PassGuard

TESTING REPORT

TESTING AND RESULTS

TESTING AND RESULTS

SOLUTION VS DESIGN SPECIFICATIONS

The completed solution very well matched the initial design specifications and in fact went further by adding additional functionality. Comparing the initial planning and designing of the solution to the final product, slight changes were made. Changes that were made were due to having to adapt the program to comply with swift apps written in swift and designed using Xcode.

The first major change includes the inclusion of @AppStorage to store all values as opposed to the previously planned Firebase database. Not only was this change made for security purposes (as sensitive data is safer by being kept locally on the device), as well as compatibility and simplicity. An error that arose from this is that @AppStorage is only able to store primitives (e.g. String).

This required the creation of an extra 4 algorithms that were needed to convert and Array of Accounts and Cards into String representations of those to Arrays and convert them back. In this way, @AppStorage was treated as a sequential file and included delimiters of a “\n” as this is not tapeable by the user.

Apart from this, smaller additions such as functions to check whether or not a particular string was valid for specific input were included later on in the development process.

GENERATING TEST DATA

Test Data was generated based on what algorithm was being tested. The aim of the test data being used was to test all possible cases and inputs that it would receive, and to see if the algorithm would fail.

This included volume testing to see if a very large input would cause an error if the algorithm was not able to handle such large input. Along with volume testing, path testing was used to see all the possible paths that could be taken by an algorithm based on what inputs were entered.

For example, the Check_Valid() function is very specific in that it checks if a string conforms to rules that are required to be followed by PassGuard as a part of data validation. In this particular case, strings can only be a valid if they only contains characters that have an ASCII value between 32 and 126 inclusive.

Looking at this example, we see that it is important to test this algorithm with inputs that test the boundaries of the algorithm which are in this case 32 and 126. To do this, multiple test values (strings) can be used which contain all valid characters as well as a single character with and ASCII value close to the boundary.

It is important to also keep in mind that there should be a test value with expected input to serve as a control and see that the algorithm functions properly under normal conditions.

ACTUAL VS EXPECTED OUTPUT

After planning and testing the algorithms in pseudocode, the expected output did meet the actual output on most occasions when they were re-written in swift. Sometimes minor changes were needed when converting pseudocode into actual swift code, due to swift conventions and for the sake of efficiency.

One particularly crucial comparison of expected vs actual output was the Find_Strength() function which functioned fairly unexpectedly when implemented into the application. Although the test data in the algorithm tested to ensure that the algorithm was free of bugs, and could handle any input thrown at it, it did not provide a very logical response when it was pushed to the rest of the project. Because of this, the Find_Strength algorithm had to be revisited and modified to better reflect how the function should function.

Another issue that arose only after the implementation, was the Encrypt() and Decrypt() functions. Again, similar to the Find_Strength() function, when planning, and desk checking the algorithms, no issues were found regarding the functioning of the algorithm as it functioned as expected and was able to handle any input that was thrown at it. It was noticed only after observing the contents of the @AppStorage within the application, that it would be easy to identify that certain fields were encrypted, meaning that if accessed by the wrong person (e.g. cyber criminal who has gotten access to the device) they would be drawn to these fields and attempt to decrypt them specifically.

Because the encryption algorithms are quite complex, this was left to be revised for a later version, as so was kept the same for Passguard 1.0.

LEVELS OF TESTING

Varying levels of testing were conducted at different stages of the development process. It is important to note that PassgUard was not developed using a traditional Top-Down approach, but rather a Bottom-Up approach.

This was done by creating individual algorithms first, seperate from the rest of the program. Here module testing involved testing the function by itself with the user of a driver to replace the to-be-completed higher level routine. When the function has been fully tested and ready to implement, the function would be pushed to the main program.

From here, system testing would be carried out to ensure that when including the new function, the program would continue to run as expected, and would not cause any errors. If an error appeared during this stage, the function would be removed from the program and analysed again before trying again. The use of stubs and drivers were also used to attempt to locate the exact origin of the error and what was causing it.

Once all functions have been developed and tested (using the previous module and system testing methods described above), program testing would commence. This is to test that the entire program works as expected with all functions and coded included. During this stage, both black-box and white-box testing was implemented to test that the function ran as expected both from the users prospected as well as internally.

LIVE TESTING

Live testing was carried out with the use of the Xcode simulator (iPhone 13 simulator as this is what Passguard is designed for). During the entire development of PassGuard 1.0, another 2 versions were created for planning and feedback purposes.

The first being PassGuard (Alpha) which was written using swift, however, did not include a GUI and had a simple CLI. This was deemed most suitable as this allowed for easy and quick testing of functions that would later be implemented in version 1.0. PassGuard (Alpha) also allowed for more focused analysis of logical functions within the application that did not have any UI components. This is where most of the logical functions were tested.

PassGuard (Beta) included all the functionality from the alpha version as well as some additional features, however, the largest change was the inclusion of a GUI. This meant that the program could now be testing in the Xcode simulator from now on. This is also the time when new functions that were a direct cross between swift and swiftUI such as the find colour algorithm was tweaked for efficiency.

Finally, after receiving feedback regarding the UI from the beta version, PassGuard 1.0 implemented this feedback and made changes accordingly. This was also the version that managed to resolve many previous minor issues regarding unappealing screen navigation and interaction. Along with this, the source code was cleaned up and better documented which showed immediate results as when it was tested in the Xcode simulator, the application ran much faster due to the modified code.