School of Computer Science

COMP30640

Project
Password Management System

Teaching Assistant:	Thomas Laurent
Coordinator:	Anthony Ventresque
Date:	Friday 15th November, 2019
Total Number of Pages:	10

General Instructions.

- You are encouraged to collaborate with your peers on this project, but all written work must be your own. Yous should be able to explain every aspect of your solution if asked.
- We ask you to hand in an archive (zip or tar.gz) of your solution which should include your code/scripts, and a 5-10 page pdf report of your work (no need to include code in it).
- The script names, command names, arguments orders, and output messages given in the instructions must be respected, as we will be running your scripts.
- The report should include the following sections:
  1. A short introduction
  2. A requirement section that answers the question "*what* is the system supposed to do?".
  3. An architecture/design section that details *how* your solution has been designed to address the requirements described in the previous section.
  4. A series of sections that describe the different challenges you faced and your solutions. For instance, take one of the scripts, describe the difficulty you faced and your solution. These sections can be short – the objective here is to show how you crafted the solutions with the tools you have learned so far.
  5. A short conclusion.
- The project is worth 30% of the total grade for this module. The breakdown of marks for the project will be as follows:
  - Commands: 30%
  - Server: 15%
  - Client(s): 30%
  - Report: 25%
- Due date: 29/11/2019

1 Introduction

In this project you will be implementing a basic password management system in bash[1]. A password management system helps us keep track of all our passwords for different systems, instead of relying on the same password for everything because of poor memory. Here we will build our password manager on a client-server model to mimic centralised password managers such as Bitwarden (https://bitwarden.com/), Lastpass (https://www.lastpass.com) or 1password (https://1password.com/).

We will now describe the system you will have to build and its different features.

2 The Basic Commands of your Server

The server stores the passwords of different users for different services. Each user, named $user is represented by folder $user. Examples of $user are "user1" or "Thomas Laurent". Each user has passwords for difference services. Each service $service has file $user/$service which stores the password and login for this service as a key-value file. Examples of $service are "google.com", "Bank/aib.ie" or "Bank/boi".

Suppose our server has passwords for two users "user1" and "user2", and that user1 has stored passwords for services: "UCD CONNECT" and "Bank/aib.ie". In the server's working directory we would see:

```
$ ls -l
user1
user2
$ ls -l user1
UCD CONNECT
Bank
$ ls -l user1/Bank
aib.ie
$ cat user1/UCD\ CONNECT
login: 12345678
password: RextT!F4%!^|%>9h{|[QJ&p!l
$ cat user1/Bank/aib.ie
password: hunter2
login: mylogin
```

Note: The user name, service name, login, and password can contain special characters and spaces.

The server needs to implement the following operations:

Init new user Create a new folder to store a user's passwords.

Insert new service Create a new file in a user's folder containing the login and password for a service.

Show a service Read the information stored for a service

[1]There are real and very useful password managers made in bash, such as pass (https://www.passwordstore.org/)

Update a service Change the information stored for a service

Remove a service Delete the file corresponding to a service

List services List the services that a user has registered a password for

2.1 Register new user

First create a script `init.sh` that takes 1 parameter, $user, and creates the directory for the user $user. Your script should differentiate the following cases and print a corresponding message (make sure the exit message is the only thing printed in all your scripts, this will matter later in the project):

- Too few or too many parameters
- The user already exists
- Everything went well

```
$ ./init.sh
Error: parameters problem
$ ./init.sh user1
Error: user already exists
$ ./init.sh newUser
OK: user created
```

2.2 Insert new service

Now write a script `insert.sh` that takes 3 parameters, $user, $service, and $payload, and creates the file for the service $service, which contains the $payload in the directory $user. Your script should differentiate the following cases and print a corresponding message:

- Too few or too many parameters
- The user does not exist
- The service already exists
- Everything went well

```
$ ./insert.sh google.com
Error: parameters problem
$ ./insert.sh user123 google.com "login: myLogin\npassword: myPassword"
Error: user does not exist
$ ./insert.sh user1 Bank/aib.ie "password: myPassword\nlogin: myLogin"
Error: service already exists
$ ./insert.sh user1 Bank/bankOfIreland "login: myLogin\npassword: myPassword"
OK: service created
$ ./insert.sh user1 "My Games/League of Legends" "login: aaa\npassword: bbb"
OK: service created
```

2.3 Show a service

Our password manager will not be very useful if we can not retrieve our passwords. This will be handled by `show.sh`. This script will take in two arguments: $user and $service and will simply print the content of the $user/$service file, after checking that it exists:

```
$ ./show.sh google.com
Error: parameters problem
$ ./show.sh user123 google.com
Error: user does not exist
$ ./show.sh user1 google.com
Error: service does not exist
$ ./show.sh user1 Bank/aib.ie
password: hunter2
login: mylogin
```

2.4 Update a service

Sometimes a user's password or login for a website can change. We will then need to change the content of the $user/$service file. This is very similar to inserting a new service, so let us just modify our `insert.sh` script. Modify the `insert.sh` script so that it now takes 4 parameters instead of 3. If the third parameter is f then the script should create the service file if it does not exist, and update it if it already exists. If the third parameter is not f, then `insert.sh` should behave as before: create the service if it does not exist and throw an error if it does exist.

```
$ ./insert.sh google.com
Error: parameters problem
$ ./insert.sh user1 google.com f "login: myLogin\npassword: myPassword"
OK: service created
$ ./insert.sh user1 Bank/aib.ie "" "password: myPassword\nlogin: myLogin"
Error: service already exists
$ ./insert.sh user1 Bank/aib.ie f "password: myPassword\nlogin: myLogin"
OK: service updated
```

2.5 Remove a service

A user might not use a service anymore, or might not want to store the password for it in the password manager, they must then be able to delete the information for the service from the system. To handle this case create an `rm.sh` script that takes two arguments: $user and $service. Your script should differentiate between the following cases and print a corresponding message:

- Too few or too many parameters
- The user does not exist
- The service does not exist
- Everything went well

```
$ ./rm.sh google.com
Error: parameters problem
$ ./rm.sh user123 google.com
Error: user does not exist
$ ./rm.sh user1 google.com
Error: service does not exist
$ ./rm.sh user1 Bank/aib.ie
OK: service removed
```

2.6 List services

A user might forget what services they have registered in the password manager and might want to check them. To accommodate this write a `ls.sh` script that takes $user as a first argument and $folder as a second, optional argument. Your script should differentiate between the following cases and print a corresponding message:

- Too few or too many parameters
- The user does not exist
- The folder does not exist
- Everything went well

```
$ ./ls.sh a b c
Error: parameters problem
$ ./ls.sh user123
Error: user does not exist
$ ./ls.sh user1 Games
Error: folder does not exist
$ ./ls.sh user1
OK:
user1
├── Bank
│   ├── aib.ie
├── UCD CONNECT
$ ./ls.sh user1 Bank
OK:
Bank
├── aib.ie
```

Hint: You can get this structured view of the file system by using the `tree` command.

3 The server

Now you will set up the server of your application. As a first step, your server will read commands from the prompt and will execute them. Write a script `server.sh`. This script is composed of an endless loop. Every time the script enters the loop it reads a new command from the prompt. Every request follows the structure `req [args]`. There are seven different 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

If the request does not have the right format, your script prints an error message: `Error: bad request`. A good structure for your script could be the case one. Your script will look like that:

```
while true; do
    case "$request" in
        init)
            # do something
            ;;
        insert)
            # do something
            ;;
        show)
            # do something
            ;;
        update)
            # do something
            ;;
        rm)
            # do something
            ;;
        ls)
            # do something
            ;;
        shutdown)
            # do something
            ;;
        *)
            echo "Error: bad request"
            exit 1
    esac
done
```

At this point you should probably test that your server works well by sending requests to it and checking that the files are created and modified accordingly. For example run the following scenario (assuming no data registered) and make sure things looks right after each step:

```
$>./server.sh
init user1
OK: user created
init user1
Error: user already exists
insert user1 Bank/aib.ie "login: myLogin\npassword: easyPassword"
OK: service created
show user1 Bank/aib.ie
login: myLogin
password: easyPassword
insert user1 Bank/aib.ie "login: myLogin\npassword: RextT!F4%!^|%>9h{|[QJ&p!l"
Error: service already exists
update user1 Bank/aib.ie "login: myLogin\npassword: RextT!F4%!^|%>9h{|[QJ&p!l"
OK: service updated
ls user1
OK:
user1
├── Bank
│   ├── aib.ie
rm user1 Bank/aib.ie
OK: service removed
```

Eventually your server will be used by multiple processes concurrently (the various clients accessing the server). You then need to execute the commands concurrently and in the background. The problem is that with concurrent execution comes the risk of inconsistencies; for instance when two commands accessing the same file run concurrently. You then need to update all of your scripts to avoid these potential inconsistencies. An option is to use a lock: $user.lock, whenever your server's scripts try to access user $user's folder or files (you can also have more fine-grained locking). Modify the server script `server.sh` to start the commands in the background and update the scripts to run concurrently.

4 The Clients

The last component of your system is the client application, which will be the interface for the users and will send requests to the server. You will write a script `client.sh`, which will be used with the following syntax: `./client.sh $clientId $req [args]` where $clienId is the unique identifier of this client and $req is one of init, insert, show, edit, rm, ls or shutdown.

First your script has to check if it received enough arguments (at least 2), and then that the request is a valid one. Depending on the request, your client should do different things:

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

```
$ ./client.sh client1 insert user1 Bank/myNewBank
Please write login: #Written by client.sh
myLogin #Written by user
Please write password: #Written by client.sh
```

```
myPassword #Written by user
OK: service created
#The following example can be ignored. See note below
$ ./client.sh client1 insert "a user that exists" "New Folder/New Service"
Please write login: #Written by client.sh
#...
```

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.

```
$ ./client.sh client1 show user1 Bank/aib.ie
user1's login for Bank/aib.ie is: mylogin
user1's password for Bank/aib.ie is: hunter2
#The following example can be ignored. See note below
$ ./client.sh client1 show user1 "UCD CONNECT"
user1's login for UCD CONNECT is: 12345678
user1's password for UCD CONNECT is: RextT!F4%!^|%>9h{|[QJ&p!l
```

ls check that (at least) a client id and user name were given, send a show request to the server and print the result.

```
$ ./client.sh client1 ls user1
OK:
user1
├── Bank
│   ├── aib.ie
├── UCD CONNECT
$ ./client.sh client1 ls user1 Bank
OK:
Bank
├── aib.ie
```

edit check that a client id, user name and service were given, retrieve the payload for the service (same as show), 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. (delete the temp file at some point). You can create a unique temporary file using the `mktemp` command.

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

How will the clients send the requests to the servers and receive its answer? Your system will connect the client(s) and server with named pipes. The server will create a named pipe called `server.pipe` and each client will create a named pipe `$id.pipe` with $id the id given as parameter of the client.sh script (the client must send its $id to the server along with the requests).

Modify the scripts `server.sh` and `client.sh` to create those named pipes (and delete them when you are done!).

Now, clients need to send their requests on the server's pipe and, likewise, the server needs to read the requests on its named pipe. Modify both scripts accordingly. We recommend you open two terminals, one for the client and one for the server in order to test your scripts.

Note: Parsing arguments sent through a pipe can be a bit of a challenge. To make this easier you can consider that the user name and service name never contain a space. Your individual scripts still need to handle this case but you can ignore it for the client/server communication. Otherwise you can choose a special character that you will use as a delimiter and assume that this character is never in the user name or service name. If you choose this solution please make it clear in your report.

The server now needs to send the replies to the clients' named pipes and not on the terminal. The client script needs to receive what's been written on the client's named pipe and process it accordingly.

Figure 1 shows a simple architecture diagram of the communication between processes using named pipes.

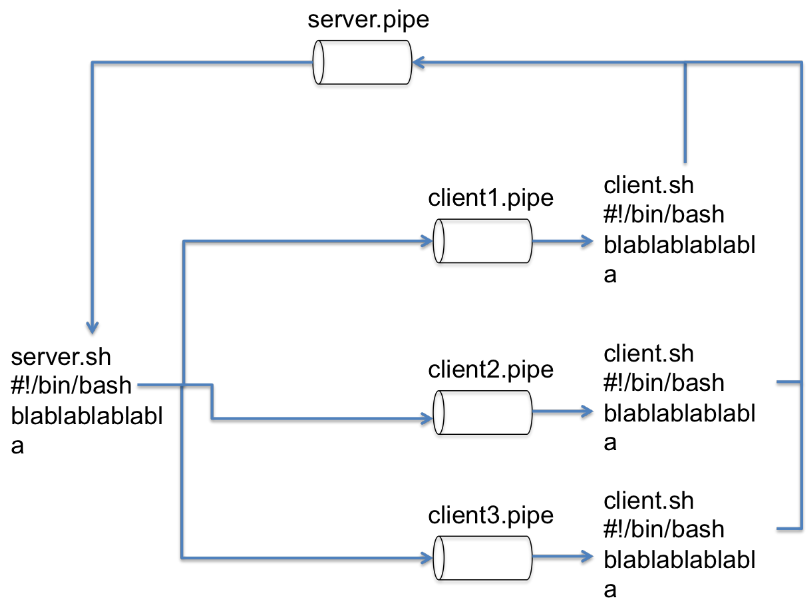
server.pipe
client1.pipe
client.sh #!/bin/bash blablablablabl a
server.sh #!/bin/bash blablablablabl a
client2.pipe
client.sh #!/bin/bash blablablablabl a
client3.pipe
client.sh #!/bin/bash blablablablabl a

Figure 1: This diagram details the communication architecture of your application.

Conclusion

This is a complex project – yet every component is made of simple scripts/commands that you are/will be familiar with. As for every large project, do not be scared by the complexity but try to understand the overall picture and start coding small parts that you understand. For instance Section 2 should be ok so start with these scripts and focus only on them – later on you can integrate small pieces together, they should make more sense then.

Good luck!

Bonus

If you finish the project early (or if you got addicted to writing bash scripts ;)), there are up to 10% bonus points available.

To get bonus points, implement an extra feature for our password manager system and document it in your report: what it does, how to use it, etc. The feature should have some substance to it, printing "Welcome to the client" when starting the client is not a new feature worth extra points.

Examples of features you could implement are:

- Encrypting the payload in the client before sending it to the server, and decrypting it when retrieving it. An encrypt script and a decrypt script are given on brightspace.
- Adding a generate command to the client that generates a random password and stores it in the server.