Secure Code Review

Project Name:

Task 5: Secure Code Review

Review Date:

31.10.2024

Reviewer:

Nijat Mazanli

Introduction

1. What is secure code reviewing?
2. Why is this important?
3. Where do we check while reviewing code?
4. What I found in the Github repository?
5. Best practices and Recommendations

Note:

First of all, I want to mention that I did this review on my old Github project. I wrote this project when I started to learn JavaScript. So, I start with what is Secure Code Review.

What is the Secure Code Review?

Secure Code Review is a process used by cybersecurity researchers to identify vulnerabilities and injections in code. It's crucial for all kinds of development because it helps enhance security and protect applications from potential threats.

Why is this important?

Secure Code Review is important like intrusion prevention tools. Every product should be checked in this way before it goes online. Due to the hundreds of lines of code, the developers may make an important mistake and this mistake may cause a serious vulnerability. Like that repository İ checked. Although, underlining where I check while reviewing this code helps you to understand.

Where do we check while reviewing code?

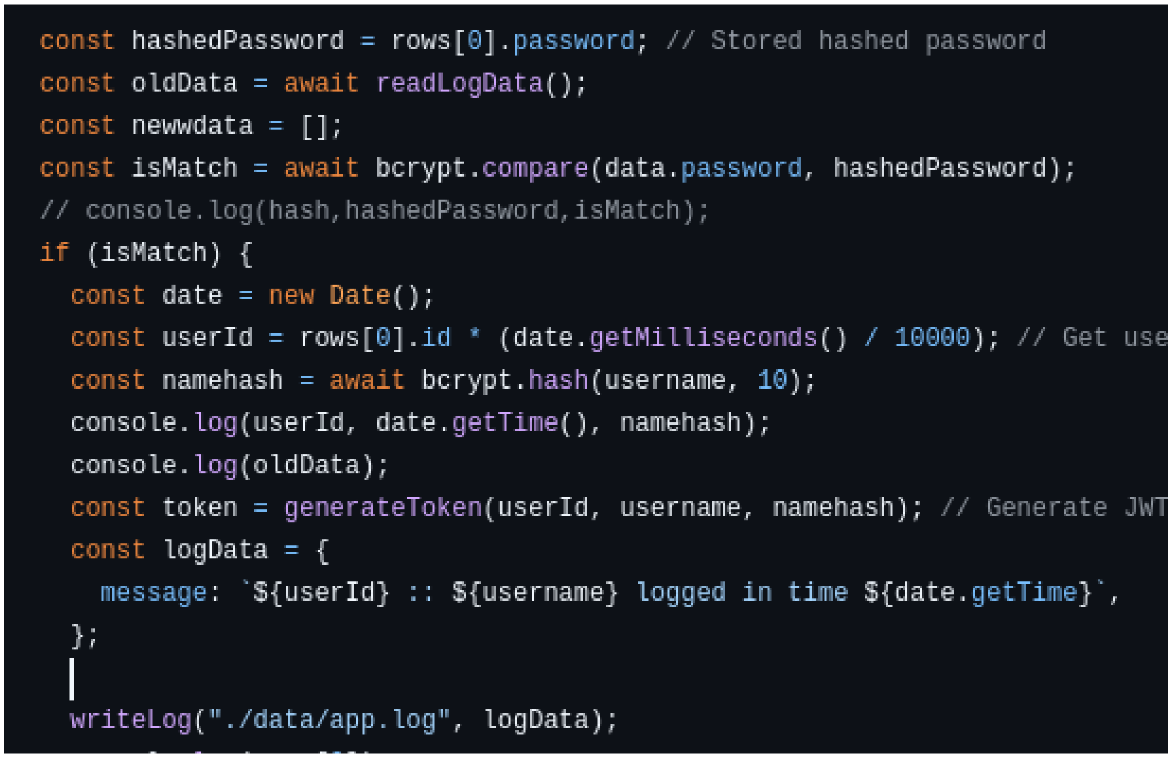
Commonly, I looked for:

1. **Input Validation and Sanitization:**
   - For finding any possible injections like XSS and SQLi.
2. **Authentication and Authorization:**
   - For finding broken authentication and how to access sensitive or unauthorized files.
3. **Session Management:**
   - Implementing secure session management techniques to prevent session hijacking and other related attacks.
4. **Cryptography:**
   - Weak cryptographic methods are a green light for hackers that you can steal my sensitive data.
5. **Error Handling and Logging:**
   - For finding Log poisoning attacks and finding which data are logging in log files.
6. **Security Configuration:**
   - For detecting outdated libraries and frameworks. Some versions have serious vulnerabilities..

What I found in the Github repository?

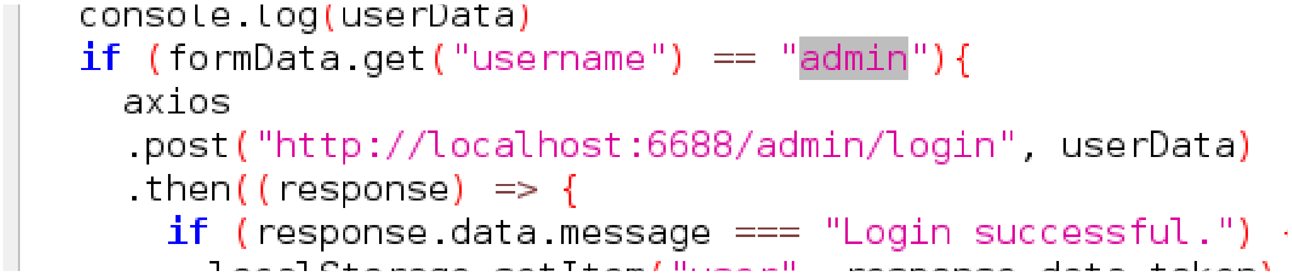
I found several vulnerabilities and injections in the Github project. The code has over one hundred lines of code. So I will put only where the project's source has a vulnerability.

- **Log Poisoning**
  - **Location:** userRoutes.js, line 175,138,211.
    adminRoutes.js, line 162,182,261.
  - **Description:** Log Poisoning is an attack vector that an attacker injects malicious code to a log functionality of a server and , for example, you inject a code that reveals servers local ip or username. It has two common techniques used in this attack: Log Forging(creating fake logs,corrupting log files) and Log Injection. Our situation contains both of them. In this project you cannot access logs but when you gain access to logs, you can see poisoned log files on the server.
    This code snippet was taken from the Github project. As you can see, no sanitization was applied. So this can lead to Log Poisoning injection.



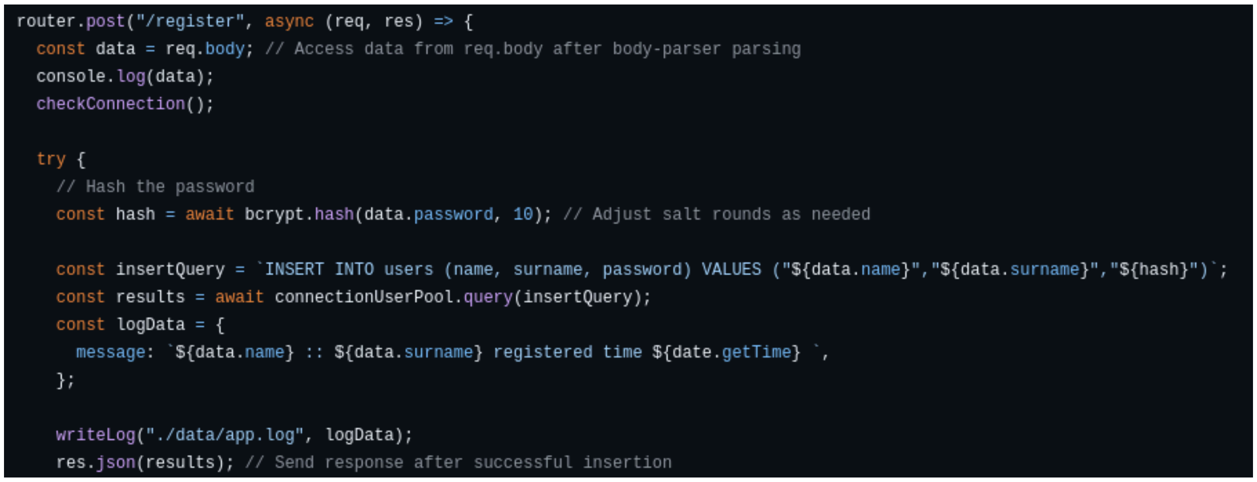
```
const hashedPassword = rows[0].password; // Stored hashed password
const oldData = await readLogData();
const newwdata = [];
const isMatch = await bcrypt.compare(data.password, hashedPassword);
// console.log(hash,hashedPassword,isMatch);
if (isMatch) {
  const date = new Date();
  const userId = rows[0].id * (date.getMilliseconds() / 10000); // Get use
  const namehash = await bcrypt.hash(username, 10);
  console.log(userId, date.getTime(), namehash);
  console.log(oldData);
  const token = generateToken(userId, username, namehash); // Generate JWT
  const logData = {
    message: `${userId} :: ${username} logged in time ${date.getTime}`,
  };

  writeLog("./data/app.log", logData);
```

- **Information Disclosure: admin username found**
  - **Location:** All frontend JavaScript files.
  - **Description:** The username of the admin is open and can easily be found in JavaScript code belonging to the frontend. This allows the attacker to narrow the attack range and take over the admin user faster.



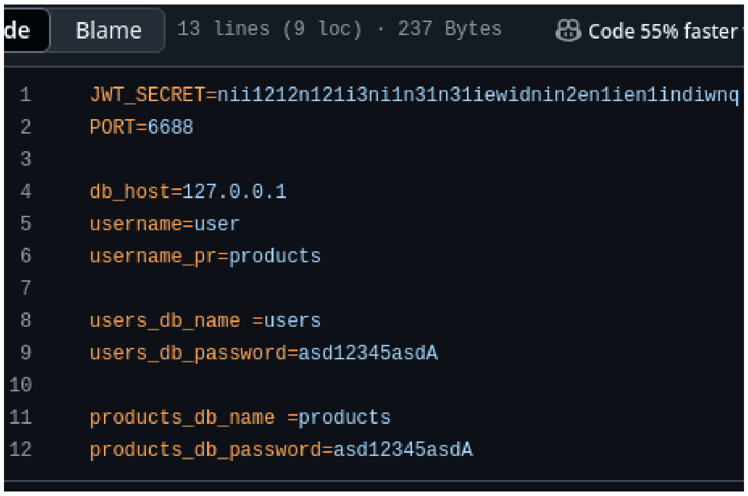
```
console.log(userData)
if (formData.get("username") == "admin"){
  axios
  .post("http://localhost:6688/admin/login", userData)
  .then((response) => {
    if (response.data.message === "Login successful.")
```

- **SQLi**
  - **Location:** userRoutes.js, line 176.
  - **Description: SQLi (SQL Injection)** is one of the critical injections. With this injection, the attacker can access and dump the database of a web server or web site easily. SQLi has 3 categories:*In-bound, Blind, Out-of-bound*. In this situation, you can see Blind SQL injection. In this code snippet, you can see that the developers didn't perform any Esanitization on variables and put them directly into the query. This fault increases the success rate of SQLi.



```
router.post("/register", async (req, res) => {
  const data = req.body; // Access data from req.body after body-parser parsing
  console.log(data);
  checkConnection();

  try {
    // Hash the password
    const hash = await bcrypt.hash(data.password, 10); // Adjust salt rounds as needed

    const insertQuery = `INSERT INTO users (name, surname, password) VALUES ("${data.name}","${data.surname}","${hash}")`;
    const results = await connectionUserPool.query(insertQuery);
    const logData = {
      message: `${data.name} :: ${data.surname} registered time ${date.getTime} `,
    };

    writeLog("./data/app.log", logData);
    res.json(results); // Send response after successful insertion
```

- **Sensitive Data Exposure**
  - **Location:** .env
  - **Description:** Exposure of confidential data of company or web server. If you take a look at folders and files in the Github repository, you see this file : “.env”. In JavaScript the ".env” file keeps envoiremental variables inside. DB name, DB password and other things that are important for project stored in this file.



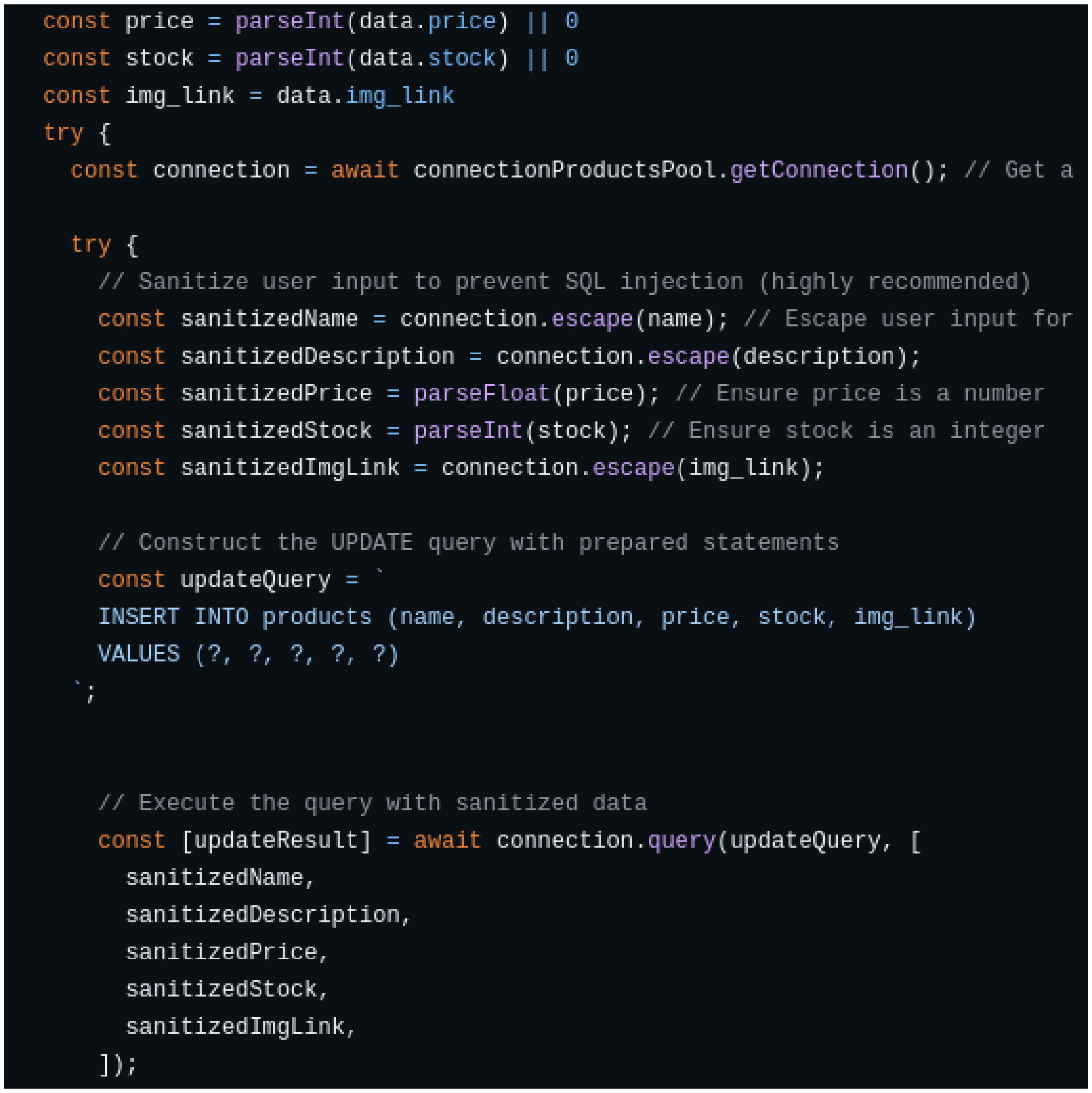
```
JWT_SECRET=nii1212n121i3ni1n31n31iewidnin2en1ien1indiwnq
PORT=6688

db_host=127.0.0.1
username=user
username_pr=products

users_db_name =users
users_db_password=asd12345asdA

products_db_name =products
products_db_password=asd12345asdA
```

As you see, JWT token, db usernames, passwords are revealed. This is also one of the biggest mistakes that developers make. If an attacker is able to access this file, this will be the worst scenario ever.

Normally, developers must put this file only on the server, but in this example, the developers uploaded it to Github.

Best practices and Recommendations

I recommend sanitizing all inputs, wherever they come from. Most of these injections are based on weak and unsanitized inputs. Developers must add filters to input fields like only number or only string. I give another great example for this in the same Github repository. In the next page, you see that developers added sanitizing to product adding functionality. All inputs are sanitized and this will reduce the success percentage of SQLi or XSS injections.For this job, developers can use the express-validator library. This library is the best option for developers. Additionally, developers' way to insert data to SQL query also reduces risk of SQLi. Sanitization also affects Log poisoning injection, but not prevents it. Adding more security methods can prevent Log injection and Log Forging.



```
const price = parseInt(data.price) || 0
const stock = parseInt(data.stock) || 0
const img_link = data.img_link
try {
  const connection = await connectionProductsPool.getConnection(); // Get a

  try {
    // Sanitize user input to prevent SQL injection (highly recommended)
    const sanitizedName = connection.escape(name); // Escape user input for
    const sanitizedDescription = connection.escape(description);
    const sanitizedPrice = parseFloat(price); // Ensure price is a number
    const sanitizedStock = parseInt(stock); // Ensure stock is an integer
    const sanitizedImgLink = connection.escape(img_link);

    // Construct the UPDATE query with prepared statements
    const updateQuery = `
    INSERT INTO products (name, description, price, stock, img_link)
    VALUES (?, ?, ?, ?, ?)
  `;


    // Execute the query with sanitized data
    const [updateResult] = await connection.query(updateQuery, [
      sanitizedName,
      sanitizedDescription,
      sanitizedPrice,
      sanitizedStock,
      sanitizedImgLink,
    ]);
```

Example of secure and sanitized input validation.

For Sensitive Data Exposure, actually, developers put Frontend and Backend to separate folders or separate servers. This means the attacker can only access the frontend side of the site, not server side. This method also prevents many injections and is used by many companies, but putting sensitive data to public spaces like Github is also a security breach.

For Information Disclosure, hiding any user's username or password is nessesary for improving security of applications. Any kind of data is confidential and must not be accessible for anyone except the company employees. To prevent this, developers must be educated about cyber attacks on websites.

Thanks for reading.