

1. Introduction

Nowadays, people have a large number of passwords for different usages, such as social media sites and online banking. Although it is important to use different and strong passwords at each site, remembering all passwords can be a daunting task. With the password management system, people can effectively manage their account names and passwords.

A password management system is a software application or a hardware device for storing and managing a person's passwords. It is a powerful tool that allows users to insert, store, update and remove their login names and passwords at any time. In this report, the requirements of creating the basic password management system will first be discussed, then a detail design of the system will be expatiated. Finally, the challenges faced when doing the project and creating the password management system and some extra features created for the project will also be specified.

2. Requirements

This basic password management system is composed of three parts: basic commands written in different scripts, a server, and a client application. In the system, each user is represented by a \$user folder. Within each folder, there can be various service files with \$service name, representing different services that the user(s) want to save. Each service file contains a payload with \$payload, which composed of login and password of that service. Each service can have one payload at a time. Also, in each script, appropriate exit code will be given according the the condition: exit 1 is the exit code that shows something wrong happened and exit 0 is an exit code 0 means everything went well.

2.1 Basic commands

The basic commands will allow the user(s) to perform various actions and they all need to be implemented within the server. We must ensure that the exit message is the only thing printed in all scripts and all user folder, service file and payload can contain spaces.

The commands that the system needs to have includes:

- Init new user: The user will create a new folder with a given name to store a his/her passwords with this command.
- Insert new service: The user will create a new file in his/her own folder with a given name containing the login and password for a service with this command. In part 2.2 only three parameters are allowed and needed in this command, but it will be modified after part 2.4 and takes 4 parameters in total. If the third parameter is not f, then it has same behaviour, if it is f, its behaviour should change, details of this change will be introduced in the description of "Update a service" command.
- Show a service: The information stored for a service will be read with this command.
- Remove a service: The file corresponding to a service will be removed with this command.
- List services: The services that a user has registered a password for will be listed(can be structured in tree form) with this command. Both user folder and inner folder within user can

be accepted as parameters and all services inside the particular folder should be listed out when requested.

- Update a service: This command is in the same script with the “Insert” command. Whenever the user enter ‘f’ as the third parameter, this command will be implement and the user will be able to change the information stored for a service.

All of these basic commands have to implement proper and suffice error handling and the user must notified by an error message in the following cases:

- The input is not in the correct form (e.g. too few or too many parameters)
- The user folder that the user is trying to create already exists when implementing “Init new user” command
- The user folder that the user is trying to insert service into does not exist or the service that the user is trying to create already exists when implementing “Insert new service” command or “Update new service” (when the third parameter is not ‘f’)
- The user folder or service file does not exist when implementing “Show a service” or “Remove a service” command
- The user folder or inner folder containing service(s) inside user folder does not exist when implementing “List services” command

If the request is in the correct format and can be performed, the user will then be notified with a message that everything went well.

2.2 The server

The server will manage different requests and will be operated in the background . The server will need to read commands from the prompt and will execute them. It will be composed of an endless loop and every time the script enters the loop it reads a new command from the prompt(named pipes will be needed to get instructions, which will be mentioned later). Every request follows the structure req [args].

The server have to include proper error handling since every request must follow a specific format and an error message must be sent to the user if a request is in the wrong format.

The server will accept 7 types of requests:

- init: \$user: which creates a user
- insert: \$user \$service \$payload: which creates file \$service containing \$payload in \$user
- show: \$user \$service: which prints the payload of file \$service in \$user
- update: \$user \$service \$payload: which updates file \$service in \$user with \$payload
- rm: \$user \$service: which deletes file \$service in \$user
- ls: \$user [\$folder]: which prints the services for \$user [in \$folder]
- shutdown: the server exits with a return code of 0

As the server will be accessed by multiple users sending different requests concurrently, all commands will need to executed concurrently and run in the background.

In order to prevent potential inconsistencies(e.g. showing a payload while updating it at the same time) when users sending different requests, lock must be set to command scripts whenever the server try to access user \$user's folder or files.

2.3 The clients

The client script will be used with following syntax: `./client.sh $clientId $req [args]` and will send requests to the server. Each client will need a unique identifier(\$clientId). The client script must check if it received enough arguments (at least 2) and the request is in the correct format. If the request is correct it will be sent to the server.

The client take the following requests:

- init: check that a client id and user name were given and send an init request to the server
- insert: check that a client id, user name and service name were given, ask user to input login and password, and send init request to server
- show: check that a client id, a user name, and a service were given, send a show request to the server and print the result.
- ls: check that (at least) a client id and user name were given, send a show request to the server and print the result.
- edit: check that a client id, user name and service were given, retrieve the payload for the service, store it in a temporary file, open this file with a text editor (vim/nano) to let the user modify the payload, send an update request to the server with the new payload. (the temp file should be deleted tat some point).
- rm: check that a client id, user name and service were given and send a rm request to the server shutdown send a shutdown request to the server

The clients and the server will have to be connected with named pipes. The server will create a named pipe(server.pipe) and each client will create its own pipe(\$clientId.pipe). With this method, the clients and server can communicate effectively, the client is allowed to send requests to the server and the server to send the outcome of the request back to the client.

In client/server communication, it can be assumed the user name and service name will never contain a space due to complexity of parsing arguments sent through a pipe, although basic commands part still need to deal with this case.

A valid exit of client and server must also be implemented. Whenever user send a shutdown request, both client and server must exit.

3. Architecture/design

In order to address the requirements described in the previous section, the following solutions has been designed:

3.1 Creation of Basic Commands

First I created the scripts init.sh, insert.sh, show.sh, ls.sh and rm.sh

init.sh :

When I was creating this script, I first check if the number of parameters inputted by the user is correct (if not print error message and exit 1) then create a directory (which represents the user) with the name given by the user (\$user). After that, an message would be printed and a corresponding exit code would be returned if the user already exists (error message, exit 1) or the user is successfully created(exit 0).

Insert.sh¹:

Similarly, I first check if the number of parameters inputted by the user is correct and if user directory exists (if not print error message and exit 1), then if the service entered by the user contained in a folder, I create that folder(\$dir_name of \$service). After that, an message would be printed and a corresponding exit code would be returned if the service(\$basename of \$service) already exists (error message, exit 1) or the service and payload(\$payload) is successfully created(exit 0).

Show.sh:

In this script, I first check if the number of parameters inputted by the user is correct and if user directory and service file exists (if not print error message and exit 1). After that, I view the contents of the service using the cat command and exit 0.

ls.sh:

Again, the first step that I do is to check if the number of parameters inputted by the user is correct and if user directory exists(if not print error message and exit 1), then if user also enter a folder name(\$folder) and that folder doesn't exist, a error message will also be printed and exit 1. After the above checks, I will separate the outcome into two scenarios: If user doesn't enter a folder, an confirmation message would be printed and the content inside the user directory will be listed out in tree form (and exit 0); else if user does enter a exist folder, an confirmation message would be printed first, then we will navigate to the user directory and list out the content of the folder in tree form(and exit 0).

rm.sh:

Similar to all other scripts, I first check if the number of parameters inputted by the user is correct and if user directory and service exists (if not print error message and exit 1), then I will remove the service entered as the user wish and print out an confirmation message(exit 0).

Then I update the insert.sh script to add an update function in it.

Modified version of insert.sh:

I first check if the number of parameters inputted by the user is correct(it now takes 4 parameters instead of 3 after modification) and if user directory exists (if not print error message and exit 1). then if the service entered by the user contained in a folder, I create that folder(\$dir_name of \$service).

¹The code of insert.sh will be changed when implementing the update part, but in this moment I just state the original version of the script without any update functionality

After that, now if the third parameter is `f`, the script will create the service file if it does not exist, and update it if it already exists, a confirmation message would be printed for both cases and they both return exit 0. On the other hand, if the third parameter is not `f`, the insert script will behave as before: an message would be printed and a corresponding exit code would be returned if the service(`$basename of $service`) already exists (error message, exit 1) or the service and payload(`$payload`) is successfully created(exit 0).

3.2 Setting up the server

The `server.sh` script includes an endless loop that reads input constantly. It reads input directly from the terminal at first for testing purposes, after the creation of the client, it then reads from a named pipe.

Messages will then be shown to show the received input to have a better understanding of what command the server actually received. In order to handle the input effectively, the input is casted from string into an array which is split to different parts and each part is given an index after read. After that, each input index is assigned with a variable:

`index[0]` is `clientID`

`index[1]` is users request(basic command)

`index[2]` is user directory

`index[3]` is service file

`index[4]` and `[5]` is login and password of payload

The `server.sh` then checks if the request is valid and in a correct format, if it is, it will call the corresponding basic command script (e.g. if request is `init` it will call `init.sh`). Each script will then take and execute the arguments given by input. The server will recognised seven commands including six basic commands that mentioned before.

Note that here the insert and update command in server side will call the same script, the only different between them is when the request is update a `"f"` will be passed as a third argument to `insert.sh`; if it if insert then `""` will be passed. This is to use the third parameter to activate different functionality of the insert script according to the user's request. Also, the payload is separated to `$login` and `$password` in both update and insert command before send in order to allow the inclusion of spaces in payloads without any inaccuracy. E.g.:

```
./insert.sh "$user" "$service" "" "login: ${login}\npassword: ${password}" > "$clientID.pipe" &
```

The server will also recognised `"shutdown"`, which is to exit the server(exit 0). If the command read is not recognized, an error message will be printed and the server will also exit(exit 1).

3.3 Creation of semaphores and allow processes run in background

Since the server will eventually be used by multiple clients concurrently, we must execute the commands concurrently in the background and ensure there is no potential inconsistencies(e.g.

when two commands try to modify the same file simultaneously). In order to let processes running in the background, I added ampersand “&” at the end of all basic commands called in server.sh.

Also, in order to avoid potential inconsistencies(synchronisation issues), I created two semaphores scripts P.sh and V.sh, and added them to the critical section of all five basic commands scripts.

Basically, the mechanism of P.sh is that it adds a link with atomic operation which called \$1-lock, this lock will then be cut out when the code inside critical section have been executed. This can ensure that if more than one user attempts to perform the same operation simultaneously, only one user can successfully perform the task, other users must wait for the user to complete the task and wait for V.sh to unlock.

As the critical sections of different basic commands are different, the semaphore will be added in different places:

In init.sh, the user directory is locked before the action of creating user directory by mkdir command in order to prevent the creation of more than one user folder at the same time.

In insert.sh, similarly, the user directory is locked before the action of creating folder in order to prevent the creation of more than one folder in the same user directory at the same time. Also, semaphores are applied before trying to create or update a service to prevent the creation of more than one service at the same time, and also prevent any modification of the service until the previous user is done his/her job with the critical section and unlock the service by V.sh.

In show.sh, the user directory is locked before the action of viewing service with cat command to prevent the possibilities that someone viewing the payload of the service in the user directory while others changing it or deleting it at the same time.

In ls.sh, the user directory is locked before the action of listing any contents in user directory or folder and removed the lock after that in order to prevent the possibilities that someone view the list of service in the user directory or folder while others deleting it or adding other service at the same time.

Note that if user choose to list out the folder, we have to change back to the user directory first before removing the lock by V.sh. Otherwise, the lock can't be remove properly as it does not exist in the directory that we are currently in.

Finally, in rm.sh, the user directory is locked before the action of removing the service in order to prevent the possibilities that the service already removed by others before the user attempt to remove it, creation of more than one user at the same time.

3.4 Setting up the client

The client is the interface for the users and will send requests to the server. In the client.sh script, I first check if the number of parameters inputed by the user is enough(at least two)(if not print error message and exit 1). Then based on the request, the script will check if parameters are given in the right amount and if they are in the correct format(if not an error message will be shown). Here, as error handling has already been implemented in basic command scripts and in the server.sh script, the client.sh script only check the number of parameters given is correct or not. Then if the right

amount of parameters are inputted but the command doesn't exist, the server will still receive the request and the client will get an error message from the server(Bad request).

If the user sends init, ls, and rm request in client, the request will be send to server right away and the output will be view using cat command, after checking parameters number.

If it is an insert request, the client will instruct the user to input login and password and read them separately, then send to server. A "IFS=" command will be added before reading the login and password in order to allow spaces contain in payload with the help of encryption(details will be provided in next section)

If it is an show request, after receiving the output from server, the client will show it in a specific format:

```
echo ""$user"'s login for $service is: $decryptedLogin"  
echo ""$user"'s password for $service is: $decryptedPW"
```

If it is an edit request, the client will insert "show" as second parameter of input then send to the server in order to activate the show command first, then we will make a temporary file(\$MY_TEMP_FILE) to store the output from server. After that, we will read one line from the \$MY_TEMP_FILE first, if the line is an error message, we will show the error message to the user instead fo letting them edit the payload.

Else, we will cast the payload in a "key:value" format, where key is "login:" or "password:" and value will be the corresponding payload. Then we will store it and a comment that remind user not to change the "key:value" format and reserve the space after ":" to \$MY_TEMP_FILE , and open this file with a text editor vim to let the user modify the payload. Some operations are made regarding the edit in order to allow the user to contain empty lines or lines with characters other than the key or reverse the order of payload when entering, some error handling are also set if key are not properly reserved, the details of this part will be introduced in the "challenges" section.

Once the user is done, we will send the input with "update" as second parameter and with the new payload to the server. And finally the client will receive the output from server and the user can view it with cat command.

Also, when the user send a "shutdown" request in the client, it will be send to the server first to exit the server, then print an exit message and exit the client.

So the communication between server and client is accomplished through name pipe, after the request is passed to the server through server.pipe, the client will get the server response through the clientID.pipe. So if, for example, the user directory(\$user) entered doesn't exist when sending an ls command, the server will still receive the request and send to the ls.sh script, and the result "Error: user does not exist" will be received by the client.

3.5 Communication through name pipe

As mentioned above, the server and client are communicated through name pipe. As soon as the system is initiated, the server.sh will create a server.pipe if it does not exist, the client.sh will also create a unique pipe for each client id as clientID.pipe after checking it does not exist. So how does they work? The client.sh will redirect the input entered by clients to the server through server.pipe.

And so the server.sh is able to read the instruction from server.pipe and perform the corresponding request (the input is casted into an array after read to better manipulate each input). Then, the server will send the output of the request back to the client through each unique clientID.pipe. Using a unique clientID.pipe to send result can ensure the server always benign the output to the correct client.

After receiving command output from clientID.pipe, the client.sh will show it to the user. After each request is done, the clientID.pipe must be removed.

4. Extra features

I have implemented 2 extra features to the system: encryption & decryption to protect the privacy of the user, and a piece of code to handle the situation if server is not responding.

4.1 Encryption & decryption

The payload need to be encrypted when user enter insert request, and need to be decrypted when enter show and edit(and encrypted again when done editing) request:

-Whenever the user want to insert new service in client, encrypt.sh will be called twice and the login and password will be encrypted separately with separate key: login with \$cliendID and password with "2\$cliendID" respectively. I use \$cliendID as a key of encryption is to prevent other users from getting the payload of the user that enter the payload, in order to further protect privacy of the user. Also, I add a "2" with the key of password to distinguish it from the key of login, in order to store the payload properly without any error. The output of encryption will then be assigned to two variables(\$encryptedLogin, \$encryptedPW) and sent together with other inputs to server using server.pipe. Now since after encryption, the space will also change to other characters, the payload can include spaces, with a "IFS="'"command added before to make sure \$password and \$payload are sent in one line.

-Whenever the user want to show payload of a service in client, a temporary file(\$MY_TEMP_FILE_one) will be make to store the output from getting from clientID.pipe(the encrypted payload) first, then the first line of \$MY_TEMP_FILE_one will be read, if the line is en error message, we will show the error message to the user. Else, we will cast the payload inside \$MY_TEMP_FILE_one(login and password are in separate lines due to \n added inside server.sh, as shown in the screenshot in "setting up server" part)into one line using awk command first, then we will cut the encrypted login and password separately using cut command. After that, we decrypt them by decrypt.sh using the key we entered during insert request in client, and assign variables to them respectively(\$decryptedLogin, \$decryptedPW) Finally, we show the decrypted payload with specific format to the user.

-Whenever the user want to edit payload of a service in client, after making temporary file(\$MY_TEMP_FILE) and checking content inside clientID.pipe does not contain error message, we cast the payload inside \$MY_TEMP_FILE into one line, cut the encrypted payload, decrypt them, and cast them with specific "key:value" format. We then remove the clientID.pipe and make a new one to prevent inconsistencies when using the same pipe to send update request after edit. Now we open clientID.pipe with vim to let the user change the payload. We will cut the login and

password to get the value when the key is here(using space as delimiter and get all information after the first space). And if key is not here, an error message will be printed and both server and client will exit(exit1). Before encrypting them again, a “IFS=’ ’”command is added in order to allow spaces contain in the payload. After that, we can call the two encrypt.sh(using \$cliendID and “2\$cliendID” as keys again) to encrypt the payload and sent it to the server to do the rest operations as mentioned in the “setting up client” and “Communication through name pipe” part.

4.2 Non-response handling

In client.sh, before doing any operation about the request, I first check that if server.sh is currently running: I show all processes for all users, then used the output to find lines that match “server.sh” using grep, then then used the output to remove the lines that contain grep (to unsure there would be lines that contain both Connected to and grep but not Connection refused), and finally then used the corresponding output to prints the line count of the result.

If the result is greater than one(which means if server.sh is not currently running), a message is printed to notify the user. Then the client will be in a while loop that will continuously check if the server is started in 10 seconds. After 10 seconds, if the server.sh is still not responding, an error message is printed, the \$clinetID.pipe will be removed and client will exit(exit 1)

5. Major Challenges

5.1 Infinite loop in server.sh

My server.sh once got into an Infinite loop suddenly when I try to run client.sh and send the request to the server. The arguments input by client can send to server, and after server call corresponding basic commands, the client can get the output, but then the server won’t stop to wait for my next instructions and keep looping. Also, the client pipe won’t be removed at all(it should after getting the output as I have a rm pipe command) no matter I stop the loop by control C or not:

```
(base) Chung-MacBook-Pro:osAssignment chungwingki$ ./client.s
h c1 init user1
Welcome to the Client
Error: user already exist
(base) Chung-MacBook-Pro:osAssignment chungwingki$
```

```
Received: c1 init user1
Received: c1 init user1
Received: c1 init user1
Received: c1 init user1
Received: c1 init user1
Received: c1 init user1
Received: c1 init user1
^C
(base) Chung-MacBook-Pro:osAssignment chungwingki$
```

After a number of testing and trying, I find out it might be because the content of server.pipe doesn’t gone after read(which it is supposed to as a pipe). I cannot solve this problem and find out the exact cause but assume that it is because I exit the client.sh and server.sh with control C for a number of times instead of exit it properly with a “shutdown” command and it might causes some conflicts in the background and lead to the weird behaviour of my script. Luckily, I have backup my project and no weird loop happens in the backup.

But, after I try the test_example script downloaded from brightspace, something similar happens again: my client show a message that it is connected to the server before I run the server.sh in other

terminal. Now if I run the `server.sh`, an infinite loop in server occurs again. But if I run `server.sh` before `client.sh`, it works perfectly fine. **This issue happens in all my backup too.**

So then I guess it is because somehow the `server.sh` still running in the background due to synchronization problem. I enter `kill -f server.sh` in my terminal and then it becomes normal again. I failed to find a way to prevent this from happening, but every time this problem happens, I use the above `kill` command to solve it. However, this solution didn't work for the initial version of my scripts(the one describe above), infinite loop in that version of scripts didn't gone after I use the same strategies.

5.2 Allow user to edit payload in client

When letting user edit their payload with text editor, I am struggling on how to get login and password correctly if the user have some weird behaviour:

- Type in some empty lines or lines with characters other than the key("login: " or "password: ")
- reverse the order of login and password

I first use `sed -n '1p' "filename"` to extract the line with login and password separately and then use if condition to test the order of them. But then I find out it didn't work if empty lines are entered. So I change the solution to use `grep` and `regex`, now the user are allowed to store their login and password in any order as long as they respect the "key:value" format ("login: " and "password: " are found at the beginning of any sentences). The value will then be cut and encrypted separately if keys are there. And if user fail the follow the format, I also set an error handling that error message will be printed and both client and server will exit(exit 1).

5.3 Decide the key of encryption

I first use a meaningless string as a key of encryption and decryption, but then I find out it is actually not safe enough as every user can get the payload of each other if they type in the same user directory and service. So then I change the key to `$clientID`, now users must enter their own unique id correctly if they want to see or edit the payload, this can better protect the privacy.

5.4 Communication between server and client

I first use `read -a input < server.pipe` to manipulate the input read from the server named pipe, but with this strategies server can only read the first argument before a space in my input. This probably due to the input are sent as string in client. Therefore, I read the input as a string first, then cast it to array `input=($inputt)` to do further operations, this works.

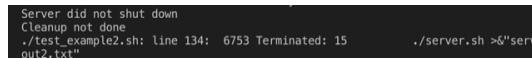
Also, when the request is insert in `client.sh`, I first simply echo "\$@" to `server.pipe`. But the login and password read were missing, so I pass `$@ $login $password` together as a string to `server.pipe`, it works.

There is one more communication problem when running the test script, and will be introduced in 5.5 part.

5.5 Error when running test script on my Macbook

When I first try to run the test script downloaded from brightspace, the script won't execute the commands after "Server seems OK" in #test server. I then find out it is because I didn't send anything to clientID.pipe when the request is shutdown, it just simply exit, therefore the script cannot cat clientID.pipe and it cannot run the code after. So after I send something to client when receiving shutdown request, the script can run till the end.

But then another problem about came up, when testing client in the test script, the following errors occur:



```
Server did not shut down
Cleanup not done
./test_example2.sh: line 134: 6753 Terminated: 15 ./server.sh >&"serv
out2.txt"
```

I first thought that it is because the communication problem between client and server: the client might run so fast that somehow viewing(cat) the output from clientID.pipe before server sending anything to clientID.pipe. So I add a "sleep0.01" right before each `cat "$clientID.pipe"` in order to make the client to wait a little bit to make sure the server have enough time to do its job. This solution works at first. But then after a few tries, when I then try to run my client.sh manually(e.g. `client.sh client1 init user1`), `server.pipe: interrupted system call` error appears in the form of infinite loop. Now if I remove the sleep in client side and add a sleep after each process is being sent in the background in server.sh, the error gone. But then if I try to run the test script again, each time different errors appears(e.g. server not shut down, not clean up, script exit when testing server..)

I finally solve this whole problem simply by **moving the whole project folder to CSserver**, then it works perfectly fine without any sleep command.

But here is one more issue: If I run client.sh before server.sh on CSserver, and then run server.sh, infinite loop in server occurs and it makes my Non-response handling useless. This problem won't happen when I try the scripts on my own computer. But since the demonstrator confirmed that they will always run servers before client.sh when grading, it should be fine.

6. Conclusion

To conclude, this was a complicated project that contains many commands, operations and challenges. However, I found it really helpful to follow the steps to create one small script at a time. It is like a puzzle, when more and more small codes and scripts are finished, the whole picture of the system becomes more clear. However, I also found it difficult to debug sometimes, because some errors are not caused by the code, but instead because of the system, and also the exact same code works for 1 time and even 1000 times doesn't mean that it won't have error when running the it in the future, you just can't make sure it will be fine all the time even if you already test it.