FastPass Devs
ICS 427
02 May 2022
Dr. Barbara Endicott-Popovsky

Introduction:

Team Name: FastPass Devs
Team Members: Anat Amar, Jolie Ching, Justin Mar, and Joshua Paino
Application Title, Description: FastPass is a web application that is designed to serve as a password manager.
Function Requirement Specification: This service allows users to store, access, and fill out login information. The login information can be exported as a JSON or CSV file for local storage. Additionally, users may automatically generate a randomized password.
Type of Program: Web Application
Development Tools: Javascript, Node.js, HTML/CSS, and MongoDB

Security and Privacy Requirements:

FastPass will be designed around 0-knowledge encryption to ensure that the storage and transfer of data will not be compromised by a hacker gaining access to either our servers or the connection to them. Therefore, only the user will have the key to access their information. A security requirement of this application is the use of AES-256, an industry standard encryption, to help protect users' data. An additional requirement is login security redundancy with multi-factor authentication, specifically in the form of two-factor authentication, that users can optionally set up. This requirement will serve as an additional layer of security to ensure that even if a user's master password is leaked or guessed, hackers will still face an extra obstacle in accessing their account. Another security requirement is the inclusion of a password generator to provide a convenient way for users to make strong randomized passwords. The password generator will create a random string of letters and numbers, with an option for special characters and a set length of 8-128 characters at the user's request. This randomization allows the user to use a strong generated password that fulfills a variety of different password requirements. The

next security requirement is the inclusion of a detection of password strength. The final security requirement is the ability to prevent a hacker from brute forcing the password.

To keep track of any security flaws, we will use testing APIs to ensure that FastPass follows our privacy and security requirements. The testing can be done manually and will run automatically on every git push, with it returning an error if a test fails. We will test login attempts to ensure that only the master password is accepted, that the master password is encrypted and isn't viewable in plain text when the user types it in, and that our system doesn't gain knowledge of the master password. We will check that a user's data can only be accessed by a user that has the key and that the user's data is encrypted in our database. Furthermore, we will also test our password generator by having it create a password and ensure that it follows the user's specifications.

Quality Gates (or Bug Bars):

The following quality gates, categorized by privacy and security, will be used to establish a standard for the level of security and privacy that our application will adhere to. These two tables detail threat scenarios, sorted by severity level, and are based on the sample docs provided by Microsoft.

Privacy Bug Bar

User Scenarios

Critical	● User notice and consent ○ Users will be made aware of the transfer of PII (personal identifiable information) from their device to the application's cloud-based service and must provide consent to the service. ● Data protection ○ PII, primarily passwords, will be encrypted on the user's device before transmitting to the cloud where only users may obtain access to the specific vault in the database. ○ In order for users to be granted access, the application will employ an authentication mechanism in the form of a master password. ● User controls ○ Application must provide an option for the user to

	remove non-essential PII data that has already been stored, or is in the process of being transferred. ● Legal controls ○ Application will not transfer user data to a third party without a legally approved contract. ● Child protection ○ Age is not part of collected data.
Important	● User controls ○ The application must provide an option for the user to remove non-essential anonymous data that has already been stored, or is in the process of being transferred.
Moderate	● Internal data management ○ Service must have a retention policy concerning data storage and disposal processes.
Low	● User notice ○ Notice of PII transfer is not obvious to users such as when stored as hidden metadata.

Administration Scenarios

Critical	● Enterprise controls ○ Application must provide opt-in consent and explicit notice of automated data transfer of sensitive PII. ● Privacy disclosure ○ Application must provide and display a privacy notice.

Security Bug Bar

Server-side

Critical	● Elevation of privilege ○ Entities will be prevented from obtaining more privilege than authorized. ■ Examples: unauthorized file system access, execution of code, SQL injection
Important	● Denial of Service ○ Server must not be easy to exploit such as with persistent or temporary amplified DoS. ● Information disclosure ○ Attackers must not be able to read information from any source, not intended to be exposed, such as PII.

	● Spoofing ○ An entity must not be able to imitate that of a specific user. ● Tampering ○ Application must prevent modifications of user data or databases, which remains even after restarting application.
Moderate	● Denial of Service ○ Temporary DoS without amplification, such as when multiple remote users hog resources. ● Targeted Information Disclosure ○ Attackers must not be able to read information from specific locations, not intended to be exposed, such as password vaults. ● Spoofing ○ An entity must not be able to imitate another (random) entity.
Low	● Information disclosure (untargeted) ○ Runtime information will not reveal sensitive information. ● Tampering ○ Temporary modification of data after submitting information that can be reversed by restarting application.

Client-side

Critical	● Elevation of Privilege ○ Application must prevent clients from executing arbitrary code or receiving more privilege than intended. ○ This can result in execution of code such as SQL injections or unauthorized file system access.
Important	● Elevation of privilege (local) ○ Local low privilege users may not elevate themselves to receive the privileges of another user. ● Information disclosure (local) ○ Attackers must be prevented from locating and reading information from anywhere on the system, not intended to be exposed, such as unauthorized file system access. ● Spoofing

	○ Ability for an attacker to present an imitation of the UI such as with a fake login prompt or a slightly different URL to get users to submit their data.
Moderate	● Information disclosure (targeted) ○ Attackers can locate and read information from a specific location on the system, not intended to be exposed, such as unauthorized file system access.
Low	● Temporary DoS ○ Performing a function on the application should not cause a crash. ● Spoofing ○ The user clicks on a malicious website intended to imitate the application and is vulnerable to a browser bug. ● Tampering ○ This category includes temporary modification of data that may be removed with a restart such as changes made within the browser's development tools.

Risk Assessment Plan for Security and Privacy:

- When will the public first have access to this project?: Upon initial deployment to a cloud hosting service.
- Who on your team is responsible for privacy?: All members of the development team.

Privacy Impact Rating and Assessment Plan:

The conclusion was ultimately made that the proposed project falls under a high privacy risk (P1). This privacy impact rating was concluded upon due to the idea that the application would exhibit P1 behavior that includes storing personally identifiable information such as usernames and associated passwords in a server that can be optionally transferred and stored onto a user's computer in the presentation of a JSON or CSV file. Furthermore, moderate privacy risks (P2) may also be exhibited that would include the transfer of anonymous data.

Therefore, in consideration of a P1 rating and possible P1 scenarios, our risk assessment plan entails an analysis of featured PII and how they might be manipulated as well as identifying aspects of the application that need threat modeling and security reviews:

1. Describe the PII you store or data you transfer:

    a. The application will contain documents of usernames and associated passwords used to log into other sites or applications, which are to be stored in a server-side database or optionally transferred to a user's local machine. Any PII that is stored and transferred is only accessible by the user that created them.
2. Describe your compelling user value proposition and business justification:
    a. Our application aims to offer an alternative platform for saving and creating passwords in an accessible manner through online and offline methods that are not restricted by subscription models or paid pricing.
3. Describe any software you install or changes you make to file types, home pages, or search page:
    a. One of the features that we plan to implement is the ability to export unencrypted data. This feature may involve retrieving a logged-in user's data within MongoDB and turning it into either a JSON or CSV file for download.
4. Describe your notice and consent experiences:
    a. Explicit and implicit notice will be presented in regard to how PII's will be protected and handled. These notices and consents will be presented on various pages, including the landing page and sign-up page.
5. Describe how users will access your public disclosure:
    a. Dedicated web pages will be created that present and document any non-confidential information. Additionally, the project's repository will be publicly available at GitHub so that users may fork and examine the code.
6. Describe how organizations can control your feature:
    a. Organizations will have the same privileges as users. In other words, they will not be able to access users' information unless there is a legally approved contract that users, the development team, and the organization are fully aware of.
7. Describe how users can control your feature:
    a. Users will have full control of the PII that they present to the site (such as usernames and passwords). Specifically, users can edit their own PII such that they have control of how much data they want to present within the web application itself. However, they will not be able to view any PII of other users.

To have additional control, users can also optionally set up multi-factor authentication to have an extra layer of security when accessing their data.

8. Describe how you will prevent unauthorized access to PII:
    a. We will employ an account system where a user's key will be verified before being able to access their data within the database. Furthermore, accounts can also be locked-out if an incorrect master password is provided multiple times.
9. What portions of the project will require threat models:
    a. One of the portions that will require threat models is that of unauthorized attempts to access another user's account that is secured with different layers of security including a master password and multi-factor authentication. Additionally, the utilization of exported data will also require a threat model, as the user who has access to it will have full control of its contained information.
10. What portions of the project will require security design reviews:
    a. Server-side storage is one portion that will require a security design review as it is to be protected by encryption and various other protection methods. Furthermore, the password generator and its randomized strings will also need a security review in regard to how secure it is in protecting created accounts of other sites or applications.

Design Requirements:

Based on our privacy and security requirements, to abide by 0-knowledge protocol, FastPass will not collect the plain-text master password to prevent a leak. Our next design requirement is to secure user data in the event our database is compromised or accessed, thus we will use AES-256 encryption -- the current industry standard. An additional requirement is the option for a redundant security method, which will be achieved with a multi-factor authenticator that generates a one-time use code that is to be inputted alongside an email and password when logging into the application. The final security requirement is a way for the user to create strong passwords for their logins, which will be done with a password generator. The master password strength will be determined by how many unique features it contains such as an uppercase letter, undercase letter, number, special character, and length. To prevent hackers from brute forcing a

user's login information, the user's account will be locked for a period of time if more than three attempts have been made.

An additional design requirement for FastPass is a server to store user data. This requirement would most likely have to be done on the cloud due to a lack of dedicated server machines. FastPass would also need a domain name to allow users to go to a website and login to access their data stored on the server. The next design requirement is the option to set a hint for the master password that will be sent to a user's email address or phone number. This requirement is needed because FastPass has no knowledge of users' master passwords, thus it is impossible to recover or reset it in the event that the password is forgotten. Another design requirement is fields setup in each login information. These fields will be username, password, URL, and custom fields. The fields will allow users to organize the login information, and the custom fields can be used for any notes concerning the login. The next design requirement is a convenient feature for users to autofill login information. This feature will be implemented with a button click that fills out the username and password fields for the listed URL. The final design requirement is to allow an easy way to change password managers by exporting the user's login information. In order to allow it, the login information will have to be exported unencrypted in a JSON or CSV file, which is supported by most password managers.

Attack Surface Analysis and Reduction:

FastPass will have 2 levels of privilege. The first level will be admin users who can manage the website and database. The second level will be everyone else, who can create an account with a master password and store and access their information.

The places of attack for our application, FastPass, includes servers that store the user's data. This data should be encrypted with the industry standard and only the user should be able to access their data. Therefore, the application scores low for attackability, or about 0.3. For the connection to our server, however, we will apply 0-knowledge encryption. With this protocol, the service cannot access the user's master password, thus FastPass should again have a low attackability, or about 0.3. This feature is similar to other password managers like Bitwarden and LastPass, who also encrypt user's information and use 0-knowledge encryption, thus they would score similarly, leading to a score of about 0.6 for all.

Our password manager is web-based, therefore its usage requires a constant internet connection, leading to the application being vulnerable in a public setting or in the event the router is compromised. This threat is severe, because unlike other password managers, there is no way to store or decrypt the user's information on their system, thus it scores high at about 0.8. Overall, FastPass receives a score of 1.4, while Bitwarden and LastPass have 0.6. However, both Bitwarden and LastPass do employ the usage of browser plugins, which is a point of vulnerability for them. Browser plugins are more vulnerable than the application that is typically used. Thus, the usage of a browser plugin will give them a moderate score of about 0.5. This analysis gives FastPass a score of 1.4 while Bitwarden and LastPass have 1.1.

The next point of vulnerability is the resetting of the master password. All the password generators employ 0-knowledge, thus it is not possible to recover the master password for any of them. However, for LastPass, it is possible to reset your master password if set up to do so in the mobile app with biometric data. While complicated to achieve, it is possible to falsify this information, thus this feature scores low at about 0.3. This increases LastPass' score to 1.4 in comparison to FastPass' 1.4 and Bitwarden's 1.1.

The final point of vulnerability is the exporting of user data. FastPass, Bitwarden, and LastPass allow the export of encrypted user data for import into another password manager. FastPass is web-based, thus the connection may be tapped into, and the passwords can be stored by a hacker. Bitwarden and LastPass can be used offline, thus FastPass receives a moderate score of 0.5, while the former two have a low score of 0.1.

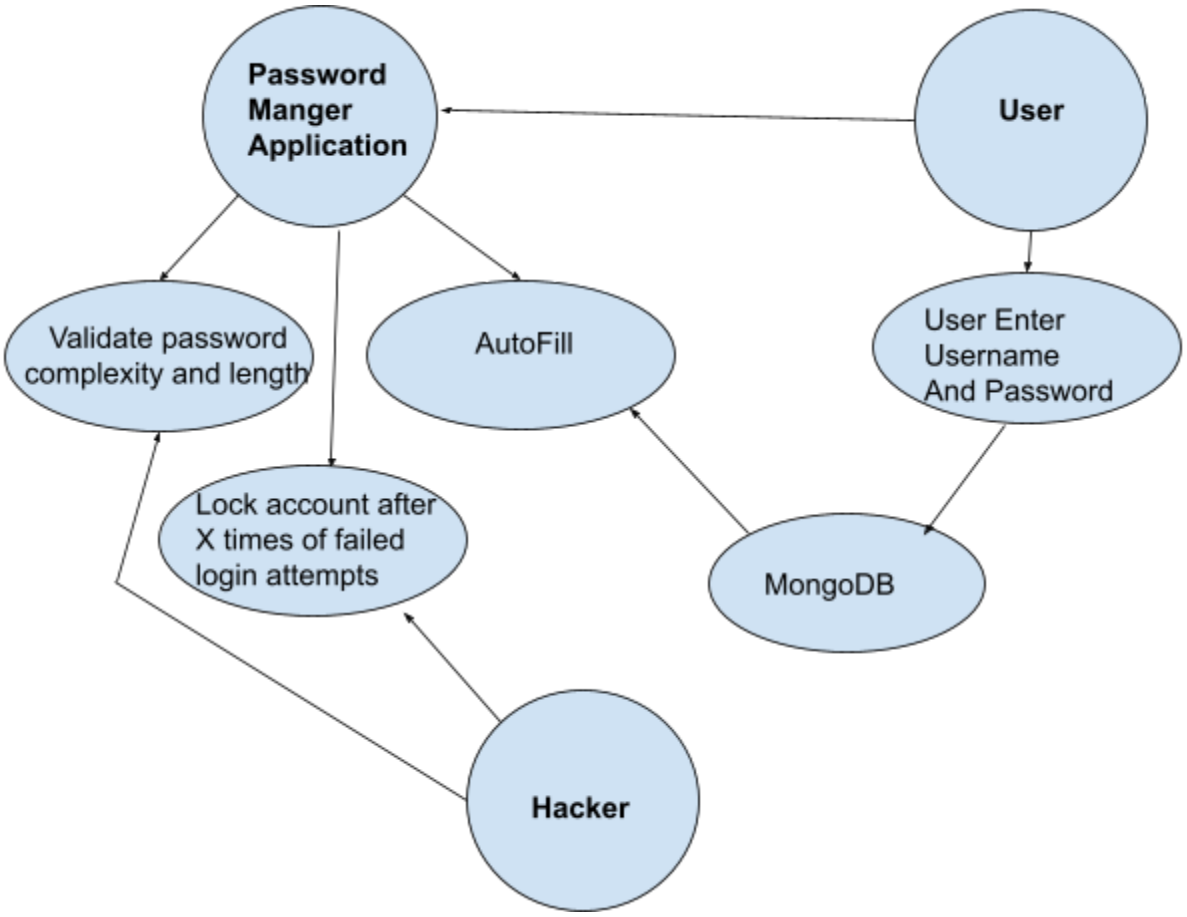
Overall, the attack surface analysis gives the following results:

FastPass - 1.9

Bitwarden - 1.2

LastPass - 1.5

Threat Modeling:



Implementation

Approved Tools:

Tool Name	**Required Version**	**Notes**
Versioning Tools Github/ Github Desktop	Github Desktop: v.2.9.6	For our project, we used several tools. First, we created a GitHub repository named “Fast-Pass” which is accessible by all team members. GitHub allows us to manage and organize the

		project with Kanban boards to divide and keep track of tasks. These boards have the following categories: “To-do”, “In Progress”, and “Done”. Versioning is directed to our online repository from our local environment via GitHub desktop, where we implement project enhancements by creating and merging branches to divide each branch to each page of the website.
IDEs IntelliJ IDE Visual Studio Code	IntelliJ IDE: v2021.2 Visual Studio Code: v1.63.2	To implement and design our project, we use IntelliJ IDE and Visual Studio Code to write the code.
Languages HTML/CSS Javascript React.js Node.js Meteor **Packages** Semantic UI	React.js: v17.0.2 Node.js: v16.14.0 Meteor: v2.6 Semantic UI: v2.1.2	The programming languages we used to create our webpage are: HTML, CSS, and JavaScript. We also use the Semantic UI user interface, a front-end development framework that helps us to create a more beautiful and responsive layout while using HTML and CSS. To compliment Semantic UI, we used React.js, an open-source JavaScript library for building user interfaces or UI components. The Meteor web application framework, an open-source isomorphic JavaScript web framework, was used as well to quickly set up our application alongside using Node.js.
Web Browser: Chromium	Chromium:v98.0.4758.102	Since our program is that of a web application, we used Chromium web browsers as a means to

(Google Chrome and Brave)		interact with the site.
NoSQL Database: MongoDB	MongoDB: v5.1	MongoDB is a NoSQL open-source document-oriented database that is designed to store any data we're entering and will provide official driver support for Node.js and JavaScript. This will allow us to store our collections where we can store documents and our data in the form of fields.
Testing Tools TestCafe ESLint	TestCafe: v1.7 ESLint: v7.0.0	The testing tools that will be used are TestCafé, ESLint, and AppVerifier. TestCafé is a Node.js end-to-end free and open source automation tool that is used to test our web pages. In addition, we will use a static analysis tool known as ESLint to ensure our code meets industry standards.

Deprecated/Unsafe Safe Functions:

Regarding Node.js which is used to handle server-side related activities, there were a few functions that could fulfill our security, privacy, and design requirements but are unfortunately deprecated or unsafe according to the Node.js v16.14.0 documentation (2022). Two deprecated functions that could have been useful, but were removed, are the “crypto.createCredentials” API and the “crypto.Credentials” class within the crypto module. Instead, the documentation recommends using a few functions within the tls (Transport Layer Security) module such as “tls.createSecureContext()” and “tls.SecureContext” to properly satisfy standard cryptographic protocols for web applications. Within the tls module, however, held another potentially useful, but deprecated function. This function in question is the “tls.CryptoStream” class which could be used to represent a stream of encrypted data. However, the safer alternative as listed by the

documentation would be to use “tls.TLSSocket” which functions similarly by performing transparent encryption of written data.

React.js also holds a few methods that are deprecated and unsafe as of React v16.9.0 (Abramov and Vaughn 2019). These methods include “componentWillMount()”, “componentWillReceiveProps()”, and “componentWillUpdate()”. The usage of these methods can allow for the triggering of API calls, however, its utilization can lead to unsafe practices and outcomes such as memory leaks. Thus, these functions were deprecated and labeled as “UNSAFE" if used in versions past v16.9.0. The alternative is to adhere to React’s component life cycles and to call certain other methods in a particular order for purposes such as mounting and updating. For instance, mounting requires calling the following methods in order: “constructor()”, “getDerivedStateFromProps()”, “render()”, and “componentDidMount()”.

Looking at the Meteor changelog documentation, the things that were deprecated are Meteor._debug to display messages to the console, we should use Log.debug, with the advantage being that Log.debug doesn't display in production. When creating indexes in the collection, we shouldn't use Mongo's _ensureIndex function, but Meteor's createIndex function. In the callback-hook, we iterate through elements with the forEach function instead of each. Meteor had an update to use camelcase for their packages, thus we shouldn't use the non-camelcase packages. Meteor's http package was deprecated for the fetch package. Meteor's uuid package had its development stopped in favor of the npm uuid package. Finally, Meteor's onRendered, onCreated, and onDestroyed functions for Template.foo.rendered were changed to be used with Template instead. An unsafe practice for Meteor is allowing the client to run code. The first is to define database functions in Meteor.methods instead of having them in the client, and finally, to avoid allow/deny to run MongoDB queries directly from the client. We shouldn't pass the user ID as an argument when we write functions to prevent it from going to the client.

Static Analysis Tool:

For our JavaScript-based code, the static analysis tool our group will be using is ESLint. Specifically, ESLint is a linter, which is a type of static analysis tool that checks for formatting, styling, and logic errors. It bases these calls off of a set of base rules such as “camelcase” which enforces a specific naming convention for variables. Rules may be added or removed based on a team’s preferences.

By running the command, “npm run lint”, the terminal returns a detailed list of errors, descriptions, and the file paths and line numbers they occur on. This output is possible because the command calls on a script called “lint”, which we declared in our package.json. The lint script checks all of our files with the .js and .jsx extensions against a set of rules we declared in our .eslintrc file. In this file, we appended three additional rulesets to the vanilla ESLint: AirBnB, Meteor-recommended, and React-recommended rules. Including the rulesets used by AirBnB is not only helpful due to allowing us to align our standards of coding with the industry, but because it is currently the only accessible, open-source ESLint ruleset used by a Fortune 500 company.

Before merging our sub-branches to the main branch, one member of our team runs ESLint. For this step of the project, we mainly focused on developing the user interface. Therefore, the majority of the errors ESLint returned were “no-unused-vars” -- an error triggered when we define variables but never use them. This is a result of using Professor Phillip Johnson’s meteor-react template. To adjust the template to our own project, we did not utilize all the variables (specifically, component names) that were originally declared in the import statements.

This is helpful because virtually all of our layouts utilize a variety of components from multiple packages. For example, our sidebar component alone requires over 10 import statements from React, Meteor, icon, and navigation packages. Like this, it is easy to forget to remove component names that one ends up not using. Unnecessary or unused variables affect the readability of our application from a programmer’s point of view due to cluttering the file with extraneous information. By using ESLint in this way, we can ensure that our code is convenient to read and debug.

Verification

Dynamic Analysis:

The dynamic analysis tool we used was TestCafe. We used it to test the client-server interaction to ensure that our app properly follows the zero-knowledge procedure. This was accomplished by having TestCafe act as both the client and server, sending and receiving messages to each other. With the following code fragment, TestCafe simulates a user inputting their login information on a sign-up form:

```
async signUpUser(testController, username, password) {
	await textController.typeText('#signup-form-email', username);
	await textController.typeText('#signup-form-password', password);
	await testController.click('#signup-form-submit');
	await navBar.isLoggedIn(testController, username);
}
```

Furthermore, we used another TestCafe function, RequestLogger(), to intercept the message the client sends to the server upon creating a new user. This test confirms that the client does not send a user's password to the server, as per zero-knowledge protocol.

```
const logger = RequestLogger(['#signup-page'], {
	logRequestBody: true,
	stringifyRequestBody: true,
});
testController.expect(logger.request.body).notContains(password);
```

We then used the MongoClient API to create functions that query the user, ensuring that they exist. The salt, SRP (secure remote password protocol) group defined in parameter.js, and verification which was derived from the master password and salt, was sent and that the password doesn't match the verification.

TestCafe can additionally be used to check the client-server interaction when a user logs into their account. TestCafe automates logging in with a process similar to the aforementioned signUpUser code fragment, but instead using the sign-in form. After that, the server sends the

salt and SRP group back to the client. We use RequestLogger() again to intercept this message and read the information to ensure that the correct salt and SRP group was received.

The client uses the salt and master password to calculate a key x. The client then generates a random secret number a and uses the SRP Group to generate a's non-secret counterpart A. We used TestCafe's RequestMocker() to create a mock of the client sending A to the server and RequestLogger() to check that it received the server's counterpart B.

```
const mock = RequestMocker()          // Mocks the client sending A
        .onRequestTo('https://localhost')
        .repsond(A);

fixture`RequestMock`
        .page`#signin-page`
        .requestHooks(mock)

const logger = Requestlogger(['#signin-page'] {        // Logs the response (B)
        logRequestBody: true,
        stringifyRequestBody; true,
});

fixture`RequestLogger`
        .page`#signin-page`
        .requestHooks(logger)

test('Send A and Receive B', async testController => {
        // waits for RequestMocker() and store what it sends to the server
        const sentMsg = await ClientFunction(() => fetch('#signin-page').then(res =>
res.json()))();
        await testController     // Checks A was sent and B received
```

```
        .expect(sentMsg).eql(A)
        .expect(logger.contains(record => record.response.body === B)).ok()
}
```

The client uses x, a, and B while the server uses verifier, A, and b to calculate the session encryption key, which should be the same value if the master password given by the user was correct. In this process, the client encrypts a message with the key and sends it to the server, who in turn decrypts and verifies the message. The server encrypts its own message and sends it back to the client who also decrypts and verifies it.

We use a similar format as the above to imitate the client sending its message and receiving the server's encrypted message. After the client decrypts it, we check if the message was correctly verified, before running the function on the server's side and sending another message to TestCafe if the server verifies the client's message.

Attack Surface Review:

Tool Name:	**Current Version:**	**Changes/Updates/Patches/Vulnerabilities:**
Versioning Tools Github/ Github Desktop	Github Desktop: v.2.9.6 -> v2.9.12 (Release Notes…c2022)	Despite an update to v2.9.12, there were no new vulnerabilities or changes that could impact our project as the release notes primarily detailed minor fixes.
IDEs IntelliJ IDEA Visual Studio Code	IntelliJ IDEA: v2021.2 -> v2021.3.3 (IntelliJ IDEA…c2000-2022) Visual Studio Code: v1.63.2 -> v1.65.2 (Visual Studio…c2022)	For IntelliJ IDEA and Visual Studio Code, the updates to v2021.3.3 and v1.65.2 respectively reported no major vulnerabilities or changes that could impact the development of our project.
Languages HTML/CSS	React.js: v17.0.2 *no change (React Versions…2022)	Regarding the four listed tools, React.js and Semantic UI did not receive any updates since

Javascript React.js Node.js Meteor **Packages** Semantic UI	Node.js: v16.14.0 -> v16.14.2 (Node.js…[date unknown]) Meteor: v2.6 -> v2.7 (Changelog…2022) Semantic UI: v2.1.2 *no change (Semantic UI…[date unknown])	our latest report while Node.js and Meteor received some noteworthy additions. Node.js received 2 minor updates which included addressing a high-risk vulnerability regarding an infinite loop when parsing certificates; other commits were also made but were not listed as notable in the documentation. However, commits and changes were made to the crypto module which is something we may integrate into our application. As for Meteor, the framework was updated to v2.7 which included a new package called "accounts-2fa". Coincidentally, one of our proposed features included implementing multi-factor authentication and thus the recent addition of this new two-factor authentication package in Meteor would ease its implementation if utilized.
Web Browser: Chromium (Google Chrome and Brave)	Chromium:v98.0.4758.102 -> v99.0.4844.83 (Git repositories…[date unknown])	Despite an update to Chromium, examination of the latest update revealed no new substantial changes. This should therefore not affect our ability to examine our web application even if version numbers begin to differ between team members.
NoSQL Database: MongoDB	MongoDB: v5.1 *no change (MongoDB 5.1…c2021)	No new updates or changes to MongoDB have been made since our previous report.
Testing Tools TestCafé	TestCafé: v1.7 -> v1.18.2 (TestCafé…c2012-2021)	Updating TestCafe to the latest version of v1.18.2 introduced new additions that may

ESLint	ESLint: v7.0.0 -> v8.12.0 (ESLint…[date unknown]).	further assist in our dynamic analysis testing. These new features include the ability to define custom variables that can be referenced in testing, the addition of testing with multiple browser windows, improvements to using TestCafe in macOS, as well as numerous bug fixes. ESLint also received a new update since the previous report; however, it was mostly bug fixes and thus there were no new noteworthy additions that could impact our project.

Fuzz Testing:

1. Insecure Direct Object Reference (IDOR):

The layout of our application consists of individual pages linked to a central App.jsx file via React routing. One of these pages allows users to edit an existing saved password item. In order to render this page and its data, an ID is required to fetch the document from the collection password items. Because this ID is displayed as part of the URL, this design raises a possible IDOR vulnerability where one user, by guessing the correct ID sequence, may gain unauthorized read and write access to another user's saved password.

To perform this hack, an ID was retrieved from an existing password item owned by an account seeded with default data. This initial step was done by logging into the account and navigating to the edit page where the ID is shown within the URL. For example, logging into the account john@foo.com and editing one of his passwords presents the URL, "http://localhost:3000/#/edit/WaSeJbbk399BEQCzS", where "WaSeJbbk399BEQCzS" is the ID for the password item. We then took this URL and ID and attempted to access it without being logged in. This attempt resulted in the user being redirected to the sign-in page as the edit page is a protected route that requires a user to be logged in. In another test, we took the URL and ID once again but this time, attempted to access it in another account. From this attempt, the edit page was accessible, but it did not present any data associated with the ID within the form inputs; in other words, the form was simply blank. Furthermore, attempting to write new data within the form was unsuccessful as there is a design measure in place to verify whether the logged-in user

is the owner of the password item; if the email of the logged-in user does not match the owner, then the user will be unable to submit the form and its contents.

The success rate for this hack can therefore be calculated as 0%. Despite the presence of the password's ID in the URL, we have security measures in place that restrict unauthorized read and write access to sensitive password items such as protected routes and owner verification.

2. Cross-site scripting (XSS):

To perform this attack, we attempted to inject a malicious script within the 'Add Password' web page. We inserted this code snippet into the form input:

```
<SCRIPT type="text/javascript">
                    var adr = '../evil.php?cakemonster=' +
                    escape(document.cookie);
                    </SCRIPT>
```

This fragment allows attackers to steal a cookie from a user as it is recognized as code rather than text and will be executed as such. However, this can only work if the application fails to validate the input data before sending it to the server. Because our application processes all form data as a string, the attack was successfully thwarted and the code fragment, rather than being run, was simply written into the text field.

Our next step was to test our program for vulnerabilities. For this, we used an online XSS Scanner tool, https://pentest-tools.com/website-vulnerability-scanning/xss-scanner-online. It tests for XSS vulnerabilities with a range of requests. First, the scanner injects a simple string in the target URL and checks whether it is reflected on the response page. Once we entered our webpage link, "https://fastpass.meteorapp.com/", into the search bar on the website, the results returned "Nothing was found for Cross-Site Scripting" in the overall findings. Therefore, the success rate for this hack can be considered 0% because adding malicious script within the Not Found Page resulted in an unsuccessful hack. Moreover, using the Cross-Site Scripting website proved that our program is protected from this hack.

3.Brute forcing/Broken authentication:

Our seed data contains a set of default accounts where one of these accounts possesses the "admin" role. The presence of this account, due to its elevated privileges, may pose a security risk if an attacker were to brute force the system to gain access to its credentials. This may lead to broken authentication of administrative power within the site.

To verify the potential of this vulnerability, we utilized a tool publicly available on GitHub called “Brute-Force-Login” (https://github.com/Sanix-Darker/Brute-Force-Login) that attempts to brute force any login form within a site with an extensive list of common passwords. After running the program, the results ultimately presented a successful breach to the site. This outcome is due in part because the default admin password is set to “changeme”, which happens to be a part of the common password list.

Naturally, this result is logical given that the password consists of only 8 lowercase characters. As such, the possible success rate of a trial-and-error brute force hack can be concluded to be high at around approximately 80-90%. To fix this vulnerability, we plan to change the admin password to one that is more complex and consists of uppercase letters, lowercase letters, numbers, and symbols. However, the very existence of the admin credentials within the default data poses a high risk and thus, the best fix is to not include a default admin account within the deployed version of the site.

Static Analysis Review:

As discussed in our previous reports, our chosen static analysis tool is ESLint. This tool enforces coding style, thereby limiting redundant code and improving upon our code’s readability. To achieve this, not only did we import ESLint’s extensive Javascript rulebase, we also included AirBNB’s open-source ESLint config to ensure we are able to adhere to an industry-standard level of code.

In order to run the linter, we call “npm run lint” in the terminal to invoke our script titled “lint”, which we defined in our application’s package.json. This script, consisting of the command, “eslint --quiet --fix --ext .jsx --ext .js ./imports”, allows us to draw rules from both ESLint and AirBNB’s rulebase and check these rules against all Javascript files. Errors that cannot be fixed are instead printed out in the terminal for manual review.

The most common error we came across was redundant code. In particular, we encountered six errors of “no-unused-imports”. Across our application, we have 16 unique pages and more than 10 imports per page. These imports are necessary to form design components such as buttons, icons, and grids as well as navigation routes. Rather than import items one at a time, we often copy and paste a commonly-used set of imports when creating a page. As the page is developed, imports that we do not need are gradually removed; however, there are occurrences

where human error causes us to miss a few unused imports. Unnecessary imports can slow down the loading speed of our application as the browser must load all imports before it can render the page to the user. This is where an automatic linter like ESLint comes in handy, returning the location of where an unused import should be deleted and optimizing our website's performance.

A less common error we encountered was "no-console", which prohibits outputs to the console. Console statements not only add to code redundancy but are also important to remove for the security and professionalism of our application. In the development phase, console outputs are helpful in debugging the application. This command returns raw data and allows developers to output error messages. For example, a command such as console.error("This array is empty.") can warn us that no data is being returned from the server or client. However, the output of these commands can also be seen by the public through their browser's Developer Tools function. This allows malicious entities to potentially reverse engineer or find clues to a vulnerability in the code. While console logs are helpful for debugging, they must be removed when deployed to production to prevent accidentally revealing sensitive information about the application's internal workings.

Dynamic Review

We continued using TestCafe to dynamically analyze our application. Using our previous tests, our application still authenticates and authorizes users without sending any sensitive information from client to server or server to client.

In our previous dynamic analysis, we were unable to implement testing for all of the features available on our application. This time, our dynamic analysis was further expanded by testing various pages in our application: the home page, sign-in page, sign-up page, and landing page. Pages were validated by storing them in a pageSelector variable with the following code fragment and checking for their existence.

```
async isDisplayed(testController) {
    await testController.expect(this.pageSelector.exists).ok();
}
```

We also included tests for the creation of login information that the user creates. We have two tests: the first is a simple login with only the name, username, and password filled out. This was done by targeting the form's fields and using "typeText" to input text into the fields.

```
async basicLogIn(testController, username, password) {
        await this.isDisplayed(testController);
        await testController.typeText('#username', username);
        await testController.typeText('#password', password);
        await testController.click('#save');
```

Additionally, a second test ensures that the password generator creates a random password with the specified parameters the user gives. A further test confirms that the log-in information is properly retrieved by the application. To do this, we extract the username and password information retrieved from the first test by clicking the copy icon and checking that it matches the form's values.

Release

Incident Response Plan:

Privacy Escalation Team:

Role & Assigned Member:	Responsibilities:
Escalation Manager - Anat Amar	Anat, as the escalation manager, will be responsible for organizing appropriate representation with privacy, business, legal, and public relations experts. Ultimately, she will oversee the entire internal process of the privacy escalation from start to finish.
Legal Representative - Jolie Ching	Jolie, as the team's legal representative, will be primarily responsible for addressing and resolving any legal concerns, liabilities, or disputes that may have arisen from the privacy incident.
Public Relations Representative - Joshua Paino	Joshua, as the team's public relations representative, will be primarily responsible for addressing any public concerns from the

	users of the software or the media that may have arisen from the privacy incident.
Security Engineer - Justin Mar	Justin, as the security engineer, will be responsible for initiating security reviews in order to identify and resolve the root cause of the privacy incident. These security reviews will also be overseen by Justin who will evaluate the security corrections made by the development team that would resolve the vulnerability or bug related to the incident which would, in turn, mitigate the chances of a similar event from occurring in the future.

Emergency Contact Email: uhfastpass@gmail.com

Incident Resolution Procedures:

In the event that a privacy-related incident arises, the procedures will progress as follows to arrive at a resolution or a set of resolutions. First, the escalation manager will lead and conduct an investigation to gather additional information related to the incident that has been reported. Specifically, the escalation manager must identify and document information such as the origins of the incident (who reported it and why), affected areas of the software, legal implications as well as public implications to ultimately determine the severity and scale of the situation. The legal representative, public relations representative, and security engineer will then review and confirm the information with the escalation manager who will announce to the organization the details of the incident in order to further proceed with a resolution.

Afterward, the escalation manager will find appropriate contacts for the legal and public relations representatives to collaborate with while the security engineer works with the software engineers to address related bugs and vulnerabilities within the product. The legal representative will then attempt to resolve any legal liabilities as a result of the incident in accordance with the law. Concurrently, the public relations representative will also communicate with the public and users of the software about the incident and be as transparent as possible regarding the steps that are being taken to arrive at a resolution.

The escalation manager will then report the actions and progress made by each member of the privacy escalation team. In particular, the escalation manager will review whether all legal liabilities have been resolved (by the legal representative) and whether the fixes to the software

pass a final security review (as determined by the software engineer). Finally, the organization will then issue a formal apology to all affected parties of the incident to which the escalation manager will then proceed with further resolutions. This action may include internal restructuring or continued enhancement of the software's security to prevent a similar situation from occurring in the future.

Final Security Review:

We will assess whether our application has passed the final security review by examining the following key areas: 1) Threat model, 2) Bug Bar, 3) Static Analysis, and 4) Dynamic analysis.

The threat model describes a general layout for the structure of the FastPass password manager. Referring to our earlier diagram, our system successfully allows the user to create an account with password authentication provided by the Meteor Accounts system. It then stores this information in our MongoDB database where the data is encrypted from the client to our database cluster. While we have implemented handling for inactive login sessions to prevent the hijacking of user accounts, we are working on adding the additional security of locking accounts after a certain number of failed login attempts. Therefore, the grade based on our threat model is determined to be "Passed FSR with Exceptions".

According to the bug bar specified in our first report, our application passes the most critical user scenarios in the privacy bug bar. This includes user notice and consent of the transfer of personally identifiable information, user controls, legal controls, privacy disclosures, and child protection. However, we were unable to encrypt all user information aside from what is covered by MongoDB and Meteor's built-in encryption. Therefore, the grade shall be assessed as "Passed FSR with Exceptions".

As discussed in our previous report, our chosen static analysis tool is ESLint. This tool enforces coding style, limiting redundant code and improving our code's readability. To achieve this, we imported ESLint's Javascript rulebase and included AirBNB's open-source ESLint configuration to ensure we are able to adhere to an industry-standard level of code. Due to running our linter and addressing ESLint errors periodically throughout development, our final ESLint check was devoid of any errors; whereas previously, our most common errors were "no-unused-imports" and "no-console". Therefore, our static analysis is "Passed FSR".

While developing FastPass, we used TestCafe to ensure that FastPass follows our detailed privacy and security requirements. We tested the login attempts, and as a result, we found that only a user's master password will be accepted by our authentication system. When the user types a plain text master password, the client does not expose it and ensures the system does not plainly record it. Moreover, an account's data can only be accessed by this password. Furthermore, we tested our password generator and ensured that the user could generate a random password according to their specification of any combination of lower/upper case, numbers, and/or symbols. Therefore, according to our dynamic analysis, we have determined that we have "Passed FSR" for this area.

Considering our four core areas of review, the overall grade we have assigned to our project is "Passed FSR with Exceptions". While we have hit most of the major requirements we outlined at the beginning of the semester, there are still additional security functions we can develop to maximize data security.

Certified Release & Archive Report:

The main feature of Fast Pass is its ability to store user passwords. Users can create accounts on our FastPass website and add multiple password entries. These entries, in addition to the passwords themselves, include other details such as site name, address, username, extra notes, and category type (social, retail, entertainment, and miscellaneous). This information is displayed on a user's dashboard. The list of passwords can be further sorted by category. The total number of passwords overall and the number of passwords in each category are also displayed.

To assist users with password creation, we developed a password generator that allows users to specify multiple conditions such as password length, special characters, and uppercase to create their ideal password.

Users may also edit their password entries and modify any details where necessary. They may use our hyperlink display in the dashboard password list to simply click on the name of their password's site and visit the link in their browser. Before this, users can press a button to copy the password to the clipboard. In addition, FastPass also allows users to export their password list into a .csv to facilitate offline storage.

For security purposes, users may optionally enable multi-factor authentication. This additional layer of logging in can be activated for an account in the form of two-factor authentication by scanning a generated QR code from the "2FA settings" page into the "Google Authenticator" mobile app and inputting the subsequent one-time six-digit code. Afterward, any future log-ins will require inputting the one-time code generated from the app alongside the correct email and password. Users will also have the ability to disable 2FA should they so choose in the future. Furthermore, users will be automatically logged out of their session after 30 minutes of inactivity.

Our future plan is to expand our web application into a desktop client or browser add-on so users can autofill their passwords into registered sites. This update is necessary in order to extend the usability of our application. We hope that users, rather than having to repeatedly visit our deployed site for their password information, can simply visit the login page of their chosen website and populate the login form with one click.

On the security side of the application, there are still aspects that can be improved. In particular, we aim to better support zero-knowledge encryption with AES-256, an encryption algorithm that many popular password generators use. Additionally, we plan to provide locking mechanisms and recovery for accounts with too many login attempts.

FastPass supports popular operating systems such as Windows, macOS, and Linux as well as major browsers. Additionally, as a web application, it is available to be accessed via mobile. The project is free and open-source, published under the MIT license.

For developers wanting to make contributions to our repository, the installation instructions are listed below. However, these requirements are not mandatory to view our deployed site. Node.js is recommended, version 12 at minimum. However, the specific Node.js used for development is v16.14.0. Additionally, Meteor v2.7.1 must be installed. Then, a user only needs to invoke "meteor npm install" in the command terminal to receive the rest of our application's packages.

Our project is hosted on Github and may be cloned from https://github.com/FastPass-Devs/FastPass. Additionally, our deployed version is hosted via Galaxy at https://fast-pass.meteorapp.com/. Due to our site using the Galaxy free tier, Galaxy servers may take up to five minutes to cold boot and start up our application upon visiting the URL.

References:

Abramov D., Vaughn B. 2019 August 8. React v16.9.0 and the Roadmap Update [blog]. React. [accessed 2022 February 16]. https://reactjs.org/blog/2019/08/08/react-v16.9.0.html.

Changelog. c2022. Meteor; [updated 2022 March 24; accessed 2022. February 21]. https://docs.meteor.com/changelog.html.

ESLint. [date unknown]. OpenJS Foundation. [updated 2022 25 March; accessed 2022 March 27]. https://eslint.org/.

Git Repositories on chromium. [date unknown]. [accessed 2022 March 24]. https://chromium.googlesource.com/?format=HTML.

IntelliJ IDEA 2021.3.3 (213.7172.25 build) Release Notes. c2000-2022. JetBrains; [accessed 2022 March 24]. https://www.jetbrains.com/idea/.

MongoDB 5.1. c2021. MongoDB, Inc.; [accessed 2022 March 24]. https://www.mongodb.com/new.

Node.js. [date unknown]. OpenJS Foundation; [updated 2022 March 17; accessed 2022 March 24]. https://nodejs.org/en/.

Node.js v16.14.0 documentation. 2022 February 8. OpenJS Foundation; [accessed 2022 February 12]. https://nodejs.org/dist/latest-v16.x/docs/api/deprecations.html#DEP0011.

React Versions. c2022. Meta Platforms; [updated 2022 March 22; accessed 2022 March 24]. https://reactjs.org/.

Release notes for GitHub Desktop. c2022. GitHub; [updated 2022 March 23; accessed 2022 March 24]. https://desktop.github.com/release-notes/.

Security. Meteor; [accessed 2022 February 18]. https://guide.meteor.com/security.html.

Semantic UI React. [date unknown]. [accessed 2022 March 24]. https://react.semantic-ui.com/.

TestCafé. c2012-2021. Developer Express Inc.; [updated 2022 January 17; accessed2022 March 24]. https://testcafe.io/release-notes.

Visual Studio Code. c2022. Seattle (WA): Microsoft; [updated 2022 February, accessed 2022 March 24]. https://code.visualstudio.com/.