Test Plan

DAC Project

CS 448

Austin Bos, Aaron Saucedo, Ivan Nieto

Test plan 001 for notetaking application requested by the Army Data and Analysis Center

Date

04/15/2020

Introduction

The goal of this project is to develop an application that aids the CCDC-DAC analysts to collaborate and incorporate each other's work when constructing reports on the cyber vulnerabilities found for an investigated system or location. This application will not only merge the findings of the analysts into a single report but also help organize the investigations by allowing the lead analyst to assign tasks to the other analysts.

This document will explain how we will test our application. We will need to test that our application is reliable, user friendly and optimize to run without error. The system is segmented into different parts, the frontend, which allows navigation, access to the database and user input, and the backend which will implement the syncing or communication interface.

1.1 Major Testing Issues

Since our system is a desktop application it will require a lot of user interaction. The application is installed as an executable and should not require a lot of work to set up, besides the LAN connection. The pages must be reliable, simple to navigate, and be able to be updated in real time with findings and other needed information. Another issue is security, it should run on a secured LAN connection between machines (leads and members). However, due to our current situation this test might not be available as we are now working remotely and have no access to our CS lab or any other resources. Also, the application should be able to sync data between the team lead and team member(s) (lead should sync to all members or a single one). The application is straightforward and needs to be easily accessible by the analysts. Lastly, the application should be run on Linux operating systems.

With these issues in mind testing overall will be complicated because of the large number of individual components that make up the application. For instance, the application itself will be created using React wrapped inside an Electron process. This means that we can test the React components independently of the Electron application fairly easily, but we will not be able to test the same components once they are integrated into Electron. The same can be said about the database. Since the database is completely dependent on the node server that operates it, all testing related to the database will be focused on the interactions between the database, server, and frontend and not the database exclusively. Furthermore, Electron does not have a testing

software so the only way we will be able to test the final build of the application will be by hand through a simulated use of the application as the product's customers might do.

Testing Strategy

We will be using a combination of unit testing and manual integration testing in our project. This is because we have components in our application that cannot be tested programmatically. For this reason, we will be testing components individually and doing integration testing wherever possible with the help of the Jest testing library. This library allows us to test all code in our project written in JavaScript, which is the frontend and backend. With Jest we will be able to get good validation on the accuracy and reliability of our program. Jest testing will mock different aspects of the UI such as button presses and user inputs and then test the results of these changes.

2.1 Frontend Testing

We will be conducting Jest tests for the frontend and backend. Jest testing will allow us to unit test each individual component as well as do some minor integration testing. Since the database is coupled to the frontend, it can be tested simultaneously to an extent with the frontend.

Jest testing will consist of calling functional components from the frontend and comparing their return values to predetermined expected results. For example, if a component that should render a button is being tested the expected return value of that component would be a button. A return value of any other kind for this test would result in a failed test. This will be the overall strategy in testing our functional components and our backend functions.

2.2 Communication Testing

For the communication testing we had planned on working with the computers in the CS lab to connect through a LAN connection and test the sync functionality. As stated earlier, we do not have access to the CS lab so testing for this section will have to be done using only the virtual machines. To ensure data integrity, we will use sample data that is predetermined, and we know. This will allow us to ensure data integrity through the syncing functionality. To test this, we will

check data before it is sent, and again on the receiver side to ensure all data is not altered, or lost in any way in transit. To see security of communication, check section 2.3.

2.3 Security Testing

Since we do not have much experience with pen testing we will not be testing the security of our application but rather we will keep the security of the information on our application in mind when deciding on any design decisions for the application. The only exception to this would be in our testing of secure connections between team members and the team lead. This is less of an audit, as much as it is ensuring there are not obvious flaws in the implementation. To test these, we will ensure proper tls connections are made. If the customer does decide they require mutual authentication between connected computers, we will make sure this is working. To test this, we will attempt to create bogus connections, invalid key pairs used, and attempt simple man in the middle attacks. The first test, for bogus keys, we will use a random key pair, not associated with any team members, and make sure the connection is rejected. Testing man in the middle attacks is negligible from this point, because if a bogus key pair cannot be used, there can be no spoofing. Creation and sharing of key pairs are out of the scope at this point in time for us, so we assume the customer has a robust mechanism to safely share public keys with each other before starting the application. Since ports will be open on the user's computer temporarily, it will only accept valid tls connections as mentioned above. The only testing to be done here is to ensure there are no known vulnerabilities that may be exploited through a gRPC port. We will do this through a search for any known vulnerabilities against the version of gRPC we ship.

Test Environment

Our system is a desktop application that will only be accessed by the DAC analysts, and must run within a secure LAN connection. The system must be tested within a Linux operating system as according to our requirement documentation. For communications, and synchronization mechanisms, unit testing, hopefully with support for regression testing will be used. The tools have not been decided on yet for this. The idea is to set up many unit tests for all synchronization, and communication functionality and put them into small, unit tests. The regression testing will be done at all stages of testing, to ensure new features, or bug fixes do not break already working code.

Performance and Stress Testing

Our web application will only be accessed by a small group of analysts, so we do not expect any overload or stress to our application. Despite this, there will be periods of heavy read and writes from the database, as these will come from the synchronization of data. For this testing, we will simply create a large sqlite3 database, and start pulling as much data out as fast as we can. The data will be pre-determined so we can verify the integrity of the data as it comes out. Then we will do the reverse, where we stuff as much data into an empty database. Again, the data will be predetermined, so after we can go through the database, pulling out data, and verifying its integrity.