

REMUSNET  
Norrec Nieh  
CS5008 Capstone Proposal

## Introduction

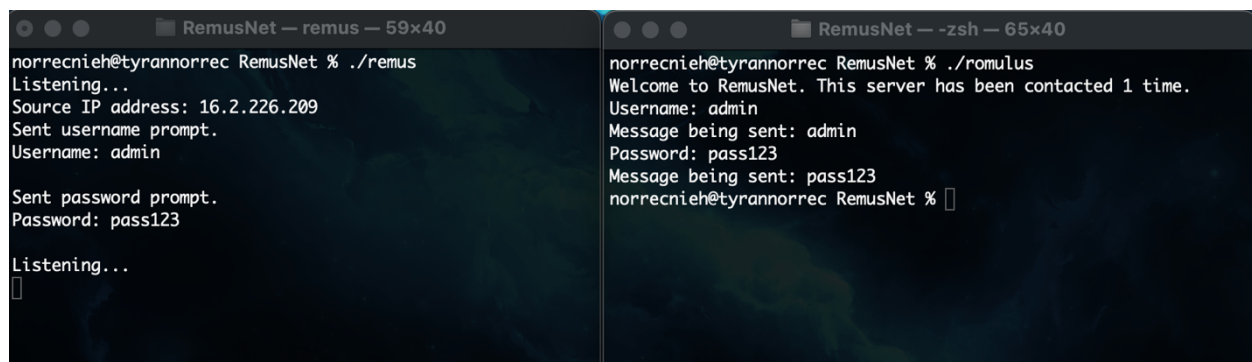
For my CS5008 capstone, I will be designing and implementing a rudimentary framework for a honeypot. Placed at the edge of a network, a honeypot is a cybersecurity system that theoretically diverts attacks by malicious actors from functioning servers and hosts by simulating a vulnerable server within a controlled environment. Honeypots are used as research labs to collect data on patterns of attack against particular protocols or entire systems, depending on the scope and use case of the honeypot. As opposed to a “high-interaction” honeypot, which sets up a real system for attack, I will be implementing a “low-interaction” honeypot, which typically emulates common attack vectors such as the SSH (port 22), Telnet (port 23) or FTP (port 21) protocols. Due to the language and time constraints of the capstone, however, I will forgo emulating one of these application-layer protocols and will focus instead on a simplistic emulation of communication between a server and client. Due to the mirrored relationship between the two programs, I have chosen Remus and Romulus as names for my server and client, after the twins central to Rome’s foundation myth. Using these two programs, I will approximate attempts to penetrate a server, and collect information on the attack via the honeypot, including usernames and passwords used in the “attack,” as well as the perpetrator’s source IP address.

## Description

Implementation will be divided into three main programs: the server, client, and log parser.

### 1) **Server** (remus.c)

The “server” program represents the actual honeypot. Listening on port 8080, the honeypot will accept any incoming connections and feign a password authentication process in plain text. Meanwhile, it will log essential details about the session, as well as the combinations of usernames and passwords deployed in the attack. Ultimately, the authentication process will timeout and fail, and any information gathered during the session will be written to a master log file, which will be read in a separate log parser program.



The image shows two terminal windows side-by-side. The left window, titled 'RemusNet — remus — 59x40', shows the server program running. It displays the following text: 'norrecnieh@tyrannorrec RemusNet % ./remus', 'Listening...', 'Source IP address: 16.2.226.209', 'Sent username prompt.', 'Username: admin', 'Sent password prompt.', 'Password: pass123', and 'Listening...'. The right window, titled 'RemusNet — -zsh — 65x40', shows the client program running. It displays the following text: 'norrecnieh@tyrannorrec RemusNet % ./romulus', 'Welcome to RemusNet. This server has been contacted 1 time.', 'Username: admin', 'Message being sent: admin', 'Password: pass123', 'Message being sent: pass123', and 'norrecnieh@tyrannorrec RemusNet % '.

## 2) **Client** (romulus.c)

The “client” program will act as the means through which attackers can engage with the server. It will be written to respond to the server’s requests for password authentication by prompting the user for input via the terminal. The authentication process will continue until the server terminates the connection.

## 3) **Log Parser** (logParser.c)

To read and parse the log file, I will be implementing a third C program; this program will make use of an ADT to process, filter, and output data sets based on several menu options, making pertinent use of binary search and an  $n \log n$  sorting algorithm to expose patterns and statistics in the information gathered.

```

-----
1 -- FIND ENTRIES BY SOURCE IP
2 -- SORT ENTRIES BY SOURCE IP
3 -- SORT USERNAMES BY NUM OF OCCURRENCES
4 -- SORT PASSWORDS BY NUM OF OCCURRENCES
5 -- PRINT USERNAMES IN LEXICOGRAPHICAL ORDER
6 -- PRINT PASSWORDS IN LEXICOGRAPHICAL ORDER
7 -- QUIT
-----
Enter choice (1-6): █

```

### - ADTs

- Entries in the log will be represented by an Entry struct, which stores the source IP, time of session, and the username/password combinations as char arrays.
- I will be using a nested Linked List to store Entries of the log in the heap.
- The ADT will be adapted from our implementation of the HashMap.
- It will include a populate() function, as well as all of our standard methods.
- Entries will be placed in order by IP address by way of insertFront() and insertRear().
- The find() method will implement binary search.

### - Sorting Algorithm

- For menu options 2 to 5, I will use different versions of mergesort to sort the char arrays representing source IPs, usernames, and passwords. Option 2 could be used in the future to parse IP addresses by region if used with a geolocation API. Options 3 and 4 will extract the usernames and passwords from the linked list and sort them by number of occurrences. Options 5 and 6 will operate similarly but sort the usernames and passwords lexicographically. These sorts will be relatively similar in implementation and have  $n \log n$  complexity.
- I plan to use the clock() function to time each menu option.

### - Binary Search

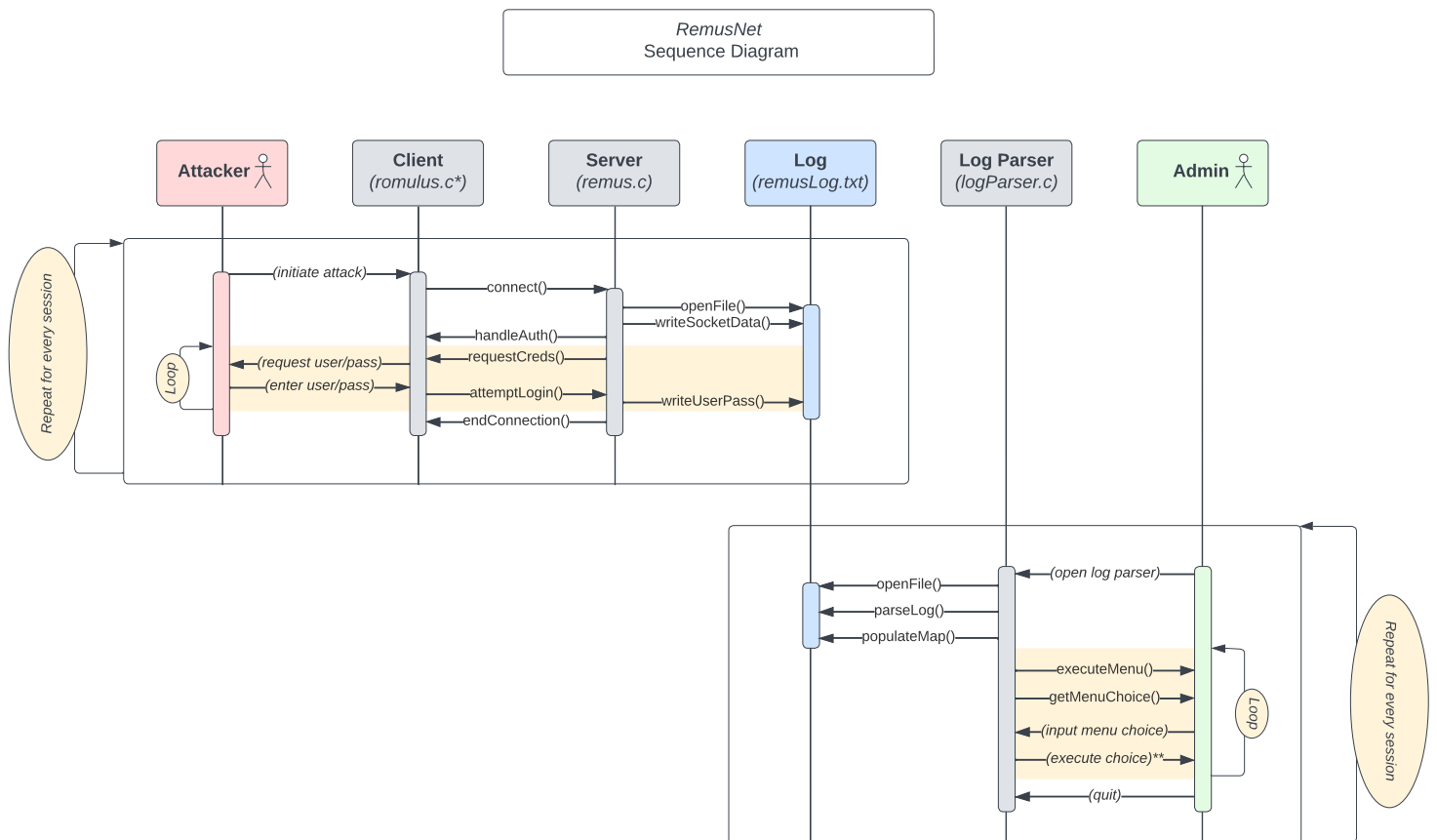
- Menu option 1 will use binary search via the find() method to find the correct IP address on a sorted Linked List. This will also have an  $n \log n$  complexity.

## System Architecture

The server will listen for incoming connections on an unending loop; when a client requests a connection, the server will use the `handleAuth()` function to initiate a loop, wherein it continually requests for a user's username and password via `requestCreds()` until a timeout occurs. The server will continually log these entries into the log file by way of the `writeUserPass()` function.

In the log parser, most of the code is contained in the Linked List ADT, as we have seen in previous implementations. Functions such as `populate()`, `insertFront()`, `insertRear()`, and `find()` take up the brunt of the work. However, for some menu options, the log parser file will alternatively extract the username and password by traversing the linked list, placing the char arrays in a larger array, and then calling a `mergeSort()` function to sort the char arrays. An additional sub-function, `stringCompare()`, will need to be written and customized to determine which of two strings is "less than" or "greater than" the other.

This sequence diagram depicts the way the different elements of the project connect.



\* For the purpose of simulation, our own client will be used by the "attacker".

\*\* Menu choices expressed here as a single step for simplicity's sake. See the following diagrams for details.

## Conclusion

In summary, this project is a framework for a honeypot meant to be built upon. Fundamentally, it is a means by which I aim to acquire a working knowledge of programming for networks by implementing a basic server and client to emulate a honeypot system, while also implementing a menu by which data collected via the honeypot can be parsed in a few different ways. While the server/client pair acts as a generator of raw data, the log parser program will implement binary search, sorting algorithms, and a nested linked list to manipulate that data. It will offer six different options to sort and print subsets of that data. Ultimately, the ambition would be to make this system viable to be placed on a small network, in order to generate some small insight on patterns and modalities seen in password attacks. It may also provide some protection and/or deterrence against less seasoned hackers when placed within a functioning network. However, the core desire is to simulate such a task.

In the future, the project can be improved upon by implementing an application-layer protocol such as FTP on top of the socket connection used in my server and client. The server can then be hosted on a cloud service such as Amazon AWS to retrieve some real-world data. I would expect to see more traffic there than if hosting on my own network, since the range of IP addresses used by AWS is widely known and commonly targeted by attackers. Additionally, I can make use of a Geolocation API and visualize the regions covered by the source IP addresses using a heat map. Another idea would be to code up a Python script to automate a hacking attempt.