Project: Password Management System

Operating Systems (COMP30640)

Niall Delahunty - 13397526

1. Introduction

The idea of this project was to create a password management system using Bash. The idea of the password management system is to allow users to store login details (i.e. username and password) for different services in one centralised system. In this system the user can add new services with login details, look at existing passwords for a given service as well as remove any details no longer needed. It is built on a client-server model, which is made to mimic a centralised password manager such as Bitwarden.

2. Requirements (What is the system supposed to do?)

The system is supposed to manage login details (login name and password) for specific services for different users. This means creating, updating, removing and showing the login details for a given service. The system has two main aspects, the server and the client. The client (from running script client.sh) using their client ID and indicating their desired intent should be able to perform 6 different tasks with regards to password management:

(1) Create a new user (init.sh)
(2) Insert a new service with a given login and password (insert.sh)
(3) Show an existing login details for a given service (show.sh)
(4) List (tree structure) all the user services or services within a subfolder of a user (ls.sh)
(5) Remove a service (rm.sh)
(6) Update the login details (show.sh and insert.sh).

Given the client has entered a clientID, a valid action and a specific user (existing or new), the query is then subject to further constraints dependent on the action desired (e.g. inserting a new service - client needs to also enter a service and login details). With appropriate error handling in place the client side (client.sh) should interact with the server side (server.sh) passing the appropriate information in. On the server side based on the request, scripts are called which carry out the specifics actions. Once finished or if there is an error at any point, this should be conveyed to the client. Concurrency issues should be accounted for to allow multiple clients using the service at the same time (i.e. protect against two users altering data at the same time e.g. one trying to update a service that the other one is removing).

3. Architecture/design

Similar to how the project brief was presented I will discuss my solutions for each aspect first starting off with the individual scripts for each function (e.g. insert, remove etc.) specifying why I have chosen to design it in the manner I have. Overall the design involved two primary scripts client.sh which dealt with the client interface and input, and server.sh which dealt with taking the information given and carrying out the necessary actions through interaction with specific scripts.

A. Basic commands

(**Note:** Semaphores - see P.sh and V.sh scripts)
To explain the semaphores that are mentioned in the scripts below, P.sh is a script that forms a link between the files P.sh and a file with the name of the argument given with "- lock" concatenated i.e. "$1-lock" (creates this file). P.sh is run before a critical section and if the lock file already exists the process sleeps and tries again over and over in a while loop. V.sh is run after the critical section and removes the lock file allowing the next process trying to enter the critical section to form a link. I explain the arguments I give in each script that uses this semaphore structure. The purpose of these semaphores is to ensure consistency with concurrent users.

(i) Register a new user (init.sh)

The script saves first argument (username) as variable name, it has a condition which outputs an error message if the number of parameters is not equal to 1 (and exits). It has a second condition statement which checks if the username given already exists and prints out an error message if it does (and exits). If both conditions are passed a directory is created with $name. and confirmatory message is outputted. Putting "$1" allows for user name with white space.

(ii) Insert a new service/update an existing service (insert.sh)

Parameter checks
The script saves first argument (username) as variable 'username' and the second argument (service) as variable 'service' This script has the functionality of both inserting a new service and of updating existing ones. Therefore it first checks if there are 4 parameters (i.e. update mode) and sets variable 'update' to the third argument (should be where "f" is entered - will fail to update otherwise) and variable 'payload' to argument 4 (login details). Otherwise update will be set to "none" and $payload to argument 3 (login details). The variable 'path' is set up as the path to the file/service , variable 'respect' is assigned the value "f", this is used in the comparison when updating. The script checks if there is less than 3 or greater than 4 arguments given(i.e. only accepts 3 or 4 arguments). It uses '-e' to check if a folder of the username given already exists.

lock
A variable called 'tempbase' contains the filename derived from calling the *basename* command on $service (e.g. aib.ie). A second variable called 'lockname' contains $user with a hyphen and then $tempbase all in one string (i.e "$user-$tempbase"). This design is so it doesn't matter if the service has folders or not it will still form a link with the name of the user and the service (e.g. user-aib.ie-lock or user-google.com-lock).
I use $lockname as it creates a lock specific for not only the user but with the service they are trying to access. This prevents a given service from being modified simultaneously (concurrency) while still allowing the users other services to be accessed (granularity). The P.sh script is called before it checks if the path exists to prevent two processes from entering the critical section with the instruction to create the exact same service and login details (i.e. both could pass check if outside critical section then both attempt to create it). A check using RegEx checks if $service is a folder (i.e. ends in '/'), as otherwise an error would occur at the creating file point.

Update
Once the correct parameter number and existing username is confirmed, '-e' is used to check if the file path exists (i.e. service has been created) if so then it will check $update to see if it is "f". If $update is equal to "f" the new $payload overwrites the old information in the given service (uses '>'). If it is not "f", an error message is outputted. In both cases there is an exit from the script.

Insert
If the path does not exist i.e. service is new, a variable called 'name' contains the result of calling the *dirname* command on $path (dirname command on the whole file path gets everything bar the

filename). If $name does not lead to a directory (e.g. if bank/aib.ie is the service $name would be user/bank/) then all folders that do not exist in that path before the file are created (mkdir -p). In the case of a non folder service (e.g. user/google.com) $name would be "." and so no new directory is created (not necessary).
The $payload is put into a new file by the name of the service (echo -e is used so that the escape sequence '\n' is supported - needed to put login and password on two different lines.)
The non-error outputs are followed by exit 0 to show desired action. Regardless of outcome once in the critical section V.sh is called under each condition path to ensure a process doesn't get stuck in the critical section.

(iii) Show login details (show.sh)

Parameter checks

Similar to the other scripts there are conditionals testing that the number of parameters is 2, that the username folder does exist, that the service exists as it is stated (i.e. if it is in a folder, the client must specify bank/aib.ie can't just input aib.ie) and finally a check that the service entered is a service name and not a folder name. If any of these conditions are not met, an error message its outputted.

lock

P.sh is given the same argument format to insert.sh (user-servicename). The semaphore is called just before checking for the path (Same as insert.sh).

Show

Through the client side passwords are entered as
"login: login details \n password: password details" , which stores them as:
login: "login details"
password: "password details"
To get the login and password grep is used and ':' is used as the delimiter with the rest of that line saved as the login or password. This means white spaces are fine and if the details get put on a different line (e.g. from editing) as long as they are prefixed by login: and password: they will be correctly outputted. This could be argued as a weakness but given the constraints of the system I felt this was a decently robust mechanism for presenting the details. When the script is run individually the output is different from the brief. It is not
login: "login details"
password: "password details"
but instead
"login details"
"password details"
This is to do with how it is dealt with on the client side. Therefore it deviates from the brief slightly however I would argue that it performs the same function and if necessary the individual script could be changed to reflect the other output.
V.sh is called before any exit in the critical section.

(iv) Remove a service (rm.sh)

The same checks as in previous scripts were used to test for the correct number of parameters, that the username exists and that the path exists. Then once this is established enter the critical section (P.sh). It is checked if the service leads to a file, if so delete the service (do not delete any folder that it may be in e.g. bank). Otherwise an error message is given. In both cases V.sh script is called and the script stops (exit). Currently the rm.sh is designed so that only the service and not the service folder is deleted as this seems most suited for purpose. If I was to implement this extra functionality I would create a condition that checks if the folder is empty and if so provide the option to delete it. I decided against this as the service folder (e.g. Bank) is likely to be used again.

However at present from the client side there is no mechanism for deleting service folders once created (this may be the result of a misunderstanding of the brief on my behalf).

(v) List a service (ls.sh)

The same parameter checks are carried out and that the username exists. If there is a second argument given (i.e. service folder) it is checked if "$user/$service" leads to a folder (e.g. user/bank), if so result of the *tree* command on $path ($user/$service) is outputted (tree structure of folders, subfolders and files), otherwise an error message is given.
If there is only one parameter the result of the *tree* command on $user is given. I decided to not put in locks as no data is being changed or manipulated and therefore I thought it was unnecessary. The only risk is that the client might see a folder structure that is simultaneously being changed. But the there is no risk of the actual data being altered due to concurrent processes.

B. Client and Server

(i) Client side (Client.sh)

This script covers the client facing aspect of the system (i.e. takes raw input and transfers it over to the server script and outputs the outcome of the action).
Before accessing anything the first two parameters are assessed (the clientId and the req i.e. instruction) if they are empty (-z) then an error message is outputted. If they are not empty, variables 'client', 'req' , 'user' and 'service' are created from the entered parameters. If the client specific pipe file(clientid.pipe) doesn't exist then it is created.

A trap function is included, which is called upon 'control + c' being entered (used to end a process before it has finished). To ensure that an exit is definitely the desired output a counter variable is declared ($ctrl_count) which is incremented if the function is called. On first call a message is outputted to check if the user definitely wants to exit. If it is called more than once (i.e. $ctrl_count > 1) then the client pipe is removed and the script is exited. This function was taken from: https://rimuhosting.com/knowledgebase/linux/misc/trapping-ctrl-c-in-bash .

Case Statement

A case structure is used which carries out a different action dependent on $req:

- *init:* the $client, $req and $user are passed into the server pipe (server.pipe). The return from the client pipe is read into the variable 'message' and outputted. A variable was used here as the output is only going to be a single string (i.e. ok message or error message).

- insert: $user and $service are checked to see if they are empty or not (error message if so), the user is prompted and input is read into variables 'login' and 'password'. These inputs are combined into a single variable string called 'payload' separated by '\n' so that when inserted into the file, login and password are put onto two different lines. $Client, $req, $username, $service and $payload variables are passed into the server pipe. Output from the client pipe is read into variable 'message' (same as above).

- show: $user and $service are checked to make sure they aren't empty variables. $client, $req, $user, $service and $payload variables are passed to the server pipe. The client pipe (containing returning information from server) is read into the array 'login'. Check if array length is greater than 2 (error messages will be greater than two). If so output the first two elements of the array i.e. the login and password details. Otherwise print out the whole array (error message).

- ls: $client, $req, $user and $service details are passed into the server pipe. The cat command is performed on the client pipe (so that the returned tree structure is seen).

- rm: $user and $service are checked to see if they are empty or not (error message if so). $client, $req, $user and $service are passed into the server pipe. The return from the client pipe is read into the variable 'message' and outputted.

- Edit: $user and service are checked to make sure they aren't empty variables. $client, "show", $user and $service are passed into the server pipe. (i.e. call show script). The output is read into an array. The same error handling as shown in 'show' case statement is used. A temporary file is created, an instruction message as well as the existing login details are written into the temporary file, the user is then presented with the temporary file to edit (vi). Using grep the login and password are isolated and saved into appropriate variables ($payload). $client, "update", $user, $service, "f" and $payload are passed into server pipe. (i.e. initiate update protocol on server). The file is updated and error or confirmatory messages are read into a variable and outputted. The temporary file is deleted.

- shutdown: $client, $req are passed to the server pipe. This causes the server.sh to shutdown and return a message. This is displayed on the client side by using *cat* command on the client pipe. On shutdown the client pipe is deleted and the script is exited.

- *: if there is a bad request i.e. a non-existent req is passed in, an error message is displayed, the client pipe is removed and the script is exited.

(ii) Server side (Server.sh)

This script covers the server side i.e. takes in instruction from client side and acts on those inputs to deliver the desired outcome. First the server pipe is created if it did not already exist. The same trap function as described in the client.sh section is included in server.sh. It is the exact same except server pipe is removed not the client pipe.

In a loop (while true), output from server pipe (from client side) is read into an array. The index of the array corresponding to the action (index 1) is passed into a case statement. In each case the output is passed into the client specific pipe (clientid is index 0). In each case (except ls.sh) the script call is performed within 'echo $(.. &)' and the output of the script is passed into the client pipe ('index 0'.pipe). '&' is used to run the process in the background.
The case statements are similar to client.sh:

- init: checks if index 2 is empty ($user), if not init.sh is run with that as the parameter, otherwise echo error message. This is to prevent a folder of an empty string being created.

- insert: checks if index 4 till the end of the array is empty (login details - $payload). This is to make sure that empty login details are not entered into the service. If it is not empty, insert.sh is run with $user (index 2), $service (index 3) and $payload (index 4).

- show: show.sh is run with $user (index 2) and $service (index 3). The show.sh script contains all the appropriate error handling.

- update: As 'f' is passed in after $service in the client.sh script in this case there is a check if index 5 till the end of the array is empty (login details i.e. $payload). Insert.sh is run with $user (index 2), $service (index 3) , "f" (index 4) and and $payload (index 5).

- rm: rm.sh is run with $user (index 2), $service (index 3). The rm.sh script contains all the appropriate error handling.

- ls: ls.sh can be run with one ($user) or two ($user and $service) parameters. If the second applicable parameter (index 3) is empty than just $user (index 2) is run as the parameter for ls.sh, otherwise both are run.

- shutdown: Echos a message to the client pipe, removes server.pipe and exits server.sh script.

- *: if there is a bad request i.e. a non-existent case, an error message is displayed, the server pipe is removed and the script is exited.

Bonus Scripts

(i)Encryption
Using the scripts provided (encrypt.sh and decrypt.sh) I encrypted the password after the user had entered it in client.sh by running ./encrypt.sh with the master-password “I<3Bash” as the first argument and the password as the second. In show (case statement in client.sh) I decrypt the stored encrypted password by running ./decrypt.sh using the same master-password and using the second array element (encrypted password) as the second argument. In ‘edit’ in client.sh I decrypt the password before the user can edit (so it is understandable) and then encrypt again with the user changes.

(ii) Random password
This was done using the script ‘generate.sh’. The user is prompted for Y/N input to generate a random password (Error handling -> only accepts Y or y, otherwise prompts user to enter password). If accepted the generate.sh script is called, otherwise the user is prompted for a password. In the script a string containing the alphabet, digits 1-9 and some random symbols is saved into a variable. 8 characters from this string are selected using the command ‘shuf’ to generate a random number, the range for shuf is the number of characters in the string (0-42) selecting one each time. A single character is then selected using this integer to define which index. This is appended onto an empty string variable which is returned. This was done using a script as it allows the password generation protocol to be easily updated (e.g. make more secure) or changed without having to interfere or stop the client.sh script.

4. Challenges faced

I have no doubt I encountered far more challenges than documented below, but due to the countless iterations of each script I have forgotten many of the obstacles I overcame, a huge proportion were syntax related and so I have not included these, the following are the main challenges I could recall:

(i) Semaphore argument and positioning
Fortunately we had covered semaphore locks in our practical sessions. I used the P.sh and V.sh scripts we had developed in that practical. Originally I had the semaphore argument as “sema” for all scripts and quickly realised this would not work as it would mean one person could only do one specific thing on any user at one time (very slow). I originally also had the P.sh and V.sh scripts called either side of the scripts in each case statement in the server.sh scripts. This seemed like a good solution until I realised that only one client could perform a specific action at once. I solved the positioning issue by calling the scripts around the critical sections of each script. It took me a lot longer to work out a suitable argument name to pass into P.sh (i.e. what the lock file would be called). I wanted to have it designed so that only one person could access a specific service of a specific user folder at once. This was to prevent two antagonistic actions on the same service file. I used “$user-$service” initially which worked well until the service had a folder in it (e.g. bank/aib.ie). I initially solved this using regex (if $ =~ ./.) but realised a much simpler solution was to just use the *basename* command to get the last element of $service which will be the service file as

long as it exists . A weakness in this approach is that if a service is entered in the format ‘ bank/ ‘, a semaphore ‘lock’ file is formed, ‘$user-bank’, and an error message is thrown when it tries to create a file (as it is a directory). To solve this I put in a condition prior to the critical section to check if the service ends in ‘/‘ using regex (=~ $/).

(ii) Reading arguments

I believe this was a common problem and I had a lot of trouble with this initially. This was first encountered in the context of the server.sh script, reading in from the user prompt and then from the server pipe. I was trying to predefine a number of variables however if all variables were not used (i.e. less parameters given than defined variables) the rest were saved as empty strings which could cause issues in the scripts themselves. Additionally if there was a whitespace in any argument it would be read into two separate variables (The login details in particular). This was solved by reading into an array, then using the appropriate index for an argument. For login details which could have whitespaces but were the last arguments, I passed the remainder of the array as a single variable (e.g. ${input[*]:4}).

(iii)Folders in service for insert.sh

I originally encountered the problem with insert.sh when bank/aib.ie was used as the service. I couldn’t create the file if the folder didn’t exist. Two solutions I found, one use regex (=~ ./.) to identify a folder or what I used was use the *dirname* command on the $path (user/service) and save into variable ‘name’. Then check $name if it leads to a directory (-d), if it does not it would mean that the folder before the service needs to be created (in the case of a normal service e.g. google.com the $name is a directory and so passes this condition). mkdir -p was put in incase there is more than one folder in the service (makes any necessary parent folders).

(iv)Edit on client.sh (and show.sh)

I had an issue when the edit case was called, ensuring that login and password details were presented for updating. Particularly to account for the password or login details not being on the first two lines. Instead I depended on the use of login: and password: as key words and used grep to get the line after these keywords. (this is how they are entered in through insert). In the case of them being deleted in an update I have a condition that prompts the user to rerun the script call if the product of grep for login or password is empty.

(v)Tree structure on client.sh

I was having an issue with displaying the tree properly on the client side. I tried reading it in many different forms (variable, array etc). The solution was to not echo the script output on the server side and to use the cat command on the client pipe as opposed to reading in from it.

5. Conclusion

Overall I found this project to be quite challenging and time consuming, however it was also thoroughly enjoyable with plenty of opportunity for creative problem solving. I also feel my capabilities in writing bash scripts has increased enormously, something I really struggled with during the practicals. It is very rewarding to create something which to me seems quite complicated and intricate compared to other problems faced thus far. I feel the password system meets the requirements as described in the brief and despite some weaknesses or design flaws, can function.

In the edit case on client.sh I feel there should probably be some semaphore or a locking mechanism to prevent context switches between the show and insert scripts, however I did not have time to flesh this idea out. The idea behind these proposed semaphores locks are

commented out in the client.sh script (‘editlock’) which uses the user name - “edit” e.g. (user-edit) as the argument for P.sh. However this was causing some issues afterwards (client login and passwords were being printed out) and I did not have the time to properly solve this issue. It is likely a relatively easy fix and might just require some work around with the way information is outputted or piped between the client.sh and server.sh, but I think my logic is correct.

If I was to continue developing this system further I think it would be interesting to build more on the interactive side of things but also to delve deeper into concurrency and learn how to scale such a system for thousands of users.