12SDD 2019
Dylan Lom

Major Project Report

Project Outline

PassMe is a password manager, using an Ethereum Smart Contract as a backend/server.

The goal of *PassMe* was to create an alternative to current monthly-payment model services, and give the user the reassurance that their data has been immutably stored on the blockchain.

I developed a solidity Smart Contract, which communicates with the Ethereum blockchain in order to store and retrieve user data, based on the existing NoteChain[1] smart contract. I also developed a web front end for users to read and write passwords from, using HTML, CSS and JavaScript.

PassMe was developed with a more agile approach in mind, based on the fluctuating nature of web applications (and their wide audiences), however the smart contract was developed in a more structured approach, since it needed to be a sturdy basis for the front-end, and specifications shouldn't change for it.

[1] https://notechain.github.io/

Interface Design

Target Audience

When developing a software solution's design specifications, it's important to consider the target audience's requirements, in order to make sure that your solution is adequate and appealing to the client. Since a password manager is a piece of software that all users should be encouraged to use, I tried to make my program as simple to interact with as possible.

However, since *PassMe* is still relatively juvenile, and also depends on the in-development Ethereum blockchain and Web3 library, the project remains targeted at more technical users, who are somewhat familiar with Smart Contract development process and blockchain transactions.

One example of such introduced complexity is the manually-input contract address, which would appear confusing and unclear to users unfamiliar with how smart contracts work, however this was included to enable development on various test networks, while the PassMe contract isn't permanently deployed on the mainnet.

Ergonomic Issues

Ergonomics in a software environment, relates to the relationship between the end-user and the software solution, through the user interface. By considering the target audience's needs, developers can create an interface that is easy to use and effectively allows users to interface with functionality. The interface should be intuitive, follow standards and a simple set of design rules, correctly use interface elements (such as text inputs, checkboxes, etc.), appropriately use colour and font, align data entry elements and be consistent.

The interface of *PassMe* aims to be as simple as possible, allowing unacquainted users to be presented with usage information in small 'pages', each of which performs a simple task (e.g. searching saved passwords).

The visual design is simple and monochromatic, aiming to be compatible with the current web standards (HTML5, CSS3) and popular web browsers (Google Chrome, Mozilla Firefox and Opera). Since the user interface has very few interface elements, very little differentiation between elements was required; one example where it was used was to differentiate navigation buttons (grey, flat as defined by CSS) with confirmation buttons (white, browser default). Text input and button styling is kept consistent through the use of element and class based CSS

styling rules. The interface also makes limited use of icons for ‘action’ buttons, which follow common iconography standards (ie. a ‘bin’ to delete a password).

Since the target audience isn’t yet the general public, some level of manual interaction remains in the application — along with the contract address input, another of this is the requirement of manually requesting that the user’s vault be updated; while this could be automated via Solidity events, this was not a priority in early versions of the project, and can be added to later version of *PassMe*, when public interest develops, an advantage of taking an agile approach to development.

Since general practice for passwords is to allow only a subset of the ASCII character set (letters, numbers and ‘symbols’) *PassMe* encrypts with the assumption that user input is in ASCII. If the input is not a part of the ASCII standard, the program will not fail to execute, however the data *will be malformed in the encryption process and potentially lost.*

Program Development and Techniques

Correct Code Generation

I thoroughly tested code throughout the development process to ensure that it ran correctly.

Modularisation of Code

Code modules allow for developers to create reusable, self-contained procedures, simplifying processes of updating and maintaining software.

Both the Smart Contract and JavaScript web frontend of *PassMe* employ functions to reduce repetitive code, simplify processes and make the programs more reusable. In addition to functions, the front end is broken into individual files, isolating modules based on their different purposes (ie. the *crypt.js* file contains functions only relating to the encryption and decryption of user data). This also increases the re-usability of functionality, making it easier to import groups of modules into other projects.

Effective and Efficient Use of Control Structures

Control structures form the basis for all computer algorithms, the three basic forms are sequence, decision and repetition.

Sequence relates to the order in which modules are run, which is mainly controlled by the user's interaction with various buttons and inputs (as well as the linear structure in which the browser interprets the JavaScript), which execute various modules which modify the appearance and send/retrieve information from the blockchain. The frontend also uses the JavaScript *promise* object, which allows for asynchronous actions to be awaited on, this is mainly relevant when interacting with the Ethereum blockchain via the Web3 library, which returns promise events on function calls.

In JavaScript decision translates into the *if...else if...else…* structure, which is used throughout the program for decisions based on the state of various internal variables and conditions.

Common repetition structures in JavaScript are the posttest *do...while*, pretest *while*, counted pretest *for* loop, the latter of which *PassMe* utilises at various points, mainly for the purposes of iterating through arrays, to retrieve, interpret and display information. An important application of counted loops was in the master key derivation function, which needed to be repeated 10,000 times, in order to derive a key that was more resilient to brute-force attacks, while not taking so long that older hardware wouldn't be able to perform the calculation.

While *while* and *do..while* loops can be useful control structures in more interactive applications, such as games, where quick screen-refreshing or AI control modules need to repeat infinitely until a condition is met (ie. the game is quit), *PassMe* only needed counted loops, since user-interaction is relatively limited. For this reason *while* loops were not utilised in *PassMe*.

Effective Use of Appropriate Data Structures

Appropriate use of data structures increases program efficiency and can help to minimise introduced errors into a program. In *PassMe's* smart contract, this is important as inefficient use of data structures can increase cost to the user and impact on the blockchain's performance. For the web front-end, JavaScript's weakly-typed, object-based language means that choosing the relevant data-types is not required, as the interpreter does this for me.

In addition to standard Solidity and JavaScript data types, *PassMe* also uses a couple of purpose-built structures for storing data. Two examples of which are:

1. HTElements (web)
    An object of objects (similar to a multidimensional array) created from the HTML's DOM. The purpose of this structure is to reduce repetitive document.getElementById *function calls in the program.*
2. Pass (smart contract)
    A data structure in the smart contract, which is used to store the user's information. It contains…
        unsigned integer: metadata *— which is used to notate whether the password has been marked as 'deleted' (since the blockchain is immutable, passwords can not ever be completely removed, only updated). This can also later serve to differentiate different encryption methods used, so that users could increase/decrease the level of security used on their information.*
        string: href *— which is used to store the domain of the website which the password relates to.*
        string: pass *— which is where the user's encrypted password is stored. PassMe currently uses the AES-256-CBC encryption method, using a master-key derived from a user's master-password and ethereum account address.*

User Documentation

Installation Guide

An installation guide leads the user through the steps required to install and run the software, assuming the user has very limited knowledge of the computer and software.

Since the application is designed for a desktop web browser, the installation guide was made available on the same website. It can be viewed here[2]. There is also a local copy of the HTML included in the submission.

User Manual / Technical Specifications

A user manual is a type of external documentation which presents explanations of the program's features and how to use them. It includes

- Clear explanations of the program's features and how to use them.
- Diagrams and screenshots as illustrations.

Since PassMe has relatively few user-actions that can be taken, a user-manual wasn't necessary, and instead I have written a set of technical specifications about how the user's data is encrypted to protect private information. It can be viewed here[3], again there is a local copy included in the submission.

How to guides / Walkthrough

Tutorials lead a user step-by-step through processes in the product so that users can experience real world use of the application, before personally using it. Tutorials are often split into several 'lessons', each of which cover different areas / processes of the program.

However since *PassMe* is relatively simple, a single video walkthrough was created based off of the setup guide, which includes real applications of using the password storage facilities at the end. The video also includes some basic information about how the blockchain operates, and some password security advice. It is available here[4]. A local copy (webm format) is included in the submission.

[2] https://dylan-lom.github.io/PassMe/docs/getting-started.html
[3] https://dylan-lom.github.io/PassMe/docs/technical.html
[4] https://youtu.be/_7TEOEa1Ly8

Technical Documentation

Intrinsic Documentation

Intrinsic documentation is a type of internal documentation, and is about using a self-explanatory, clear and concise coding style. This includes:

- Use of white space and indentation
- Meaningful variable and function identifiers/names

In *PassMe*, code blocks are indented with white space to help identify what code relates to which module, and where control structures are used.

Internal identifiers (for both functions and variables) are mostly self-explanatory, where relevant further explanations are provided.

Extrinsic Documentation

Extrinsic documentation relates to the inclusion of additional statements, such as comments, within code to simplify more complicated sections of code. Code comments are helpful to inform a developer the purpose of a module or variable without having to read sections of code to extrapolate this information.

PassMe uses code comments to clarify the purpose of modules and variables, and to simplify long expressions.

Project Blog

A logbook allows a developer to monitor their progress, throughout the various stages of software development. Inclusions in *PassMe's* project logbook includes:

- Tasks achieved.
- Difficulties encountered, and subsequent solutions.
- Ideas and thoughts on the project's direction, and any deviations to original design specifications that may be required.
- Changes introduced (and their timing), which can be useful for backtracking when debugging future issues.
- Reflect on progress.
- Upcoming tasks.
- References to resources used to aid development / learning.

The project blog for *PassMe* is available here[5]. There is also a local copy included in the submission, under the *log* directory.

The projects development progress is also partially contained within commit messages recorded by the git VCS, available here[6].

[5] https://dylan-lom.github.io/PassMe/log/index.html
[6] https://github.com/dylan-lom/PassMe

Communication

Agenda for Progress Meeting

- How detailed should documentation be?
- How to perform end user testing (project won't work on school PC's/network)?
- Is a survey enough data for user testing?
- How to do a user manual for my project, based on it's limited user interaction?
- What to write for areas that aren't applicable, (e.g. single line stepping isn't available for a JS environment)?

Minutes of Progress Meeting

- *Detail Required*
    - There isn't a word limit, make sure you satisfy the top band of requirements
- *End User Testing*
    - Try to use laptops, etc. If you can't do a lot, explain why in documentation.
- *Survey sufficient*
    - Yes, make sure you detail the test plan.
- *User Manual*
    - Due to the nature of a password manager, it would be more relevant to write a technical document explaining the security considerations taken.
- *N/A Areas*
    - Demonstrate that you understand what the dot point is referring to, and explain why it wasn't relevant to your project.
- *Other*
    - A video walkthrough of using the project is better than a how-to guide.

Testing the Software Solution

Comparison to Original Design Specifications

At the end of a project, it's important to test it's compliance to the design specifications, which may have been modified during the development cycle. The requirements should be simple to test, with a pass or fail allocation to each requirement.

Over *PassMe's* development cycle, some significant changes to the original design have been changed, as a result it doesn't meet all of the original specifications.

Specification	**Compliance**	**Notes**
Browser extension	Fail	Due to time requirements, and the project's dependency on the MetaMask extension, developing an extension became infeasible.
Accessible to a variety of systems/browsers	Pass	
Ethereum blockchain back-end	Pass	Because *PassMe* is not ready for public use, the smart contract is currently deployed on the Kovan test network, so that real money is not required. Kovan replicates most of the Ethereum network's main features, with the major exception that it uses a proof of authority model (instead of proof of work), meaning that authorised nodes mine all blocks instead of the public.
Automatically scrape websites for password fields	Fail	Since the project was unable to be implemented as a web extension, actions that directly interacted with web-pages were not relevant, as the project runs in an isolated environment.
Autofill saved passwords	Fail	Unmet for the same as above

Level Testing

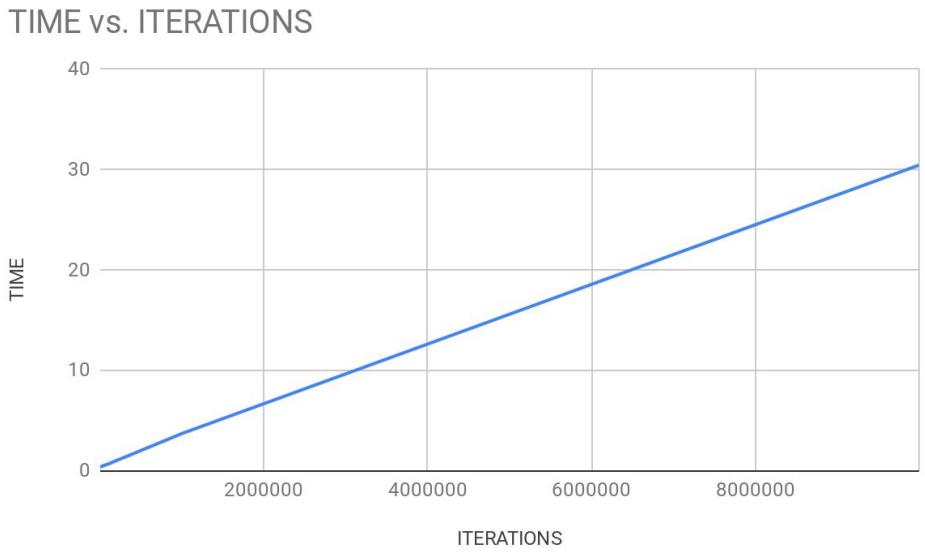
Top-down programs allow for flexibility when testing, as individual subroutines and modules can be tested independently from the main program.

In *PassMe's* development, both module and program level testing was used.

Most module testing was done using either the NodeJS interactive console, or Mozilla Firefox's developer console, where individual modules could be run in isolation. Black-box testing was performed on imported functions from Web3, MetaMask and Crypt, in order to establish that these functions worked correctly without looking into their source code. White-box testing was performed on modules that I wrote for *PassMe*, where I could inspect the source-code and make changes when I had made errors within them. For testing the Smart Contract, I used the Remix IDE which abstracts the Web3 function calls, allowing for simpler testing of the smart contract's implementation.

Below: a table and graph that I produced when black-box testing the PBKDF2 function's time scaling (discussed further in blog post #13)

ITERATIONS	TIME
1	0.423
10	0.408
100	0.414
1000	0.407
10000	0.444
100000	0.711
1000000	3.73
10000000	30.457



Below: a testing table for the addPass function (on the web front-end), where *h(...)* is used to indicate the string is an encrypted copy

Input	**Expected Output**	**Output**
href: “mysite.com” pass: “password”	Pass pushed to passwds array on blockchain with metadata: 0 href: h(“mysite.com”) password: h(“password”)	Expected
href: “” pass: “”	Pass pushed to passwds array on blockchain with Empty href and pass values	Expected
href: “11111111111111111111111 11111111111111111111111 11111111111111111111111 11111111111111111111111 11111111111111111111111 1.com” pass: “11111111111111111111111 11111111111111111111111 11111111111111111111111 11111111111111111111111 11111111111111111111111 2”	Pass pushed to passwds array on blockchain with metadata: 0 href: h(“111… .com”) password: h(“111… 112”)	Expected
href: ‘☃’ pass: ‘snowman.utf’	Pass is stored on blockchain, however since the *encrypt* function intends only to deal with ASCII characters, decryption of the href property will fail.	Expected

Further examples of when/where I used testing can be found in the project blog.

Program testing was done at various stages throughout development, and I iteratively built full demo versions upon one-another to test the program at significant points in development. These have all been preserved (although early iterations are very environment specific, and won't function correctly unless the environment they were developed in has been replicated) in the testing branch of my project’s Git repository[7].

[7] https://github.com/dylan-lom/PassMe/tree/testing

The submission version of *PassMe* has also been thoroughly tested, by both myself and other users (see End User Testing). System/Hardware testing was supposed to be performed mainly by end users, as all development was done on debian based linux systems (I did, however, test the program's compatibility with both Firefox and Chrome), but as discussed in the End User section, this didn't go as planned.

Live Test Data

User data input is often the most likely source of errors in a software program, and it's important to test a solution against a variety of input data to ensure it's reliability in live conditions. Common issues in systems when subjected to live test data are

- Response time, where the system struggles and slows down under load.
- Volume data, where the system may become unstable when large amounts of data are supplied in parallel.
- Interfaces between modules, when external modules (such as remote APIs) become unavailable, it is important for the program to have a fallback mechanism in case of such events.

  Response time: is dependant on the nodes mining on the ethereum network, and so testing it on a test network is pointless, since the Kovan network doesn't allow users to mine, and has a guaranteed block rate of once every 4 seconds, which does not reflect how the mainnet operates. If I was preparing to deploy *PassMe* on the mainnet, this would be a consideration, however.

  Load testing / Volume data: how this is handled is, again, dependant on the nodes mining on the ethereum network, and wasn't relevant to test.

  Interfaces between modules: *PassMe* performs an initial test to determine whether MetaMask is available on the system, if it were to become unavailable during operation of the application, local operations (such as searching the user's vault) would still work, however remote operations, relating to the blockchain would fail, as there wouldn't be a broker on the system to confirm transactions (MetaMask's role). *PassMe* works on the assumption the blockchain is immutable and will not disappear, if it were to disappear, the user wouldn't be able to connect to the contract, and no harm would be done to the user's data.

Software Debugging Techniques

Manual software debugging techniques can be used to detect errors in a program, examples of debugging techniques are:

- Function stubs — allow a developer to test higher-level subroutines without a subroutine that it requires having been written. In early stages of *PassMe's* development, stubs were used to test the smart contract's methods.
- Drivers — were not used in *PassMe's* development, since they are more suited to a bottom-up design method, whereas a top-down approach was taken in *PassMe's* development.
- Debugging output statements — JavaScripts *console.log* and *console.error* output statements were used extensively when performing module-debugging, to determine the values of internal variables to isolate where the source of a problem.

```
// CREATING THE VISIBILITY BUTTON
let visButton = document.createElement('input');
  visButton.type = "image";
  visButton.src = "show.svg";
  visButton.alt = "Show Password";
  visButton.passId = ret[i][3];
  visButton.pass = ret[i][2]; //stores password text for when it's 'hidden'
  visButton.onclick = function(){
    if (visButton.src.substring(visButton.src.lastIndexOf('/')+1) == "show.svg"){
      visButton.src = "hide.svg";
      console.log('pass'+visButton.passId);
      document.getElementById('pass'+visButton.passId).textContent = visButton.pass;
    } else {
      visButton.src = "show.svg";
      document.getElementById('pass'+visButton.passId).textContent = "*".repeat(visButton.pass.length);
    }
  }
vis.replaceWith(visButton);
```

Pictured: the use of console.log to establish that the passId value is stored correctly, and appended to 'pass' correctly. Used when debugging the error: document.getElementById(...) is undefined

Software Debugging Tools

CASE (Computer-aided software engineering) tools can help to detect errors in a program with less developer interaction than manual debugging techniques. Examples of software debugging tools are:

- Breakpoints — temporarily halt execution of code when reached.
- Program traces — track the status of internal variables through execution of a program.
- Single-line stepping — halts execution after each line, to observe changes to variables after each line.

My JavaScript development environment, a lightweight text-editor and a web browser, didn't implement these debugging tools, and so they weren't utilised in the development of *PassMe*. In the development of the smart contract I used the web-based Remix IDE, which didn't implement any of these techniques either.

End User Testing

Test Plan

A test plan for end-user testing includes a schedule and timeline of events that will occur during the testing process, and additional information about the user who completed the test, including the software (ie. web browser) and system software (ie. operating system) used.

Users were instructed to do the following

1. Follow the installation guide[8], to set-up your PassMe vault.
2. Writes three arbitrary URLs and passwords to your vault.
3. Refreshes your vault, and checks that all passwords have been written.
4. Closes the PassMe webpage, reload it and reaccess your vault.
5. Toggles the visibility of all your saved passwords on and off
6. Delete all your saved passwords
7. Refreshes your vault and checks that all passwords have been deleted.
8. Respond to google forms survey (here[9])

Test Data

The actual test data, reported by users via the survey, and expected data for comparison. If any issues were encountered, information on how to recreate the issue is also invaluable.

Since the requirements to test are reasonably high, and I didn't finish my project very far in advance, I could only get two responses to my user testing.

While it would've been ideal to get a variety of responses from users of varying levels of understanding, in the time frame I had left I could only allocate to get more tech-literate users to trial the software. This would've been particularly important in getting feedback on how easy the Getting Started and Video Walkthrough were to follow.

Question	**Expected Results**	**Actual Results**
Was the Getting Started Guide easy to follow?	Yes	1. Yes 2. Yes
What web browser are you	Mainly chrome, possibly a	1. Chrome

[8] https://dylan-lom.github.io/PassMe/docs/getting-started.html
[9] https://forms.gle/KZv11AJgNwQgqtUf7

using	few Firefoxes	2. Chrome
What operating system are you using	Mainly MS Windows, possibly a few OSX's	1. Windows 2. Windows
Were you able to complete all steps in the testing plan without issues	Yes	1. Yes 2. Yes
If no, what step did you encounter an issue on? How can the issue be recreated?		
Were you satisfied with the speed of the application?	No, I don't think that most users will understand the reason why the transaction takes several seconds to process	1. No 2. Yes
How do you compare this to other Password managers	Less features, basic user interface, too complicated to use	No answers given
Additional Feedback		1. User interface is uninviting, could use more colour. Tooltips could be used to give additional information about what buttons do

Test Results

Test results are stored as a record to justify any recommended changes. Results form the basis of new Design Specifications.

- While not a direct response to the test surveys, it was too difficult to try and help users test the software with its current dependence on the user having the time to learn how to use the MetaMask extension, and have a GitHub account to get currency on the test network.
- An explanation in the text version of the Getting Started guide as to why the transactions take a few seconds to occur would be helpful to education users anaquanted with how Ethereum networks works.
- The user interface could be brighter and more user friendly. Tooltips should be added to clarify the function of interface elements.