Table 1: Revision History

Date	**Developer(s)**	**Change**
October 26, 2017	Thomas Mullen	Rev.0 of Document
...	...	...

3XA3 Requirements Document

Group 20 (2020Vision)
Mullen, Thomas - mullentc - 001406837
Pavlich, Phillip - pavlicpm - 001414960
Bauer, Ivan - bauerim - 001418765
L02

1 Introduction

Testing is a crucial step when building a project. It ensures that the structure of the source code is sound. It verifies the each module of the code meets its functional requirements and is able to communicate with other modules to accomplish a specific goal. Failure to perform a structured set of tests for a project can result in unforeseen failure and can potentially harm the stakeholder. The engineers responsible for the design on the project are responsible for the consequences that arise from poorly tested source code. The sections below break down all the different forms of tests that we are performing on our project to ensure correctness and functionality.

2 Static Testing

Static testing involves performing test cases without any program execution. In this section, we will be verifying the structure of our code, verifying proper coding styles, insuring that there are no syntax errors, and confirming the standard setup of the folder system.

Folder System: Must be broken down into an icons folder for the chrome extension icon, the src folder for the source code and the manifest file for building the chrome extension. Inside the src folder, we must have each module broken down individually. We have three different modules: browser_action which contains all code for the user interface, module which contains hashing algorithms and properties for the back end of the code, and options which contains the source code for the options and preferences page. Inside these folders, there is a folder to contain any JavaScript files, another folder for any css files and all html files are in the source folder.

We will be reading all new code additions verifying there is not any syntactically incorrect pieces of code.

Finally, we will be reading over all code changes to make sure that camel case is being followed and proper JavaScript conventions have been met. The same rules should be used across all scripts.

3 Dynamic Testing

Dynamic testing requires the program to be executed. Tests are preformed during execution of the program to verify the methods and functionality of the code. For this section of testing, we will be using Tape. Tape is a behavior-driven development framework for testing JavaScript code. The JavaScript scripts will have their methods tested to ensure that the functions are returning the correct output for the function.

We will be required to perform exhaustive testing to ensure all inputs for these methods function as expected. Criteria will be chosen wisely for representative test cases to meet all possible boundary conditions and common scenarios that may arise.

4 Logical Structure

4.1 Unit Testing

Unit testing involves finding the differences between the design model of a unit and its corresponding implementation. We have modularized the code so that each section performs a specific section. We have an options/preferences section, a user interface section and a back end section. All three of these sections require unit testing to verify that the design for our project is matching the implementation for the source code. Below are examples of some test plans that we will be implementing. For our test plan coverage, we provide a unit with an input and compare the actual output to the expected output. If these outputs match, the unit test has ensured that the design matches our implementation.

4.2 Integration Testing

Integration testing is required to test the composition of the system components.

To verify that the software meets integration requirements, the automated unit tests in section 4.1 must be able to execute and pass in the 4 major browsers (Chrome, Firefox, Safari, Edge) as well as a Node.js V8 environment.

Table 2: src/browser_action/js/browser_action.js

Test Types	Manual, Black Box, Functional, Dynamic
Unit or System	Unit
Test Factors	Correctness, Compatibility, Reliability
Initial State	Extension in deployed in Google Chrome.
CASE 1:	
Input	"Show User Key" UI element is toggled
Output	User key field content is hidden/shown.
Schedule	See Section 10

4.3 System Testing

System testing ensures that the whole system complies with the functional and non-functional requirements.

The majority of the system testing involves manual tests of the User Interface. These may be automated if resources permit it, but it is not required.

4.4 Acceptance Testing

Acceptance testing allows the customer to verify all the requirements to ensure that the system has satisfied these. It is the formal description of the way a piece of software behaves. Black box testing will be the method that is used for this process.

A third party belonging to the "user" shareholder group will be brought in an briefly explained the usecase of the software. The user will then be allowed to experiment and then interviewed on whether the software would meet their password storage needs.

5 Manual Versus Automated Testing

Testing code consistently to ensure correctness can be very exhaustive and time consuming. Our goal is to have the back end testing automated so that upon execution, we can verify the outputs of every method. This saves us time and these test cases can be used every single time we push a new commit to be merged with our master branch. The user interface and graphics will

Table 3: src/module generatePassword method

Test Types	Automated, Black Box, Functional, Dynamic
Unit or System	Unit
Test Factors	Correctness, Compatibility, Reliability
Initial State	Node.js module src/module is started in a V8 environment.
CASE 1:	
Input	Master password string and domain string
Output	Generated password.
CASE 2:	
Input	Two strings and a constraints object
Output	Generated password that follow constraints.
CASE 3:	
Input	strings of all unprintable unicode characters.
Output	Generated password.
CASE 4:	
Input	Two non-string objects.
Output	Exception thrown "TypeError: First argument must be a string, Buffer, ArrayBuffer, Array, or array-like object.".
CASE 5:	
Input	Two strings and two seperate calls to generatePassword.
Output	Two generated passwords that are identical.
CASE 6:	
Input	Two strings and constraint object with length 1000 (testing iterative algorithm).
Output	Generated password with length 1000.
Schedule	See Section 10

Table 4: src/module setOptions method

Test Types	Automated, White Box, Functional, Dynamic
Unit or System	Unit
Test Factors	Correctness, Compatibility, Reliability
Initial State	Node.js module src/module is started in a V8 environment.
CASE 1:	
Input	Valid complete options object
Output	Nothing returned and module._opts is now the object.
CASE 2:	
Input	Object with unknown option names.
Output	Unknown options are NOT added to module._opts.
CASE 3:	
Input	Valid but incomplete options object.
Output	module._opts has default values for all missing options.
CASE 4:	
Input	null.
Output	Exception thrown "TypeError: Cannot read property 'test' of null".
Schedule	See Section 10

Table 5: src/module setSalt method

Test Types	Automated, White Box, Functional, Dynamic
Unit or System	Unit
Test Factors	Correctness, Compatibility, Reliability
Initial State	Node.js module src/module is started in a V8 environment.
CASE 1:	
Input	String
Output	Nothing returned and module._salt is now a Buffer containing the input string.
CASE 2:	
Input	Buffer.
Output	Nothing returned and module._salt is now the Buffer.
CASE 3:	
Input	null
Output	Exception thrown "TypeError: First argument must be a string, Buffer, ArrayBuffer, Array, or array-like object."
Schedule	See Section 10

need to undergo manual testing as it is a preference. The look of the user interface must be easy to work with and easy to understand.

6 Code Coverage

For this project we will be using the Node.js code coverage tool "covert" to provide code coverage methods.

6.1 Equivalence Testing

Equivalence Testing is a black-box testing method. We will use it to divide the space of all possible inputs into equivalence groups so that the program will behave the same for each group. This process will be done in two steps. First the values of input parameters will be partitioned into equivalence groups. Then the test input values will be chosen. For our input parameters we will partition the value space into one valid and one invalid equivalence class.

6.2 Boundary Testing

Boundary Testing is a special case of Equivalence Testing which focuses exclusively on the boundary values of input parameters. Elements from the edges of the equivalence class will be selected, including zero, min / max, empty values, and null.

6.3 Control-Flow Testing

Control flow statements involve covering all possible paths that a user can navigate through and ensuring that everything flows in the system as expected. All statements that can be reached through some path of execution must be tested to verify that a user doesn't perform some action that results in a program failure. For our project, we will write tests to test the path to the options page and back to the user interface. We will verify that the info button displays a description of how the application functions so the user is aware of how to use it. There will be a test to make sure that the user can write into the user key text box. There will be a test to confirm that the user cannot manually alter the password. These are the paths of navigation that are addressed in the control-flow testing phase.

7 Functional System Testing

Functional system testing involves performing tests to determine if the system has satisfied all requirements and that it can perform its function. Our test case involves attempting to utilize the Chrome extension by providing the system with a user key and attempting to obtain a unique password that follows all preferences that have been given in the options page. Changes to the preferences section will be done to ensure that the outputted password matches the preference requirements.

8 Parallel Testing

The previous implementation "HashPass" does not have any tests. Due to the password generation implementations diverging, output will not be identical and so the two projects cannot be directly compared. However,

from a non-functional level the two can be compared. Performance, security and usability metrics should be compared between the two projects.

9 Proof of Concept Test

The proof-of-concept test consists of the following steps.

1. Install the extension in Chrome.
2. Navigate to a webpage.
3. Open the options page of the Chrome extension.
4. Set some password constraint options.
5. Open the browser action of the Chrome extension.
6. Enter a master password and copy the generated password.
7. Verify the password has the desired constraints.

10 Test Plan Schedule

Please refer to figures 1&2. This contains the complete outline of the testing schedule, including deliverable task IDs and milestones.

Figure 1: Schedule of Test Deliverables

Test Document	Objective	ID	Deliver Date
Test Plan - Revision 0	Set of initial requirements for testing PretzelPass	0	Fri. Oct 27, 2017
Test Plan - Revision 1	Versions of the Test Plan.	1	November 27, 2017.
Test Report - Revision 0	Report tracking progress, history and results of the test cases as they are being performed.	2	November 27, 2017.
Final Documentation	A report documenting requirements, design and testing for PretzelPass. Information from Test Report - Revision 0 may be used.	3	December 6, 2017

Figure 2: Schedule of Test Cases with Milestones

Milestone 1: Test #	Doc ID	Team Members	Comments	Date
1.1.1	1,2	Thomas Mullen	Unit Testing	Oct. 29, 2017
1.1.2	1,2	Phillip Pavlich	Integration Testing	Oct. 30, 2017
Milestone 2: Test Case #	Doc ID	Team Member	Comments	Date
1.1.3	1,2	Ivan Bauer	System Testing	Nov. 1, 2017
Milestone 3: Test Case #	Doc ID	Team Member	Comments	Date
1.2.1	2,3	Thomas Mullen	Acceptance Testing	Nov. 6, 2017
1.2.2	2,3	Ivan Bauer	Manual and Automated Testing	Nov. 8, 2017
1.2.3	2,3	Phillip Pavlich	Manual and Automated Testing	Nov. 8, 2017
Milestone 4: Test Case #	Doc ID	Team Member	Comments	Date
1.3.1	3	Thomas Mullen	Equivalence Testing Testing and Proof of Concept Test	Nov. 24, 2017
1.3.2	3	Phillip Pavlich	Boundary Testing and Proof of Concept Test	Nov. 24, 2017
1.3.3	3	Ivan Bauer	Control Flow Testing and Parellel Testing	Nov. 24, 2017

11 Conclusion

This concludes the components of our test plan. This document has helped our group discuss the objectives and requirements that must be achieved. We have explored the possible boundary conditions that that we may run into as we are testing our project. Using this test plan, we will be able to create a series of tests to ensure that our project has satisfied all functional and non-functional requirements.