Client Report

100287756 – Alfie Arundell-Coote
100287814 – George Beales
100301701 – James Wright
CMP-6045B

1 Introduction

At a basic level, a food blog website is intended for users to post their meals for other users to see. The complexity can range dramatically, however with more extravagant features then in turn comes more of a chance for a security breach. With our website we have created a simple front end with just the essential features, this means that we can maximise the security potential for our site.

We have two main objectives with this site: the first is making it easy to use so that any user of any level can utilise our site. Our second objective is to ensure that our website is watertight and safe from a multitude of potential cyber attacks. In order to complete our first objective we have decided to utilise white space to our advantage by not crowding our site with unnecessary colours, images or buttons. For example our navigation bar only has 3 buttons and a search bar as we felt like if we added much more it could confuse some users.

Our second goal is achieved via implementing certain mitigation's in order to counter security vulnerabilities which are: account enumeration (2), session hijacking (6), SQL injection (3), cross-site scripting (4) and cross-site request forgery (5). We believe these to be the main 5 common vulnerabilities that could be abused in our site. These security mitigation's do not affect the usability, for example our cookies expire after one hour so that if the user quickly clicks off of the site, they can get back online without the needs to generate new cookies.

2 How ethics are essential to secure web development

When creating a secure website there are ethics that come into play. Having secure encryption is ideal so that people's data is safe and secure and not liable to theft from malicious sources. A downside to encryption is that since everything is encrypted, if anything malicious is to take place on our site, it could be very difficult to actually find the malicious people in question. We however did come to the conclusion that we need to focus on the user, as our product is designed for them and by protecting them, we believe this to be the ethical approach. We also make use of a system administrator who can delete posts that may be inappropriate.

Ethics come into play in other scenarios such a data breach. Companies can be faced with the conundrum that if there is a breach, do they either tell the users immediately or do they try and mitigate and contain the problem before it spreads. We attempt to do both as we make our users sign up with an email address so in the event of a data leak we can alert them via the use of email.

3 What mitigation's were coded for each security vulnerability, and should include discussions of pre-built libraries, hashing, encryption, usability etc.

3.1 Account enumeration

In order to protect against account enumeration is via implementing a captcha system so we can stop quick enumeration of usernames, this means that a malicious user cannot simply use bots to check if a username exists as they would need to verify every time. We do not have a way of telling the user that a username is already taken without exposing its existence as a whole, however we have figured that this is the best response as it does not impact the user too heavily. We do however make sure that the HTTP response time taken for an incorrect password is the same as the time taken for an already taken username. (2)

3.2 Session hijacking

We parse our sessions in cookies as we believe this solves a lot of the session hijacking possibilities For session hijacking we made sure to use the HTTP only tag for all cookies in order to insure they cannot be altered via javascript. In our project we would have ideally utilised HTTPS to force browsers to use TLS protocols, this would have ensured encryption between the user and the server for everything, including cookies. (6)

3.3 SQL Injection

We have countered against traditional string SQL injection as we use something called MongoDB which utilises a specific data format called BSON where queries are represented as binary data (ones and zeros) so direct injection through a string (text) is not available. There is no SQL code at all, as everything is d-one through a schema, all SQL statements are based on a model which stops a user from injecting SQL as they have no access to the SQL code whatsoever. This is essentially input sanitisation. (3)

3.4 Cross-site scripting

We implemented a node library called helmet (Helmet) which allowed us to use a content security policy. This allowed us to specify where we want the browser to run scripts and also use some built in features of the helmet library. This stops a malicious user from being able to inject HTML into text that the server will serve to the user because the script will not run from places that where we did not specify it could not run in the first place. (4)

3.5 Cross-site request forgery

The use of captcha helps us as the user needs to authorise before confirming important actions (such as login), this prevents cross-site forgery as a malicious user cannot automate a request without authentication from a captcha. All cookies we have implemented cannot be used in cross-site request forgery as they are all unique. We also make use of a secret cookie that retrieves a key from a .env file, utilising .env files means that we do not store sensitive information on our git meaning it cannot be stolen if an attack is to occur. (5)

4 What Authentication method/s were coded, and evidence of how they increase both usability and security.

4.1 2FA

Hackers do not care if a business is big or small, if it is liable to an easy attack they WILL go for it, this is why we have decided to utilise 2FA (2 factor authentication) when logging in to our site. 2FA halts an attack via the use of bots which essentially rapidly guess passwords against an account as once the password is guessed correctly the user will receive a 5 character code that they then need to input into the site so they can log in. This means that if a hacker would like to gain access to a persons account on our site, they need to also have access to said persons email address which is significantly harder to obtain than just a username and password for our site.

4.2 Passwords

Passwords are the base line of defense against unauthorised access to an account. The user can pick their own password, ideally something memorable to them but not anything too specific such as their name or just "password". The stronger the password the more protected the account and therefore the less are the chances for it to be breached.

4.2.1 Hashing our Passwords

We have irreversible hashing that means that with our B-crypt (bcrypt) library that if the same word was hashed for a second time it would never be the same. We authenticate passwords via comparing hashes to one another and also utilising a practice called "salting" which encrypts a password even further, we do this 8 times.

4.3 Captcha

We have lastly implemented captchas in another bid to halt brute force attacks on our users. Having a captcha present on login prevents the use of bots to rapidly attempt to log in to our site as I mentioned earlier in the 2FA section as it requires the user to verify a number of images before logging in. The second place we have implemented a captcha is in registration, this prevents the use of bots to create a multitude of accounts on our site which could lead to many fake accounts being created and in turn wasting resources.

5 Discussion and evidence of user testing

when a post is deleted, the ID of the post gets added to the URL. If the URL was changed before the URL gets checked, the cookie/session is checked to see if the person deleting it has actually made this request.

For our user testing we carried out multiple tests in order to make sure that the core functions of our website worked as well as the key security and authentication features.

6 Discussion and evidence of system unit testing

In this section I will detail the unit tests we carried out to identify bugs. The unit tests were carried out using the 'jest' library, which is a module for unit testing, with each test only being able to pass or fail

To make our code easy to test, we split up a lot of the code in the routes into functions with return values to test. Individual tests were written after each function was designed. The tests are located in the tests folder.

The first tests were carried out on auth.routes.js. The functions tested were authFn, encryptpass and authenticateEmail. The goal of the first was to ensure that authFn returns true or false depending on the session it reads. This helps identify any bugs if the session was incorrect. The encryption test was mainly to check if the hashing was valid by using compareSync to compare a string and a hash. This was expected to return true. The last test involved the process of sending emails. For this function we had to mock the nodemailer module and then get the results of the mock when calling the method.

The second test was carried out on getPosts() in post.routes.js. In this test mongoDB had to mocked through the library 'mockingoose'. This means we can get the model to return an element when a particular function is called. The method is called to check for any errors and then we retrieve results from the model, with the assertion that the model has returned the exact same object as our test data.

The final test was on user.routes.js. This checked the user existing method by also mocking mongoDB with test data. The function is then called and the assertion is that it returns true. The find operation is then carried out on the model to asser that the result matches our test data. Upon an error in the try/catch loop the test will fail with the message of the error.

All these tests can be found in our completed plan document. Our final coverage was 60 percent of lines being tested in the routes folder, 35 percent in the db.js file and 100 percent coverage on all of the models in the models folder.

7 Conclusion

If we had more time I think our main goal would be to secure against more against Cross site request forgery. Even though we believe that our captcha stops most rouge requests we believe it would be more beneficial to add csrf tokens to all our forms and make sure that every form that is submitted to the server contains the correct tokens in order to submit properly.

We definitely could improve on our authentication by adding another layer or even an easier and more secure layer such as using a mobile device's authentication app. This would be easier for the users to use and adds a physical level or security as you would need to have access to a singular mobile device.

Overall we believe that our food blog website would rate very well for security as we had designed our system from the ground up to be resistant to the most common forms of attacks that hackers would try. The fact that it is a blog website defines our security to be much less than say, a banking website but, any website that uses accounts should take very seriously the impacts of housing user information.

References

[bcrypt] bcrypt. Npm bcrypt package. `https://www.npmjs.com/package/bcrypt`.

[2] Hacksplaining. Avoiding user enumeration. `https://www.hacksplaining.com/prevention/user-enumeration`.

[3] Hacksplaining. Protecting against sql injection. `https://www.hacksplaining.com/prevention/sql-injection`.

[4] Hacksplaining. Protecting your users against cross-site scripting. https://www.hacksplaining.com/prevention/xss-stored.

[5] Hacksplaining. Protecting your users against csrf. https://www.hacksplaining.com/prevention/csrf.

[6] Hacksplaining. Protecting your users against session fixation. https://www.hacksplaining.com/prevention/session-fixation.

[Helmet] Helmet. Npm helmet package. https://www.npmjs.com/package/helmet.

A Appendix

Test Plan Creation Date	12/04/2022	
Created By	James Wright, George Beales	
Project Name	UG17 - Food Blog	

Sr.No.	**Module**	**Sub-module**	**Pre-Requisite**	**Steps to be followed**	**Expected Result**	**Actual Result**	**Comments**	**Status [Pass / Fail]**
1	Testing auth.routes	N/A	Nodemailer and bcrypt library. Auth.routes must be written. Nodemailer mock file set up	1. Run tests on 3 functions, mock any use of libraries. 2. Assert each function returns the expected value. 3. Check function produces no errors	authFN returns true when cookie created. False otherwise encryptpass returns no errors authenticateEmail successfully sends an email	encryptpass verified to be working by comparing the hashes. Authfn only returns true when the auth value is true in the cookie. authenticateEmail produced no errors and the console logged succesfully	nodemailer is mocked.	Pass
2	Testing post.routes	N/A	mockingoose library. Function getPosts must be written	1. Create test data 2.Mock the db library 3. Run the function getPosts to check for errors. 4. Assert the db library returns the test data	No errors found. The results from the model must match the test data object	No results can be found from the DB library	Incorrect model being mocked	Fail
3	Testing post.routes	N/A	mockingoose library. Function getPosts must be written	1. Create test data 2.Mock the db library 3. Run the function getPosts to check for errors. 4. Assert the db library returns the test data	No errors found. The results from the model must match the test data object	Results yielded match the test data. Correct result logged	Had to mock the model from posts instead of using the index	Pass
4	Testing user.routes	N/A	mockingoose library. Function checkUserExists must be written	1. Create test data 2.Mock the db library 3. Run the function checkUserExists to check for errors. 4. Assert the db library returns the test data 5.Assert the function returns true	No errors found. The function must return true and produce no errors	Results yielded match the test data. Correct result logged. Function returned true		Pass
5	Login System	User Testing on the login	Server running, and user details are known	1.Enter the user details 2. Complete captcha 3. Get authentication code 4. Check for login	Quick and complete login. Should be obvious to the user they are logged in by the navbar offering a profile page	Login took 45 seconds to complete with no autofill. Page for register/login changed to Profile. 2FA code is too long and captcha was too hard.	2FA code shortened to increase usability	Fail
6	Login System	User Testing on the login	Server running, and user details are known	1.Enter the user details 2. Complete captcha 3. Get authentication code 4. Check for login	Quick and complete login. Should be obvious to the user they are logged in by the navbar offering a profile page	Login took 30 seconds to complete with no autofill. Page for register/login changed to Profile.	2FA code shortened to increase usability	Pass
7	Registration	User testing on the Registration	Server running	1. Navigate to register page 2. Enter details for a new user 3. Complete captcha 4. Click register 5. Complete Authentication via e-mail	The user should be logged in and the account should be created	Registration took ~1 minute to complete. Authentication page loaded correctly and the website logged me in after registration. The account was also created.	Everything worked as expected	Pass
8	Viewing posts	User testing on viewing posts	Server running	1. Access the home page	The user should see all the posts that have been created fill up the screen.	All the posts stored on the website display for the user.	N/A	Pass
9	Creating a post	User testing on creating a post	Server running and user is logged in.	1. Navigate to the Create a post screen from the navbar. 2. Enter the post details 3. Upload an image for the post. 4. Click the create a post button.	The user should be able to easily upload information for their desired post and the post should be quickly made and made visible to the user on the home page immediately after creating the post.	The post gets created and the user is sent to the home page.	N/A	Pass
10	Deleting a post	User testing on deleting a post	Server running and user is logged in.	1. Navigate to the home page 2. Find a post that you have made on the login you are logged in on. 3. If you have not made a post refer to test 9. 4. Click the delete post button.	The user's post should get deleted and should no longer be visible.	The post gets deleted and the user can no longer see the post.	N/A	Pass
11	Editing a post	User testing on editing a post	Server running and user is logged in.	1. Navigate to the home page 2. Find a post that you have made on the login you are logged in on. 3. If you have not made a post refer to test 9. 4. Click the edit post button.	The user's post should be edited with their changes	The post is edited	N/A	Pass
12	Using the search bar	User testing on using the search bar	Server running	1. Navigate to the home page 2. Click on the search bar 3. Type what you would like to find 4. Press enter and find it	The user should be able to search via the search bar	They can utilise the search bar	N/A	Pass
13	Captcha system	User testing on the captcha system	Server running	1. Navigate to log in page 2. Attempt to login without captcha 3. See if you can successfully log in	The user should not be able to log in	The user cannot log in	N/A	Pass