Assurance Document

`Jammed` Final Release (3.0)

5.5.15

I. UserData

A. Description: This Class securely encrypts/decrypts the users data on the local machine, with a secret Key which remains only on client machines, as well as obfuscating the locally stored key with a user supplied login password. It contains the functions that deal with the actual manipulation of the user data in terms of an arraylist. This class is responsible to reading and writing the user keys to and from a specified location on local machines or usb drives.

B. Testing: To test this class a program was made which using print statements to check functionality does the following operations.

1. creates and stores a Key to the local machine (enrollment)
2. takes an input which simulates a "userdata" and passes it through encoding
3. passes encoded data and IV used to a variable
4. loads the key from the machine.
5. uses the key and IV to decode the encrypted user data
6. displays decrypted data
7. Additional functional testing was done to make sure that files were reading and writing to the proper locations.

C. Additional testing was done on the portion of the class that transforms a list of objects representing user data into a string representation of that data for storage in a database, and vice versa.

1. `stringToList`: This method was tested by creating both valid and invalid input strings by hand and running the function on them. Their validity in list form could be verified by simply printing out the values in the list.
2. `listToString`: This method was tested by manually creating a list of LoginInfo objects and calling the method on it, assuring us that the method would execute correctly on arbitrary cases.

3. Tests were also performed to ensure that the function `listToString(stringToList)` was an identity function on a string, ensuring that our method would always be able to properly parse its own output.

II. Request and Subclasses

A. Description: `Request` is the Superclass of all `Request` objects, which act as a message packet, for sending information between the client and server. Subclasses are divided by type of request; `login`, `log`, `userdata`, `account deletion` and `termination`. Each request has a corresponding response so that the issuer can establish a success or failure of their request.

B. Testing: This was done in place with actual transmission through the session between `Jelly` and `Jammed`. Since these classes do not have functions but primarily are holders for data, no real testing was necessary.

III. UserInterface

A. Description: This is the class that displays the (human-readable) user data and allows the user to make changes to it. This class simply prints to and reads from the command line; however, in order to facilitate the integration of a GUI, it has been made into its own class. UserInterface and Jammed are the paired classes that function for the command line application.

B. Testing: As this is an interactive class, a test program was constructed that would allow us to interact with this class independently of other classes. We verified that the data was output in a correct and readable fashion and that changes input to the data were correctly saved and returned.

IV. Jammed

A. Description: This is the main class for the client application. It consists of a main method which prompts the user for a file location to access/save the user keys, enrollment or login information, verifies this with the server, and requests/displays the user's data. If the user chooses to make changes to this data, it then uploads the new version of said data to the server for storage.

B. Testing: As this component is very difficult to unit test, testing for this was done by running it on a variety of inputs and user behaviors, and seeing how it behaved.

V. LoginGUI

A. Description: This class displays the login GUI for the client application. It allows users to enter the username and password, and select a directory to store their key and IV. It allows users who have entered valid input to login or register to create an account with the server. Upon successful entry, this GUI closes so that the MainGUI class can open.

B. This component is difficult to unit test, so testing was done by running the interface and ensuring that button clicks caused the correct actions to take place.

VI. MainGUI

A. Description: This class contains the code for the GUI for the main application, which allows users to interact with their data after they have received it from the server by adding, deleting, and modifying data entries as well as changing the password to or deleting their account. It allows users to input information pertinent to the actions they want to take, such as the data to be modified or a new password.

B. This class was tested by creating a main method that would display the GUI and allow it to be interacted with in order to ensure that the button clicks and data entry fields worked properly.

VII. GuiJammed

A. Description: This class contains the code that maintains the login and main GUI during the run time of the client application. It also maintains the connection with the server application and acts as the bridge between the GUIs and the server so the client can store their information on the server, change their master password, get their log, or delete their account.

B. Testing: This class is difficult to unit test, testing was done by running it to ensure it properly maintained the GUIs, accepted requests from the GUIs and passed them along to the server, received a response and then relayed the response to the GUI as necessary.

VIII. Jelly

A. Description: This is the main class for the server application. It consists of a main method which creates the server and database, and initializes all SSL related configurations, reading `server.config` and changing any default values as needed.

B. Testing consisted of running it with `Jammed` and ensuring that basic functionality was present. Additionally, response type and data correctness was checked to

ensure the server was sending back correct information, later confirmed again by checking after reception on the client side.

IX. Communication

A. Description: This class acts as a connection handler thread for the multithreaded `Jelly` server. It and `Jelly.java`'s functionalities were shuffled around in order to create a stable server capable of multiple client communications at a time. The server state machine is now found in the run method of this class.

B. Testing consisted of running `Jelly` and `Jammed`, debugging as needed.

X. ClientCommunication

A. Description: This class acts as an SSL wrapper class for `Jammed`, and is used for all client side communication. After initializing any SSL configurations needed, it abstracts sending and receiving through Object streams.

B. Testing of this class consisted mainly of running `Jelly` and `Jammed`, debugging as needed.

XI. DataBase (DB)

A. Description: This classes serves as a utility class for the server application. Its purpose is to abstract file management away from the server, and provide methods for initializing the file tree, reading and writing files, creating new users, searching for the existence of a user, and reading and writing user specific files. Because there is no instance information being maintained, and due to the utility nature of this class, it was decided that it would be a static class. This class was written by `mvp34`, and reviewed by the other members of the group.

B. Testing was performed on an individual method basis, a main test method in a junit test class was used to make method calls that would expected from the server, and the output was observed directly through the directory structure, and by comparing the data that was read with the data that was written to ensure that they were equivalent. Tests were run after any change or new feature was implemented.

1. `public static boolean initialize()`: This method was tested by calling it in the main method, and then checking the directory structure to ensure that the proper directories and files were initiated. This method was written under the assumption that the server would only ever call it once upon initial install, but was tested with multiple calls nonetheless. Upon multiple calls, the method simply returns true without any action, because it checks that the file structure already exists.
2. `public static boolean newUser(String uid, String uPWD)`: This method was tested by calling it in the main method with a

sample input and then checking the file structure for the creation of the new user folder, along with a file containing the bytes of randomly generated salt and the hashed uPWD. Then it was tested by calling it again with same input to ensure no action was taken because the method checks to ensure that no user already exists with user id input.

3. `public static boolean searchUser(String uid)`: This method was called with sample input to ensure that it returned true if the given uid had already been created, and false otherwise.
4. `public static boolean deleteUser(String uid)`: This method was tested by first creating a user folder and the files associated with that user, and then calling the method and checking to ensure that the files and the directory were deleted.
5. `public static boolean writeServerLog(String dataToLog)`: This method was tested by calling it with sample input and ensuring that it appended data to the log by viewing the log file after it had been written to.
6. `public static String readLog()`: This method was tested by calling it in the junit test framework and comparing it to the expected output.
7. `public static String readUserLog(String uid)`: This method was tested by ensuring that it read the data that had been previously written to a specific user log, and returned null otherwise, such as a non existent user.
8. `public static boolean writeUserLog(String uid, String fileData)`: This method was tested by writing sample data to a user log, and reading the data back to make sure that it was recorded correctly. The method was also tested against bad input to ensure it performed no action and returned false.
9. `public static byte[] readUserData(String uid)`: This method was tested by reading data that had been previously written, and ensuring that the information matched. It was tested to ensure that no action was performed if the user did not exist, and that the correct files were read from.
10. `public static byte[] readUserPWD(String uid)`: This method was tested by reading data that had been previously written, and ensuring that the information matched. It was tested to ensure that no action was performed if the user did not exist, and that the correct files were read from.

11. `public static byte[] readUserIV(String uid)`: This method was tested by reading data that had been previously written, and ensuring that the information matched. It was tested to ensure that no action was performed if the user did not exist, and that the correct files were read from.
12. `public static boolean writeUserData(String uid, byte[] fileData)`: This method was tested by writing sample data to each of the file types and then reading it back to ensure it was written correctly. This method was tested on bad input, such as a user not existing.
13. `private static boolean writeUserPWD(String uid, byte[] fileData)`: This method was tested by writing sample data to each of the file types and then reading it back to ensure it was written correctly. This method was tested on bad input, such as a user not existing.
14. `public static boolean writeUserIV(String uid, byte[] fileData)`: This method was tested by writing sample data to each of the file types and then reading it back to ensure it was written correctly. This method was tested on bad input, such as a user not existing.
15. `public static byte[] getNextSalt()`: This method was tested by calling it a million times and ensuring the outputs were 16 bytes and different.
16. `public static byte[] hashPwd(String pwd, byte[] salt)`: This method was tested by ensuring the output was the result of hashing pwd with salt using PBKDF2WithHmacSHA1 and 20000 iterations and a key length of 512 bits.
17. `public static boolean storeUserPWD(String uid, String pwd)`: This method was tested by ensuring that pwd was hashed and stored inside a correctly named file inside a directory of name uid.
18. `public static boolean checkUserPWD(String uid, String givenPWD)`: This method was tested by creating a user and storing a hashed password and salt, and then attempting to authenticate with various incorrect passwords and then a correct one, and ensuring it returned true only when the correct password was given.