Password Management System

Yuqian Shu
19200491

COMP30640: Operating Systems

UCD School of Computer Science

Introduction

The password management system is a type of pf software which can be employed to store passwords for multiple users. In other words, by using the password management system, users can get rid of remembering various password.

The password management is a reliable tool which contains encrypt and decrypt functionalities and allows users to create, edit, update, remove, list and show information. Once a user folder created, the password system enables users to store and retrieve username and password of it.

The key functionalities of this Password Management System are:

1. Register new user
2. Insert new service of an existing user
3. Show the username and password of an existing service
4. Retrieve, list and remove an existing username and password
5. Encrypt and decrypt the username and password
6. Generate random string as the password
7. Access this management system concurrently by multiple users

Requirements

By using this password management system, each users' password will be stored at a folder called *$user.* Within each user folder, there can be multiple files named by the organization of the stored password.

The password management system has three main sections: basic commands, the server-side, the client-side.

<u>Basic commands:</u>

The basic commands allow users to perform different operations. Those basic commands can also be implemented via call *server.sh*.

The commands that this password management system have:

1. Create user folder: the user can create a new folder with a given name to store password and login strings
2. Create a new service: the user can create a new service in the user folder, this new service can be a single file or a file under a directory. When the user creates a new service, the strings that need to stored need to be given as well.
3. Show a service: this command allows users to see the stored information inside a specific file
4. Update a service: this command enables the user to update password and login strings
5. Remove a service: this command allows the user to delete information from the system
6. List services: this feature enables the user to see the tree structure of an existing file

Each of those commands that described above contains error handling, including too few or too many parameters, the user folder does not exist, the service file does not exist. Besides, they will also respond to the scenario that everything is going well.

<u>The server</u>

The server is a *.sh* file which needs to execute in the background, reading the input of the client-side, then executing the related request. The error handling of the server.sh is the same as the error handling of the basic commands. The server-side can process the following request: create user, insert service, show service, update service, remove service, list service and shutdown.

The server side is designed to be used by multiple users sending different requests at the same time. Moreover, the server-side is running in the background and allow events concurrency happened. To enable process can be run in the background, the *&* has been added at the end of each basic commands in the server.sh. Because concurrency has the risk that two users might overwrite the user file. To avoid this situation, P.sh and V.sh have been applied, which can create a file locker to ensure that there is only one process enters the critical section at the same time.

<u>The client</u>

The client is used to send a request to the server-side; each client needs to have a unique client identifier. Similar to the server-side, the error handling of client.sh is the same as basic commands. Because the client-side will send a request to the server-side. The client-side need to generate a pipe named by the client identifier and the server-side need to create a server pipe as well. By using pipes, the client requests can be read by the *server.sh*.

Architecture of the password management system

The system is designed as following section:

<u>Basic commands:</u>

The basic commands consist of *init.sh, insert.sh, ls.sh, show.sh, rm.sh.*

1. init.sh: This script is used to register a new user. We first check that the number of input parameters entered by the user is correct or not. Then, create a user folder with the name given by the second argument ($1). The error message or finish message will be print out if the user folder already exists or folder is created.

2. insert.sh: This script has four allowed arguments user ($1), service ($2), override ($3), payload ($4). It is used to insert a new service. Similar to init.sh, we first check the input arguments, the responding messages will be print out if the user folder is not existing or the service is already existing. Then create a file which is the service under the user folder

($3 is “”), the created file contains the information given by the fourth argument ($4), and the service can be updated if the third argument ($3) is f.

3. ls.sh: this script allows the user to check the service they have registered in this password manager. It has two arguments, user ($1) and folder ($2). The error message will be print out corresponding too many or too few arguments, and user or folder does not exist. After the check, the tree structure of the registered file or folder will be print out.

4. show.sh: This script has two arguments, user ($1) and service ($2). Similar to above, this script will check the number of arguments, user folder or file exists or not first and show the corresponding message. If everything corrects, the login and password stored in registered service will be printed out.

5. rm.sh: this script has two arguments, the user ($1) and service ($2), it will check the number of arguments; user folder or file exist or not first and show the corresponding message. If everything corrects, the registered service will be removed.

Server:

The server.sh contains an infinite loop which can constantly read inputs from *server.pipe*. In order to read the input, we cast it into an array which gives an index to each part. We assign a variable name to each index, the ClientID is $array[0], the command is $array[1], the user is $array[2], the service is $array[3], the login is $array[4], the password is $array[5] and the payload is ${array[@]:4} which means that the input part with index larger or equal than 4 all belong to payload.

The script will check the command is valid or not; if it is, server.sh will call the related basic commands. The server can recognize 7 commands including 4 basic commands, an update command which is used to re-write the service content and shutdown which will terminate the server. Other invalid commands will lead to an error message.

Client:

The client.sh assign the first input as ClientID ($1), the second input as request ($2), the rest of inputs as args (${@:3}). The script first checks if there are enough arguments (at least 2) entered, then check if the request is valid or not. The server can recognize 7 commands including 4 basic commands, and an edit command and a shutdown command. The edit command, which is used to retrieve the service payload, store it in a temporary file and let the users modify the payload, then sent an update request to the server with the new payload, the shutdown command will terminate the server. Other invalid commands will lead to an error message. The client.sh will be sent requests to the server.pipe and identifies each client with an ID when executing.

For the edit section in the client-side, our password management system can verify the valid information. For example, if a user types a bunch of string that is neither login nor password; then, our system will not store it.

Communication:

Because the server-side and the client-side require to communicate with each other, to achieve this goal, we use the pipe to transmit strings. Once initiated, server.sh will create a *server.pipe,* and the client.sh will create a pipe named as the ClientID. Then, the client.sh will send arguments to the *server.pipe*; the server.sh read from the s*erver.pipe,* response to the request and sent results to the *clientid.pipe.* Finally, the clinet.sh capture results from *clientid.pipe* and remove the *clientid.pipe*; the *server.pipe* will be removed as well once server.sh is completed.

Exit

There are two exit functions in this password management system, one in the server-side, another one in the client-side. For the server-side, once user hitting CTRL+C, the exit function will be executed, and the server.pipe will be deleted. For client side, once user hitting CTRL+C the exit function will be performed and the ClientID.pipe will be deleted.

Challenges and solutions

Print multiple line

When the string includes the newline character \n, *echo* cannot switch a newline.

This problem be solved by using *echo -e.*

Server.pipe: Interrupted system call

This error message is printed whenever I am running server.sh. It did not occur before I set the server-side can running in the background. However, after I add *&,* this error appeared.

To solve this, I add a *sleep 5* after *&.* However, our scripts will be exanimated on Anthony' s server, which allows programs to run in the background without *sleep*.

Read multiple lines from temporary file and send they to server.pipe

I used *read -r* first, however, it cannot read multiple lines, and it only read the first line.

To solve this, I used a while loop to read all the lines and save each line to an array. For example, the string of the first line is $array[0], the second line is $array[1]. Hence, by using ${array[@]}, all the lines can be sent to *server.pipe.*

Newline character missing when communication via pipe

In bash, the newline character is *\n* and it always missing when I transmit a string contains it via pipe. After test, I realized that the newline character is been recognized as a space. To solve that, I set the transmit string as an array which will not lose anything even if there is a space inside. Then, use sed command, change space to newline character.

Show a service which contains multiple passwords

It is hard to print out all passwords if the service stored multiple passwords when using a keyword to capture strings.

To solve that, I use a while loop to read lines from the service and capture each string with a delimiter : , then no matter how many passwords stored, they will be print out correctly.

Strengthen the security

To ensure user privacy, our system can encrypt login string and password string when user insert/edit a specific file. In order to protect user privacy, all the information that user stored in this system is encrypted. The decrypted string can only be shown via calling show request in the client side.

User password/login string consist space

In the original version of this system, some information may lose when there is a space contains inside the password or login.
To solve this problem, I put encrypt and decrypt on the client-side, which allow the space inside the string.

Encrypt letter limitation

While testing multiple scenarios that may occur when using this system, I recognized that the length limitation of the encrypted string is 32, which cannot be changed as Thomas gives the encrypt script. So, I assume that our user will not store an ultra-long string.

Conclusion

It is a challenging project, and you will not know what kind of bug will occur next time. However, it is not always painful; during this project, I became more familiar with Linux and bash. Moreover, it is much easier to break the whole project down into small sections. By debugging, I learned the idea of problem-solving and knew how to use google to find out the correct information.