interfaces and Implementations", Addison-Wesley, 1997

BIO

Throughout his career, Mike Schiffman has been involved in most every technical arena computer security has to offer. He has researched and developed many cutting-edge technologies including tools such as firewalk and tracerx as well as the low-level packet shaping library libnet. Mike has led audit teams through engagements for fortune 500 companies in the banking, automotive and manufacturing industries. He has spoken in front of NSA, CIA, DOD, AFWIC, SAIC, and others. Mike has written for numerous technical journals and has worked on several books. Currently, Mike is the director of research and development at Guardent, the leading provider of professional security services.