Password Management system

Introduction

With extensive span of internet, generating accounts and maintaining credentials is a tedious task. Since web based apps are connecting our wallet as well as our intellectual property; web security is a prime challenge for sustainable internet usage. A password manager assists in generating and retrieving complex passwords, potentially storing such passwords in an encrypted database or calculating them on demand. Password managers are locally installed or locally accessed software or web based services.

Purpose of System

In broader sense, Password management system allow user to store their login credentials in encrypted format at local/ remote server. They also facilitate user by providing Automatic password generation facility.

During web transactions, password managers automatically submit credentials for user to website. This is done through consent of user on the go or though already set permissions by user. Basic versions of password management system are available on modern day browsers which provide users autofill options.

Architecture

For the scope of this project, system developed emulates server side and client side. Passwords are user defined and stored in raw form. Both client and server are represented by separate scripts viz. server.sh, client.sh. For system operation, server invokes multiple scripts viz. init.sh, insert.sh, ls.sh, rm.sh, show.sh to generate users, make entry of credentials for the user, remove the user etc.. p.sh and v.sh are internally called before and after the critical section in each of the above scripts.

- **server.sh**
  Server is connected with the world through a single named pipe named “server.pipe”. When script is instantiated, server generates the pipe if not already present. Until server is shutdown, Server infinitely reads this pipe together requests from client.
  Client feeds data into the pipe in syntax-
  >[client] [operation] [userName] [serviceName] [special argument] [payload]
  Pipe data is read into array structure. And a pipe for particular client pipe is made using argument [client]. Further script finds is valid operation keyword exists for determined input syntax. If so, arguments for the operation are taken inside an array arg. Case statement matches the operation keyword. Subsequent routine calls the script for operation and provides arg array elements as argument. Data echoed by script is caught in variable pipeOut. This variable is passed to client pipe for the client side to read it.

- **client.sh**

  Client script is invoked with arguments by user. Syntax designed is-

  >./client.sh [client] [operation] [userName] [serviceName] [special argument] [payload]

  Alike server script, client script cases through the operation keyword. It checks if server pipe exists or not. For each keyword, specific number of arguments are echoed over server pipe.

During insert operation; spaces are replaced with special pattern "--". User inputs are restricted from entering this pattern as part of login or password. Conversely, show function replaces "--" pattern with a space. This operation is implemented using regex and bash string manipulation.

Edit operation invokes show function and stores returned content into a temp file. The file is opened to allow user edition in credentials. **USER IS RESTRICTED NOT TO ADD NEWLINE DURING EDITION THAT IS; TEMP FILE MUST CONTAIN ONLY TWO LINES AFTER EDITION AS PRIOR.** The temp file is read into variable pipeIn update service is invoked through server pipe.

- **Init.sh**

  Each user is represented by a folder. Init script checks if a user directory already exists or not and accordingly creates the user folder. Init.sh is called by server to create a new user. Directory creation is a critical section when multiple clients attempt to use a user service. Hence static semaphore guards code integrity.

- **Insert.sh**

  Insert.sh script is used to generate new user service as well as to update existing one. Hence it extensively checks the number of arguments and depending on the presence of special argument "F", present user service is edited or preserved. Semaphore guard directory validation loops, file validation loops and file creation commands. Services may have subservices, hence multiple service folders may reside within service folder. They can be created using -p [parent] attribute to mkdir. But a file and directory of same identifier can not coexist at same address hierarchy. Hence, directories are created recursively and at each stage, it is checked if a file of same name exists or not.

- **Ls.sh**
  This function lists the tree of a user and or a user service. This is a UI facilitation function with no direct impact on operations. This function does not need critical section handling.

- **rm.sh**
  rm.sh enables client to remove unwanted service entries. A service must exist prior to removal. Hence, a file validation loop and file deletion command exist in rm.sh. These critical sections are protected by semaphore.

- **Show.sh**
  show.sh is used by client to access stored credentials. The function checks if requested service exists. If so, contents for the service are concatenated to standard output. Show is also invoked during edit operation at client end.

Challenges

- For transportation over pipes, spaces need to be regularized with some unique pattern. And that pattern can not be reutilized in credentials. This can be tricky in case of encoding the passwords.
- During edition operation, contents of service are fetched and altered in a temporary file. This file needs to be read into a variable. As file has two lines, carriage return needs to be replaced

with literal “\n”. Making this payload string is complicated. It was solved using regex and string manipulation.

- P.sh and v.sh synchronization is critical to operation. To avoid server overwhelming; Nested while loops are used to read server pipe data.
- Shell syntax for loops which check multiple conditions in tandem is tricky. It is time consuming to debug syntax errors for the particular case. Editing in linux based editors is tricky. For readability and debugging, proper indentation is important which consumes some time.

Conclusion

This project was a good way of exploring bash operations, semaphores and code complexity. I understood and moreover, achieved synchronization in my server section.