**ECE 2220: System Programming Concepts** **Fall 2020**
**Machine Problem 5** **Due: 11:59 pm, Monday, November 09**

The goal of this machine problem is to learn to use function pointers, structure pointers, passing structures and pointers to structures to functions, dynamic memory, passing binary data between programs using Shell I/O, and use of the library function **`qsort()`** with function pointers and casts. In this lab, each student is to complete two programs called **`snort6.c`** and **`lab6.c`**. Initial versions of both programs are provided. The first program, **`snort6.c`**, simulates the behavior of the *snort* program to process intercepted packets and produce a collection of records with key information from the packets. The output of this program is used as input to **`lab6.c`**. The second program takes as input from **`stdin`** a collection on binary data using a specified structure and sorts the data on one of the fields in the structure. An additional option determines if there is a port scan attack. Finally, the binary data is printed as ASCII text to **`stdout`**.

# Input

These programs are to use Shell I/O redirection techniques. The **`snort6.c`** program generates a list of **`Records`** as binary output, which it sends to **`stdout`**. The first 4 bytes it produces represent the number of **`Records`** produced, and the remaining bytes contain the details of each **`Record`**, sequentially.

```
 struct Record {
    int seqnum;
    float threat;
    unsigned int addrs[2];
    unsigned short int ports[2];
    char dns_name[8];
};
```

where the values for each must be initialized as given below:

- **`seqnum`** – a unique value for each record (this is provided and should not be changed)
- **`threat`** – A floating point number with a random value $x$, such that $0.0 \le x < 1000.0$
- **`addrs[2]`** - An array of two integers, each with a random value selected with a uniform distribution as described next. The first address, $x$, is the destination IP address and is randomly selected from the range **`DEST_IP_MIN`** $\le x \le$ **`DEST_IP_MAX`**. The second address is the source IP address from the range **`SRC_IP_MIN`** $\le x \le$ **`SRC_IP_MAX`**. However, the source IP address CANNOT be any address in the range of destination IP addresses (i.e., the source is not on the same network as the destination). The constants are defined in **`snort6.c`**, and notice there are a small number of possible destination IP addresses and a very large range of source IP addresses.
- **`ports[2]`** - An array of two short integers representing the destination and source port numbers, respectively. Each port number, $x$, is a random value in the range $0 \le x <$ **`NUMBER_PORTS`**. The two port numbers could be equal to each other.
- **`dns_name[8]`** - A random number (2 to 7) of random characters that are lower case letters (‘**`a`**’ to ‘**`z`**’)

The **`seqnum`**’s have already been implemented in the provided template for **`snort6.c`**. You must complete the code in **`snort6.c`** to create the randomized data for the other fields using the **`drand48()`** library function. Additional details about **`drand48()`** are given in the notes section below. The **`snort6.c`** program takes three command line arguments: the first is the number of records to

generate, the second indicates if a port scan attack should be created (0 means no, 1 means yes), and the third is the seed for the random number generator. The third argument, the seed, is optional, and if not provided **snort6.c** will use the system time to seed the random number generator. Note that most of the **snort6.c** code is complete including creating the port scan attack; you just fill in the details for generating the randomized data.

The **lab6.c** program gathers input from **stdin** as binary data. The data is redirected from a file or by piping directly from the **snort6.c** program. You should assume that there are no errors in the input stream for simplicity. The provided **lab6.c** template program gathers the data from **stdin** and dynamically allocates an array of structures using the form **struct Record** shown above. The template also collects and processes additional input from the command line. The output from this program uses both **stdout** and **stderr**; the resulting sorted data is converted in ASCII and sent to **stdout** (which can be redirected to a file) while the time required to sort the data is sent to **stderr** (which is displayed on the terminal).

The **lab6.c** program accepts command line arguments for two different modes. For Mode I, there are two command line arguments. The 1st argument specifies the field in the structure that should be used for sorting, which should be one of the four values:

1 – Sort the structures by **seqnum**.
2 – Sort the structures by **threat**.
3 – Sort the structures by **dns_name**.
4 – Sort the structures by **address**.

The second argument specifies if bubble sort should be used (if the value is **2**) instead of the default sorting method of **qsort** (if the value is **1**). If the type of sort is **0**, then the data is not sorted but just printed.

For Mode II, there is only one command line argument, and its value must be equal to 5. In this mode, perform a port scan attack analysis (as described below). The type of sort is not relevant with this option so there is only one argument.

After collecting all the input, if the program is directed to sort the data, then use the **clock()** command to measure the time required to sort the data, and print the result (in milliseconds) to **stderr**. In addition, print all of the data to **stdout** in text format instead of binary format. Examples of how to use **clock()** and print the output are provided in the **lab6.c** template. While the **lab6.c** template file has a few examples of parts of the code, you must develop most of the details.

The program must have the following supporting functions which **qsort()**, bubble sort, and a validation function will use. The functions specify how to compare two records.

- **SeqNumCompare()** compare two structures by their seqnum numbers. This function is provided.
- **ThreatCompare()** compare two structures by their threats.
- **AddrCompare()** compare two structures by their IP addresses. First consider the destination IP address in **addrs[0]**. If the destination addresses of the two structures are equal, then sort by the source IP address in **addrs[1]**.
- **DNSNameCompare()** compare two structures by their DNS names

It is **critical** that when you print the records at the end of **lab6.c** that you have **not** changed the values in any of the records.

You code must be able to handle all combinations of redirection, such as

Generate 5 binary records and simply print them as ASCII characters. The records are not sorted:

```
./snort6 5 0 321 | ./lab6 1 0
```

Generate 10 records and store them in a file. The file contains binary data

```
./snort6 10 0 123 > file.data
```

Using the binary data in **file.data**, sort the **threat** field with qsort. Store the records in a file.

```
./lab6 2 1 < file.data > file.txt
```

Generate 10,000 records. Use bubble sort on the DNS name field and store in a file

```
./snort6 10000 0 231 | ./lab6 3 2 > file.txt
```

Generate 1,000,000 records. Use qsort on the IP addresses and discard the standard output

```
./snort6 1000000 0 312 | ./lab6 4 1 > /dev/null
```

Generate 1,000,000 records with a port scan attack. Perform a ports scan analysis and print details about each attack

```
./snort6 1000000 1 | ./lab6 5 > /dev/null
```

Your bubble sort implementation should be able to sort about 10,000 records in about 1 second, while **qsort** should be able to handle lists at least 100 times larger! Notice that the output for very large lists should be directed to **/dev/null**. This tells the system to delete the output data instead of wasting disk space storing the file.

For bubble sort, just like **qsort**, you must have a single function that accepts a function pointer to determine the field to use in sorting. Your prototype of bubble sort must be:

```
void bubblesort(struct Record *ptr, int records,
      int (*fcomp)(const void *, const void *));
```

where **ptr** is the pointer to the array of Record structures, **records** is the count of the number of records in the array, and **fcomp** is the function to compare records. For an additional example see the validate function.

After each call to qsort or bubblesort, you code **must** call the validate function. This function is used during grading and verifies that the list is in sorted order and contains all of the elements. The function is provided and you just need make sure it is called after the completion of any sort.

# Performance Evaluation

For each of the 2 sorting algorithms and each of the 4 field values, determine the largest list size that can be sorted for each option in less than one second. In the comments at the top of your **lab6.c** file, you must have a table showing this list size for each of the 8 combinations.

# Port Scan Attack

A port scan attack is said to occur when there is a set of packets from the same source IP address that are sent to one destination IP address, and the packets cover at least 16 consecutive destination port numbers (with no gaps). The block of port numbers can start at any value, but a block cannot wrap from the largest port number back to zero. The packets for an attack do not have to arrive in order (and could arrive in any random pattern). It is possible that there is more than one block of attacks from a particular source to one particular destination IP address, or that a single destination is under attack by more than one source. Also, more than one destination IP address may be subject to an attack. You code must print out all destination addresses that are determined to be under attack. For an attack print the destination IP

address, the source IP address associated with the attack, the first (or lowest) destination port number that starts a block, and the size of the block. If there are no attacks then print a message that no attack is detected. You must use the **`printf()`** format provided in the **`lab6.c`** template.

# Notes

1. The **`drand48()`** function.

Note, the random number generator `rand()` has very poor statistical properties and should never be used. Instead use `drand48()` and `srand48()`, which are included in the C standard library `stdlib.h`.

`srand48`(*x*) initializes (or *seeds*) the sequence of numbers, where *x* is a 32-bit integer. This function should be called one time at the start of the program. Setting the seed ensures that the same sequence of numbers will be produced by `drand48()` each time the program is run. To get a different sequence of numbers change the seed.

`drand48`(void) returns a double in the range [0.0, 1.0 ), and has the property that the number is uniformly distributed over the range. To generate a number in the range [0.0, 1000.0) use:

```
float f = 1000 * drand48();
```

to create a integer in the range {0, 1, 2, …, 9} just take advantage of type conversion:

```
int i = 10 * drand48();
```

2. The **`clock()`** function

To measure the performance of a sorting algorithm use the built in C function `clock` to count the number of cycles used by the program.

```
#include <time.h>
clock_t start, end;
double elapse_time;  /* time in milliseconds */

start = clock();
// call qsort or bubble sort here
end = clock();
elapse_time =  1000.0 * ((double) (end - start)) / CLOCKS_PER_SEC;
```

where `CLOCKS_PER_SEC` and `clock_t` are defined in `<time.h>`.

3. Submission requirements

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. To receive credit for this assignment your code must correctly sort the input using qsort for all four field values. Code that does not compile or fails to pass the minimum tests will not be accepted or graded.

4. Put your source files **`snort6.c`** and **`lab6.c`** in a folder called **`lab5`**. Compress folder **`lab5`** to **`lab5.zip`**. Submit your **`lab5.zip`** to Canvas.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments. Work must be completed by each individual student. See the course syllabus for additional policies.