Lab 2: C Programming Primer and Parsing Network Packets

Part 1 due: Sept. 21st at 11:59pm Part 2 due: Sept. 25th at 11:59pm

Overview

The goal of this lab is to familiarize yourself with nework packet headers by parsing some yourself. This assignment must be written in C (not C++), and will require a fairly deep understanding of pointer manipulation, memory layouts of basic and structured data types, and formatted printing. To ensure you can complete these tasks, you will first complete a sequence of smaller programs (Part 1). Following, you will use the skills you developed in Part 1 to complete the packet parser in Part 2.

Deliverables

Your submission should minimally include the following programs for each part of the lab.

- **Part 1**
  - `data_sizes.c`
  - `bit_wise.c`
  - `Makefile`
  - `README`
- **Part 2**
  - `parse_headers.c`
  - `Makefile`
  - `README`

Your `README` file should contain a short header containing your name, username, and the assignment title. The `README` should additionally contain a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. Additionally, if there are any short answer questions in this lab write-up, you should provide well marked answers in the `README`, as well as indicate that you've completed any of the extra credit (so that I don't forget to grade it).

Submission Instructions

Submission will occur via `handin43`. You can retrieve relevant files via `update43`.

1 C Programming Primer

In Part 1 of this lab, you will complete a sequence of short programs to review some basic C programming skills. Note that C is a subset of the C++ language; however, there are subtle difference that can trip you up. In this lab, I expect you to only include C's `stdlib.h`, `stdio.h`, and `string.h` and no more unless otherwise specified in sample code provided. A quick note about strings in C, there are none. Strings are represented as `char` arrays, where the last character is NULL (`'\0'`).

1.1 Data Types and Memory Sizes

1. Write a short program, `data_sizes.c`, that when executed will print the data width's of each the basic C data types: `int`, `short`, `long`, `float`, `double`, `char`, and the pointer `void *`. You will find the built-in macro `sizeof()` to be particularly useful to complete this task. As an example of what I'm looking for: The data width of a `char` is 1 byte.

2. Add the following structured data type to `data_sizes.c`:

```
struct record{

  char first_name[20];
  char last_name[20];
  char middle_initial;

  long account_num;
  short account_type;
  double account_balance;
};
```

   Have your code print the size of this data type when `data_sizes.c` is executed. Additionally, declare a data type of this structure on the stack and propagate it with your name and (made-up) numbers for the account (you'll find `strcpy()` like functions useful for this task). Print out the record information as well when the program executes. (See below for an example of a formatted print).

3. Embedded in the string below is the data for a `struct record`:

```
char data_record[] = "\x41\x64\x61\x6d\x00\x7f\x00\x00\x30"
                     "\x57\xf1\x6c\xff\x7f\x00\x00\x00\x00"
                     "\x00\x00\x41\x76\x69\x76\x00\x4b\xf1"
                     "\x6c\xff\x7f\x00\x00\x85\xb8\x7e\xc5"
                     "\xc3\x91\x98\xa0\x4a\x00\x00\x00\x00"
                     "\x00\x00\x00\x32\x79\x06\x00\x00\x00"
                     "\x00\x00\x0a\x00\x00\x00\x00\x00\x00"
                     "\x00\x8d\x97\x6e\x12\x83\x00\xa0\x40";
```

   Your task is to extract that data and print out the information below:

```
Name: Adam J. Aviv
Acc#: 424242
Acc Type: 10
Balance: 2048.256000
```

4. **Extra Credit (4 pts)**: Submit your own data embeded string and a formatted print in your code.

Questions: Answer the following questions in your `README`.

1. What is the bit-width of your CPU? 32bit or 64bit? How can you tell from the output of this program?
2. What is the largest signed `short`, `int`, and `long`? What is the largest unsigned values?
3. Explain casting in C? Does it affect the underlying data or just the interpretation of that data?
4. Hexidecimal notation (or hex) is incredibly useful tool for programmers, and you've seen an example of that above where I embedded data into a string using hex. Why do you think we use hex so readily? How much information can two hex digits store? How much information can one byte store? What is 1024 in hex? What is 1025 in hex?

1.2 Bit-Wise Operations

To solve the problems below, you should familiarize yourself with the standard bitwise operators:

- `a & b`, **AND**: Take the bit-wise AND of a and b
- `a | b`, **OR**: Take the bit-wise OR of a and b
- `a ^ b`, **XOR** operation: Take the bit-wise XOR of a and b
- `a << b`, **left shift**: Shift the bits of a to the left by b bits and pad extra bits to the right with 0. For example, $1 << 3$ is 8, or $1 = 1_b$ in binary and shifted to the left by 3 gives $8 = 1000_b$ in binary.
- `a >> b`, **right shift**: Shift the bits of a to the right by b bits and pad extra bits to the left with 0. For example, $16 << 2$ is 4, or $16 = 10000_b$ in binary and when shifted to the right by two $4 = 100_b$.

Program solutions to the following problems and put them in a new file named `bit_wise.c`. In the `main()` function of your program, you should provide code to access and test these function, verifying your solutions. Feel free to include all test code you used when programming.

1. A bit-mask is used to extract information embedded at the bit level. For example, consider the embedding of bit-wise flags within a byte. There are 8 flags that can be set, one for each bit, and suppose we are only interested in the flag in the least significant bit. We can describe a mask of 0x01 in hex (or 00000001 in binary) to extract that bit in the following way.

```
char flags = 0xFB; // 11111011b
char mask  = 0x01; // 00000001b

if(flags & mask){
  printf("Flag 1 set!\n");
}
```

   Now consider the TCP flags embedded within a TCP header (we'll discuss the meanings of the flags later in the semester). There are 9 TCP header flags, and you can consider them stored within the data-width of a `short`. Write a function `tcp_flags()` that given a `short` representing the flags, will print out which flags are set. The TCP flag layout is as follows:

```
bit-offset   0  < --- >  6   7   8   9   10  11  12  13  14  15
           +---------------+---+---+---+---+---+---+---+---+---+
           |               | N | C | E | U | A | P | R | S | F |
values     |     unused    | S | W | C | R | C | S | S | Y | I |
           |               |   | R | E | G | k | H | T | N | N |
           +---------------+---+---+---+---+---+---+---+---+---+
```

Do not worry about what each flag means. Your program can just print the short-hands for each flag set. For example, a call to your function like this `tcp_flags(0x01A1)` should provide the following formatted output:

```
Flags: NS, CWR, URG, FIN
```

2. Write a function `print_bits()` that takes a char and prints a sequence of 1's and 0's representing the char in terms of its bits. Below is a function outline to work from:

```
void print_bits(char c){
  int i;
  for(i = 0; i < sizeof(char)*8; i++){ // 8 bits per byte

    if ( .... ){ /*i^th bit is 1, print 1, else print 0*/
      printf("1");
    }else{
      printf("0");
    }
  }
}
```

3. **Extra Credit (3 pts):** Change your function above such that it can take in arbitrary data and print the bits of that data. This function should have the following definition:

```
void print_bits_EC(unsigned char * ptr, size_t len);
```

4. Consider a `short` which contains two different values in the first (most significant) byte and the second (least significant) byte. Write a program that will print the (unsigned) values of each part of the short. Below is a function outline to work from:

```
void split_short(unsigned short s){
  unsigned short first,second;

  //extract first and second byte from s

  printf("first: %u\n", first);
  printf("second: %u\n", second);
}
```

5. Most modern computers encode data in Little Endian format, where the most significant bits appear on the left in the data encoding, such as the encoding of 13 in binary $13 = 1101_b$. An equivalent encoding is where the most siginficant bits apear on the right. This is called Big Endian notation; for example, 13 is encoded as $13 = 1011_b$ in Big Endian.

   Due to legacy issues, all network packet are encoded in byte-wise Big Endian (also called Network Order). Within each byte, the ordering is Little Endian, but sequences of bytes are encoded such that the must significant byte occurs on the right.

This causes many headaches for network programmers, but the byte order can easily be reorder using bit-wise operators. Add a function to your program that given a `short` will swap the first and second byte of the short, printing it in both formats. Here is a function outline to work from:

```
void swap_bytes(unsigned short s){
  printf("Lil'Endian: %u", s);

  //swap the bytes

  printf("Big Endian: %u", s);
}
```

6. **Extra Credit (3pts)**: Write a function that will reverse the byte order of an arbitrary data type. Here is a function definition to start from:

```
unsigned char * reverse_bytes(unsigned char *, size_t len);
```

2 Packet Parsing

In this part of the lab, you will be provided with a network packet capture, and you must parse the Ethernet, Internet, and TCP header information from the packets and print out requisite information as described below. The packet capture was done using the `tcpdump` program and is accessed by `libpcap`. The point of this lab is not to learn `libpcap`, but rather parse the packet header information. So: I have provided you with some skeleton code to get started.

This code makes use of `libpcap`, the packet capturing libraray, and you will need to tell the compiler to dynamically link the library during compilation. If you are using `gcc` (which I highly recommend), then this is easily accomplished using the `-l` flag. For example, here is a standard compilation command for your program:

```
bash> gcc -g -lpcap parse_headers.c -o parse_headers
```

If `libpcap` is install, then it should be automatically linked against your compiled binary at execution time by the dynamic linker.

In addition to the skeleton code, I have also provide two traces for you to use to test your program: `capture1.pcap` and `capture2.pcap`. As an example of the expected output, here is some sample output from parsing `capture2.pcap`:

```
(...)
-------- Packet 16 ------------
Size: 66 bytes
MAC src: 00:25:90:26:be:da
MAC dest: 7c:c3:a1:89:31:e8
IP src: 130.58.68.137
IP dest: 130.58.68.200
Src port: 80
Dst port: 63770
-------- Packet 17 ------------
Size: 1514 bytes
MAC src: 00:25:90:26:be:da
MAC dest: 7c:c3:a1:89:31:e8
IP src: 130.58.68.137
IP dest: 130.58.68.200
Src port: 80
Dst port: 63770
-------- Packet 18 ------------
Size: 1514 bytes
MAC src: 00:25:90:26:be:da
MAC dest: 7c:c3:a1:89:31:e8
IP src: 130.58.68.137
IP dest: 130.58.68.200
Src port: 80
Dst port: 63770
(...)
```

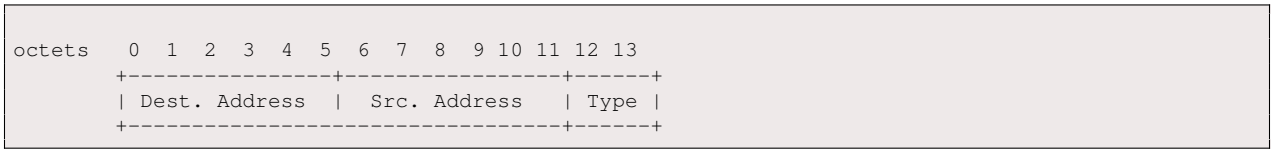
You are not required to match this format precisely, but all this information must be present.

2.1 Packet Layout and Offsets

As discussed in class, a network packet is really the encaspulation of multiple headers for different protocols being use. In the captures provided to you, the ethernet, internet, and TCP headers are encapsulated in a single packet. The outermost header is the ethernet header, whose payload defines the start of the internet header, whose payload defines the start of the TCP header, whose payload is the payload. Below, the information and offsets of each of the headers is provided.

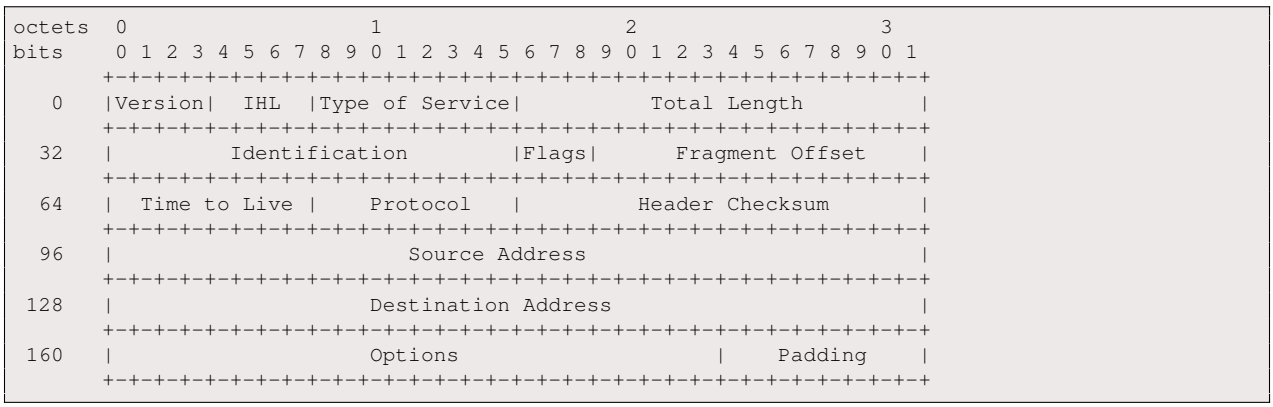
Ethernet Header

Note that the link-layer addresses are MAC addresses, not IP addresses, which occur in the Internet packet (i.e., in the network layer).



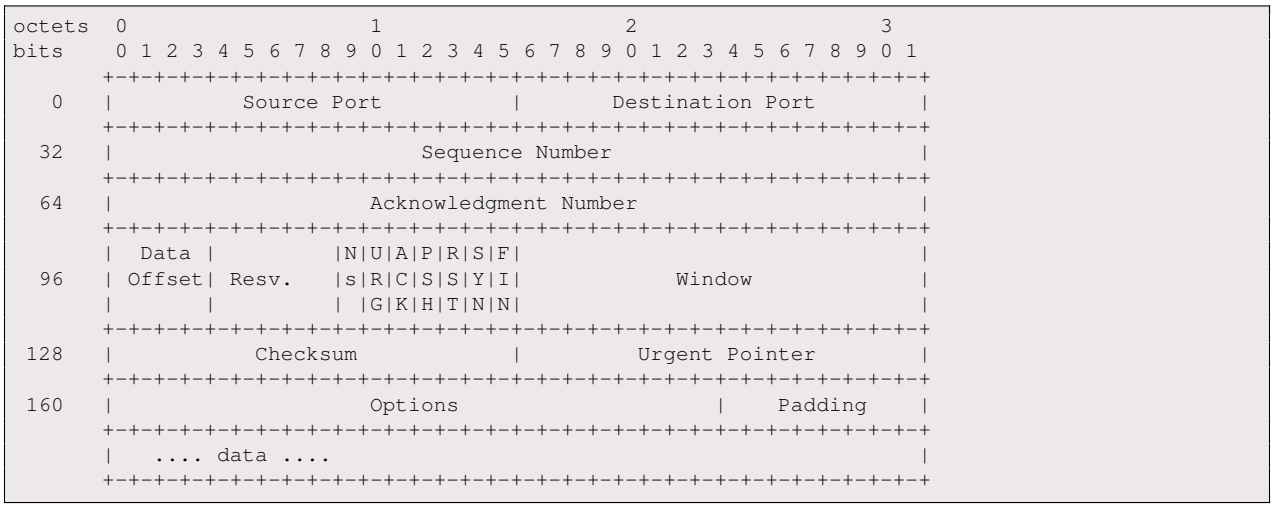
Internet Header

Below is the IPv4 header. Note that the length of the header is defined by IHL, which describes how many data words are in the header. Each word is 4 bytes longs, and if IHL is 5, then we expect the header to be 160 bits (or 20 bytes) Thus, the "Options" and "Paddding" field may not be set and are optional.



TCP Header

Similar to the Intenet header, the TCP header can be variable length. The length of the header is defined in the data offset field, and again is described in data word units (or 4 byte units). Normally, the data payload offset is set to 5, or the data begins at the 160th bit. Again, anything following the 160th bit is optional.



2.2 Extra Credit (6 pt): Verify the Internet and TCP checksum

Review the Internet checksum in Kurose and Ross, and implement a routine that will verify both the TCP and Internet header check-sums.

2.3 Extra Credit (2 pt): Collect your own trace

Review the man page for `tcpdump` (or install it on your personal machine). Use `tcpdump` to capture a trace of you visiting the `www.google.com` and only visiting `www.google.com`. You must use a filter to do this, and be sure to capture traffic flowing in both directions, from your computer to google and vice versa. Submit the capture trace file as well as the `tcpdump` command you use to capture the trace.

2.4 Extra Credit (2 pt): Analyzing the payload

In `capture2.pcap`, I was performing a specific task when I captured that trace. Analyze the payloads of the packet to reveal what I was doing and describe your process in the `README`. If you do not describe the process, you will receive zero credit.