

Faculty of Computer Science
CS6417
Software Security
Project Report – Winter 2023

by
Bhavesh Motiramani

1st April 2023

Table of Contents

- 1. Abstract.....3
- 2. Introduction3
- 3. Code Analysis.....5
- 4. Code Fix.....8
- 5. Discussion14
- 6. References16

1. Abstract

This report discusses the code analysis and fixes made to a Node.js, Express.js, and MongoDB-based blog website. The report highlights the vulnerabilities and issues found during the code analysis and the fixes to address them. It aims to analyze and understand different security issues that go unnoticed by developers and cause significant security breaches. It emphasizes using static code analysis tools and how they can be beneficial in solving security-related problems.

The report discusses the security issues in the blog web app, majorly the importance of secure random integer generators, proper input sanitization, disabling technology fingerprints, and avoiding hardcoding sensitive information in web applications. It explains the risks associated with each vulnerability and the corresponding fix applied to address it.

The report concludes by emphasizing the importance of secure coding practices and the need to prioritize security in web application development. It highlights the importance of implementing static code analysis tools to identify vulnerabilities and provides insight into best practices for fixing security issues. It also highlights the lessons learned about vulnerabilities and how to avoid them in the future. Lastly, it talks about the pros and cons of the static code analysis tools used in developing secure software.

2. Introduction

Static code analysis tools are used to analyze software code; the output of static code analysis tools gives the developers a depth understanding of security issues in the code. Static code analysis tools do not execute the code; they read the lines of code and compare it with the rule base to check if the application is following standards. These tools not only aid in finding the issues but also determine other essential metrics such as lines of code, code coverage, code smell, and many others, thus improving the code quality.

Using static code analysis is beneficial for developers. First, developers need to know certain practices in a particular programming language. For example, a Java developer does not have to worry about pointers, while a C programmer does have to check for memory references. Secondly, static code analysis tools help to identify bugs and vulnerabilities in a visually pleasing way, which is much easier to understand as a human. Thirdly, they provide possible solutions to the bugs or vulnerabilities, including the reference vulnerabilities from trusted websites like CWE and OWASP, which is beneficial for developers.

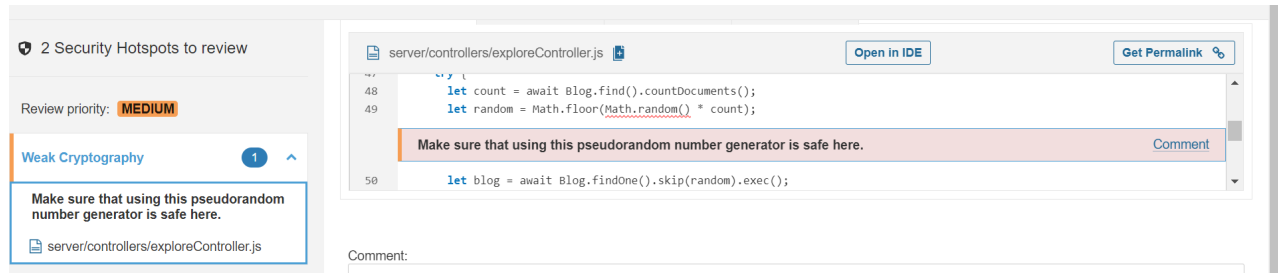
This project introduces a blog website created with HTML, Bootstrap, CSS, JavaScript, Express.js, and MongoDB, providing users with a visually appealing and user-friendly platform to publish their blogs. The website includes several pages such as Home, Write, Contact Us, and Template, each serving a unique purpose to enhance the user's experience. The Write page enables users to create and publish their blogs, while the Contact Us page allows them to ask questions and

concerns. The Template page lets users create personalized templates for their blog posts. The project uses MVC architecture as a clean code practice; it is divided into different folder structures – controller (stores the backend logic), views (stores the ejs pages), models (stores the schema files), routes(stores the different routes), public(stores the CSS, js pages).

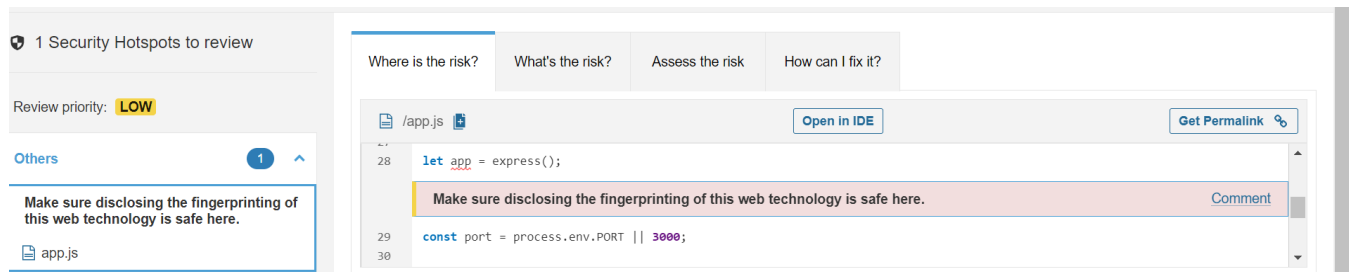
SonarQube and Synk, static code analysis tools, were used to find the security issues in the application. SonarQube scans HTML and JavaScript pages, while Synk is a static code analyzer tool for Node. js-based applications. These tools were chosen because of their ability to analyze the code in detail while providing potential bugs and vulnerabilities. The SonaQube tool provides a detailed analysis of bugs, code smell, and vulnerabilities. The Synk tool scans .ejs files, which SonarQube does not support. The Synk tool provided a detailed approach and explanation for resolving issues for the application.

3. Code Analysis

1. The below screenshot shows there is a vulnerability in the code as it is using ***Math.random()***. Using pseudorandom number generators such as ***Math.random()*** is prone to security risks because it may be possible for the attacker to guess any random number, thus compromising the security of our application.



2. The below image shows there was an issue with **disclosing technology fingerprints**, which refers to revealing the unique characteristics of an application. This code does not disable the '**x-powered-by**' header and may send it in HTTP responses. This could reveal that the server is powered by Express.



3. The secret value is **hardcoded** as shown in the below snippet. When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. Attackers can use hardcoded values to their advantage by exploiting vulnerabilities in the software that uses these values.

H Hardcoded Secret

SNYK CODE | CWE-547

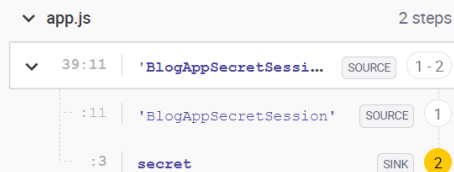
Data flow

Fix analysis



Avoid hardcoding values that are meant to be secret.
Found a **hardcoded string** used in [here](#).

Data flow - 2 steps in 1 file



Find out how to remediate this issue through our
[Fix analysis](#)

```
app.js
32
33 app.use(express.urlencoded({ extended: true }));
34 app.use(express.static('public'));
35 app.use(expressLayouts);
36
37 app.use(cookieParser('BlogAppSecure'));
38 app.use(session({
39   secret: 'BlogAppSecretSession',
40   saveUninitialized: true,
41   resave: true
42 }));
43 app.use(flash());
44 app.use(fileUpload());
45
46 app.set('layout', './layouts/main');
```

4. This is an example of **Cross Side Scripting**. The below image shows the input that the input is not sanitized in code.

Unsanitized input from an uploaded file flows into `send`, where it is used to render an HTML page returned to the user. This may result in a Cross-Site Scripting attack (XSS).

H Cross-site Scripting (XSS)

SNYK CODE | CWE-79

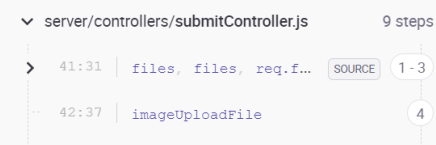
Data flow

Fix analysis



Unsanitized input from an **uploaded file** flows into `send`, where it is used to render an HTML page returned to the user. This may result in a Cross-Site Scripting attack (XSS).

Data flow - 9 steps in 1 file



Find out how to remediate this issue through our
[Fix analysis](#)

```
server/controllers/submitController.js
40
41 imageUploadFile = req.files.image;
42 newImageName = Date.now() + imageUploadFile.name;
43
44 uploadPath = require('path').resolve('.') + '/public/upload';
45
46 imageUploadFile.mv(uploadPath, function(err){
47   if(err) return res.status(500).send(err);
48 })
49
50 }
51
52 const newRecipe = new Blog({
53   name: req.body.name,
54   description: req.body.description,
```

5. Below is an example of **sensitive data exposure**.

An error object flows to `send` and is leaked to the attacker. This may disclose important information about the application to an attacker.

The error in the `send` can leak sensitive data about the website.

M Information Exposure

SNYK CODE | CWE-200

Data flowFix analysis

An error object flows to `send` and is leaked to the attacker. This may disclose important information about the application to an attacker.

Data flow - 3 steps in 1 file

server/controllers/submitController.js3 steps

47:47err, err, sendSOURCE1-3

:47errSOURCE1

:47err2

Find out how to remediate this issue through our [Fix analysis »](#)

server/controllers/submitController.js

```
40
41     imageUploadFile = req.files.image;
42     newImageName = Date.now() + imageUploadFile.name;
43
44     uploadPath = require('path').resolve('.') + '/public/upload:
45
46     imageUploadFile.mv(uploadPath, function(err){
47         if(err) return res.status(500).send(err);
48     })
49
50 }
51
52 const newRecipe = new Blog({
53     name: req.body.name,
54     description: req.body.description,
```

4. Code Fix

1. Below image shows the fix applied to avoid the risks associated with **Math.random()** function.

Math.random() was replaced with **crypto.randomInt(0,count-1)**.

A random integer generator that is cryptographically secure guarantees that the values produced are difficult to predict and not inclined towards any specific range of values.

Before Fix:

```
/**
 * GET /explore-random
 * Explore Random as JSON
 */
exports.exploreRandom = async(req, res) => {
  try {
    let count = await Blog.find().countDocuments();
    let random = Math.random()*count;
    let blog = await Blog.findOne().skip(random).exec();
    res.render('explore-random', { title: 'Blog - Random', blog } );
  } catch (error) {
    res.status(500).send({message: error.message || "Error Occured" });
  }
}
```

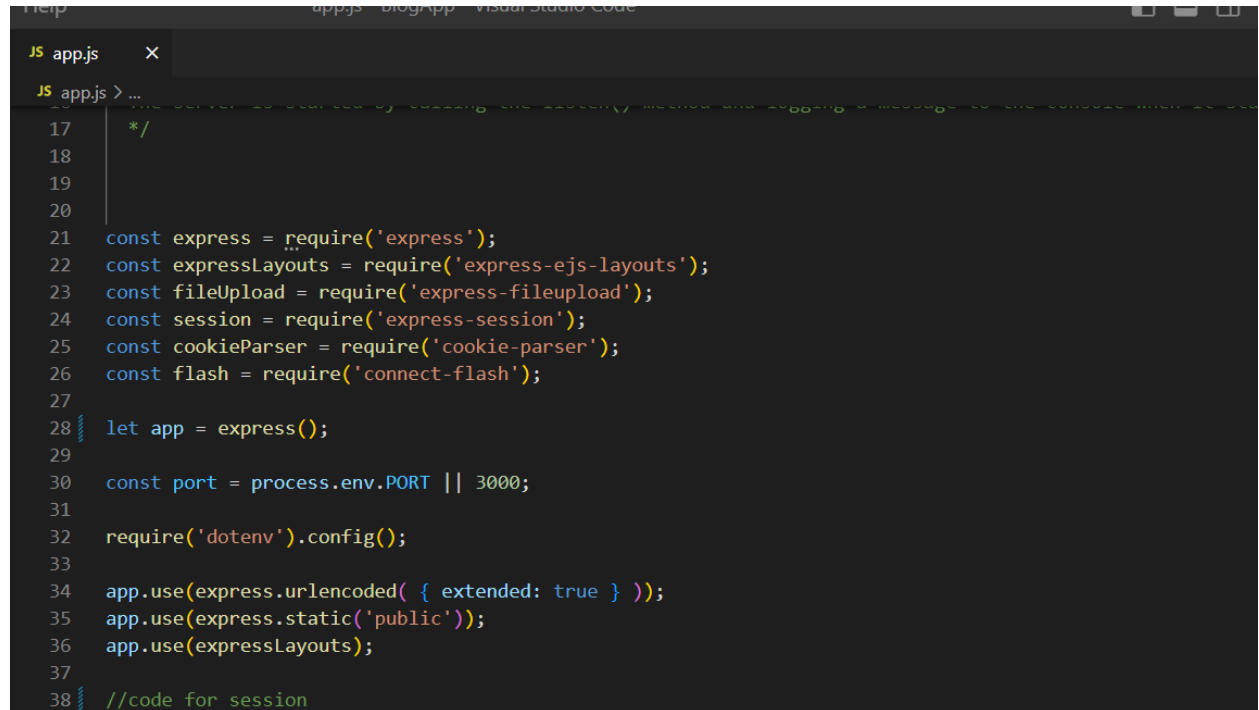
After Fix:

```
/**
 * GET /explore-random
 * Explore Random as JSON
 */
exports.exploreRandom = async(req, res) => {
  try {
    let count = await Blog.find().countDocuments();
    let random = crypto.randomInt(0, count - 1);
    let blog = await Blog.findOne().skip(random).exec();
    res.render('explore-random', { title: 'Blog - Random', blog } );
  } catch (error) {
    res.status(500).send({message: error.message || "Error Occured" });
  }
}
```


2. To avoid disclosing technologies used on a website, **x-powered-by HTTP** header was disabled.

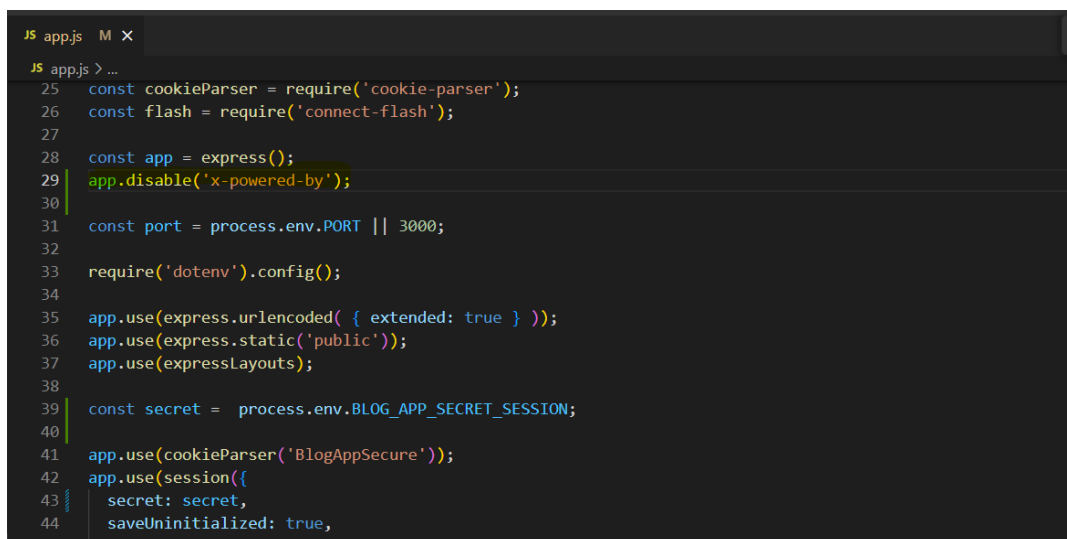
A new line of code was added: **app.disable('x-powered-by')**. This helps to avoid the risk of revealing information about the application.

Before Fix:



```
JS app.js x
JS app.js > ...
17  */
18
19
20
21  const express = require('express');
22  const expressLayouts = require('express-ejs-layouts');
23  const fileUpload = require('express-fileupload');
24  const session = require('express-session');
25  const cookieParser = require('cookie-parser');
26  const flash = require('connect-flash');
27
28  let app = express();
29
30  const port = process.env.PORT || 3000;
31
32  require('dotenv').config();
33
34  app.use(express.urlencoded( { extended: true } ));
35  app.use(express.static('public'));
36  app.use(expressLayouts);
37
38  //code for session
```

After Fix:



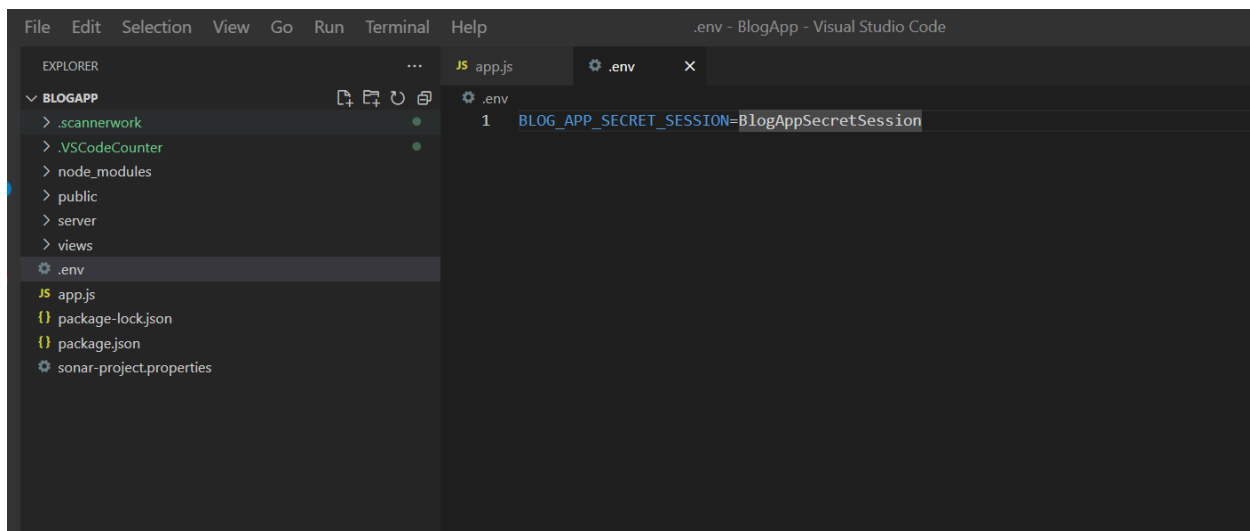
```
JS app.js M X
JS app.js > ...
25  const cookieParser = require('cookie-parser');
26  const flash = require('connect-flash');
27
28  const app = express();
29  app.disable('x-powered-by');
30
31  const port = process.env.PORT || 3000;
32
33  require('dotenv').config();
34
35  app.use(express.urlencoded( { extended: true } ));
36  app.use(express.static('public'));
37  app.use(expressLayouts);
38
39  const secret = process.env.BLOG_APP_SECRET_SESSION;
40
41  app.use(cookieParser('BlogAppSecure'));
42  app.use(session({
43    secret: secret,
44    saveUninitialized: true,
45    resave: true
```

3. The issue of hardcoded values was fixed by creating a **.env** file and secret key was added to it. The .env file is used to store sensitive information and configuration variables that are used in the application code.

Before Fix

```
JS app.js
JS app.js > ...
22 const expressLayouts = require('express-ejs-layouts');
23 const fileUpload = require('express-fileupload');
24 const session = require('express-session');
25 const cookieParser = require('cookie-parser');
26 const flash = require('connect-flash');
27
28 let app = express();
29
30 const port = process.env.PORT || 3000;
31
32 require('dotenv').config();
33
34 app.use(express.urlencoded( { extended: true } ));
35 app.use(express.static('public'));
36 app.use(expressLayouts);
37
38 //code for session
39
40 app.use(cookieParser('BlogAppSecure'));
41 app.use(session({
42   secret: 'BlogAppSecretSession',
43   saveUninitialized: true,
44   resave: true
45 }));
46 app.use(flash());
47 app.use(fileUpload());
48
49 app.set('layout', './layouts/main');
50 app.set('view engine', 'ejs');
```

Fix:



Created a .env file and added the **secretkey** value pair in the section.

Next, the variable defined in .env file was loaded in app.js using the `require('dotenv').config()`.

A variable `secret` was created to access the secret session variable from the .env file.

```
const secret = process.env.BLOG_APP_SECRET_SESSION;
```

Finally, `secret` variable was used to access the secret.

secret: **secret**,

```
const app = express();
const port = process.env.PORT || 3000;

require('dotenv').config();

app.use(express.urlencoded( { extended: true } ));
app.use(express.static('public'));
app.use(expressLayouts);

const secret = process.env.BLOG_APP_SECRET_SESSION;

app.use(cookieParser('BlogAppSecure'));
app.use(session({
  secret: secret,
  saveUninitialized: true,
  resave: true
}));
app.use(flash());
app.use(fileUpload());

app.set('layout', './layouts/main');
app.set('view engine', 'ejs');

const routes = require('./server/routes/BlogRoutes.js')
app.use('/', routes);
```

4. The issue was fixed by **sanitizing** all the inputs to avoid XSS attacks.
A new package was imported in the application named *sanitize-html*. The package sanitizes the HTML input.
All the inputs in *submitBlogOnPost* function were updated to take **sanitized-input**.

Before Fix:

```
var uploadPath;
var newImageName;

if(!req.files || Object.keys(req.files).length === 0){
  console.log('No Files where uploaded.');
```

```
} else {

  imageUploadFile = req.files.image;
  newImageName = Date.now() + imageUploadFile.name;

  uploadPath = require('path').resolve('./') + '/public/uploads/' + newImageName;
```

After Fix:

```
server > controllers > JS submitController.js > ...
37   var newImageName;
38
39   if(!req.files || Object.keys(req.files).length === 0){
40     console.log('No Files where uploaded.');
```

```
41   } else {
42
43     imageUploadFile = req.files.image; // user can add malicious file input
44     newImageName = Date.now() + sanitizeHtml(imageUploadFile.name);
45
46     uploadPath = require('path').resolve('./') + '/public/uploads/' + newImageName;
47
48     imageUploadFile.mv(uploadPath, function(err){
49       if(err) return res.status(500).send(err);
50     })
51   }
52
53   const newRecipe = new Blog({
54     name: sanitizeHtml(req.body.name),
55     description: sanitizeHtml(req.body.description),
56     email: sanitizeHtml(req.body.email),
57     category: sanitizeHtml(req.body.category),
58     image: sanitizeHtml(newImageName)
59   });
60
61   await newRecipe.save();
62
63   req.flash('infoSubmit', 'Blog submitted...')
64   res.redirect('/');
65 } catch (error) {
66   // res.json(error);
67   req.flash('infoErrors', error);
68   res.redirect('/');
```

```
69 }
70 }
```

5. The fix was made by creating a string error message and adding the error message in console.log for the developers. This way users can't see the error details revealed and developers get to see the errors if any.

In this code, we are logging the error to the server console using **console.error()**, and sending a generic error message to the client using **res.status().send()**. This way, we are not revealing any sensitive information to the attacker.

Before Fix:

```
JS submitController.js
server > controllers > JS submitController.js > submitBlogOnPost > submitBlogOnPost > imageUploadFile.mv() callback

33  /**
34   * POST /submit-blog
35   * Submit Blog
36   */
37  exports.submitBlogOnPost = async(req, res) => {
38    try {
39
40      var imageUploadFile;
41      var uploadPath;
42      var newImageName;
43
44      if(!req.files || Object.keys(req.files).length === 0){
45        console.log('No Files where uploaded.');
```

After Fix:

```
imageUploadFile = req.files.image; // user can add malicious file input
newImageName = Date.now() + sanitizeHtml(imageUploadFile.name);

uploadPath = require('path').resolve('.') + '/public/uploads/' + newImageName;

imageUploadFile.mv(uploadPath, function(err){
  if(err) {
    console.error(err);
    return res.status(500).send("An error ocuured while uploading image");
  }
});
```

5. Discussion

This project helped me better understand Node.js, Express.js, and MongoDB. Through this project, I gained knowledge about different types of routes (GET, POST, PUT, DELETE) and implemented them to create a blog website. I was able to integrate and use static code analysis tools to find bugs and vulnerabilities in the project.

Further, the project taught me about different vulnerabilities that developers often ignore, which can compromise the security of applications, and hands-on best practices to fix those security issues. Some of the lessons learned from analyzing issues were:

- We should not rely on functions provided by programming languages such as `Math.random()`. These functions can sometimes be vulnerable to attacks. Cryptographically secure random integer generators are essential for many security applications, such as encryption, key generation, and digital signatures, where the quality of randomness is critical to the system's security. By ensuring that the generated values are unpredictable and unbiased, a cryptographically secure random integer generator helps to protect against attacks that rely on the predictability or bias of the generated values.
- Disclosing technology fingerprints such as the `x-powered-by` HTTP header allows attackers to gather information about the technologies used to develop web applications and perform attacks. To prevent this, it is recommended to disable the `x-powered-by` HTTP header rather than setting it to a random value.
- Hard-coding security-relevant constants are also considered bad coding practices and should be avoided if present. Instead, symbolic names or configuration lookup files should be used. Another adverse effect of hard-coding constants is potential application performance unpredictability if the development team fails to update every instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants are considered lousy coding practices and should be remedied if present and avoided in the future.
- Improper input sanitization can lead to Cross-Side-Scripting attacks. The browser unknowingly executes the malicious script on the client side to perform actions otherwise typically blocked by the browser's Same Origin Policy. It is essential to sanitize all the data input in HTTP requests or user input before reflecting it, ensuring all data is validated, filtered, or escaped before echoing anything back to the user, such as the values of query parameters during searches. Using and enforcing a Content Security Policy or allowing users to disable client-side scripts can avoid XSS attacks.
- It is bad practice to send data with errors on a webpage. Always try to give the user as little detail as possible about the application. Sensitive data exposure can aid the attacker in attacking using the application details. It is crucial to encrypt data permanently, whether in transit or at rest, to avoid sensitive data exposure. Use store passwords with secure, salted hashing functions.

SonarQube and Snyk were used as static code analysis tools for this project. Some pros and cons of tools are discussed below:

SonarQube:

- Pros:
 - Provides comprehensive details about lines of code, code smell, vulnerabilities, bugs, code coverage, and other metrics.
 - It provides details about vulnerabilities/bugs, the issue, and a detailed description of fixing it.
- Cons:
 - Produced false positive results for node_modules in my code.
 - No option to scan the .ejs file extension.
 - Did not provide security issues for .js extension files.

Snyk:

- Pros:
 - Provides detailed vulnerability information for open-source components and dependencies.
 - Provides remediation and detailed guidance on solving vulnerabilities.
- Cons:
 - Requires access to GitHub code; only partially secure.
 - Does not provide metrics such as code smell.

6. References

- [1] Static code analysis (no date). "Static Code Analysis | OWASP Foundation." Available at: https://owasp.org/www-community/controls/Static_Code_Analysis
- [2] Synk. "Node.js Security." Available at: <https://snyk.io/>
- [3] SonarQube. "Static Code Analysis." Available at: <https://www.sonarqube.org/>
- [4] Node.js. "About." Available at: <https://nodejs.org/en/about/>
- [5] Express.js. "About." Available at: <https://expressjs.com/>
- [6] MongoDB. "About." Available at: <https://www.mongodb.com/>
- [7] OWASP. (n.d.). Fingerprint Web Application Framework. OWASP Web Security Testing Guide. Available at: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/01-Information_Gathering/08-Fingerprint_Web_Application_Framework
- [8] Express.js. "Production Best Practices: Security." Available at: <https://expressjs.com/en/advanced/best-practice-security.html>
- [9] Bootstrap. "Introduction." Available at: <https://getbootstrap.com/docs/5.3/getting-started/introduction/>
- [10] OWASP. (n.d.). Cross-Site Scripting (XSS). <https://owasp.org/www-community/attacks/xss/>
- [11] Understanding how to prevent Sensitive Data Exposure. <https://owasp.org/www-pdf-archive/Owaspsuffolk-20190319.pdf>