REMUSNET
Norrec Nieh
CS5008 Final Project

<< USER MANUAL >>

==========

Introduction

RemusNet is a cybersecurity system which simulates a low-interaction honeypot. Typically, 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 of the honeypot. This system emulates the process by which such collection could occur, by having a server (remus.c) request credentials from the client (romulus.c) repeatedly and subsequently print those credentials to the terminal on the server side. Additionally, the system includes a log parser program (parserMenu.c), which contains options to manipulate and display the data (see descriptions below). Initially, the design of the server included the capability to log credentials sent by the client directly into a text file, which could then be read by the parser program, but issues with the C language's internal memory management have precipitated the shelving of that functionality. Instead, a log file (remusLog.txt) has been provided to show the functions of the log parser application, which utilizes the following coding elements:

1. Menu
2. ADTs – Nested Linked List; Arrays of Structs
3. File access – Reading / Populating
4. Sorting algorithms – Quicksort and Mergesort
5. Binary search

==========

Description

In the `code` directory you will find two sub-directories – `parser` and `server_client`. `parser` includes the files necessary to run the log parser application, while `server_client` contains the server and client programs.

The `server_client` directory contains FOUR files:

1) **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 print essential details about the session, as well as the combinations of usernames and passwords deployed in the attack, on the server terminal.

2) **remus.h** -- The header / config file for the server program.

3) **romulus.c** -- The "client" program will act as the means through which theoretical 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 (after three iterations).

4) **romulus.h** -- The header / config file for the client program.

The `parser` directory contains FOUR files:

1) **parser.c** -- The implementation file for the log parser. Includes most of the code for this program, including functions to manipulate the structs and linked lists, as well as those to manipulate and compare IP address strings and username/password strings.

2) **parser.h** -- The header file for the parser program. Defines the methods, structs, and types required to run parserMenu.c, as well as the arrays used to hold the nested structs.

3) **parserMenu.c** -- The application file for the log parser. Implements a menu through which users can select options to variously parse the data from a specially formatted log file (see parser menu options for details).

4) **remusLog.txt** -- The specially formatted sample text file containing data to be read and parsed by parserMenu.c.

==========

parserMenu.c Options

1) POPULATE SESSION ARRAY

Reads in `remusLog.txt` and parses it for data on every session; each session is represented as a session_t object with an attached linked list of creds node, detailing the credentials used within each session. All sessions are stored in sessionArray in order of their ID. This menu option also initializes userArray and passArray, two lexicographically ordered arrays holding all the unique usernames and passwords seen in the sessionArray, as well as their number of occurrences. Two other arrays, userArrayOcc and passArrayOcc, are also initialized with the same usernames and passwords, mergesorted by num of occurrences for ***O(n log n)*** complexity.

2) FIND ENTRIES BY SOURCE IP ADDRESS

Deploys binary search on the array of sessions to find one with a specific ID address. The array is sorted implicitly using quicksort, for a total of ***O(n log n)*** complexity.

3) FIND ENTRY BY ID

Finds a session with a specific ID by employing linear search, for ***O(n)*** complexity.

4) SORT ENTRIES BY SOURCE IP ADDRESS

Deploys quicksort to sort sessionArray by non-decreasing IP address values. Methods from arpa/inet.h are used to check if the input string holds a valid IP address. Then, IP addresses are compared by way of a method in parser.c that splits the string by periods and compares the integer values of each octet. Since we are calling quicksort on an array of sessions, and each comparison is constant time, the average time complexity comes out to ***O(n log n)***.

5) SORT ENTRIES BY ID

Deploys another version of quicksort to sort the array of session_t objects by their ID attribute. The time complexity of this sort is also ***(n log n)***.

6) SORT USERNAMES BY NUM OF OCCURRENCES

The sort is already implicitly done in menu option 1, upon initialization of the arrays. This option prints the contents of userArrayOcc in order, for ***O(n)***.

7) SORT PASSWORDS BY NUM OF OCCURRENCES

The sort is already implicitly done in menu option 1, upon initialization of the arrays. This option prints the contents of passArrayOcc in order, for ***O(n)***.

8) SORT USERNAMES BY LEXICOGRAPHICAL ORDER

The sort is already implicitly done in menu option 1, upon initialization of the arrays. This option prints the contents of userArray in order, for ***O(n)***.

9) SORT PASSWORDS BY LEXICOGRAPHICAL ORDER

The sort is already implicitly done in menu option 1, upon initialization of the arrays. This option prints the contents of passArray in order, for ***O(n)***.

10) DELETE ALL SESSIONS

All arrays used to store data are reset.

11) QUIT

Terminates the program.

==========

==========

Conclusion

In summary, this system is a framework for a honeypot meant to be built upon. Fundamentally, it is a means by which I aimed 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 intention of the project was for the server to write the credentials it received to a text file, which could then be parsed by the log parser, the current iteration relies on the data being collected from the server's terminal and manually transferred over to a log. This represents a weakness in the current implementation and is something to be improved upon in a future iteration. Meanwhile, the log parser program successfully deploys binary search, sorting algorithms, and a few different structs and data types, such as a nested linked list, to manipulate the data theoretically acquired from the server-client interaction. It provides ten menu options to find, 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 was to simulate such a task.

==========