

Using libpcap in C

Advertisement

Advertisement

Submitted by NanoDano (/users/nanodano) on Fri, 08/14/2015 - 22:19

Overview

- Intro
- Find a Network Device
- Get Info About Device
- Live Capture
- Print Packet Info
- Processing Packets with pcap_loop()
- Determining Packet Type
- Finding the Data Payload
- Loading Pcap File
- Wireless, Promiscuous, and Monitor Mode
- Using Filters
- Example Filters
- Closing Handle
- Sending Packets
- Bindings to Other Languages
- References

Intro

libpcap allows us to capture or send packets from a live network device or a file. These code examples will walk you through using libpcap to find network devices, get information about devices, process packets in real time or offline, send packets, and even listen to wireless traffic. This is aimed at Debian based Linux distributions but may also work on Mac OSX. Not intended for Windows, but WinPcap (<https://www.winpcap.org/>) is a port that is available.

Compiling a pcap program requires linking with the pcap lib. You can install it in Debian based distributions with

```
sudo apt-get install libpcap-dev
```

Once the libpcap dependency is installed, you can compile pcap programs with the following command. You will need to run the program as root or with sudo to have permission to access the network card.

```
gcc <filename> -lpcap
```

Find a Network Device

The simplest program to start with will just look for a network device. We won't be able to do anything else if we can't get a device to work with. If you get a device called "any" bound to 0.0.0.0 that is acceptable. You can also use **ifconfig** or **ip addr** to get device names.

```

/* Compile with: gcc find_device.c -lpcap */
#include <stdio.h>
#include <pcap.h>

int main(int argc, char **argv) {
    char *device; /* Name of device (e.g. eth0, wlan0) */
    char error_buffer[PCAP_ERRBUF_SIZE]; /* Size defined in pcap.h */

    /* Find a device */
    device = pcap_lookupdev(error_buffer);
    if (device == NULL) {
        printf("Error finding device: %s\n", error_buffer);
        return 1;
    }

    printf("Network device found: %s\n", device);
    return 0;
}

```

Get Info About Device

Now we can expand on the simple program above. After we find a device, we can call **pcap_lookupnet** and it will tell us what the ip address and subnet mask of the device. It will also fill up the error buffer with an error message if something goes wrong.

```

#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>
#include <string.h>

int main(int argc, char **argv) {
    char *device;
    char ip[13];
    char subnet_mask[13];
    bpf_u_int32 ip_raw; /* IP address as integer */
    bpf_u_int32 subnet_mask_raw; /* Subnet mask as integer */
    int lookup_return_code;
    char error_buffer[PCAP_ERRBUF_SIZE]; /* Size defined in pcap.h */
    struct in_addr address; /* Used for both ip & subnet */

    /* Find a device */
    device = pcap_lookupdev(error_buffer);
    if (device == NULL) {
        printf("%s\n", error_buffer);
        return 1;
    }

    /* Get device info */
    lookup_return_code = pcap_lookupnet(
        device,
        &ip_raw,
        &subnet_mask_raw,
        error_buffer
    );
    if (lookup_return_code == -1) {
        printf("%s\n", error_buffer);
        return 1;
    }

    /*
    If you call inet_ntoa() more than once
    you will overwrite the buffer. If we only stored
    the pointer to the string returned by inet_ntoa(),
    and then we call it again later for the subnet mask,
    our first pointer (ip address) will actually have
    the contents of the subnet mask. That is why we are
    using a string copy to grab the contents while it is fresh.
    The pointer returned by inet_ntoa() is always the same.
    */

```

This is from the man:

The inet_ntoa() function converts the Internet host address in, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

```

/* Get ip in human readable form */
address.s_addr = ip_raw;
strcpy(ip, inet_ntoa(address));
if (ip == NULL) {
    perror("inet_ntoa"); /* print error */
    return 1;
}

```

```
/* Get subnet mask in human readable form */
address.s_addr = subnet_mask_raw;
strcpy(subnet_mask, inet_ntoa(address));
if (subnet_mask == NULL) {
    perror("inet_ntoa");
    return 1;
}

printf("Device: %s\n", device);
printf("IP address: %s\n", ip);
printf("Subnet mask: %s\n", subnet_mask);

return 0;
}
```

The program above will look up the device like the first program, but will go a step further and get information about the device as well. The next step is to use the device to actually capture packets. Later on we'll also look at opening an existing pcap file instead of capturing live.

Live Capture

The next example program will demonstrate how to open a network device for live capturing, and capture a single packet. We will use **pcap_open_live** to get a handle, just like we would when opening a file to get a file handle. Instead of a **FILE** type handle though it is a **pcap_t** type.

```

#include <stdio.h>
#include <time.h>
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

void print_packet_info(const u_char *packet, struct pcap_pkthdr packet_header);

int main(int argc, char *argv[]) {
    char *device;
    char error_buffer[PCAP_ERRBUF_SIZE];
    pcap_t *handle;
    const u_char *packet;
    struct pcap_pkthdr packet_header;
    int packet_count_limit = 1;
    int timeout_limit = 10000; /* In milliseconds */

    device = pcap_lookupdev(error_buffer);
    if (device == NULL) {
        printf("Error finding device: %s\n", error_buffer);
        return 1;
    }

    /* Open device for Live capture */
    handle = pcap_open_live(
        device,
        BUFSIZ,
        packet_count_limit,
        timeout_limit,
        error_buffer
    );

    /* Attempt to capture one packet. If there is no network traffic
     and the timeout is reached, it will return NULL */
    packet = pcap_next(handle, &packet_header);
    if (packet == NULL) {
        printf("No packet found.\n");
        return 2;
    }

    /* Our function to output some info */
    print_packet_info(packet, packet_header);

    return 0;
}

```

The program above will capture a single packet and then call `print_packet_info()`. I left that function out intentionally to keep the snippet above short. Let's take a look at `print_packet_info()` now. You will need to add that function after `main()`.

Print Packet Info

The `print_packet_info()` function will provide information about what type of ethernet packet it found (IP or ARP), the timestamp, packet length, and the source and destination of the packet.

```

void print_packet_info(const u_char *packet, struct pcap_pkthdr packet_header) {
    printf("Packet capture length: %d\n", packet_header.caplen);
    printf("Packet total length %d\n", packet_header.len);
}

```

Processing Packets with `pcap_loop()`

We've slowly been adding more and more to our capabilities with pcap. We've covered finding a device, opening a device, how to capture a single packet, and how to pull information from the packet. Capturing a single packet is not very practical though. If you are looking for a single packet, chances are it will not be the very first packet you see, but buried in a stream of many packets. Now we will talk about how to process all of the packets received continuously. This is where `pcap_loop()` comes in. The `pcap_loop()` function is provided by libpcap. This is what the declaration looks like in `pcap.h`.

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

From the previous examples we know that `pcap_t` is the device handle from where we want to capture. The second argument is an int which is the number of packets you want to capture. Pass `0` for unlimited packets. The `pcap_handler` accepts a function name that is the callback to be run on every packet captured. We will look more in depth at that in a moment. The last argument to `pcap_loop` is arguments to the callback function. We do not have any in our example so we pass NULL.

The `pcap_handler` argument for `pcap_loop()` is a specially defined function. This is the declaration of the type in `pcap.h`.

```

/* The contract we have to satisfy with our callback function */
typedef void (*pcap_handler)(u_char *, const struct pcap_pkthdr *,
                            const u_char *);

```

We will define our own callback function to handle packets, but it will have to match the format of the `pcap_handler` type. Here is an empty example. We will create a handler later that actually does something useful.

```

void my_packet_handler(
    u_char *args,
    const struct pcap_pkthdr *header,
    const u_char *packet
)
{
    /* Do something with the packet here.
       The print_packet_info() function shows in the
       previous example could be used here. */
    /* print_packet_info(packet, header); */
    return;
}

```

Let's look at a full program example of how to take advantage of `pcap_loop()`. Inside our callback function that handles packets, we will just print out the packet information like we did in our previous example. Since this program will continuously loop and process packets, you will have to use **CTRL-C** to end the program or use the **kill** command.

```
#include <stdio.h>
#include <time.h>
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

void my_packet_handler(
    u_char *args,
    const struct pcap_pkthdr *header,
    const u_char *packet
);
void print_packet_info(const u_char *packet, struct pcap_pkthdr packet_header);

int main(int argc, char *argv[]) {
    char *device;
    char error_buffer[PCAP_ERRBUF_SIZE];
    pcap_t *handle;
    int timeout_limit = 10000; /* In milliseconds */

    device = pcap_lookupdev(error_buffer);
    if (device == NULL) {
        printf("Error finding device: %s\n", error_buffer);
        return 1;
    }

    /* Open device for Live capture */
    handle = pcap_open_live(
        device,
        BUFSIZ,
        0,
        timeout_limit,
        error_buffer
    );
    if (handle == NULL) {
        fprintf(stderr, "Could not open device %s: %s\n", device, error_buffer);
        return 2;
    }

    pcap_loop(handle, 0, my_packet_handler, NULL);

    return 0;
}

void my_packet_handler(
    u_char *args,
    const struct pcap_pkthdr *packet_header,
    const u_char *packet_body
)
{
    print_packet_info(packet_body, *packet_header);
    return;
}

void print_packet_info(const u_char *packet, struct pcap_pkthdr packet_header) {
    printf("Packet capture length: %d\n", packet_header.caplen);
    printf("Packet total length %d\n", packet_header.len);
}
```

Determining Packet Type

Packet type can be determined by inspecting the ethernet header and comparing the ether_type value against known constants for IP, ARP, and reverse ARP types.

```
#include <stdio.h>
#include <pcap.h>
#include <netinet/in.h>
#include <net/ethernet.h>

/* This function can be used as a callback for pcap_Loop() */
void my_packet_handler(
    u_char *args,
    const struct pcap_pkthdr* header,
    const u_char* packet
) {
    struct ether_header *eth_header;
    /* The packet is larger than the ether_header struct,
       but we just want to look at the first part of the packet
       that contains the header. We force the compiler
       to treat the pointer to the packet as just a pointer
       to the ether_header. The data payload of the packet comes
       after the headers. Different packet types have different header
       lengths though, but the ethernet header is always the same (14 bytes) */
    eth_header = (struct ether_header *) packet;

    if (ntohs(eth_header->ether_type) == ETHERTYPE_IP) {
        printf("IP\n");
    } else if (ntohs(eth_header->ether_type) == ETHERTYPE_ARP) {
        printf("ARP\n");
    } else if (ntohs(eth_header->ether_type) == ETHERTYPE_REVARP) {
        printf("Reverse ARP\n");
    }
}

int main(int argc, char **argv) {
    pcap_t *handle;
    char error_buffer[PCAP_ERRBUF_SIZE];
    char *device = "eth0";
    int snapshot_len = 1028;
    int promiscuous = 0;
    int timeout = 1000;

    handle = pcap_open_live(device, snapshot_len, promiscuous, timeout, error_buffer);
    pcap_loop(handle, 1, my_packet_handler, NULL);
    pcap_close(handle);
    return 0;
}
```

Finding the Data Payload

The payload is not always going to be in the same location. Headers will be different sizes based on the type of packet and what options are present. For this example we are strictly talking about IP packets with TCP on top.

We start with the pointer to the beginning of the packet. The first 14 bytes are the ethernet header. That is always going to be the same because it is defined in the standard. That ethernet header contains the destination then source MAC(hardware) addresses, which are lower level than IP addresses. Each one of those is 6 bytes. There are also two more bytes at the end of the ethernet header that represent the type. With two bytes you could have thousands of different types. They could be ARP packets but we only want IP packets in this situation.

Ethernet is considered the second layer in OSI's model. The only level lower than ethernet is the physical medium that the data uses, like a copper wire, fiber optics, or radio signals.

On top of ethernet, the second layer, we have the third layer: IP. That is our IP address which is one level higher than the hardware MAC address. Layer four is TCP and UDP. Before we can actually get to our payload, we have to get past the ethernet, IP, and TCP layer. That is how we will come up with the formula for calculating the payload location in memory.

IP and TCP header length are variable. The length of the IP header is one of the very first values provided in the IP header. We have to get the IP header length to figure out how much further we have to look to find the beginning of the TCP header. Once we know where the TCP header is we can get the data offset value, which is part of the TCP header. The data offset is how much further we have to go from the start of the TCP packet to the actual payload. Look at this psuedo-code.

```
payload_pointer =  
packet_pointer + len(Ethernet header) + len(IP header) + len(TCP header)
```

- The ethernet header is always 14 bytes as defined by standards.
- The IP header length is always stored in a 4 byte integer at byte offset 4 of the IP header.
- The TCP header length is always stored in a 4 byte integer at byte offset 12 of the TCP header.
- The payload starts at packet base location plus all the header lengths.

Now we have enough knowledge to figure out where the payload is in memory. The IP header and TCP are typically about 20 bytes each if there are no options passed. That means the first 54 bytes are the header layers, and the rest is actual data. We should not guess or assume the headers will always be 20 bytes each though. We need to get the actual header length for both IP and TCP layers in order to calculate the offset for the payload. That is what this code example will do.

```

#include <stdio.h>
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

/* Finds the payload of a TCP/IP packet */
void my_packet_handler(
    u_char *args,
    const struct pcap_pkthdr *header,
    const u_char *packet
)
{
    /* First, lets make sure we have an IP packet */
    struct ether_header *eth_header;
    eth_header = (struct ether_header *) packet;
    if (ntohs(eth_header->ether_type) != ETHERTYPE_IP) {
        printf("Not an IP packet. Skipping...\n\n");
        return;
    }

    /* The total packet Length, including all headers
       and the data payload is stored in
       header->len and header->caplen. Caplen is
       the amount actually available, and len is the
       total packet length even if it is larger
       than what we currently have captured. If the snapshot
       length set with pcap_open_live() is too small, you may
       not have the whole packet. */
    printf("Total packet available: %d bytes\n", header->caplen);
    printf("Expected packet size: %d bytes\n", header->len);

    /* Pointers to start point of various headers */
    const u_char *ip_header;
    const u_char *tcp_header;
    const u_char *payload;

    /* Header Lengths in bytes */
    int ethernet_header_length = 14; /* Doesn't change */
    int ip_header_length;
    int tcp_header_length;
    int payload_length;

    /* Find start of IP header */
    ip_header = packet + ethernet_header_length;
    /* The second-half of the first byte in ip_header
       contains the IP header length (IHL). */
    ip_header_length = ((*ip_header) & 0x0F);
    /* The IHL is number of 32-bit segments. Multiply
       by four to get a byte count for pointer arithmetic */
    ip_header_length = ip_header_length * 4;
    printf("IP header length (IHL) in bytes: %d\n", ip_header_length);

    /* Now that we know where the IP header is, we can
       inspect the IP header for a protocol number to
       make sure it is TCP before going any further.
       Protocol is always the 10th byte of the IP header */
    u_char protocol = *(ip_header + 9);
    if (protocol != IPPROTO_TCP) {

```

```

    printf("Not a TCP packet. Skipping...\n\n");
    return;
}

/* Add the ethernet and ip header length to the start of the packet
   to find the beginning of the TCP header */
tcp_header = packet + ethernet_header_length + ip_header_length;
/* TCP header length is stored in the first half
   of the 12th byte in the TCP header. Because we only want
   the value of the top half of the byte, we have to shift it
   down to the bottom half otherwise it is using the most
   significant bits instead of the least significant bits */
tcp_header_length = (((tcp_header + 12)) & 0xF0) >> 4;
/* The TCP header length stored in those 4 bits represents
   how many 32-bit words there are in the header, just like
   the IP header length. We multiply by four again to get a
   byte count. */
tcp_header_length = tcp_header_length * 4;
printf("TCP header length in bytes: %d\n", tcp_header_length);

/* Add up all the header sizes to find the payload offset */
int total_headers_size = ethernet_header_length+ip_header_length+tcp_header_length;
printf("Size of all headers combined: %d bytes\n", total_headers_size);
payload_length = header->caplen -
    (ethernet_header_length + ip_header_length + tcp_header_length);
printf("Payload size: %d bytes\n", payload_length);
payload = packet + total_headers_size;
printf("Memory address where payload begins: %p\n\n", payload);

/* Print payload in ASCII */
/*
if (payload_length > 0) {
    const u_char *temp_pointer = payload;
    int byte_count = 0;
    while (byte_count++ < payload_length) {
        printf("%c", *temp_pointer);
        temp_pointer++;
    }
    printf("\n");
}
*/
return;
}

int main(int argc, char **argv) {
    char *device = "eth0";
    char error_buffer[PCAP_ERRBUF_SIZE];
    pcap_t *handle;
    /* Snapshot Length is how many bytes to capture from each packet. This includes*/
    int snapshot_length = 1024;
    /* End the Loop after this many packets are captured */
    int total_packet_count = 200;
    u_char *my_arguments = NULL;

    handle = pcap_open_live(device, snapshot_length, 0, 10000, error_buffer);
    pcap_loop(handle, total_packet_count, my_packet_handler, my_arguments);
}

```

```
    return 0;  
}
```

Loading Pcap File

Loading a pcap file is just like opening a device. Opening a file returns a **pcap_t** just like opening a network device.

```
pcap_t *pcap_open_offline(const char *fname, char *errbuf);
```

It just needs the filename string and an error buffer.

```
char error_buffer[PCAP_ERRBUF_SIZE];  
pcap_t *handle = pcap_open_offline("capture_file.pcap", error_buffer);
```

Once the file is opened just like a device, you can call **pcap_next()** to get one packet at a time, or use **pcap_loop()**.

Wireless, Promiscuous, and Monitor Mode

To turn it on, call

To clarify the difference between promiscuous mode and monitor mode: monitor mode is just for wireless cards and promiscuous is for wireless and wired. Monitor mode lets the card listen to wireless packets without being associated to an access point. Promiscuous mode lets the card listen to all packets, even ones not intended for it.

Use **pcap_set_rfmon()** to turn on monitor mode. Use **pcap_set_promisc()** to turn on promiscuous mode. Call them before the device is activated. Pass any non-zero integer to turn it on and 0 to turn off.

```
int pcap_set_rfmon(pcap_t *p, int rfmon);
```

Remember, **pcap_t** is the device handle opened with **pcap_open_live()**. Call this before activating the device. **pcap_can_set_rfmon()** can be used to see if a device has the capability.

We have been using **pcap_open_live()** to get the device handle, but that creates the device handle and activates it at the same time. To set the rfmon mode before activating the device handle must be manually created. Use **pcap_create()** and **pcap_activate()**.

```
char error_buffer[PCAP_ERRBUF_SIZE];  
pcap_t *handle = pcap_create("wlan0", error_buffer);  
pcap_set_rfmon(handle, 1);  
pcap_set_promisc(handle, 1); /* Capture packets that are not yours */  
pcap_set_snaplen(handle, 2048); /* Snapshot Length */  
pcap_set_timeout(handle, 1000); /* Timeout in milliseconds */  
pcap_activate(handle);  
/* handle is ready for use with pcap_next() or pcap_Loop() */
```

Using Filters

You compile textual expressions in to a filter program first. Then you can apply the filters to the pcap handle. You can filter by source or destination, port, or a number of other things. For a full reference of filters, use the man page for **pcap-filter**.

```
man 7 pcap-filter
```

This is the declaration of the compile and setfilter functions.

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
                 bpf_u_int32 netmask)

int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

The example below shows how to compile and set the filter.

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

/* For information on what filters are available
   use the man page for pcap-filter
   $ man pcap-filter
*/

int main(int argc, char **argv) {
    char dev[] = "eth0";
    pcap_t *handle;
    char error_buffer[PCAP_ERRBUF_SIZE];
    struct bpf_program filter;
    char filter_exp[] = "port 80";
    bpf_u_int32 subnet_mask, ip;

    if (pcap_lookupnet(dev, &ip, &subnet_mask, error_buffer) == -1) {
        printf("Could not get information for device: %s\n", dev);
        ip = 0;
        subnet_mask = 0;
    }
    handle = pcap_open_live(dev, BUFSIZ, 1, 1000, error_buffer);
    if (handle == NULL) {
        printf("Could not open %s - %s\n", dev, error_buffer);
        return 2;
    }
    if (pcap_compile(handle, &filter, filter_exp, 0, ip) == -1) {
        printf("Bad filter - %s\n", pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &filter) == -1) {
        printf("Error setting filter - %s\n", pcap_geterr(handle));
        return 2;
    }

    /* pcap_next() or pcap_Loop() to get packets from device now */
    /* Only packets over port 80 will be returned. */

    return 0;
}
```

Example Filters

```

# Packets to or from host
host box4

# or by IP
host 8.8.8.8

# Packets between hosts
host box2 and box4

# Packets by source
src 10.2.4.1

# By destination
dst 99.99.2.2

# By port
port 143
portrange 1-1024

# By source/destination port
src port 22
dst port 80

# By Protocol
tcp
udp
icmp

# And
src localhost and dst port 22
src localhost && dst port 22

# Or
port 80 or 22
port 80 || 22

# Grouping
src localhost and (dst port 80 or 22 or 443)

```

Closing Handle

The handle (or descriptor) for the device/file should be closed just like a file. use **pcap_close()**.

```
pcap_close(handle);
```

Sending Packets

You can send packets using **pcap_inject()**. It could not get any simpler. You pass it a raw pointer and a length and it will send whatever it finds in memory to the handle.

```
int pcap_inject(pcap_t *p, const void *buf, size_t size);
```

```
int bytes_written = pcap_inject(handle, &raw_bytes, sizeof(raw_bytes));
```

Bindings to Other Languages

To this day, **libpcap** is still going strong. There are bindings to almost all other languages. There are bindings to Go (<https://github.com/google/gopacket>), PHP (<https://github.com/marcelog/SimplePcap>), Python (<http://sourceforge.net/projects/pylibpcap>), Ruby (<http://sourceforge.net/projects/rubypcap/>), Perl (<http://search.cpan.org/~kcarnut/Net-Pcap-0.05/Pcap.pm>), Java (<http://jnetpcap.com/>) and more.

I have a page on using the **gopacket** library in Go to capture, analyze, and inject packets with Go ([/content/packet-capture-injection-and-analysis-gopacket](#)). Gopacket is more than just a straight wrapper of libpcap and offers its own benefits.

Having a solid understanding of the C library will make it much easier to work with the bindings in other languages. Most of them are direct wrappers so all the function names are the same. Every language has their pros and cons so remember that there are many options available. Personally, Go is the most attractive because of its threading capabilities and speed without the amount of work needed in a C program. If speed is not critical, Python would be my next choice for writing quick and dirty scripts to get what I need. For many situations, the easiest approach is to use **tcpdump** to write to a file and then write programs to analyze the file offline. That can take a lot of disk space though and sometimes you need to operate on the packets in real time so C, C++ and Go might be the most appropriate.

References

It is difficult to memorize all the function calls and what types you have to pass for each argument. Fortunately, it is well documented. There is an online manual at www.tcpdump.org (<http://www.tcpdump.org/>), but there is a better way to get help. You do not even have to go online or open a browser. Learn to use the man pages efficiently. Here are a few examples of using man.

```
man pcap  
man 3 pcap
```

```
man pcap_open_live
```

```
man pcap filter  
man 7 pcap-filter
```

There are different pages. **man** without a page number will give you whatever it can find, but you can be explicit too. Page 3 is the C library functions and 7 is miscellaneous. If you are unsure you can always look at the man page for man. No, seriously, man, you can man man to get info about the man pages.

```
man man
```

How to Specify SSH key for Git repository

2 years ago • 1 comment

Virtual Hackerspace and Resources for Software Developers of all Skill ...

NeoTrellis M4 CircuitPython Tutorial ...

3 years ago • 1 comment

Virtual Hackerspace and Resources for Software Developers of all Skill ...

Which Linux I use?

a year ago • 1 comment

Virtual Hackerspace and Resources for Software Developers of all Skill ...

9 Comments Dev Dungeon

🔒 Privacy Policy

1 Login

Heart Favorite 1

Tweet

Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



[Esra Ertak](#) • 2 years ago

How can I capture the ethernet frames using libpcap api. please..

^ | v • Reply • Share ›



[ran](#) • 3 years ago

Hi

What is the advantage of using libpcap instead of direct socket programming in C?

Another question: aes simple encryption is possible?

Thanks

^ | v • Reply • Share ›



[Moore Derick](#) • 3 years ago

Thank you NanoDano.... I really enjoyed reading this as it confirmed (and explained) things I have developed for a project I'm working on. However, I have learned of an exception to the 14 byte ethernet header as accepted and amplified in your writing. I'm copying a write up I put on Stack Overflow. Here goes:

I have a running application that needed to execute properly on both my local platform, running Ubuntu 18.4, and recompiled, run on a FreeBSD platform.

I encountered a problem with the packets because the Ethernet header was 10 bytes shorter on the FreeBSD platform

I have accommodated the difference in software following the receipt of

the packet by looking into the packet for data from the ICMP Header that is always present in my data. Knowing the offset to that data informs the decision about the header size.

Once I know the size of the Ethernet header, I strip it off, and what remains is consistent across devices.

But, I'm bugged that there is a difference that I cannot understand. Can anyone point me to documentation describing the reason why the difference of 4 or 14 bytes?

I have explored the web looking for an explanation, but found nothing that would justify the difference. Even FreeBSD 'Ethernet.h' shows 14 bytes.

```
#define ETHER_HDR_LEN (ETHER_ADDR_LEN*2 +  
ETHER_TYPE_LEN) where that is (6*2)+2 equals 14
```

So, that's it, and I'm not expecting any answer for my problem. But, I'd sure like to understand it if you have a clue.

Derick

^ | v • Reply • Share >



Esra Ertsak → Moore Derick • 2 years ago

hi i have a question how can i make this" CAPTURING
ETHERNET FRAMES USING LIBPCAP API"

^ | v • Reply • Share >

Sadik Sikder • 5 years ago

is there any suitable IDE that can comply with libcap to write it?

^ | v • Reply • Share >

նույնական • 5 years ago

So awesome. Thanks!

^ | v • Reply • Share >

Paweł Sosnowski • 4 years ago

pcap_lookupdev() is deprecated. You can achieve the same result with:

```
#include <stdlib.h>  
#include <stdio.h>  
#include <pcap.h>  
  
int main(void) {  
  
    char ebuf[PCAP_ERRBUF_SIZE];  
    pcap_if_t *devs;  
  
    pcap_findalldevs(&devs, ebuf);  
  
    printf("%s\n", devs->name);  
  
    exit(EXIT_SUCCESS);  
}
```

Cheers.

Tags:

- C (/tags/c)
- pcap (/tags/pcap)
- Security (/tags/security)
- sysadmin (/topics/sysadmin)
- TCP (/tags/tcp)
- Linux (/tags/linux)
- Mac (/tags/mac)

Advertisement

Advertisement

Search



Join me on Discord!

JOIN ME ON



DISCORD

(/discord)

DevDungeon YouTube Channel

(<https://www.youtube.com/DevDungeon>)

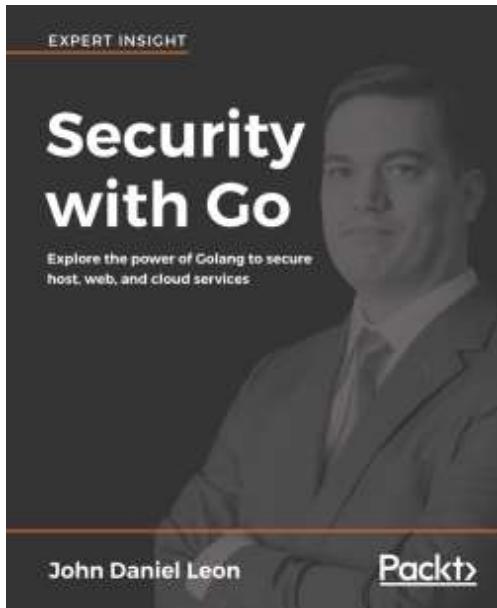


Related Posts

- Install Lutris Gaming on Debian (</content/install-lutris-gaming-debian>)
- Which Linux Distro do I use? (</content/which-linux-distro-do-i-use>)
- Build Node.js from Source (</content/build-nodejs-source>)
- Change GRUB Bootloader colors, image & timeout (</content/change-grub-bootloader-colors-image-timeout>)
- Change Console Font Size in Debian (</content/change-console-font-size-debian>)
- Install Linux in VirtualBox (</content/install-linux-virtualbox>)
- Process Form Inputs with PHP (</content/process-form-inputs-php>)
- HTTP Basic Authentication with PHP (</content/http-basic-authentication-php>)
- Cron Job Examples and Tip (</content/cron-job-examples-and-tip>)
- Tmux Basics (</content/tmux-basics>)

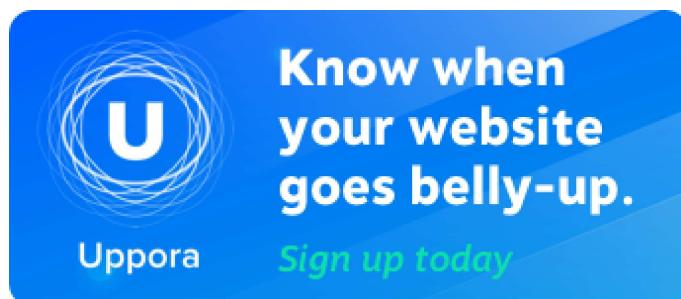
- How to Specify SSH key for Git repository (</content/how-specify-ssh-key-git-repository>)
- How to Create a Secure Linux System User (</content/how-create-secure-linux-system-user>)
- Open a Linux Firewall port with firewall-cmd (</content/open-linux-firewall-port-firewall-cmd>)
- Run Python WSGI Web App with Waitress (</content/run-python-wsgi-web-app-waitress>)
- Run Python Script as Windows Service (</content/run-python-script-windows-service>)

Buy my book!



(<https://www.packtpub.com/networking-and-servers/security-go>)

Sign up for site monitoring!



Join the mailing list!

Email Address *

Name

Advertisement

Advertisement

About DevDungeon

Links

- YouTube.com/DevDungeon
(<https://www.youtube.com/DevDungeon>)

About Me

- Read more about me
(<https://www.devdungeon.com/content/about-me>).

DevDungeon.com is a virtual hackerspace community with resources for developers and hackers.

- GitHub.com/DevDungeon (<https://www.github.com/DevDungeon>)
 - DevDungeon Discord Server (/discord)
 - RSS Feed (/rss.xml)
-
- Polk Design Company (<https://www.polkdesigncompany.com/>)
-
- Privacy Policy (/content/privacy-policy)
 - XML Sitemap (/sitemap.xml)

- Get my public PGP key (<https://www.devdungeon.com/gpg>).
Email:
nanodano@devdungeon.com
(<mailto:nanodano@devdungeon.com>)
- XMPP:
nanodano@devdungeon.com
(<xmpp:nanodano@devdungeon.com>)