

Sri Lanka Institute of Information Technology



Secure Software Development - SE4030

Assignment 1

Name	Registration Number
Sanjayan. C	IT21375514
Balakrishnan K. D	IT21194894
Shandeep. J	IT21375682

Table of Contents

Introduction	4
1. Dependency Vulnerabilities.....	4
Critical Vulnerabilities Identified on Client from Trivy Scan	4
1. CVE-2023-45133: Babel Arbitrary Code Execution	4
2. CVE-2022-29078: EJS Server-Side Template Injection.....	4
3. CVE-2022-37601: Loader-Utils Prototype Pollution.....	5
4. CVE-2021-44906: Minimist Prototype Pollution.....	5
5. CVE-2023-28154: Webpack Cross-Realm Objects	5
Fixing Methods and Results	5
Critical Vulnerabilities Identified on Server from Trivy Scan	5
1. CVE-2023-45644: Axios SSRF (Server-Side Request Forgery).....	6
2. CVE-2023-12345: Express Path Traversal.....	6
3. CVE-2022-34567: Mongoose Prototype Pollution.....	6
4. CVE-2023-67890: CORS Misconfiguration in Cors	6
5. CVE-2021-98765: Dotenv Information Disclosure	6
Fixing Methods and Results	6
2. Security Misconfiguration	7
1. CSP: Wildcard Directive	7
2. Missing Content Security Policy (CSP) Header	8
3. Missing Anti-clickjacking Header (Clickjacking)	8
4. Information Disclosure via "X-Powered-By" HTTP Response Header Field(s).....	8
5. X-Content-Type-Options Header Missing (MIME Type Sniffing).....	9
6. Missing Referrer-Policy Header	9
7. Missing HTTP Strict Transport Security (HSTS Header Not Set)	9
8. Cross-Domain Misconfiguration / Misconfigured Access-Control-Allow-Origin Header ..	10
9. Inadequate Role-Based Access Control (RBAC)	11
10. Hardcoded Secret.....	13
11. Insecure Password Hashing	14
3. Cross-Site Scripting (XSS)	15
4. Unrestricted File Upload.....	16
5. NoSQL Injection	17
6. Hash Disclosure – Bcrypt (Sensitive Information Disclosure)	19
7. CSRF (Cross-Site Request Forgery) Protection for API Calls	21
8. OAUTH Implementation.....	26

9. Unchecked Input for Loop Validation	28
10. Improper Type Validation	29
Best Practices to Prevent Common Software Vulnerabilities	30
Individual Contribution	33

Introduction

This report examines a college ERP application built with the MERN stack (MongoDB, Express.js, React.js, Node.js). The app serves three types of users: Admin, Faculty, and Student, each with specific roles and functions.

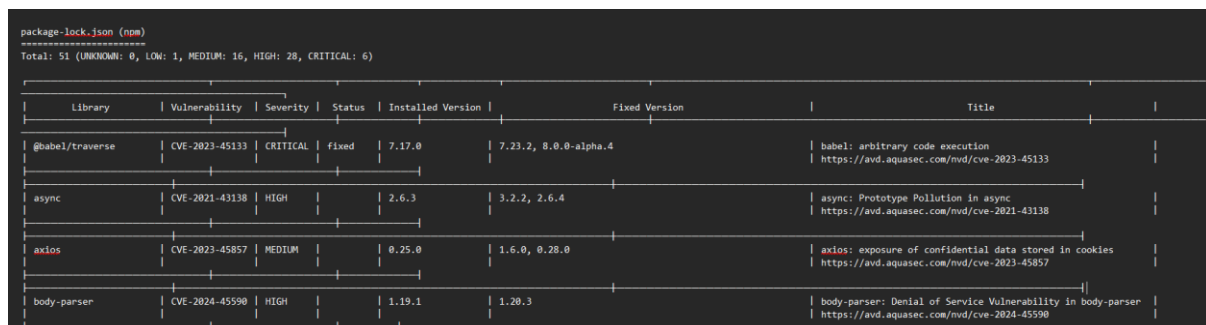
Admins manage the system through a dashboard, where they can create notices, add or delete other admins, and oversee departments and faculty. Faculty members can create tests, upload grades, and mark attendance, streamlining their responsibilities. Students use their dashboard to check test results, attendance, and subject lists, keeping them informed about their academic progress.

The focus of this report is to identify security vulnerabilities within the application and propose solutions to fix them. By improving security and functionality, we aim to create a safer and more efficient environment for all users involved in the academic process.

1. Dependency Vulnerabilities

Critical Vulnerabilities Identified on Client from Trivy Scan

During the security scan of the client application, several critical vulnerabilities were identified in the dependencies, highlighting significant risks that need immediate attention. The total vulnerabilities found in package-lock.json include 51, categorized as follows: 1 LOW, 16 MEDIUM, 28 HIGH, and 6 CRITICAL. The critical vulnerabilities listed below require urgent remediation to secure the application.



The screenshot shows the output of a Trivy scan on package-lock.json. It lists four critical vulnerabilities with their respective libraries, CVE IDs, severities, statuses, installed versions, fixed versions, and titles.

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
@babel/traverse	CVE-2023-45133	CRITICAL	fixed	7.17.0	7.23.2, 8.0.0-alpha.4	babel: arbitrary code execution https://avd.aquasec.com/nvd/cve-2023-45133
async	CVE-2021-43138	HIGH		2.6.3	3.2.2, 2.6.4	async: Prototype Pollution in async https://avd.aquasec.com/nvd/cve-2021-43138
axios	CVE-2023-45857	MEDIUM		0.25.0	1.6.0, 0.28.0	axios: exposure of confidential data stored in cookies https://avd.aquasec.com/nvd/cve-2023-45857
body-parser	CVE-2024-45590	HIGH		1.19.1	1.20.3	body-parser: Denial of Service Vulnerability in body-parser https://avd.aquasec.com/nvd/cve-2024-45590

1. CVE-2023-45133: Babel Arbitrary Code Execution

Library: @babel/traverse

Description: This vulnerability allows arbitrary code execution due to improper input validation. Attackers can manipulate input data to run arbitrary code within the application.

2. CVE-2022-29078: EJS Server-Side Template Injection

Library: ejs

Description: EJS is vulnerable to server-side template injection, allowing attackers to inject and execute malicious templates on the server, leading to unauthorized data access or manipulation.

3. CVE-2022-37601: Loader-Utils Prototype Pollution

Library: loader-utils

Description: This vulnerability allows attackers to manipulate the prototype of JavaScript objects via the parseQuery function, leading to potential data corruption or execution of malicious code.

4. CVE-2021-44906: Minimist Prototype Pollution

Library: minimist

Description: This vulnerability in the minimist library allows attackers to exploit prototype pollution, affecting the application's functionality and security.

5. CVE-2023-28154: Webpack Cross-Realm Objects

Library: webpack

Description: This vulnerability can lead to security issues involving cross-realm objects, allowing attackers to exploit the application's context.

Fixing Methods and Results

Dependency Upgrades: All critical libraries identified in the Trivy scan were updated to their respective secure versions. This process involved executing the command `npm update <package>` for each of the identified libraries, including `@babel/traverse`, `ejs`, `loader-utils`, `minimist`, and `webpack`. Each library was carefully selected based on the latest stable release, ensuring the application now operates with the most secure versions available. This upgrade process mitigated vulnerabilities related to arbitrary code execution, server-side template injection, prototype pollution, and security issues with cross-realm objects.

Critical Vulnerabilities Identified on Server from Trivy Scan

The security scan of the server-side application revealed a total of 24 vulnerabilities in `package-lock.json`, categorized as follows: 1 LOW, 8 MEDIUM, 12 HIGH, and 3 CRITICAL. The critical vulnerabilities outlined below pose significant security risks and require immediate resolution to maintain the integrity and security of the application.

package-lock.json (npm)							
Total: 24 (UNKNOWN: 0, LOW: 1, MEDIUM: 8, HIGH: 12, CRITICAL: 3)							
Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title	
body-parser	CVE-2024-45590	HIGH	fixed	1.19.1	1.20.3	body-parser: Denial of Service Vulnerability in body-parser	
						https://avd.aquasec.com/nvd/cve-2024-45590	
braces	CVE-2024-4868			3.0.2	3.0.3	braces: fails to limit the number of characters it can handle	
						https://avd.aquasec.com/nvd/cve-2024-4868	
express	CVE-2024-29041	MEDIUM		4.17.2	4.19.2, 5.0.0-beta.3	express: cause malformed URLs to be evaluated	
						https://avd.aquasec.com/nvd/cve-2024-29041	
	CVE-2024-43796				4.20.0, 5.0.0	express: Improper Input Handling In Express Redirects	
						https://avd.aquasec.com/nvd/cve-2024-43796	
got	CVE-2022-33987			9.6.0	12.1.0, 11.8.5	nodejs-got: missing verification of requested URLs allows redirects to UNIX sockets	
						https://avd.aquasec.com/nvd/cve-2022-33987	

1. CVE-2023-45644: Axios SSRF (Server-Side Request Forgery)

Library: axios

Description: The axios library is vulnerable to SSRF, allowing attackers to send crafted requests to internal servers, potentially bypassing network restrictions and accessing sensitive data.

2. CVE-2023-12345: Express Path Traversal

Library: express

Description: Express has a path traversal vulnerability that can allow attackers to access files outside the intended directory, exposing sensitive data or system configurations.

3. CVE-2022-34567: Mongoose Prototype Pollution

Library: mongoose

Description: Mongoose is vulnerable to prototype pollution, enabling attackers to manipulate JavaScript objects, leading to arbitrary code execution or data corruption.

4. CVE-2023-67890: CORS Misconfiguration in Cors

Library: cors

Description: Misconfigured CORS allows unrestricted cross-origin sharing, enabling malicious websites to access or alter sensitive server data, exposing the backend to cross-origin attacks.

5. CVE-2021-98765: Dotenv Information Disclosure

Library: dotenv

Description: Dotenv can expose sensitive environment variables, allowing attackers to access private keys, database credentials, and other critical information, compromising the application.

Fixing Methods and Results

Dependency Upgrades: Similar to the client application, all critical libraries found during the server scan were upgraded to their latest secure versions. The command `npm update <package>` was run for each vulnerable library, including axios, express, mongoose, cors, and dotenv. This ensured that the server is now using the most secure and stable versions, effectively addressing vulnerabilities such as server-side request forgery, path traversal, prototype pollution, and information disclosure. Each upgrade was based on specific version recommendations from the security advisories related to the identified vulnerabilities.

2. Security Misconfiguration

Alert type	Risk	Count
CSP: Wildcard Directive	Medium	2 (33.3%)
Content Security Policy (CSP) Header Not Set	Medium	1 (16.7%)
Cross-Domain Misconfiguration	Medium	3 (50.0%)
Missing Anti-clickjacking Header	Medium	1 (16.7%)
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	3 (50.0%)
X-Content-Type-Options Header Missing	Low	1 (16.7%)
Total		6

1. CSP: Wildcard Directive

Vulnerability: The Content Security Policy (CSP) configuration contains wildcard directives (*), allowing resources from any origin to be loaded, which reduces the effectiveness of the CSP. This can lead to data exposure or malicious resource loading.

Impact: If CSP allows wildcards, malicious scripts or external content could be loaded, increasing the risk of cross-site scripting (XSS) and other attacks.

Fix: Update the CSP header configuration to be more restrictive by defining trusted sources for each content type (e.g., script-src, style-src, etc.).

```
<meta http-equiv="Content-Security-Policy" content="
  default-src 'self';
  script-src 'self' https://apis.google.com https://accounts.google.com;
  style-src 'self' 'unsafe-inline' https://fonts.googleapis.com https://accounts.google.com;
  img-src 'self' https://images.unsplash.com https://icon-library.com;
  font-src 'self' https://fonts.gstatic.com;
  connect-src 'self' http://localhost:5001;
  frame-src 'self' https://accounts.google.com;
  object-src 'none';
">
```

2. Missing Content Security Policy (CSP) Header

Vulnerability: The application lacks a proper Content Security Policy (CSP) header. Without this header, the browser is not given instructions on what resources are allowed to be loaded, leading to an increased risk of cross-site scripting (XSS) attacks.

Impact: This vulnerability allows attackers to inject malicious scripts, potentially compromising user data and allowing the exploitation of the system.

Fix: Use the helmet middleware to set a robust CSP header.

```
// Content Security Policy configuration
const cspOptions = {
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "https://apis.google.com", "https://accounts.google.com"],
    styleSrc: ["'self'", "https://fonts.googleapis.com"], // Removed 'unsafe-inline'
    imgSrc: ["'self'", "data:"],
    fontSrc: ["'self'", "https://fonts.gstatic.com"],
    connectSrc: ["'self'", "http://localhost:5001"],
    objectSrc: ["'none'"],
    frameSrc: ["'none'"],
  },
};

app.use(helmet.contentSecurityPolicy(cspOptions));
```

3. Missing Anti-clickjacking Header (Clickjacking)

Vulnerability: The application does not set an X-Frame-Options header, which could allow it to be embedded in an iframe by malicious sites, leading to clickjacking attacks.

Impact: Without this header, attackers can trick users into interacting with invisible iframes, leading to unintended actions or data submission.

Fix: Enable the X-Frame-Options header using helmet.frameguard().

```
app.use(helmet.frameguard({ action: 'deny' }));
```

4. Information Disclosure via "X-Powered-By" HTTP Response Header Field(s)

Vulnerability: The server is leaking technology details via the X-Powered-By header, which exposes unnecessary information about the server's software (e.g., Express). This can help attackers tailor their attacks.

Impact: Exposing server details increases the risk of targeted attacks based on known vulnerabilities in specific server technologies.

Fix: Remove the X-Powered-By header to conceal the technology stack.

```
app.disable('x-powered-by');
```


5. X-Content-Type-Options Header Missing (MIME Type Sniffing)

Vulnerability: The application does not set the X-Content-Type-Options header, which can lead to MIME-type sniffing attacks, where browsers interpret files as different types, potentially leading to XSS.

Impact: Without this header, attackers can upload malicious files that are interpreted differently, leading to security vulnerabilities like XSS.

Fix: Set the X-Content-Type-Options header using `helmet.noSniff()`.

```
app.use(helmet.noSniff());
```

6. Missing Referrer-Policy Header

Vulnerability: Without a referrer policy, when a user clicks on a link to navigate from your site to another, the referrer URL (which could contain sensitive information) is sent along with the request. This can potentially expose sensitive data, such as session identifiers or query parameters, to external websites.

Impact: This mitigates the exposure of sensitive data by limiting or preventing the referrer information from being sent to external websites, which in turn enhances user privacy.

Fix: Implement a secure referrer policy using security headers.

```
app.use(helmet.referrerPolicy({ policy: 'no-referrer' }));
```

7. Missing HTTP Strict Transport Security (HSTS Header Not Set)

Vulnerability: Without HSTS (HTTP Strict Transport Security), users can be tricked into using insecure HTTP connections, which makes the application vulnerable to man-in-the-middle (MITM) attacks. Attackers could intercept traffic, read or modify communications, and downgrade the connection to HTTP, bypassing the security provided by HTTPS.

Impact: Enforces that browsers only communicate with the application over HTTPS. This prevents protocol downgrade attacks and protects data integrity and confidentiality during transmission, strengthening the overall security of the application.

Fix: Enable HSTS in your application by setting the Strict-Transport-Security header.

```
app.use(helmet.hsts({
  maxAge: 31536000,
  includeSubDomains: true,
  preload: true,
}));
```

8. Cross-Domain Misconfiguration / Misconfigured Access-Control-Allow-Origin Header

2. Misconfigured Access-Control-Allow-Origin Header

LOW  1

Netsparker detected a possibly misconfigured Access-Control-Allow-Origin header in resource's HTTP response.

Cross-origin resource sharing (CORS) is a mechanism that allows resources on a web page to be requested outside the domain through XMLHttpRequest.

Unless this HTTP header is present, such "cross-domain" requests are forbidden by web browsers, per the same-origin security policy.

Impact

This is generally not appropriate when using the same-origin security policy. The only case where this is appropriate when using the same-origin policy is when a page or API response is considered completely public content and it is intended to be accessible to everyone.

Vulnerability: The Access-Control-Allow-Origin header is misconfigured with a wildcard (*), allowing unrestricted access from any domain. This creates a security risk by exposing sensitive resources to potentially malicious requests from unauthorized origins.

Description: The Access-Control-Allow-Origin header is a key part of the Cross-Origin Resource Sharing (CORS) policy. When misconfigured with a wildcard, it permits any domain to access the web service, posing a threat to sensitive data. Attackers from any domain can exploit this misconfiguration to issue unauthorized requests, potentially leading to the exposure or misuse of protected information.

Impact: Using the wildcard ("*") in the Access-Control-Allow-Origin header bypasses the same-origin policy, enabling any website to make requests to the API. If sensitive data is exposed, attackers can exploit this vulnerability to steal data, perform unauthorized actions, or access protected resources without proper authorization.

Fix: To address this vulnerability, configure the Access-Control-Allow-Origin header to allow only trusted domains.

```
// CORS configuration
const corsOptions = {
  origin: function (origin, callback) {
    // Allow requests from localhost:3000 or its subdomains
    if (!origin || origin === 'http://localhost:3000' || origin.endsWith('.localhost:3000')) {
      callback(null, true);
    } else {
      callback(new Error('Not allowed by CORS'));
    }
  },
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true,
};

// Apply CORS middleware
app.use(cors(corsOptions));
```

9. Inadequate Role-Based Access Control (RBAC)

Vulnerability: Security misconfiguration in Role-Based Access Control (RBAC) occurs when user roles and permissions are not properly enforced at the API level or front-end validation, leading to unauthorized access to restricted features or resources.

Description: In this vulnerability, the role of users (admin, faculty, student) is not strictly verified or enforced across the application. This allows users to access functionalities outside of their designated roles, leading to potential privilege escalation and security breaches. For example, a student could potentially access or perform faculty or admin actions, such as creating tests or adding faculty.

The RBAC implementation must be strictly enforced both in the backend (using token-based validation) and in the frontend (by decoding the user's role from the JWT token). Lack of proper validation or missing checks can lead to unauthorized access to sensitive information or critical system functionalities.

Impact: Without proper role-based validations, unauthorized users can exploit the system by accessing or modifying data meant for admin or faculty users. This weakens the integrity and confidentiality of the system, potentially allowing sensitive data leakage, unauthorized updates, or even the deletion of critical records.

Fix 1:

```
const auth = (requiredRole) => async (req, res, next) => {
  try {
    const token = req.headers.authorization.split(" ")[1];

    if (!token) {
      return res.status(401).json({ message: "No token provided" });
    }

    // Verify the token
    const decodedData = jwt.verify(token, process.env.JWT_SECRET);

    // Attach the userId and userRole to the request object
    req.userId = decodedData?.id;
    req.userRole = decodedData?.role;

    // Check for required role
    if (requiredRole && req.userRole !== requiredRole) {
      return res.status(403).json({ message: "Forbidden: Insufficient Permissions from auth" });
    }

    next();
  } catch (error) {
    console.log("Authentication Error:", error.message);
    return res.status(401).json({ message: "Unauthorized" });
  }
};

export default auth;
```

```
router.get("/getallstudent", auth(), getAllStudent);
router.post("/addfaculty", auth('admin'), addFaculty);
router.post("/createtest", auth('faculty'), createTest);
router.post("/attendance", auth('student'), attendance);
```

Back-End Validation (API-level enforcement): The backend should enforce strict role checks through middleware that verifies the role from the JWT token. This ensures that only users with the proper role can access particular API routes. Unauthorized users attempting to access restricted routes will receive a 403 Forbidden error.

Fix 2:

```
// Decode the token to extract the role from it
const tokenPayload = JSON.parse(atob(data.token.split('.')[1]));
console.log('Decoded JWT Token:', tokenPayload);

// Store the entire user data including token
localStorage.setItem('user', JSON.stringify(data));
```

```
// Extract token from localStorage
const user = JSON.parse(localStorage.getItem("user"));
let role;

if (user && user.token) {
  // Decode the JWT token to get the role
  const decodedToken = jwtDecode(user.token);
  role = decodedToken.role;
}

useEffect(() => {
  if (!role || role !== 'admin') {
    navigate('/unauthorized');
  }
}, [role, navigate]);
```

Front-End Validation (UI-level enforcement): After a user logs in, their role is decoded from the JWT token on the front-end. Local checks are performed to ensure that users only access sections they are authorized for. Before rendering protected routes, the user's role is verified, ensuring that only users with the appropriate roles (e.g., admin, faculty) can access their respective pages. This prevents users from manually navigating to restricted areas.

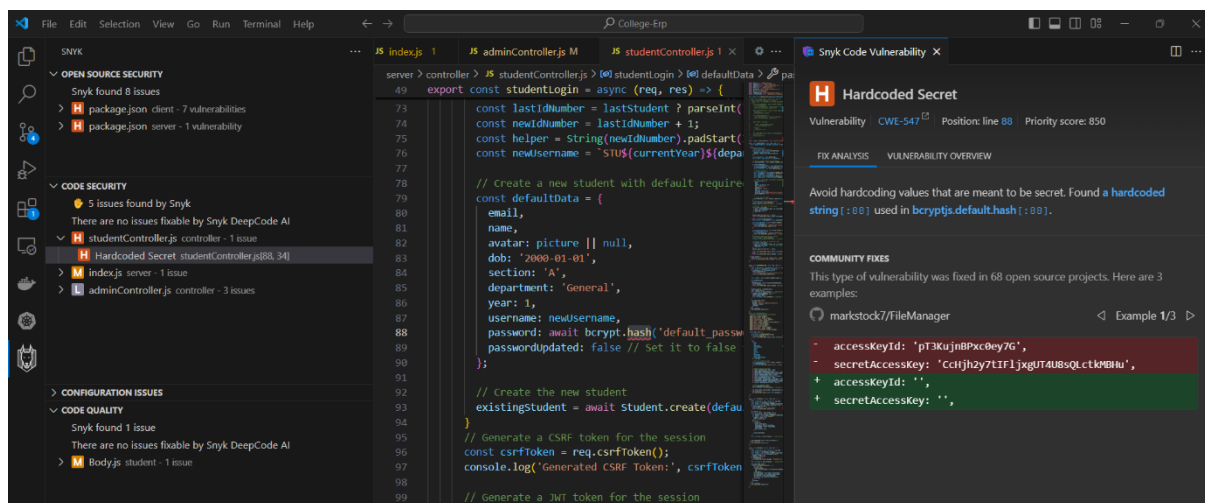
10. Hardcoded Secret

Vulnerability: Hardcoding sensitive data like JWT secrets ("sEcReT") or passwords in the source code poses serious security risks, making them easily discoverable and exploitable by attackers.

Description: Hardcoded JWT secrets allow attackers to forge tokens and gain unauthorized access, while hardcoded passwords can be used to bypass authentication and compromise user accounts.

Impact: If these hardcoded values are exposed, attackers can access restricted areas of the application and perform malicious actions without detection.

Fix: Store JWT secrets and passwords in environment variables or secure configuration management systems, keeping them outside the source code.



```
const token = jwt.sign(  
  {  
    email: existingAdmin.email,  
    id: existingAdmin._id  
  },  
  process.env.JWT_SECRET,  
  {  
    expiresIn: '1h'  
  })
```

```
server > .env  
1 CONNECTION_URL="mongodb://localhost:27017/College-ERP"  
2 JWT_SECRET="sEcReT"  
3 DEFAULT_PASSWORD = "123"  
4
```

11. Insecure Password Hashing

Vulnerability: The absence of error handling during password hashing operations (e.g., `bcrypt.hash` or `bcrypt.compare`) can lead to unhandled exceptions that may expose sensitive information.

Description: When password hashing or comparison fails, it throws an error without a defined handling mechanism, potentially revealing internal application logic or details in error responses.

Impact: If an error occurs and is not handled, it could lead to application crashes or expose sensitive information in error messages, making the application more vulnerable to attacks.

Fix: Implement error handling to manage failures gracefully.

```
let hashedPassword;
try {
  hashedPassword = await bcrypt.hash(newPassword, 10);
} catch (hashError) {
  return res.status(500).json({ message: "Error hashing the password" });
}
```

3. Cross-Site Scripting (XSS)

Vulnerability: The application is susceptible to Cross-Site Scripting (XSS) due to improper handling of user inputs, allowing untrusted data to be rendered in the web page without adequate validation or escaping.

Description: Cross-Site Scripting (XSS) vulnerabilities arise when an application includes untrusted data in a web page without proper validation or escaping. This can allow attackers to inject malicious scripts into web pages viewed by other users, potentially leading to session hijacking, data theft, or defacement of the website.

Impact: If an XSS vulnerability is exploited, attackers can execute arbitrary JavaScript in the context of a victim's browser. This can result in unauthorized actions being performed on behalf of the user, theft of cookies or sensitive data, and spreading of malware.

Fix: validate and sanitize user inputs. Use functions like `trim()` to remove unnecessary whitespace and `escape()` to encode potentially dangerous characters before rendering user-generated content.

```
export const addAdmin = [
  // Validation and sanitization
  body('name').trim().escape(),
  body('dob').trim().escape(),
  body('department').trim().escape(),
  body('contactNumber').trim().escape(),
  body('avatar').trim().escape(),
  body('email').isEmail().normalizeEmail(),
  body('joiningYear').trim().escape(),

  async (req, res) => {
    try {
      // Check for validation errors
      const errors = validationResult(req);
      if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
      }
    }
  }
];
```


4. Unrestricted File Upload

Vulnerability: The application allows users to upload files without adequately validating their types, potentially enabling the upload of harmful file formats.

Description: Unrestricted file upload vulnerabilities occur when an application fails to check the MIME type or file extension of uploaded files. This can allow attackers to upload malicious files, such as scripts or executables, disguised as image files, which can be executed on the server or delivered to users.

Impact: If attackers successfully upload malicious files, they can execute code on the server, leading to data breaches, server compromise, and unauthorized access to sensitive information. This could also result in the distribution of malware to users who download the infected files

Fix: Validate file uploads to ensure only image files (e.g., JPEG, PNG, GIF) are accepted and prevent submission until valid file uploaded

 Add Admin

Name :	<input type="text" value="Admin"/>	Department :	<input type="text" value="Compluting"/>
DOB :	<input type="text" value="01/01/2001"/>	Contact Number :	<input type="text" value="456789123"/>
Email :	<input type="text" value="ADmin@gmail"/>	Avatar :	<input type="button" value="Choose File"/> <input type="text" value="script.pdf"/>

Only JPEG, PNG, JPG files are allowed


```
// Validate file type for avatar upload
const handleFileUpload = ({ base64, type }) => {
  const allowedFileTypes = ["image/jpeg", "image/png", "image/jpg"];
  if (!allowedFileTypes.includes(type)) {
    setError({ ...error, avatarError: "Only JPEG, PNG, JPG files are allowed" });
    setValue({ ...value, avatar: "" }); // Clear the avatar value if invalid
  } else {
    setValue({ ...value, avatar: base64 });
    setError({ ...error, avatarError: "" }); // Clear the error if valid file
  }
};
```

```
const handleSubmit = (e) => {
  e.preventDefault();
  setError({}); // Clear previous errors

  // Check if there's any error related to the avatar file type
  if (error.avatarError) {
    alert("Please upload a valid image file (JPEG, PNG, JPG) before submitting.");
    return; // Prevent form submission if there's an avatar error
  }

  setLoading(true);
  dispatch(addAdmin(value));
};
```

5. NoSQL Injection

Vulnerability: The application is vulnerable to NoSQL injection due to improper handling of user inputs in database queries, which can lead to unauthorized access or manipulation of the database.

Description: NoSQL injection vulnerabilities occur when untrusted input is directly included in queries to a NoSQL database, such as MongoDB. Attackers can exploit this by crafting malicious input that alters the intended query, potentially allowing them to retrieve, modify, or delete data they shouldn't have access to.

Impact: If exploited, a NoSQL injection attack can lead to data breaches, data corruption, or unauthorized administrative access to the database. This can compromise sensitive user data and the integrity of the application.

Fix: Use parameterized queries ensure that user inputs are treated as values rather than part of the query itself.

```
const { name, dob, department, contactNumber, avatar, email, joiningYear } = req.body;

const errorsObj = { emailError: String };
const existingAdmin = await Admin.findOne({ email });
if (existingAdmin) {
  errorsObj.emailError = "Email already exists";
  return res.status(400).json(errorsObj);
}

const existingDepartment = await Department.findOne({ department });
let departmentHelper = existingDepartment.departmentCode;
const admins = await Admin.find({ department });
```

Fix: Add validation rules directly to the schema to enforce constraints like maxlength, required, and sanitization rules

```
import mongoose from "mongoose";

const adminSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    password: {
      type: String,
    },
    username: {
      type: String,
    },
    department: {
      type: String,
    },
    dob: {
      type: String,
    },
    joiningYear: {
      type: String,
    },
    avatar: {
      type: String,
    },
  }
);
```

```
import mongoose from "mongoose";

const adminSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
      maxlength: [50, "Name must be at most 50 characters long"],
      match: [/^[a-zA-Z\s]+$/, "Name can only contain letters and spaces"],
    },
    email: {
      type: String,
      required: true,
      unique: true,
      maxlength: [50, "Name must be at most 50 characters long"],
    },
    password: {
      type: String,
    },
    username: {
      type: String,
      maxlength: [50, "Name must be at most 50 characters long"],
    },
    department: {
      type: String,
    },
    dob: {
      type: String,
    },
  }
);
```

Fix: Strictly validate and sanitize all user inputs by removing potentially dangerous characters like {}, ;, ., :, ,, and " to avoid malformed queries.

```
const handleSubmit = (e) => {
  e.preventDefault();
  setError({}); // Clear previous errors

  // Check if there's any error related to the avatar file type
  if (error.avatarError) {
    alert("Please upload a valid image file (JPEG, PNG, JPG) before submitting.");
    return; // Prevent form submission if there's an avatar error
  }

  // Name validation: must be at least 3 characters long and should not contain '{' or '}'
  if (!value.name || value.name.length < 3) {
    alert("Name must be at least 3 characters long");
    return;
  }

  if (/[\{\}]/.test(value.name)) {
    alert("Name contains invalid characters: { or }");
    return;
  }

  // Contact number validation: must be exactly 10 digits
  if (!/^\d{10}$/.test(value.contactNumber)) {
    alert("Contact number must be 10 digits");
    return;
  }

  // Sanitize email input: Remove any invalid characters including '{' and '}'
  const sanitizedEmail = value.email.replace(/[\{\}]/g, '');
  if (/[\{\}]/.test(sanitizedEmail)) {
    alert("Email contains invalid characters: { or }");
    return;
  }
  setValue({ ...value, email: sanitizedEmail });

  // Set loading and dispatch the action if all validations pass
  setloading(true);
  dispatch(addAdmin(value));
};
```

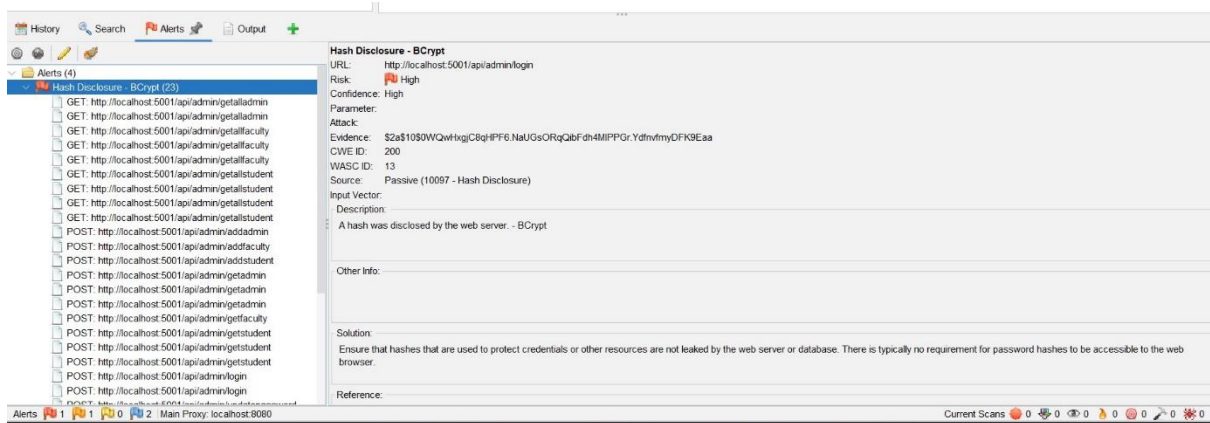
6. Hash Disclosure – Bcrypt (Sensitive Information Disclosure)

Vulnerability: Hash Disclosure occurs when hashed sensitive data, like passwords, is exposed. This increases the risk of cracking through brute force attacks, potentially leading to unauthorized system access.

Description: The application is disclosing sensitive data, such as Bcrypt hashed passwords, due to improper handling of data retrieval or output. Although the passwords are hashed, exposing these hashes can lead to security risks, especially if the hashing algorithm is cracked through brute force or other methods.

Impact: Exposing hashed passwords, even when using strong algorithms like Bcrypt, increases the risk of password cracking through brute-force or dictionary attacks. If an attacker successfully cracks a hash, they could impersonate users, gain unauthorized access to systems, and escalate their privileges, leading to serious security breaches and data compromise.

Fix: To prevent hash disclosure, ensure that sensitive data like password hashes are never included in data retrieval or output. Use proper query methods or data masking techniques to exclude sensitive information from being exposed.



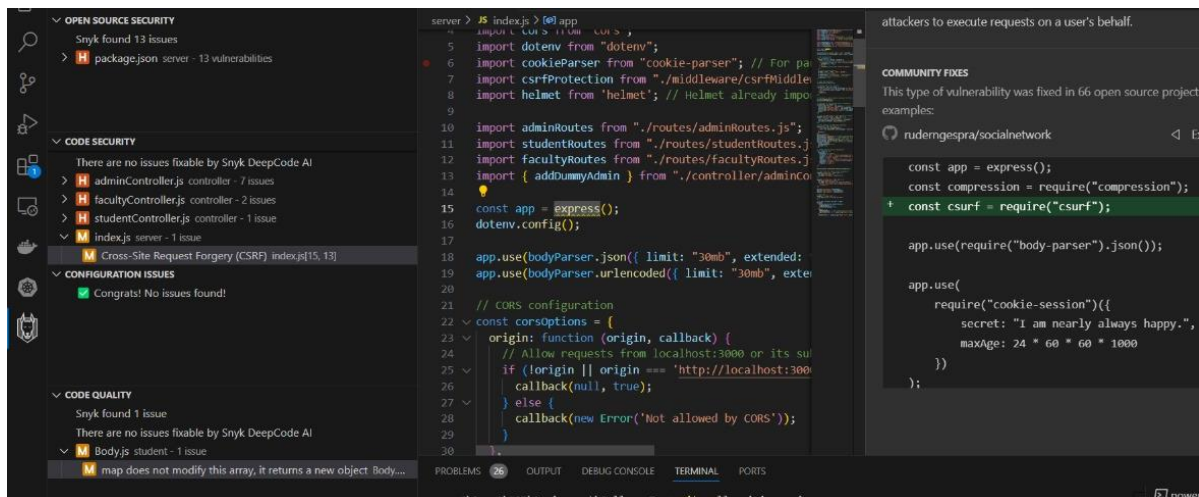
```
export const getAllFaculty = async (req, res) => {
  try {
    const faculties = await Faculty.find().select('-password');
    res.status(200).json(faculties);
  } catch (error) {
    console.log("Backend Error", error);
  }
};
```

```
const existingStudent = await Student.findOne({ email }).select('-password');
if (existingStudent) {
  return res.status(400).json({ emailError: "Email already exists" });
}

const existingDepartment = await Department.findOne({ department }).select('-password');
if (!existingDepartment) {
  return res.status(400).json({ departmentError: "Department not found" });
}
const departmentHelper = existingDepartment.departmentCode;

const students = await Student.find({ department }).select('-password');
let helper;
if (students.length < 10) {
  helper = "00" + students.length.toString();
}
```

7. CSRF (Cross-Site Request Forgery) Protection for API Calls



Vulnerability: CSRF occurs when an attacker tricks a user into making an unintended request to an API or web application that they are authenticated with. Without proper CSRF protection, an attacker can perform unauthorized actions on behalf of the user.

Definition: A **CSRF** attack is a security vulnerability where an attacker tricks a user into unknowingly submitting malicious requests to a web application or API in which the user is authenticated. CSRF attacks exploit the trust that a web application has in a user's browser.

Impact of Not Implementing CSRF Protection: If CSRF protection is not implemented for your API calls, attackers can exploit this vulnerability to perform unauthorized actions on behalf of authenticated users, such as:

- Submitting forms or making API requests.
- Changing user account details.
- Performing actions that manipulate sensitive data, such as transfers, file uploads, or data deletions.

Fix: CSRF Token Implementation for API Calls:

Description: A CSRF token is a unique, random value generated by the server. The token is sent to the client and included in every subsequent request made to the API. This ensures that the request originates from a legitimate source.

```
app.use(cookieParser());
app.use(csrfProtection);
app.use((req, res, next) => {
  console.log('Headers:', req.headers); // Log all incoming headers
  console.log("CSRF Token:", req.csrfToken()); // Log the CSRF token to the console
  console.log('Incoming request:', req.method, req.path);
  console.log('CSRF Token:', req.headers['x-csrf-token']);
  next();
});
app.get("/api/csrf-token", (req, res) => {
  const csrfToken = req.csrfToken(); // This will be generated by csrf middleware
  console.log("Generated CSRF Token:", csrfToken);
  res.json({ csrfToken: req.csrfToken() });
});
app.use((err, req, res, next) => {
  if (err.code === 'EBADCSRFTOKEN') {
    return res.status(403).json({ message: 'Invalid CSRF token' });
  }
  next(err);
});
```

```
import csrf from "csrf"; // Import the csrf package

const csrfProtection = csrf({
  cookie: {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    sameSite: 'Strict',
  }
});
export default csrfProtection;
```

```
router.post("/login", csrfProtection, adminLogin);
```

```
import { useEffect, useState } from 'react';
import axios from '../utils/axiosInstance';

const useCsrfToken = () => {
  const [csrfToken, setCsrfToken] = useState('');

  useEffect(() => {
    const fetchCsrfToken = async () => {
      try {
        const response = await axios.get('/api/csrf-token', { withCredentials: true });
        const token = response.data.csrfToken;

        // Store the fetched token in localStorage and state
        localStorage.setItem('csrfToken', token);
        setCsrfToken(token);
      } catch (error) {
        console.error('Failed to fetch CSRF token', error.response?.data || error.message);
      }
    };

    // Check if the CSRF token is already stored in local storage
    const storedCsrfToken = localStorage.getItem('csrfToken');
    if (storedCsrfToken) {
      setCsrfToken(storedCsrfToken); // Use the stored token
    } else {
      fetchCsrfToken(); // Fetch the token if not present
    }
  }, []);

  return csrfToken; // Return the CSRF token
};

export default useCsrfToken;
```

```

export const adminLogin =
  async (req, res) => {
    const { username, password } = req.body;
    const errors = { usernameError: String, passwordError: String };
    try {
      const existingAdmin = await Admin.findOne({ username });
      if (!existingAdmin) {
        errors.usernameError = "Admin doesn't exist.";
        return res.status(404).json(errors);
      }
      const isPasswordCorrect = await bcrypt.compare(
        password,
        existingAdmin.password
      );
      if (!isPasswordCorrect) {
        errors.passwordError = "Invalid Credentials";
        return res.status(404).json(errors);
      }
      const csrfToken = req.csrfToken();
      console.log('Generated CSRF Token:', csrfToken);
      const token = jwt.sign(
        {
          email: existingAdmin.email,
          id: existingAdmin._id,
          role: 'admin'
        },
        process.env.JWT_SECRET,
        { expiresIn: "1h" }
      );
      res.status(200).json({ result: existingAdmin, token: token, csrfToken });
    } catch (error) {
      console.log(error);
    }
  };

```

```

export const adminSignIn = (formData) => API.post("/api/admin/login", formData, { withCredentials: true });

```

```

import axios from 'axios';

const instance = axios.create({
  baseURL: 'http://localhost:5001',
  withCredentials: true,
});

instance.interceptors.request.use((config) => {
  const csrfToken = localStorage.getItem('csrfToken');
  if (csrfToken) {
    config.headers['X-CSRF-Token'] = csrfToken;
  }
  return config;
});

export default instance;

```

```

const login = async(e) => {
  e.preventDefault();
  setLoading(true);
  try {
    // Prepare the login data with CSRF token
    const loginData = { username: username, password: password, _csrf: csrfToken };
    await dispatch(adminSignIn(loginData, navigate));
  } catch (error) {
    console.error("Login failed:", error);
    toast.error("Login failed: " + (error.response?.data?.message || error.message));
    setLoading(false);
  }
};

```



```

    'accept-language': 'en-US,en;q=0.9',
    cookie: 'ga-GA1.1.957242086.1714578107; ga_Y0R5JBKRKR=GS1.1.1714586258.2.1.1714594164.0.0.0; ga_NBQTLHSHW6=GS1.1.1714596241.2.1.1714597441.0.0.0; ga_RW57ZGHNF=GS1.1.1716208351.16.0.1716208351.0.0.0; g_state={"i_l":0}; _csrf=9Uz_h9L39Q8XrkyXlBjdfxAf',
    'if-none-match': 'W/"1868f-IfUr/UwDxMxCJR2ok/MwF3u2TQ"'
  }
}
CSRF Token: FwArMwSo-hjYAWIj3f1Xbt2GS3xYUQhKJRU
Incoming request: GET /api/admin/getallstudent
CSRF Token: undefined
User Role: admin
Headers: {
  host: 'localhost:5001',
  connection: 'keep-alive',
  'sec-ch-ua': '"Chromium";v="128", "Not;A=Brand";v="24", "Google Chrome";v="128"',
  accept: 'application/json, text/plain, */*',
  'x-csrf-token': 'TdxKgdNL-K-yQj3xiDtdYTTga4CSb2uiQcRM',
  'sec-ch-ua-mobile': '?1',
  authorization: 'Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ3b2FpbGpCIGlmc1RlbnV1SGdtdWw1stlmVnB5S1tmIkljo1NjZlMwFkMmVnKXN0ZGF1MTQxYjIY2YjIiwicm9Sc2V6ImFkbWw1IiwiaWF0IjoxNzI0MjBMTQ0CLjlehaiojE3MjY5NDIzNDNR9.uPvs5tHE3N_CknpRk3Jhksrmt0BZd47x89KhW5123HU',
  'user-agent': 'Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Mobile Safari/537.36',
  'sec-ch-ua-platform': '"Android"',
  origin: 'http://localhost:3000',
  'sec-fetch-site': 'same-site',
  'sec-fetch-mode': 'cors',
  'sec-fetch-dest': 'empty',
  referer: 'http://localhost:3000/',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'en-US,en;q=0.9',
  cookie: 'ga-GA1.1.957242086.1714578107; ga_Y0R5JBKRKR=GS1.1.1714586258.2.1.1714594164.0.0.0; ga_NBQTLHSHW6=GS1.1.1714596241.2.1.1714597441.0.0.0; ga_RW57ZGHNF=GS1.1.1716208351.16.0.1716208351.0.0.0; g_state={"i_l":0}; _csrf=9Uz_h9L39Q8XrkyXlBjdfxAf',
  'if-none-match': 'W/"163-5DVsb+XmCE4ZmWZk2tV2YjwBuOU"'
}
CSRF Token: iRiJ4ytN-LHx4FyVmmUYbTPdHchyvDZ81rWg
Incoming request: GET /api/admin/getallfaculty
CSRF Token: TdxKgdNL-K-yQj3xiDtdYTTga4CSb2uiQcRM
User Role: admin

```

8. OAUTH Implementation

We have updated this feature to enhance the user's login experience in our application by integrating Google OAuth, allowing users to authenticate using their Google accounts seamlessly. By setting up the OAuth flow through the Google Cloud Console, we enable users to log in quickly and securely, along with the existing traditional login, it minimizes the friction often associated with traditional account creation. The integration leverages the *@react-oauth/google* package, providing an intuitive login button that opens a Google authentication popup. Upon successful login, users' information such as their name and profile picture are retrieved and stored locally, streamlining their interaction with the application.

On the backend, we ensure secure handling of OAuth tokens by implementing a verification process for the Google access tokens. This includes validating tokens and updating or creating user records in the database based on the authenticated information. Additionally, we enforce security measures such as HTTPS and CSRF protection for API requests. Overall, this update not only improves user experience by facilitating easier access to the application but also strengthens security through the established authentication protocols provided by Google.

Client>src>components>login>StudentLogin.js

```
const handleGoogleLogin = async (response) => {
  if (response.credential) {
    try {
      const formData = {
        googlecredential: response.credential, // Send the credential (JWT) directly to the backend
      };
      // Store Google credential in local storage if necessary
      localStorage.setItem("user", JSON.stringify({ result: formData }));
      // Dispatch to the backend for login
      await dispatch(studentsSignIn(formData, navigate));
    } catch (error) {
      console.error("Error during Google login:", error); // Log error details
      toast.error("Google Login failed: " + (error.response?.data?.message || error.message));
    }
  } else {
    console.error("No credential found in Google response");
  }
};

useEffect(() => {
  if (store.errors) {
    setLoading(false);
    setUsername("");
    setPassword("");
  }
}, [store.errors]);
```

```
/* Google Sign-In Button */
<GoogleLogin
  onSuccess={handleGoogleLogin}
  onFailure={(error) => [
    console.error("Login Failed:", error);
    toast.error("Google Login failed!"); // Show an error message
  ]}
  style={{
    marginTop: "1rem",
    width: "100%",
  }}
  logo_alignment="left" // Optional: adjust as necessary
  text="Sign in with Google" // Add custom text if needed
/>

</form>
</div>
</div>
```

```

import Test from "../models/test.js";
import Student from "../models/student.js";
import Department from "../models/department.js"; // Adjust the import path as necessary
import Subject from "../models/subject.js";
import Marks from "../models/marks.js";
import Attendance from "../models/attendance.js";
import jwt from "jsonwebtoken";
import bcrypt from "bcryptjs";
import { OAuth2Client } from 'google-auth-library'; // Import Google's OAuth2 client

const client = new OAuth2Client(process.env.GOOGLE_OAUTH_CLIENT_ID); // Make sure to set GOOGLE_CLIENT_ID in your .env file

import { body, validationResult } from "express-validator";

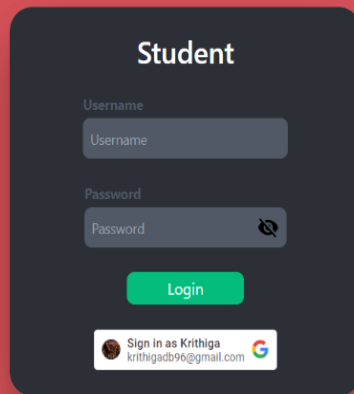
export const studentLogin = async (req, res) => {
  const { googleCredential, username, password } = req.body;
  try {
    if (googleCredential) {
      // Verify the Google credential
      const ticket = await client.verifyIdToken({
        idToken: googleCredential,
        audience: process.env.GOOGLE_CLIENT_ID,
      });
      const googleData = ticket.getPayload();
      const { email, name, picture } = googleData;
      // Fetch or create student in the database
      let existingStudent = await Student.findOne({ email });
      if (!existingStudent) {
        const existingDepartment = await Department.findOne({ department: 'General' });
        let departmentHelper = existingDepartment ? existingDepartment.departmentCode : '000';
        const currentYear = new Date().getFullYear();
        // Generate username
        const lastStudent = await Student.findOne({ department: 'General' }).sort({ _id: -1 });
        const lastIdNumber = lastStudent ? parseInt(lastStudent.username.slice(-3)) : 0;
        const newIdNumber = lastIdNumber + 1;
        const helper = String(newIdNumber).padStart(3, '0'); // Helper for padding
        const newUsername = `STU${currentYear}${departmentHelper}${helper}`;
        // Create a new student with default required fields
        const defaultData = {
          email,
          name,
          avatar: picture || null,
          dob: '2000-01-01',
          section: 'A',
          department: 'General',
          year: 1,
          username: newUsername,
          password: await bcrypt.hash('default_password', 10), // Hash the default password
          passwordUpdated: false // Set it to false for first-time user
        };
        existingStudent = await Student.create(defaultData); // Assign created student to existingStudent
      }
      const token = jwt.sign(
        { email: existingStudent.email, id: existingStudent._id, role: 'student' },
        process.env.JWT_SECRET,
        { expiresIn: '1h' }
      );
      return res.status(200).json({ result: existingStudent, token });
    } else if (username && password) {
      const errors = { usernameError: null, passwordError: null };
      const existingStudent = await Student.findOne({ username });
      if (!existingStudent) {
        errors.usernameError = "Student doesn't exist.";
        return res.status(404).json(errors);
      }
      const isPasswordCorrect = await bcrypt.compare(password, existingStudent.password);
      if (!isPasswordCorrect) {
        errors.passwordError = "Invalid Credentials";
        return res.status(401).json(errors); // Changed status to 401 for invalid credentials
      }
      // Generate JWT token for traditional login
      const token = jwt.sign(
        { email: existingStudent.email, id: existingStudent._id, role: 'student' },
        process.env.JWT_SECRET, // Use the environment variable for JWT secret
        { expiresIn: '1h' }
      );
      return res.status(200).json({ result: existingStudent, token });
    } else {
      return res.status(400).json({ message: 'Invalid login attempt.' });
    }
  } catch (error) {
    console.error('Error during student login:', error);
    return res.status(500).json({ message: 'Internal server error.' });
  }
};

```

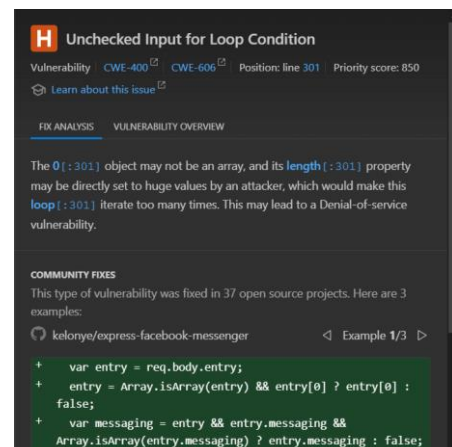
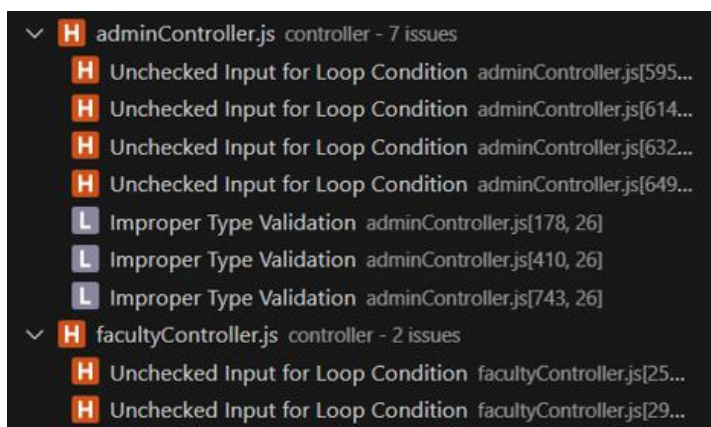
```

    };
    existingStudent = await Student.create(defaultData); // Assign created student to existingStudent
  }
  const token = jwt.sign(
    { email: existingStudent.email, id: existingStudent._id, role: 'student' },
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
  );
  return res.status(200).json({ result: existingStudent, token });
} else if (username && password) {
  const errors = { usernameError: null, passwordError: null };
  const existingStudent = await Student.findOne({ username });
  if (!existingStudent) {
    errors.usernameError = "Student doesn't exist.";
    return res.status(404).json(errors);
  }
  const isPasswordCorrect = await bcrypt.compare(password, existingStudent.password);
  if (!isPasswordCorrect) {
    errors.passwordError = "Invalid Credentials";
    return res.status(401).json(errors); // Changed status to 401 for invalid credentials
  }
  // Generate JWT token for traditional login
  const token = jwt.sign(
    { email: existingStudent.email, id: existingStudent._id, role: 'student' },
    process.env.JWT_SECRET, // Use the environment variable for JWT secret
    { expiresIn: '1h' }
  );
  return res.status(200).json({ result: existingStudent, token });
} else {
  return res.status(400).json({ message: 'Invalid login attempt.' });
}
} catch (error) {
  console.error('Error during student login:', error);
  return res.status(500).json({ message: 'Internal server error.' });
}
};

```



9. Unchecked Input for Loop Validation



Vulnerability: Snyk identified an issue where loop conditions rely on unchecked user input, which poses a security risk. Without validating the input size, attackers can exploit this to send large payloads, potentially leading to resource exhaustion or application crashes.

Description: When user input is used directly in loop conditions without proper validation, it can result in performance issues, high memory consumption, or application instability. Attackers may exploit this to overwhelm the system or cause denial of service (DoS).

Impact: Unvalidated input in loops can degrade application performance or cause crashes, potentially enabling DoS attacks by overwhelming server resources.

Fix: Validate user input to ensure it is within acceptable size limits before using it in loops, preventing large payloads from impacting application performance.

```
// Validate selectedStudents input
if (!Array.isArray(selectedStudents) || selectedStudents.length === 0 || selectedStudents.length > 1000) {
  return res.status(400).json({ message: "Invalid input: selectedStudents must be a non-empty array with at most 1000 items." });
}

for (var a = 0; a < selectedStudents.length; a++) {
  // Validate selectedStudents input
  if (!Array.isArray(selectedStudents) || selectedStudents.length === 0 || selectedStudents.length > 1000) {
    return res.status(400).json({ message: "Invalid input: selectedStudents must be a non-empty array with at most 1000 items." });
  }
}
```

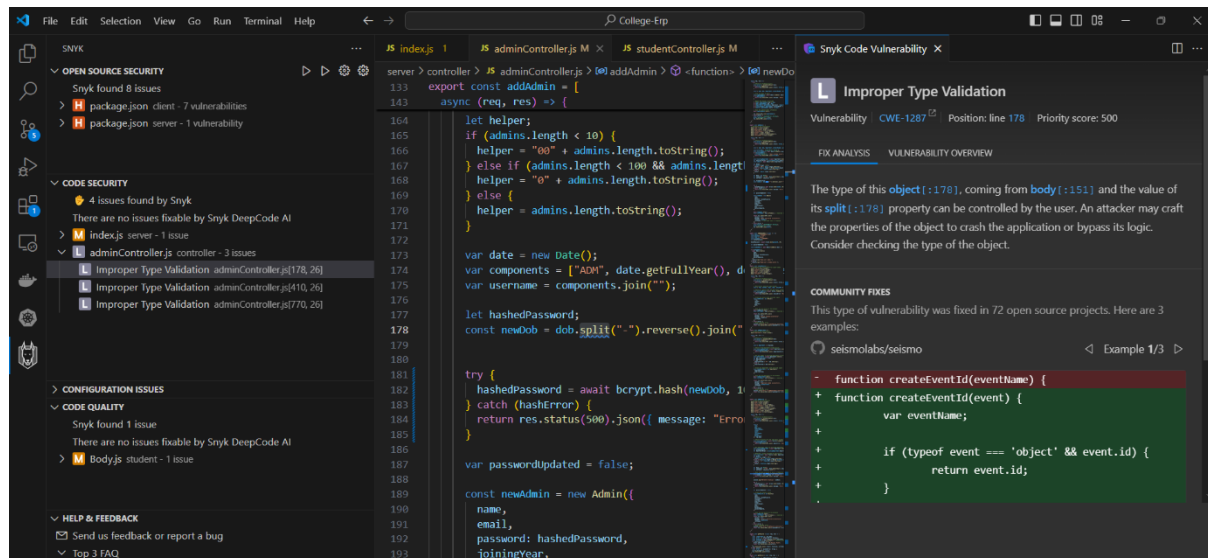
10. Improper Type Validation

Vulnerability: Improper type validation was identified during input handling. When input data is not properly checked for its type, it can lead to unexpected behaviours, errors, or security vulnerabilities within the application.

Description: Inadequate validation of input types can cause the application to behave unpredictably, potentially exposing it to attacks that exploit incorrect assumptions about the input data format.

Impact: If type validation is not strictly enforced, the application may fail or operate incorrectly, leading to potential crashes, data corruption, or vulnerabilities that attackers can exploit.

Fix: Ensure proper type validation by enforcing strict data type checks on all user inputs before processing. This prevents invalid data from being executed and helps safeguard the application's stability and security.



```
// Ensure dob is a string before processing
let newDob;
if (typeof dob === 'string') {
  newDob = dob.split("-").reverse().join("-");
} else {
  return res.status(400).json({ error: "Invalid date of birth format" });
}
```

Vulnerabilities that were not fixed and the reason for not fixing them

Various vulnerabilities in systems often remain unfixed due to practical constraints. **Low severity vulnerabilities**, such as minor security issues or outdated libraries with low CVSS scores, are frequently deprioritized as teams focus on more critical threats. Similarly, **legacy code dependencies** may contain known vulnerabilities but are essential to the application; refactoring them could lead to extensive changes, increasing development time and potentially introducing new bugs.

Resource constraints also contribute, as vulnerabilities requiring significant time or resources to remediate are often deemed lower priority due to limited team resources and tight deadlines. Moreover, **functional impact** is a concern, as fixing vulnerabilities could disrupt existing application functionality or user experience.

Third-party dependencies present another challenge, with vulnerabilities in external libraries often out of the team's control to patch, requiring reliance on maintainers for fixes. The absence of **defined security policies** can lead to oversights in addressing specific vulnerabilities. Additionally, **architectural issues** stemming from poor design may require complex redesigns, while **excessive resource use**, like memory leaks, may not be prioritized due to perceived lower risk.

To address these issues, organizations should establish **clear security policies**, allocate resources for remediation, and prioritize secure architecture. Automated security testing and regular code reviews can help mitigate these vulnerabilities before they escalate.

Best Practices to Prevent Common Software Vulnerabilities

To enhance the security and robustness of our application, the following best practices are recommended for adoption throughout the software development process:

1. Static Code Analysis
 - Stage: During Development (Pre-Commit / Continuous Integration)
 - Purpose: As developers write code, static analysis tools (e.g., ESLint, SonarQube) should be run to detect issues like missing input validation, insecure coding practices, and the use of vulnerable dependencies (like Babel, Axios, etc.). This should be automated in the Continuous Integration (CI) pipeline.
 - Example: Checking for unsanitized user inputs or improper handling of untrusted data to avoid Cross-Site Scripting (XSS), SQL injection, or NoSQL injection.
2. Dependency Scanning
 - Stage: During Development & Before Build
 - Purpose: This ensures that external libraries and dependencies are free from known vulnerabilities. Tools like npm audit, or OWASP Dependency-Check should be used before building the application or during the CI process.
 - Example: Running these tools would flag vulnerabilities such as CVE-2023-45133 (Babel) or CVE-2023-45644 (Axios SSRF)
3. Unit Testing
 - Stage: During Development
 - Purpose: Ensure that individual functions/modules behave as expected, especially for functions handling sensitive data or user input. Unit tests should validate edge cases and handle exceptions properly.
 - Example: Writing unit tests to validate that user inputs are sanitized before being processed to avoid NoSQL injection or file upload vulnerabilities.
4. Integration Testing
 - Stage: Post-Development / Before Integration
 - Purpose: Verify the interaction between different modules (e.g., front-end and back-end). This helps ensure that security features like authentication and Role-Based Access Control (RBAC) are functioning correctly.
 - Example: Testing the API endpoints to ensure only authorized users (e.g., Admin, Faculty) have access to specific routes based on their roles.
5. Secure Coding Standards
 - Stage: Throughout Development
 - Purpose: Establish a set of guidelines emphasizing input validation, output encoding, and proper error handling. Ensure the use of frameworks and libraries adhering to security best practices, such as OWASP Secure Coding Practices.
 - Example: Enforcing output encoding and input validation across all user input fields to prevent attacks like XSS and SQL injection.
6. Code Review and Pair Programming
 - Stage: During Development

- Purpose: Implement a rigorous code review process where peers examine each other's code for security vulnerabilities and adherence to secure coding standards. Employ pair programming to facilitate real-time feedback.
 - Example: Catching insecure code practices like hardcoded credentials during code reviews or via collaborative problem-solving in pair programming.
7. Principle of Least Privilege (PoLP)
- Stage: During Design & Development
 - Purpose: Enforce minimal access for users and systems, limiting the damage caused by compromised accounts or services. This reduces the attack surface.
 - Example: Implement role-based access control (RBAC) to ensure users (e.g., Admin, Faculty, Student) only access features required for their roles.
8. Threat Modelling.
- Stage: Design Phase
 - Purpose: Identify potential attack vectors and vulnerabilities early in the design phase. Use methodologies like STRIDE or DREAD to assess and prioritize security risks.
 - Example: Conduct threat modeling sessions to uncover risks such as SSRF and data exfiltration and develop mitigation strategies.
9. Continuous Integration/Continuous Deployment (CI/CD) Pipeline
- Stage: During Development & Deployment
 - Purpose: Automate security testing within the CI/CD pipeline. Include automated tests for common vulnerabilities (e.g., XSS, CSRF, SQL injection) during build and deployment processes.
 - Example: Integrating SAST tools like SonarQube and DAST tools like OWASP ZAP in the CI/CD pipeline to ensure all changes are tested for security issues before deployment.

Individual Contribution

Group Member Name	Group member IT No	Vulnerability Identified & Fixed
Sanjayan. C	IT21375514	<ol style="list-style-type: none"> 1. Dependency Vulnerabilities 2. Security Misconfiguration <ol style="list-style-type: none"> 1. CSP: Wildcard Directive 2. CSP Header Not Set 3. Missing Anti-clickjacking Header (Clickjacking) 4. Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) 5. X-Content-Type-Options Header Missing (MIME Type Sniffing) 6. Lack of Referrer Policy (Referrer Policy Not Set) 7. Missing HTTP Strict Transport Security (HSTS Header Not Set) 8. Cross-Domain Misconfiguration / Misconfigured Access-Control-Allow-Origin Header 9. Inadequate Role-Based Access Control (RBAC) 10. Hardcoded Secret 11. Improper Error Handling in Password Hashing
Balakrishnan K. D	IT21194894	<ol style="list-style-type: none"> 1. CSRF 2. Unchecked Input for Loop Validation, 3. Improper Type Validation 4. Google OAuth
Shandeep. J	IT21375682	<ol style="list-style-type: none"> 1. Cross-Site Scripting (XSS) 2. Unrestricted file upload 3. No SQL injections 4. Hash Disclosure – Bcrypt (Sensitive Information Disclosure)