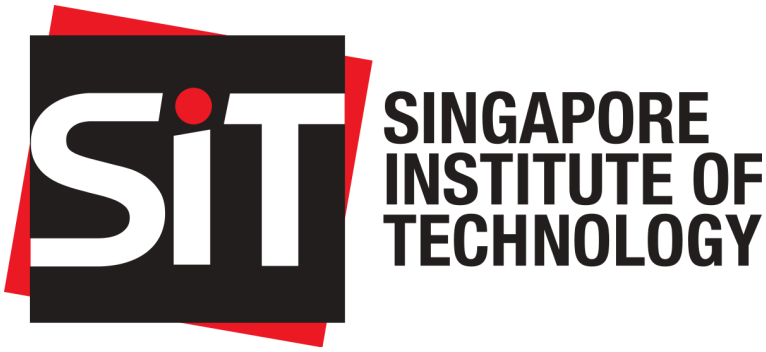


Deliverables 3 - Secure Software Requirement Analysis and Design

A report submitted to the

Singapore Institute of Technology in partial fulfillment of the requirements for the BACHELOR OF INFORMATION AND COMMUNICATIONS TECHNOLOGY (SOFTWARE ENGINEERING / INFORMATION SECURITY)

Team Name: Team 18

Submitted on: 08/07/24

GitHub handle: ICT2101-P11-6

Name	Student Number
Chiu Zheng Hao Jonathan	**2203187**
Goh Yue Jun	**2201471**
Muhammad Fiqri Adam	**2202878**
Nicholas Sng Ray Shiang	**2203197**
Denzel Low	**2202347**
Kee Han Xiang	**2201423**
Jeremy Lim Min En	**2203516**

Table of Contents

1. Implementation Walkthrough

1.1 CI/CD

Analysis	The content in their jenkinsfile shows that they have implemented OWASP Dependency check, unit testing, integration and UI test but not SonarQube Scanning.
Recommendation & Evaluations	It is recommended to use SonarQube scanning as part of the CI/CD.

1.2 Sequence Diagram does not reflect codes

Analysis: Mismatch of codes and sequence diagram made it confusing for our team to understand the flow. Our team noticed that the code checked if a user status is either not activated or not deactivated, but the diagram shows not activated or deactivated as seen in the figure below. There were also a few missing database calls to check user email, get HOTP secrets and update sessions. Deliverable 2 stated that "including calls to 3rd party components or APIs if applicable" but sequence diagram is missing logger and utils entity.

The report also failed to mention how MFA was implemented.

```
if (user_exist.activation_status !== "activated" && user_exist.activation_status !== "deactivated") {
```

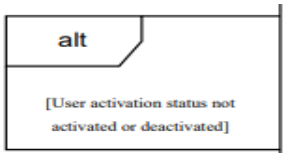


Figure 1. Sequence Diagram mismatch

The team also found incorrect code status reflected in the sequence diagram as seen in Figure 2 such as the inserting of incorrect password returned a 401 status code rather than the 403 status indicated in the sequence diagram.

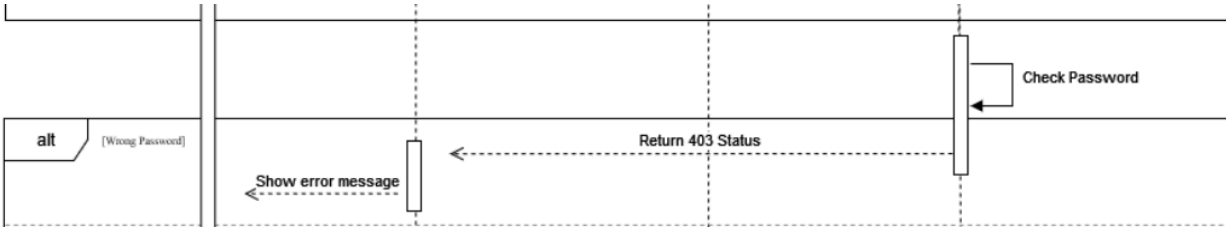


Figure 2. Incorrect input format insertion of credentials

```
// Login user if email and password are correct and account is activated
try {
    const result = await user.login_user(email);

    if (result === null) {
        logLoginAttempt(email, false, "Reason: Invalid email or password");
        return res.status(401).json({ message: "Invalid email or password" });
    }
```

Figure 3. Code depicting error response to incorrect credentials

1.3 User authentication

Analysis	D2 requirement stated "How is server authentication implemented?", but server authentication was not found in the report. It is important as it shows how the server ensure its identity is verified before making a secured connection to the client. D2 requirement also stated, "If implementing HTTPS, what cipher suites are enabled and why". The report only mentioned how they implement HTTPS and where the server's private key is stored and whether it's secure. But it failed to mention and explain the choices of cipher suites.
Recommendation & Evaluations	Ensure server authentication details and enabled cipher suites are documented for the team to verify security measures and compliance with best practices during testing and evaluation.

1.4 Password storing

Analysis	This section details the secure storage of user passwords and the implementation of a common password checker. User passwords are stored as salted-hashed passwords, using a randomly generated salt and the PBKDF2 algorithm with 1000 iterations, a 64-byte key length, and the SHA-256 hash function. Database access follows the least privilege principle, and communication is secured with SSL/TLS encryption. During account registration, a common password checker prevents weak passwords by comparing them against a curated list, ensuring they meet security criteria. This check is performed asynchronously on the backend, providing users with immediate feedback on their password's security.
Recommendation & Evaluations	The Section is well-detailed.

1.5 Session management

Analysis	This section describes the secure implementation of session management in the application, using express-session middleware with express-mysql-session to store session data on the server. Sessions are initialized with a unique session ID signed with a secret key using HMAC_256. Session data, including the user ID and role, is stored in a MySQL database. To enhance security, the application sets 'httpOnly' and 'secure' attributes for cookies, preventing them from being accessed through client-side scripts or transmitted over unencrypted connections, reducing the risk of session hijacking, replay attacks, and XSS. Additionally, sessions have a maximum lifespan of 20 minutes, after which they expire, and CSRF protection is implemented by verifying session data against the database.
Recommendation & Evaluations	The Section is well-detailed.

1.6 Access control

Analysis	This section details the secure implementation of Role-Based Access Control (RBAC) in the application. Upon logging in, users are assigned a role, and based on this role, they are directed to the appropriate landing page. Access to specific pages is restricted according to the user's role: regular users can access pages like the cart, current rentals, and listing new equipment, while admins are restricted from these pages. Conversely, admins have access to their own profile page, which users cannot access. This separation ensures that users and admins can only access the functionalities and information relevant to their respective roles, maintaining security and integrity within the application.
Recommendation & Evaluations	The Section is well-detailed.

1.7 Secure coding

Analysis	This section outlines the secure practices implemented in the application, including credential storage, input sanitization, SQL handling, communication security, and access control. User passwords are securely stored using salted-hashed methods, while reCAPTCHA keys are encrypted with AES-256 in CBC mode. Input sanitization is performed using DOMPurify, express-validator, and sanitize-html to prevent XSS attacks and other malicious inputs. SQL injection attacks are mitigated using prepared statements. SSL/TLS encryption ensures secure data transmission, and rate limiting prevents server overload from excessive requests. Authorization handling restricts access based on user roles, and equipment actions are logged for accountability.

Recommendation & Evaluations	The Section is well-detailed.

2 Source Code Review

In this section, the team will be reviewing the code by group based on the **OWASP Code Review Guide**.

The table below shows the overall source code review that was conducted by Group 18.

No.	OWASP Section	Section REF	OWASP Code Description	Tested	Exploitable/ Error
1	A7:2021-Identification and Authentication Failures	CWE-307	Lack of Account Lockout Mechanism	Yes	Yes
2		CWE-200	Improper input validation during password reset	Yes	Yes
3	A2:2021 - Broken Authentication and Session Management	CWE-613	Session not terminated after password reset	Yes	Yes
4			Exploitation of Expired Session Tokens and Privilege Escalation	Yes	Yes
5	Cryptographic Failures (ASVS 2.11 and 2.1.2)	CWE-521	Insufficient password validation	Yes	No
		-	Unnecessary features are enabled or installed	-	No
6	Sensitive Data exposure (ASVS 8.2.2)	CWE-913 CWE-315	Sensitive data exposed through local storage	Yes	No
7	A05:2021 - Security Misconfiguration	CWE-400	File upload lacks file size validation	Yes	No
8	Insecure Backend	CWE-862	Missing	Yes	Yes

	Routes	CWE-642	Authorization; External Control of Critical State Data		

2.1 Lack of Account Lockout Mechanism

- **CWE-307: Improper Restriction of Excessive Authentication Attempts**
- **OWASP A7:2021: Identification and Authentication Failures**

In the application, as you can see in Figure 4, there is no mechanism of failed login attempts and no conditional check to lock the user account after a certain number of failed attempts.This lacks of tracking leaves the application vulnerable to brute force attacks, where an attacker can continuously attempt to guess a user's password without any restriction.

```
if (!validateEmail(data.email) || !validatePassword(data.password)) {
  setErrorMessages('Invalid email or password');
  return;
} else {
  // Get reCAPTCHA token
  const recaptchaToken = await window.grecaptcha.execute(site, { action: 'login' });
  setErrorMessages('');
  try {
    const response = await axios.post('/api/users/login', { data, recaptchaToken }, {
      headers: {
        'Content-Type': 'application/json',
      },
      withCredentials: true,
    });

    if (response.status !== 200) {
      toast.error("Invalid email and password.");
    }

    navigate("/hotp-login")
```

Figure 4. Login function

Suggested Fix:

- Implementation of a mechanism to track the number of failed login attempts for each user.
- Lock the account by setting the account status to a locked user when login attempts exit a certain limit.

2.2 Request Reset Password

- **CWE-200: Improper Input Validation**
- **OWASP-163: Error Handling**

Figure 5 below has a significant security vulnerability related to improper input validation and error handling. Specifically, when password reset is requested for an email that does not exist in the database, the application returns a specific error message, "**Email not found**". This issue can be exploited by an attacker to confirm the existence of a user account through enumeration attacks.

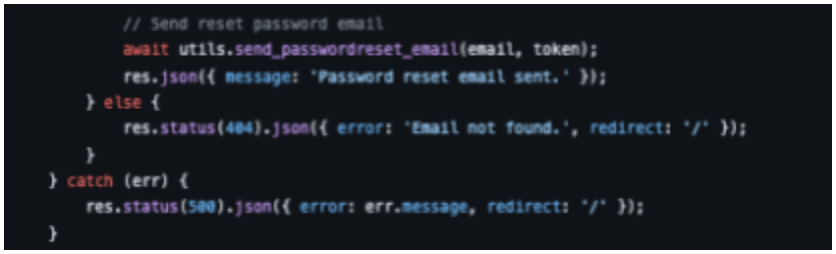


Figure 5. checkEmailExist function

Suggested Fix:

- The error message shown to the end user should be the same regardless of whether the email address is in the system. An example would be, "Password Reset Link sent".

2.3 Session are not Terminated after successful password reset

- **CWE-613: Insufficient Session Expiration**
- **OWASP A2:2021: Broken authentication and Session Management**

On Figure 6 below, the reset password function does not terminate active sessions after a successful password reset. This allows any previously authenticated sessions to remain active, posing a security risk. If an attacker has obtained a valid session token through session hijacking or token theft, they will still have access to the account even after the password has been changed

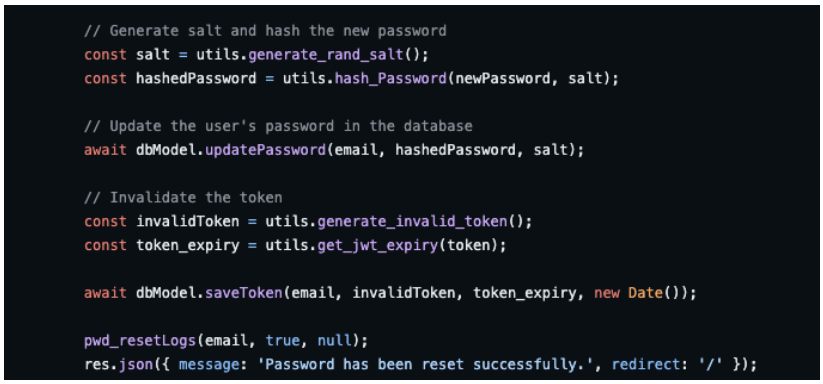


Figure 6. resetPassword function

Suggested Fix:

- Explicitly destroying all sessions for the user that has undergone a password reset.

2.4 Weak Authentication

- **CWE-521: Weak password requirement**

The Figure 7 below for the password validation function in the application reveals a weakness in the authentication mechanism. Specifically, the implementation allows passwords to be at least 10 characters long, whereas OWASP ASVS **2.1.1** and **2.1.2** recommend a minimum length of 12 characters. Additionally, the implementation does not enforce a maximum password length of 128 characters, which could potentially lead to performance issues or security vulnerabilities.

```
const validatePassword = (password) => /^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{10,}$/.test(password);
```

Figure 7. validatePassword regex

The validate password function also does not allow printable Unicode characters like space to be permitted in the password as mentioned in OWASP ASVS 2.1.4. The recommended remediation for this is to edit the validatePassword function regular expression to reject special characters that could be used in injection attacks instead of only allowing a set of special characters.

Suggested Fix:

- Set password to have a password length of 12 to 128 characters and use regex to reject special characters.

2.5 Unnecessary Dependencies

- **ASVS-14.2.2: Verify that all unneeded features, documentation, sample applications and configurations are removed.**

The figure 8 below contains packages that are better suited to be placed in devDependencies instead. Including dependencies that are not meant to be exposed in a production environment opens the possibility of security vulnerabilities. There may potentially be a performance impact with the inclusion of unnecessary packages.

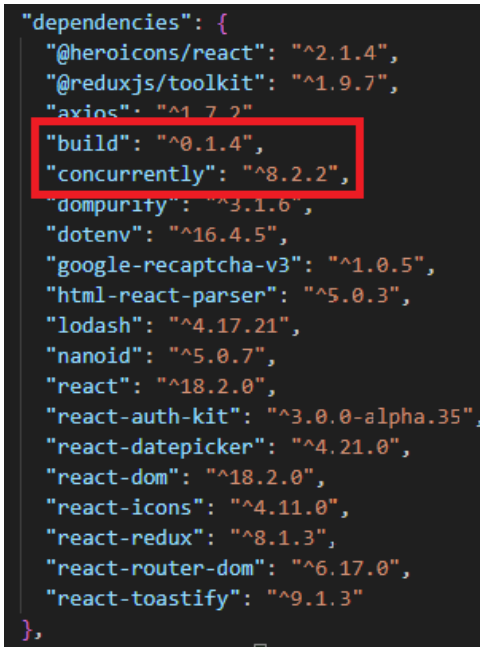


Figure 8. Dependency list

Suggested Fix:

- Include only the minimum necessary components in the production environment to reduce attack surface

2.6 Client Side Data Protection

- **CWE-922: Insecure Storage of Sensitive Information**
- **CWE-315: Cleartext Storage of Sensitive Information**

The Figure 9 below shows the client-side data storage reveals a critical security issue related to the improper storage of sensitive information in the local storage of the browser. Storing sensitive data such as **userID** and authentication states (**isLoggedIn**) in local storage exposes this information to potential attackers who can manipulate or access it to perform malicious actions.

```
const initialState = {
  userId: localStorage.getItem("userId") || null,
  isLoggedIn: localStorage.getItem("isLoggedIn") === "true",
  darkMode: localStorage.getItem("darkMode") === "true" || true,
};

const authSlice = createSlice({
  name: "auth",
  initialState,
  reducers: {
    loginUser: (state, action) => {
      const { userId } = action.payload;
      if (userId) {
        state.isLoggedIn = true;
        state.userId = userId;
        localStorage.setItem("isLoggedIn", "true");
        localStorage.setItem("userId", userId);
        toast.success("You have successfully logged in.");
      } else {
        toast.warn("Invalid user ID.");
      }
    },
    logoutUser: (state) => {
      state.isLoggedIn = false;
      state.userId = null;
      localStorage.removeItem("isLoggedIn");
      localStorage.removeItem("userId");
      toast.success("You have successfully logged out.");
    },
```

Figure 9. Local Storage of Application

Suggested Fix:

- Avoiding storing user IDs on local storage. Instead, store it to a more secure location such as a server-side session.

2.7 Listing New Products are not Sanitized

- **OWASP A03:2021: Injection**

The application currently accepts user input for listing new products functionality without sanitizing or validating user's input. This makes the application vulnerable to XSS attacks, where malicious scripts can be injected into the web pages, and be executed in the browsers of other users.

```
81      const formData = new FormData();
82      formData.append('name', name);
83      formData.append('type', type);
84      formData.append('description', description);
85      formData.append('image', image);
```

Figure 10. Lack of sanitization for form data

Suggested Fix:

- Enforce strict type, format, and length checks
- Use allowlists to restrict input to only allowed characters or values

2.8 File Upload

- **CWE-400: Uncontrolled Resource Consumption**

The current implementation of the image upload validation checks ensures that the uploaded file is either a PNG or JPG image. It is important to enforce file size quotas and maximum number of files per user to prevent denial of service (DoS) attacks by filling up storage.

```
  if (!image) {
    errors.image = "Please upload the image file.";
    isValid = false;
  } else if (!['image/png', 'image/jpeg', 'image/jpg'].includes(image.type)) {
    errors.image = "Please upload a PNG or JPG image.";
    isValid = false;
  }

  setErrorMessages(errors);

  return isValid;
};
```

Figure 11. Image lack file size validation

Suggested Fix:

- Add file size validation to not accept large files.
- Add a check to make sure a single user cannot fill up too much storage.

2.9 Exploitation of Expired Session Tokens and Privilege Escalation for admin site

- **CWE-613: Insufficient Session Expiration**
- **OWASP A2:2021: Broken Authentication and Session Management**

The application fails to properly validate and expire session tokens, which can be exploited by attackers to escalate privileges. Specifically, it allows expired session tokens to be swapped with active admin tokens without proper validation, leading to unauthorized access and control. The following code below has the absence of checks for the validity and expiration of session tokens when they are used to access sensitive

admin functions. The middleware in Figure 8 does not validate the expiration of the token, it only checks for its presence.

```
module.exports = (req, res, next) => {
    const token = req.query.token || req.body.token;

    if (!token || token === 'undefined') {
      return res.status(400).json({ error: 'No token provided.', redirect: '/forgot-password' });
    }

    next();
  };
```

Figure 12. Middleware function

Suggested Fix:

- Ensure that session tokens are checked for expiration on every transaction by using middleware that verifies the token's validity before processing any requests.
- By Regenerating sessions whenever there is a change in the user's privilege level, especially during authentication.

2.10 Insecure Backend Routes

- **CWE-862: Missing Authorization**
- **CWE-642: External Control of Critical State Data**

The Figure 13 below shows the backend routes use of POST requests for all routes without adequate validation and authorization. This approach can be exploited by attackers who can send requests to any route, potentially manipulating body parameters to extract sensitive information or perform unauthorized actions.

```
const express = require('express');
const router = express.Router();
const adminController = require('../controllers/adminController');

router.post('/admin-land', adminController.retrieveUsers);
router.post('/admin-request', adminController.retrieveUsers);
router.post('/create_request', adminController.create_request);
router.post('/approve_equipment', adminController.approve_equipment);
router.post('/reject_equipment', adminController.reject_equipment);
router.post('/delete_deactivated', adminController.delete_deactivated);
router.post('/fetch_listings', adminController.fetch_listings);
router.post('/fetch_admin', adminController.fetch_admin);
router.post('/fetch_requests', adminController.fetch_requests);
router.post('/delete_request', adminController.delete_request);
router.post('/fetch_activities', adminController.fetch_activities);
router.post('/remove_request', adminController.remove_request);

module.exports = router;
```

Figure 13. Insecure backend routes

Suggested Fix:

- Use POST request only when needed, otherwise use GET request.

3 Static Code Analysis

In this section, important open issues and security hotspots which are detected by SonarQube will be discussed below. These are impending vulnerabilities which may compromise the website.

3.1 SonarQube Scan (ReEquip)

SonarQube detected a total of 229 issues and 11 security hotspots. Out of all these,there is 1 serious vulnerability, 91 reliability issues, 137 maintainability issues and 11 security hotspots. Below, the focus will be on the filtered ascertained issues that the team has decided that were pressing issues for the development team to fix.

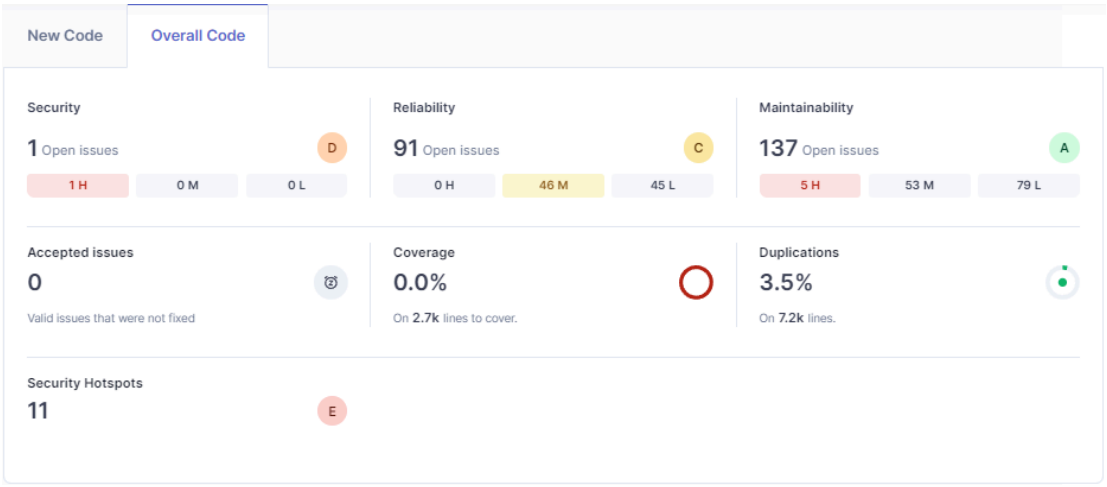


Figure 14. Overview of insecurity of ReEquip

	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage	Duplications
OWASP	—	—	—	—	—	—	—
client	3,528	0	89	103	4	0.0%	5.5%
server	2,492	1	2	29	6	0.0%	0.0%
test	120	0	0	4	1	0.0%	24.1%
.eslintrc.cjs	20	0	0	0	0	0.0%	0.0%

Figure 15. Additional details of ReEquip

3.1.1 Vulnerability

Figure 16. SonarQube padding scheme finding

AES-256-CBC has vulnerabilities due to its lack of built-in integrity and authentication mechanisms, making it prone to tampering and padding oracle attacks, where attackers exploit padding validation to decrypt data without the key. Deterministic encryption, where identical plaintexts yield identical ciphertexts using the same key and IV, also poses a risk if IVs are improperly implemented.

Suggested fix:

- AES-GCM (Galois/Counter Mode) is recommended due to its ability to offer integrity and confidentiality to the software. An authentication tag is included which detects modifications made to the ciphertext, ensuring data integrity.

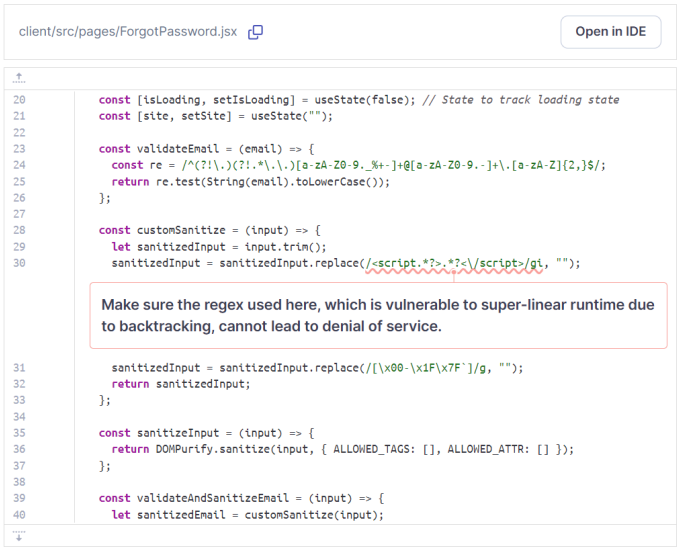


Figure 17. SonarQube regex finding

The warning message highlights a potential performance issue in the code due to the use of a regular expression that may cause catastrophic backtracking, which can lead to a DoS attack if exploited.

Suggested fix:

- Ensure that the regular expression used for sanitizing input does not cause performance bottlenecks.
- The HTML parser library “DOMParser” could be used to handle and sanitize the input.

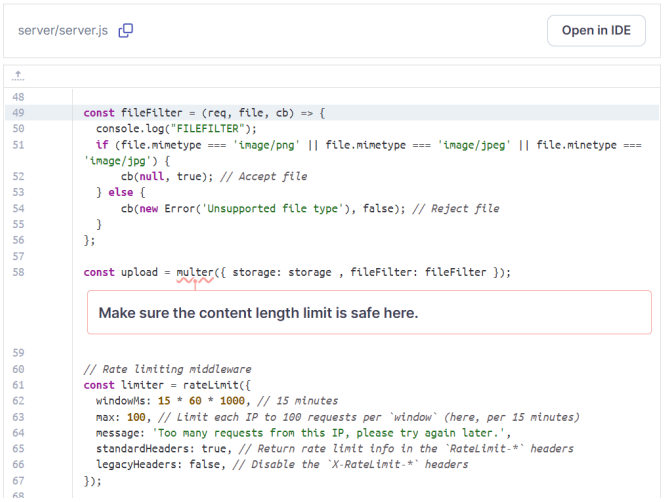


Figure 18. SonarQube mutler finding

The above warning message indicates that handling file uploads without setting a proper content length limit makes the server susceptible to denial-of-service (DoS) attacks, as it could receive excessively large files.

Suggested fix:

- Set a content length limit for file uploads. This helps in controlling network traffic intensity and resource consumption.

3.2 SonarQube Scan (TicketingHuat)

SonarQube detected a total of 224 issues and 18 security hotspots. Out of all these,there are no serious vulnerabilities, 91 reliability issues, 137 maintainability issues and 11 security hotspots. Below, the focus will be on the filtered ascertained issues that the team has decided that were pressing issues for the development team to fix.

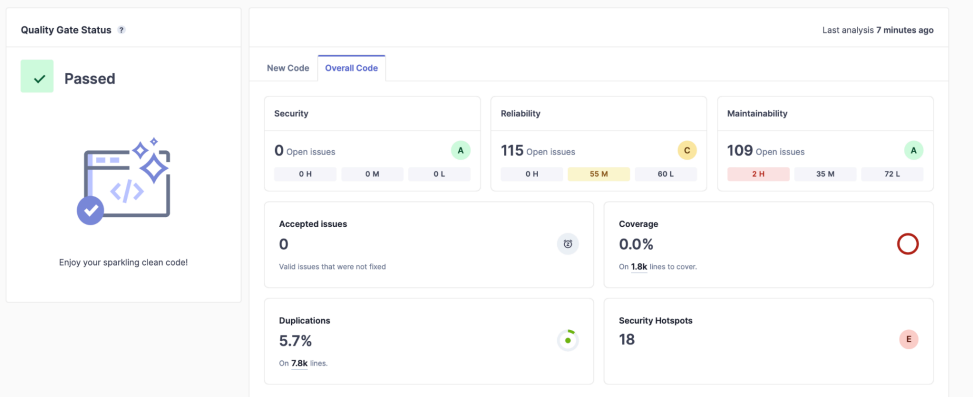


Figure 19. SonarQube TicketingHuat Scan Result

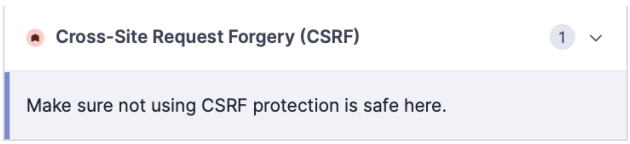


Figure 20. Missing CSRF protection

The security hotspot indicates the potential vulnerability in the code due to the absence of CSRF protection. In a CSRF attack, attackers trick authenticated users into performing actions on a web application without their knowledge. The /events router is missing the csrfProtection constant.

Suggested fix:

- Add csrfProtection constant to the POST request.

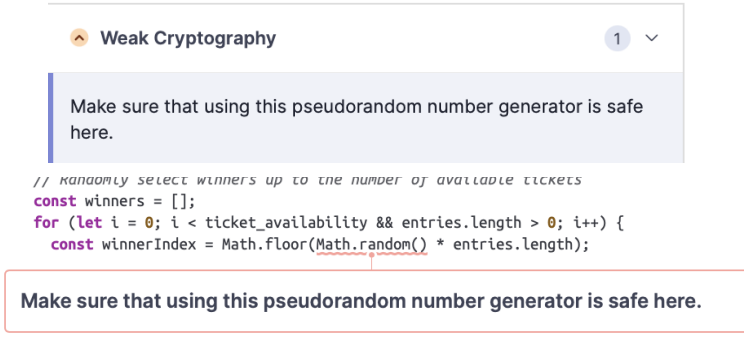


Figure 21. Weak Cryptography

The security hotspot indicates that the use of Math.random() for generating random numbers might not be suitable for all contexts due to its pseudorandom nature. While Math.random() is sufficient for non-critical operations, it may not be appropriate for scenarios where strong randomness and security are required, such as in cryptographic applications or fair lotteries which is the case for selecting winners from the ballot of the ticketing application as it can be predicted if the seed is known.

Suggested fix:

- Use a cryptographically secure random number generator (RNG) such as crypto.randomBytes in Node.js which generates a random byte which can be then converted back into a number in a desired range.

4 Dynamic App Security Testing (DAST)

4.1 Session Fixation Vulnerability

- **CWE-384: Session Fixation**
- **A07:2021 - Identification and Authentication Failures**

The session ID still exists after logging out. Can refer to the figure below.

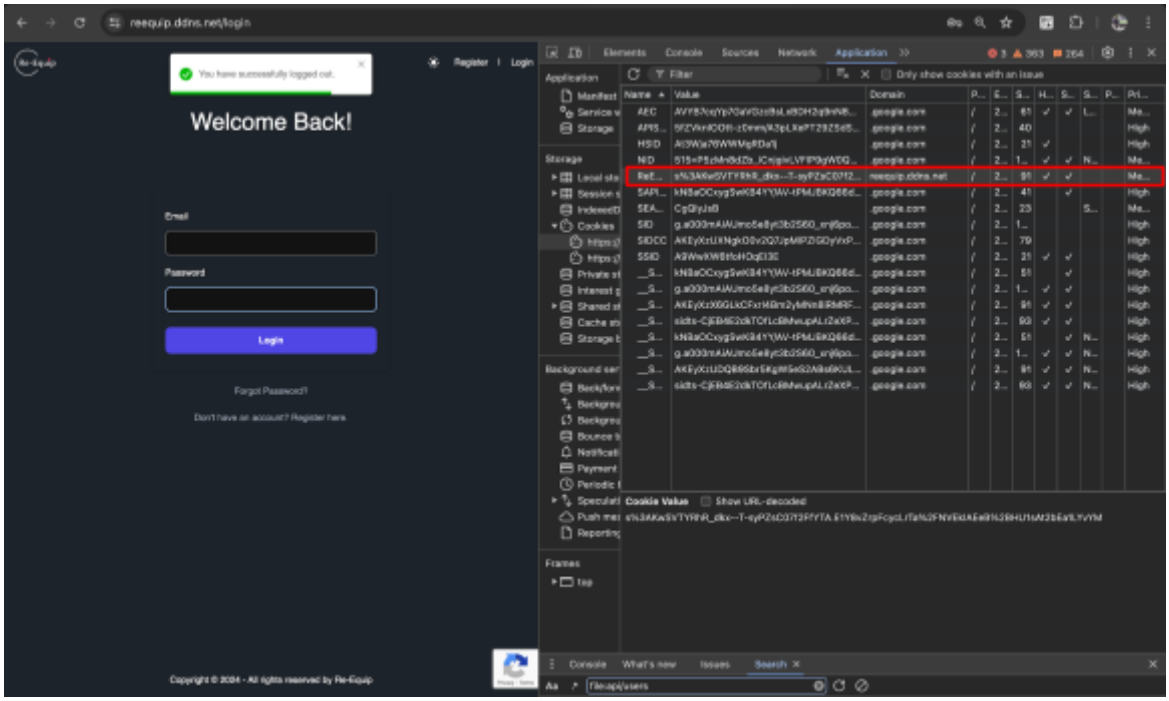


Figure 22. Session ID still exists after logging out

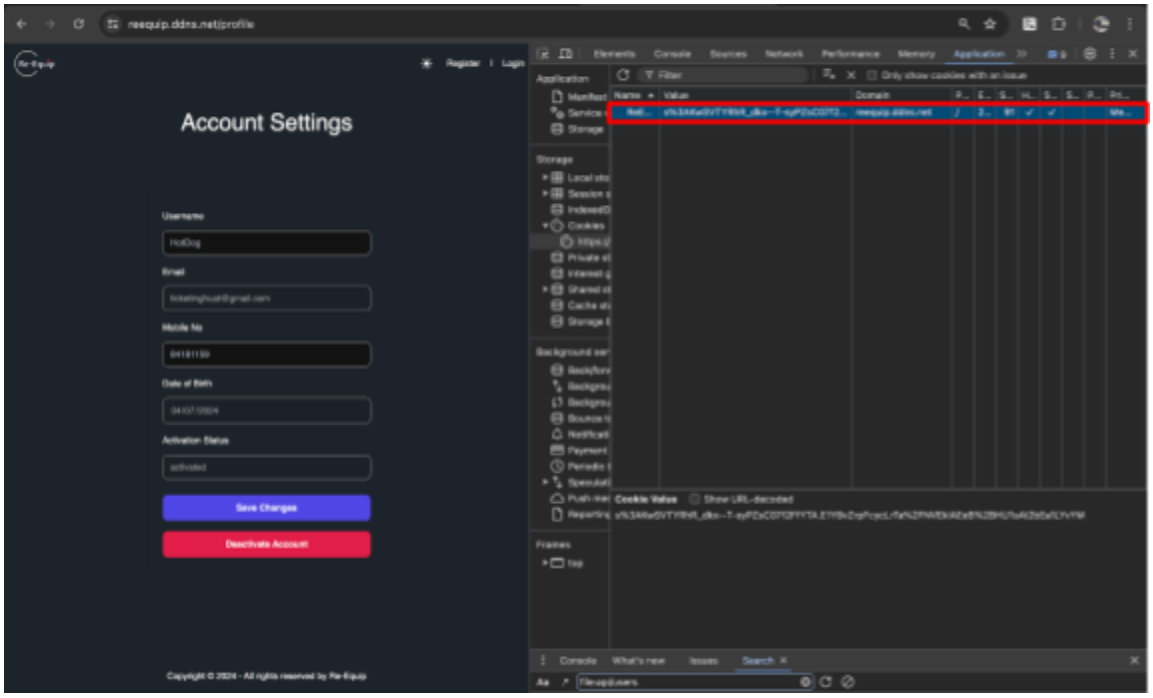


Figure 23. After copying session ID in another browser

In Figure 22 above, the sessionID still exists in the first image after logout. Copying the sessionID into another browser, we were able to access the user's profile page without being authenticated.

In the backend, there are no session checks for **all admin routes**. As shown in Figure 23, we changed the sessionID to an expired user account sessionID and accessed the backend using Burp Suite. Even though the 'set-cookie' header is missing (indicating the cookie is invalid), the API still returns a result. However, when we used the same sessionID for normal user API calls, the backend asked us to log in.

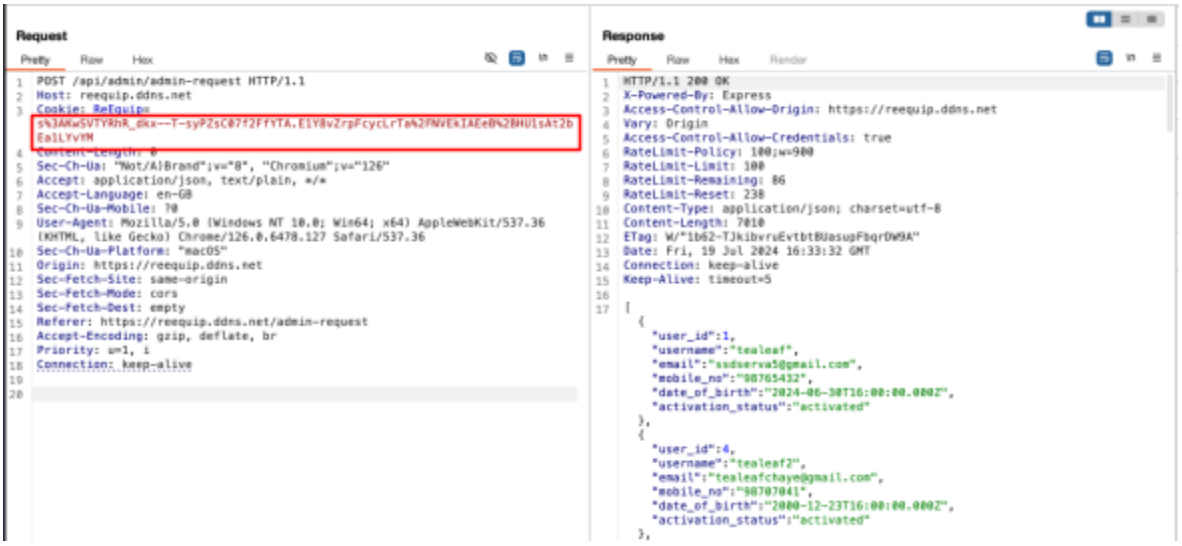


Figure 24. Using an expired session ID in burpsuite

Suggested fix:

- Invalidate session when logout function is called or when session expired.
- Invalidate session server side when expired and check if the session is still valid before handling requests.
- Add session checks for the admin side too.

4.2 Authentication Bypass Vulnerability

- **OWASP A07:2021 - Identification and Authentication Failures**
- **CWE-287: Improper Authentication**

Navigating to "**https://reequip.ddns.net/list-new-product**" without logging in will display the page. **Originally, accessing a few other pages would have crashed the server, but the team fixed the error on their vm**. Attempting to submit a new product will successfully add it into the database (They fixed this error too). This can lead to unauthorized data manipulation, which can result in malicious data entering the system. This shows that the application is not correctly enforcing authentication checks. You refer to the 3 figures below.

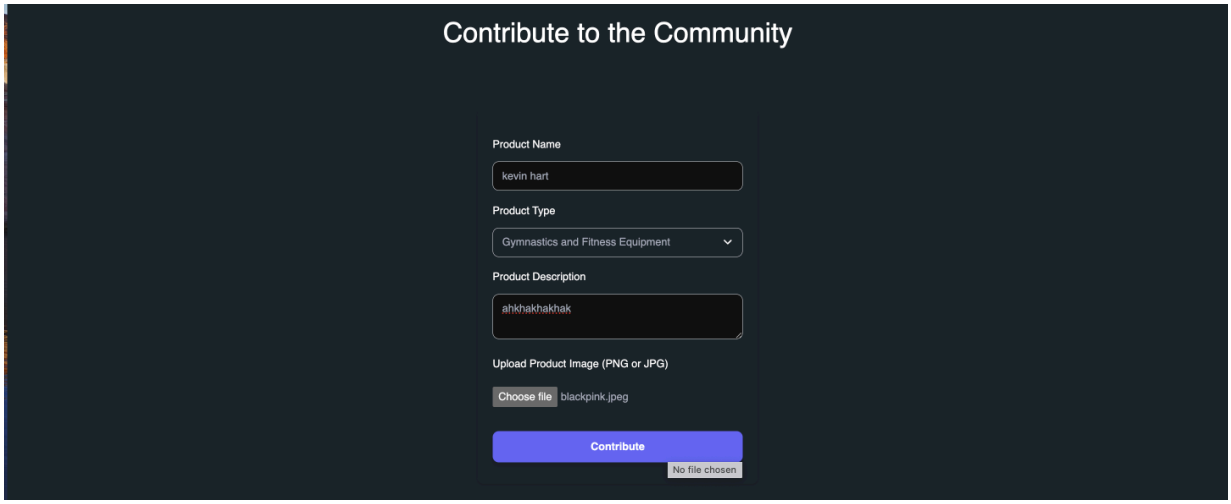


Figure 25. list-new-product page

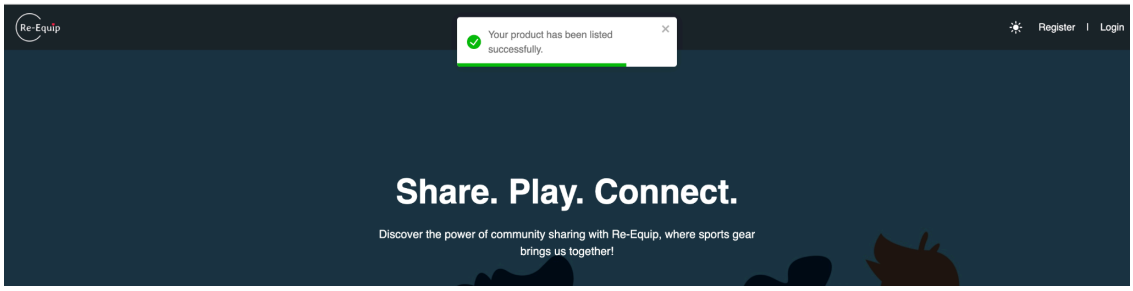


Figure 26. Managed to save products in the database without logging in.

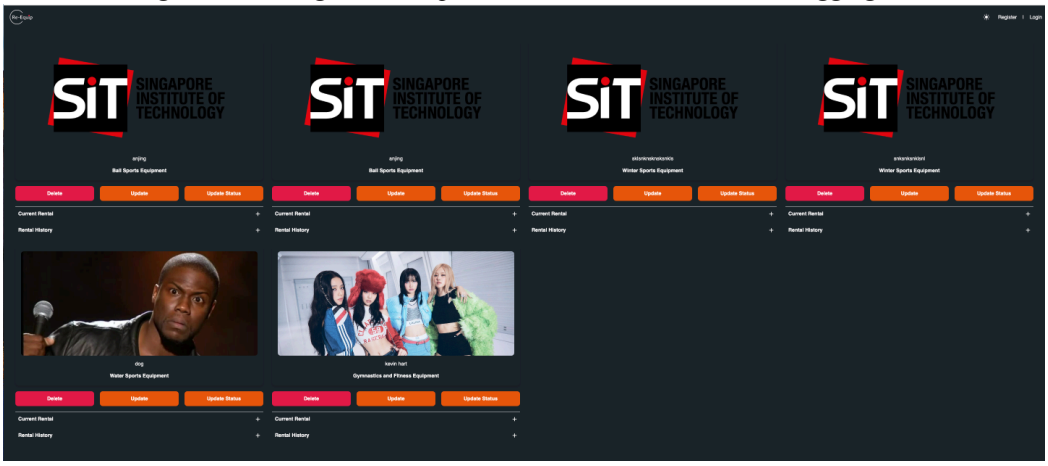


Figure 27. Successfully saved product without logging in

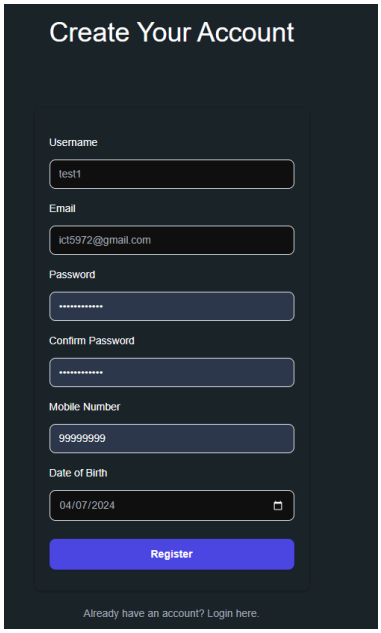
Suggested fix:

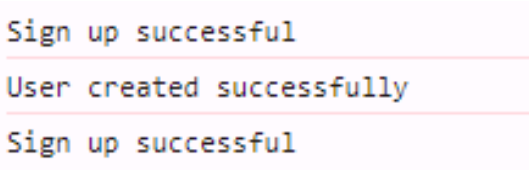
- Implement middleware to check if user is authenticated and apply it on routes.
- Also add authentication checks to data submission in the backend to ensure that user is authorized to perform certain actions.

4.3 Lack of Input Validation

- **CWE-20: Improper Input Validation**
- **OWASP A02:2021 - Security Misconfiguration**

At the sign up page, lack of birthday input validation on frontend and backend allow users to input a date before the current day. As shown in the image of the admin dashboard below, the account is successfully added into the database. **However, there was form validation when tested 2 weeks ago, suggesting code was changed**.





111	test1	dcswsewcsew@gmail.com	99999999	11/07/2024	unactivated	Delete
112	test1	ict5972@gmail.com	99999999	04/07/2024	unactivated	Delete
113	test1	ddedswcwse@gmail.com	99999999	10/07/2024	unactivated	Delete

Figure 28. Allow users to input a date before the current day

With reference to Figure 29 below, there was a lack of input sanitization for product description when updating a product details. An attacker will be able to link a malicious or a phishing website to gather sensitive information or attack users. As seen in the figure, clicking on the link brings the user to Google.

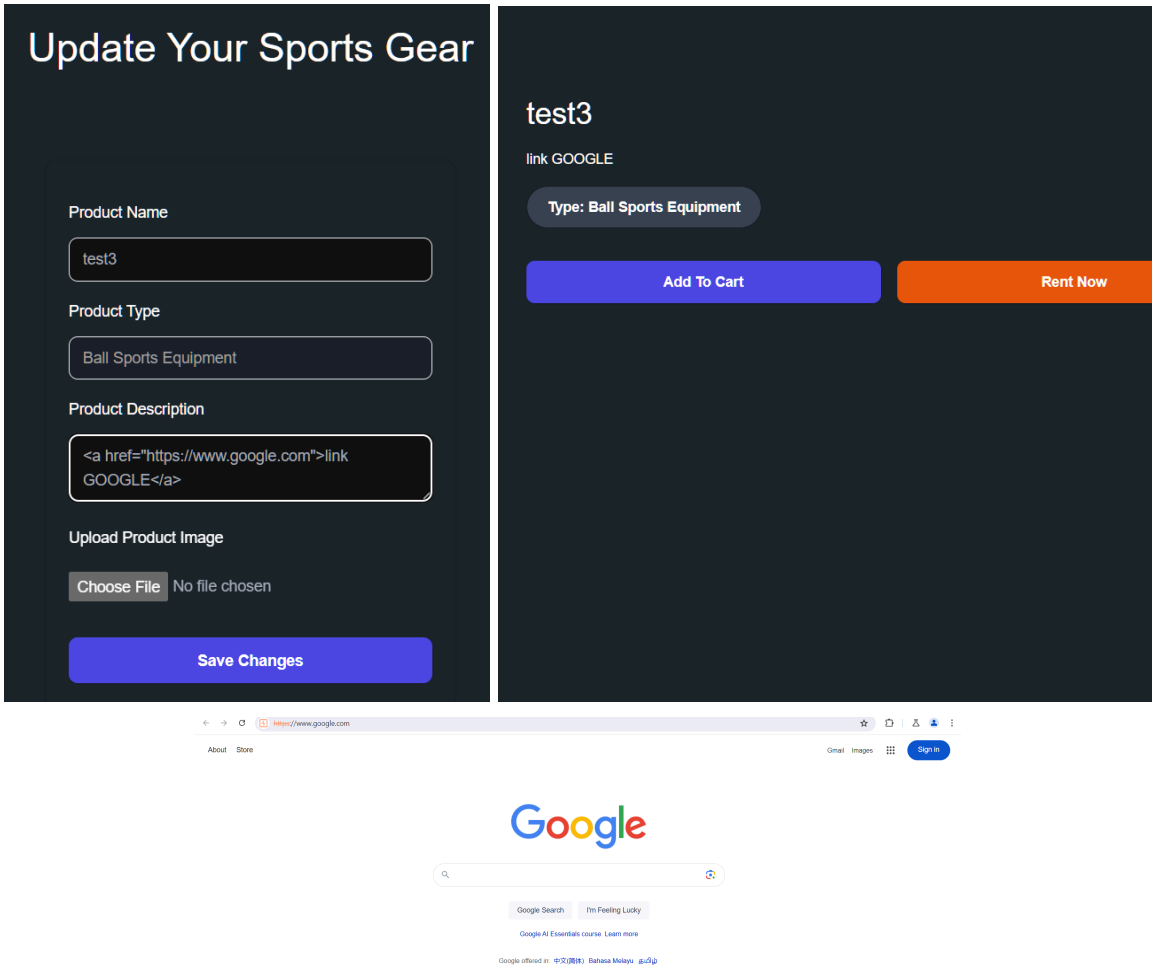


Figure 29. Lack of input sanitization for product description

Suggested fix:

- Implement frontend and backend validation for birthdays to be before the current date.
- Implement sanitization for frontend and backend on all input fields to remove malicious inputs.

4.4 Bad Error Handling

- **CWE-209: Information Exposure Through an Error Message**

Sending a POST request to the server on the backend with an unexpected output in the username field is not handled properly. From the error message in Figure 28, we can see that the backend uses express.

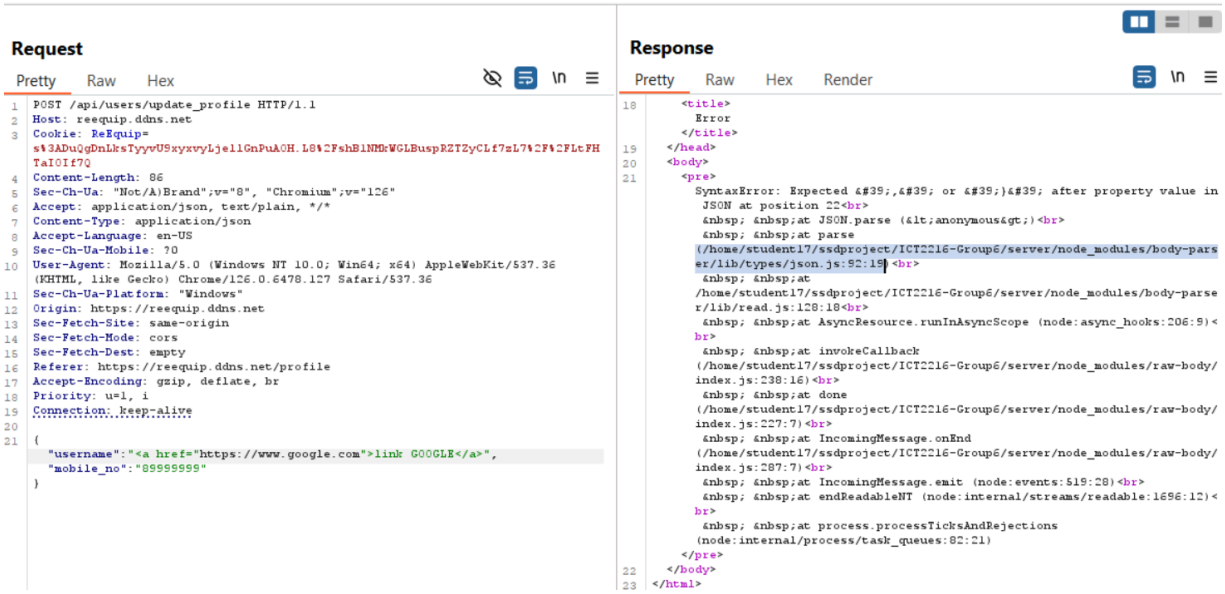


Figure 30. update_profile api via burp suite

Suggested fix:

- Implement a centralised error handling middleware to catch unexpected errors.

4.5 Insecure Direct Object Reference (IDOR)

- **CWE-639: Authorization Bypass Through User-Controlled Key**

IDOR occurs when an application provides direct access to objects based on user-supplied inputs, but it does not check if the user is authorized to access that resource. From our understanding, the product will be shown on the home page after it is approved by an admin. With reference to Figure 31 below, sending a GET request to /api/equipment/<id> will allow a user to access another user's item even though it is not approved by an admin.

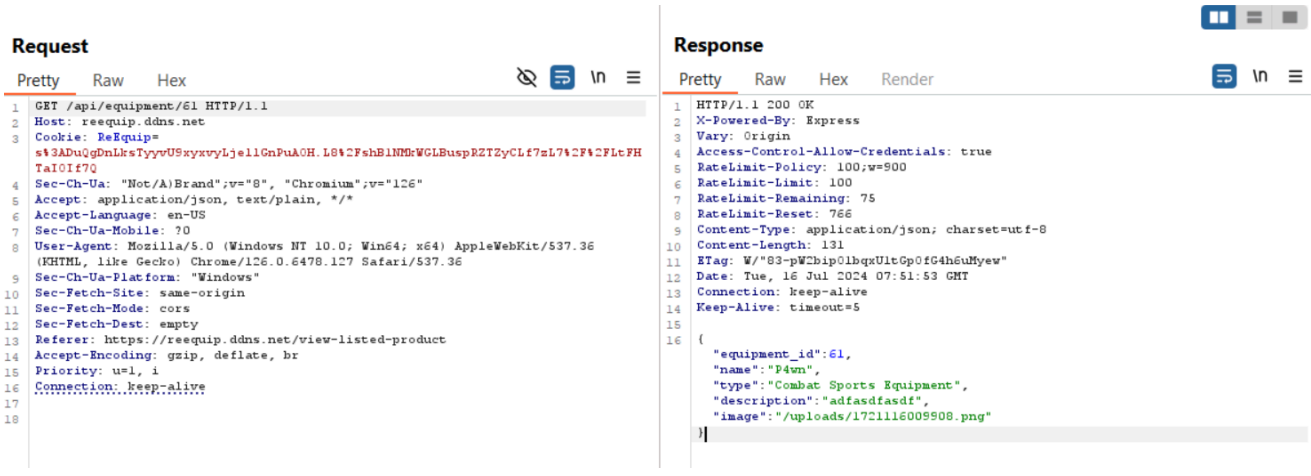


Figure 31. equipment api via burpsuite

Suggested fix:

- Ensure that the user making the request is authorized to access the api.

- Check for admin approval in database query.

4.6 Broken Access Control

- **CWE-284: Improper Access Control**
- **OWASP A01:2021 - Broken Access Control**

This affects all admin routes. A normal user can approve equipment by sending a backend request, which is an admin user functionality. A normal user cookie is used, and a random admin id is used for the request. With reference to the 1st image below, the request successfully updated the database. As seen in the Figure 33 below, admin accounts with user_id '69' have an entry for approval of equipment in their admin log. This shows that a normal user is able to masquerade as an admin.

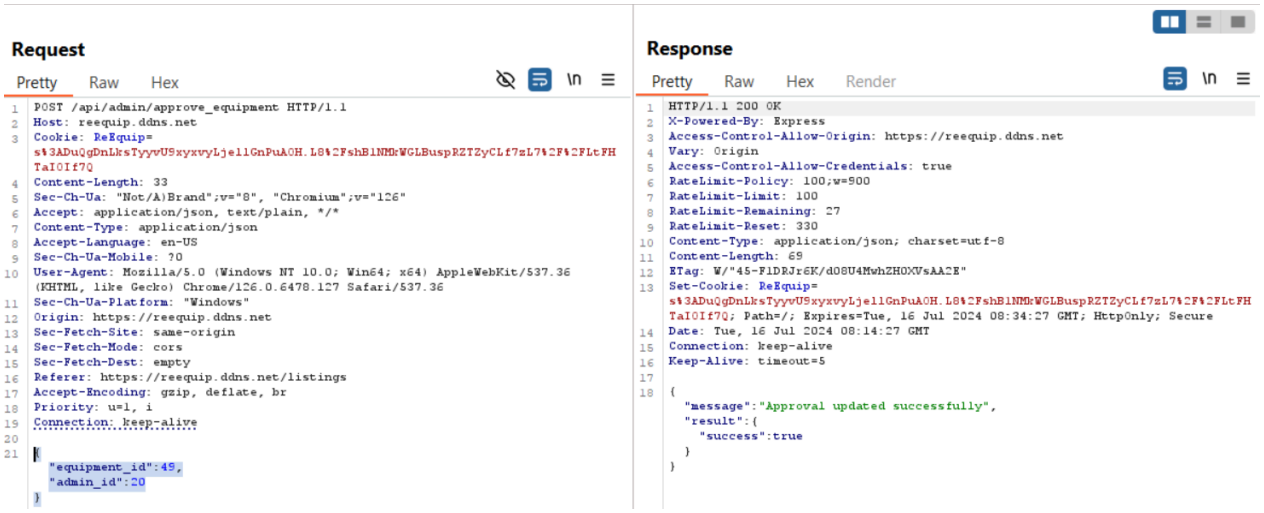


Figure 32. approve_equipment api via burp suite

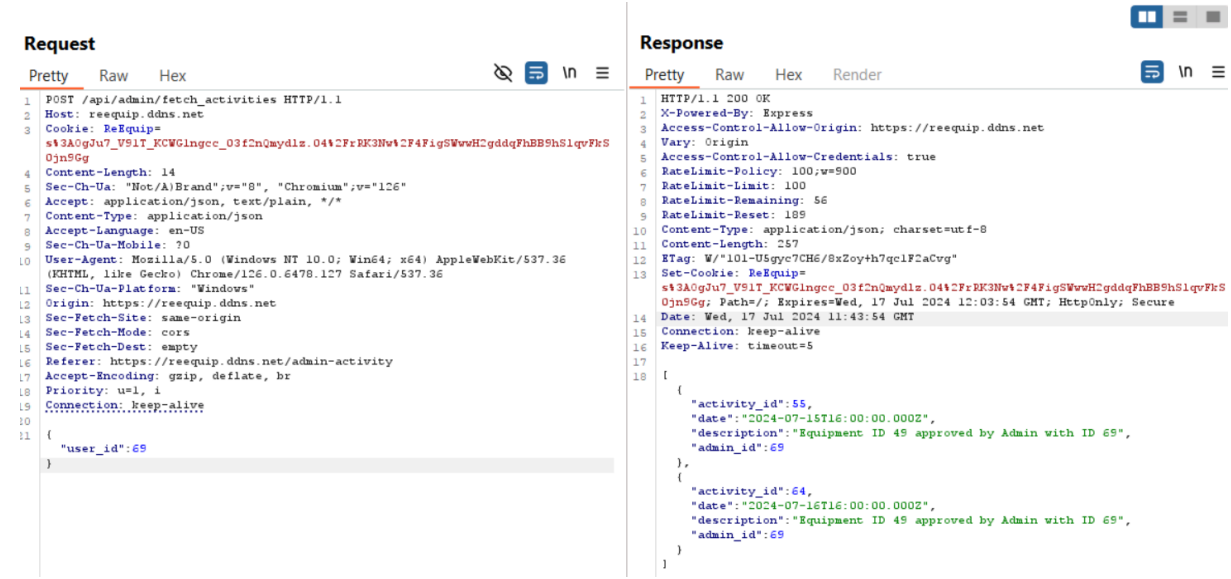


Figure 33. fetch_acitivities api via burp suite

Furthermore, using a normal user's cookie, we can get other account sensitive information, including admin accounts. As shown in the picture below, the hash and salt of an admin account can be found by IDOR. Although it will take many days, an attacker will be able to crack the password to access the admin account.

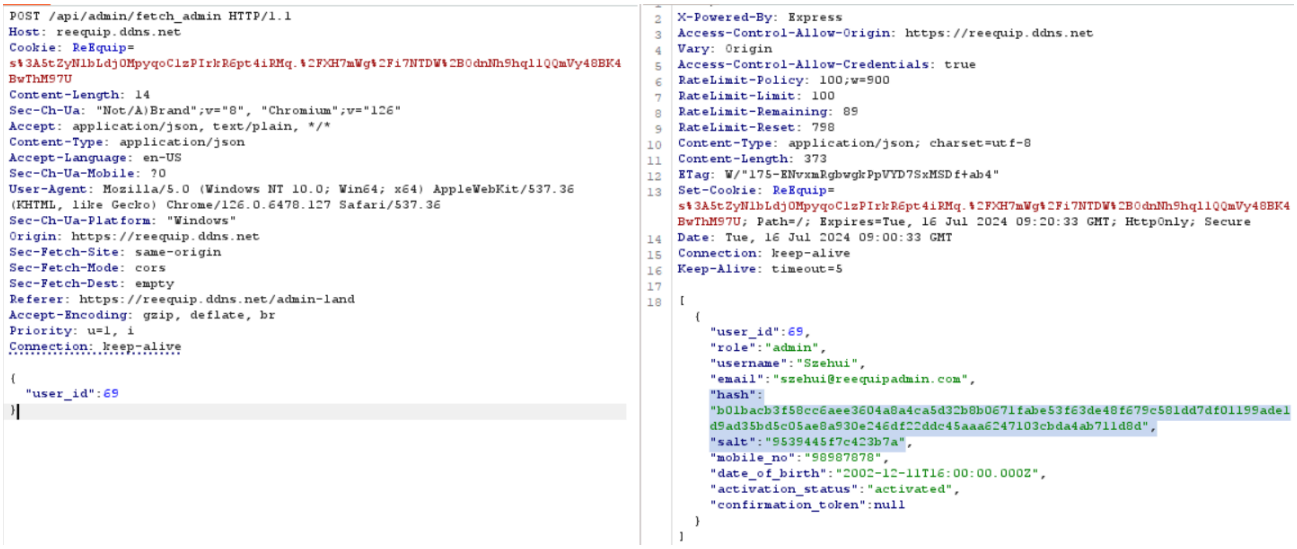


Figure 34. fetch_admin api via burp suite

Suggested fix:

- Only fetch necessary fields needed instead of fetching all fields.
- Implement session check of admin client and server side, make sure they are authorized and authenticated as an admin user.
- Instead of using POST request, use GET for retrieving information.

4.7 Cookie without SameSite Attribute

- OWASP A01:2021 - Broken Access Control
- CWE-1275: Sensitive Cookie with improper SameSite Attribute

On the login, registration, and password reset pages of the web application, a cookie is set without the 'SameSite' attribute. This means the cookie can be sent as a result of a cross-site request. The 'SameSite' attribute is important as it helps to counter Cross-Site Request Forgery (CSRF), Cross-Site Script Inclusion (XSSI), and timing attacks.

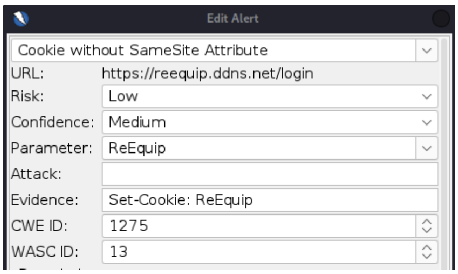


Figure 35. Cookie without SameSite Attribute

Suggested fix:

- Ensure that the SameSite attribute is set to either “lax” or ideally “strict” for all cookies.

4.8 Clickjacking Vulnerability

- **OWASP A05:2021 - Security Misconfiguration**
- **CWE-1021: Exposure of Sensitive Information to an Unauthorized Actor**

The webpage response does not include CSP header with 'frame-ancestors' directive or X-Frame-Options, which are important in protecting against Clickjacking attacks. Clickjacking attacks trick users into clicking undesired webpage elements which can potentially lead users to perform malicious actions such as downloading malware unbeknownst to them.

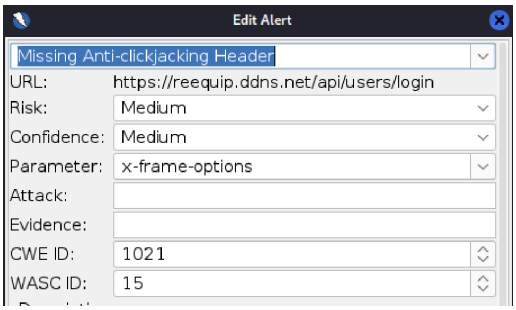


Figure 36. ZAP clickjacking

Suggested fix:

- Implement CSP HTTP header on application’s webpages.
- Implement X-Frame-Options HTTP header on webpages returned by the website.
- Consider implementing the "frame-ancestors" directive in the CSP.

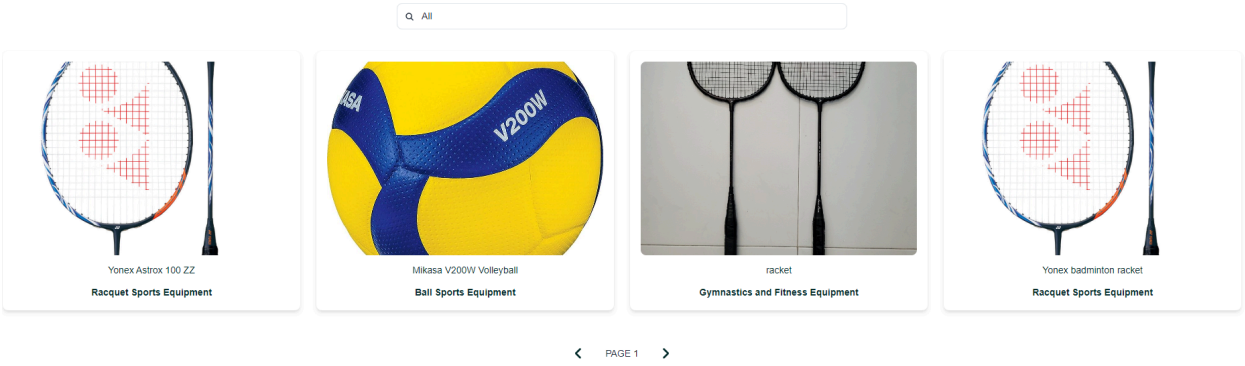
5 Conclusion

ReEquip has several vulnerabilities that need urgent addressing as highlighted in the report. Suggested fixes are in their respective bugs or vulnerabilities section for ease of reading.

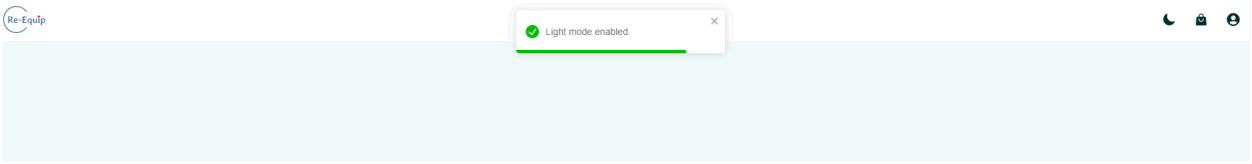
Vulnerabilities such as session management, IDOR, missing validation and sanitization, information disclosure, and improper authentication suggest a lower security level.

Usability of the application can also be improved, such as:

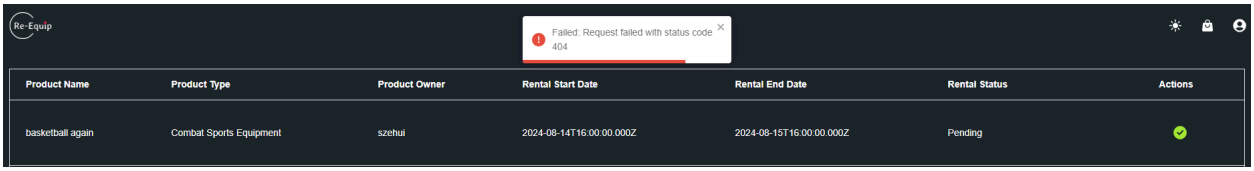
- Website has very few functions which means it is easy to use.
- Equipment approved by admins does not appear on the home page.
- There is a next page and previous page buttons instead of displaying everything, so that it is easier to use.



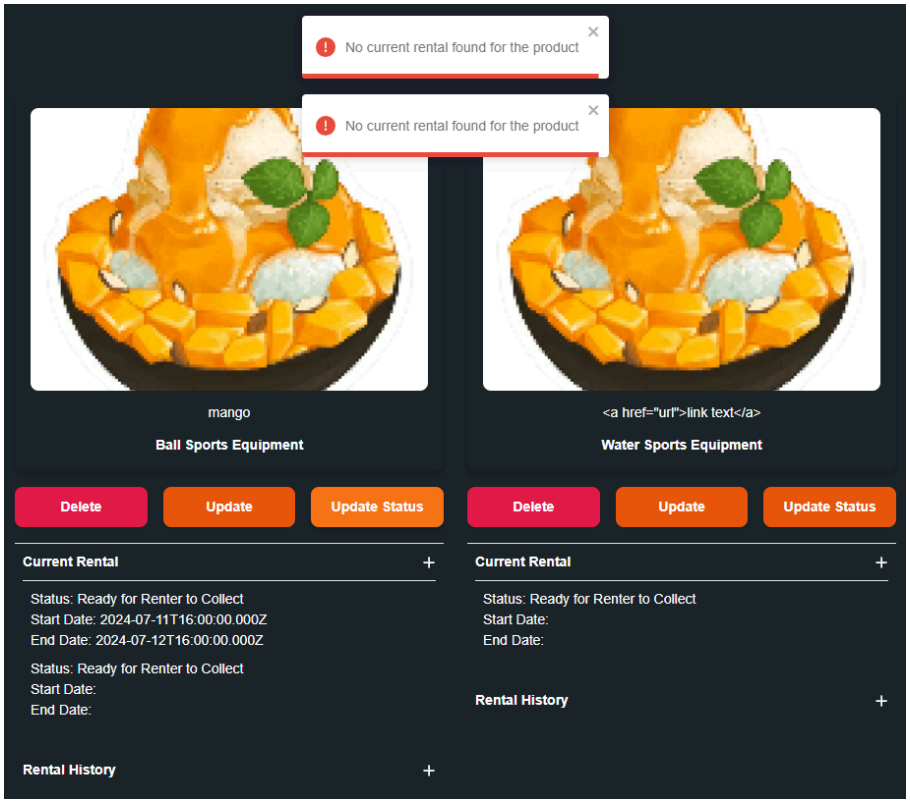
- Able to switch between light and dark mode depending on the user's preference.



- Error messages displayed to users may not be very clear when performing some actions.



- User can rent their own listed products



Taking all into consideration, the team rates the application usability and security level from 1 star, being the lowest, to 5 stars, being the highest.

Security level: 🌟🌟

Usability level: 🌟🌟

In summary, the application security vulnerabilities are an issue that needs to be fixed. Only through proper testing and review that the application can better improve its security and user experience.

References

[1] OWASP, "OWASP Code Review Guide," [Online]. Available: https://owasp.org/www-project-code-review-guide.

[2] OWASP, "OWASP Application Security Verification Standard," [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/.

APPENDIX

Words	Meaning
MiTM Attack	Man in the middle attack
SQLi Attack	SQL injection attack
DoS Attack	Denial-of-service attack
RBAC	Role-based access control
WAF	Web Application Firewall
XSS	Cross-Site
CSRF	Cross-Site Request Forgery
CDN	Content Delivery Network
MFA	Multi-Factor Authentication
DNS	Domain Name Server
RBAC	Role-Based Access Control
WFA	Web Application Firewall
CRUD	Create, Read, Update, Delete
OTP	One Time Password
CSA	Cyber Security Agency of Singapore’s
PBKDF2	Password-Based Key Derivation Function 2
TLS	Transport Layer Security
PDPA	Personal Data Protection Act
PIN	Personal Identification Number
API	Application Programming Interface
OWASP	Open Worldwide Application Security Project
SQL	Structured Query Language
HTTPS	Hypertext Transfer Protocol Secure
JWT	JSON Web Token

CSP	Content Security Policy
RCE	Remote Code Execution
CORS	Cross-Origin Resource Sharing

Table X. Data Dictionary

2.1.1	Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined). (C6)	✓	✓	✓	521	5.1.1.2
2.1.2	Verify that passwords of at least 64 characters are permitted, and that passwords of more than 128 characters are denied. (C6)	✓	✓	✓	521	5.1.1.2

Figure 35. OWASP ASVS 2.1.1 and 2.1.2

2.1.4	Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords.	✓	✓	✓	521	5.1.1.2

Figure 36. OWASP ASVS 2.1.4