

Name: Balkrishna Giri

Student Number: 152982814

## Purpose and Functionality

The Secure Blog App is designed to allow day-to-day users to create, edit, and post blogs in a secure environment. The primary purpose of the application is to demonstrate secure programming practices within a real-world context. By implementing various secure mechanisms throughout, the app addresses common vulnerabilities—such as brute-force attacks, token tampering, and cross-site scripting (XSS)—that are critical concerns in secure programming.

## User Interface and Experience

- **Interface:** The application is presented as a responsive website (web app) where users can navigate, create new posts, and manage their content.
- **User Workflow:** The interface is designed to be intuitive. Basic elements such as forms for blog creation and editing are straightforward; however, certain security-related features (like account lockout and session validation) are implemented behind the scenes. Screenshots and code snippets (including attempts at brute-force and XSS attacks) are provided in the accompanying documentation to illustrate these security measures.

## High-Level Structure

The application is built using a modern MERN (MongoDB, Express, React, Node.js) stack. Its architecture is divided into three main components:

Frontend:

- Framework: React
- Components: Includes various pages and React components for content display, forms, and state management using context.
- Libraries: React Router for navigation, Axios for API calls, Tailwind CSS and Headless UI for styling and accessible components.

Backend:

- Framework: Express running on Node.js
- Modules: Handles API requests, user authentication using JWT and bcryptjs for password encryption, and applies secure middleware for validating requests.
- Security Features: Implements account lockout mechanisms after five unsuccessful login attempts (with a 15-minute lockout period), JWT token validation, and secure error handling.

Database Interface:

- Database: MongoDB accessed via Mongoose.
- Data Storage: Responsible for persisting user data, blog posts, and security configurations.

Secure-Blog-App/

  └── backend/

    |  └── node\_modules/

    |  └── src/

      |  └── config/

      |  |  └── middleware/

      |  |  |  └── auth.js

      |  |  |  └── security.js

      |  └── models/

      |  |  └── Post.js

      |  |  └── User.js

      |  └── routes/

      |  |  └── postRoutes.js

      |  |  └── userRoutes.js

      |  └── .env

      |  └── .env.example

      |  └── package-lock.json

      |  └── package.json

  └── frontend/

    └── node\_modules/

    └── public/

    └── src/

      |  └── components/

      |  |  └── LoginForm.js

      |  |  └── NavBar.js

      |  |  └── Post.js

      |  |  └── PostForm.js

```
| | └── ProtectedRoute.js
| | └── RegisterForm.js
| └── context/
| | └── AuthContext.js
| └── pages/
| | └── Home.js
| | └── PostDetail.js
| └── App.css
| └── App.js
| └── App.test.js
| └── index.css
| └── index.js
└── package-lock.json
└── package.json
```

## System Workflow:

1. User Interaction: The frontend communicates with the backend through RESTful API calls.
2. API Processing: The backend validates input, processes secure authentication, and interacts with the database using Mongoose.
3. Response: The processed data is returned to the frontend for display, creating a secure loop of data handling and user interaction.

## Technologies and Environment

### Development Tools:

- **IDE:** Visual Studio Code (VSCode)
- **Version Control:** GitHub for source code management

### Runtime Environment:

- **Frontend:** Bootstrapped using Create React App and Tailwind CSS.
- **Backend:** Node.js environment configured to handle scalable network applications.

## Secure Design Principles:

### Data Encryption:

- Passwords are encrypted using bcryptjs.
- Data exchanges are secured using TLSv1.2 even when HTTP is used, ensuring data is encrypted.

### Authentication and Token Security:

- JWT tokens are used to manage sessions. The token's payload includes an expiry timestamp (verified by decoding on jwt.io and checking via a Unix timestamp converter).
- The application enforces token validation to prevent token tampering; any modification of the token results in a failed authentication.

## Adherence to Best Practices Using OWASP Top 10

1. Injection Attacks: The use of a NoSQL database (MongoDB) reduces traditional SQL injection risks. Input is validated and sanitized even though the risk is less prominent with NoSQL databases.

The screenshot shows a login interface for a "BlogApp". The top navigation bar has links for "Home", "Login", and "Register". The main area is titled "Sign in to your account". There is an input field for an email address containing "'OR 1=1 -". A validation message box appears, stating: "Please include an '@' in the email address. "'OR 1=1 -' is missing an '@'." Below the message is a blue "Sign in" button.

2. Broken Authentication: To mitigate brute force attacks, the login system enforces a maximum of five unsuccessful attempts, locking the account for 15 minutes upon reaching this limit. Code snippets and BurpSuite screenshots below document this implementation.

JS User.js X JS userRoutes.js ...

backend > src > models > JS User.js > userSchema

```
4 const userSchema = new mongoose.Schema({
5   password: {
6     type: String
7   },
8   isAdmin: {
9     type: Boolean,
10    default: false
11  },
12  // Lockout settings for security after 5 failed attempts
13  loginAttempts: {
14    type: Number,
15    default: 0
16  },
17  lockUntil: {
18    type: Date,
19    default: null
20  }
21 }, {
22   timestamps: true
23 });
24
25 // Hash password before saving
26
```

Pieces: Comment | Pieces: Explain

Accept Ctrl+Y | Reject Ctrl+N

JS User.js JS userRoutes.js X ...

backend > src > routes > JS userRoutes.js > router.post('/login') callback

```
36 router.post('/login', async (req, res) => {
37   return res.status(401).json({ error: 'Invalid credentials' });
38 }
39
40 // Check if account is locked
41 if (user.lockUntil && user.lockUntil > Date.now()) {
42   const remainingTime = Math.ceil((user.lockUntil - Date.now()) / 1000 / 60);
43   return res.status(401).json({
44     error: `Account is locked. Please try again in ${remainingTime} minutes.`
45   });
46
47 const isMatch = await user.comparePassword(password);
48
49 if (!isMatch) {
50   // Increment failed login attempts
51   user.loginAttempts += 1;
52
53   // Lock account after 5 failed attempts for 15 minutes
54   if (user.loginAttempts >= 5) {
55     user.lockUntil = new Date(Date.now() + 15 * 60 * 1000); // 15 minutes
56     await user.save();
57     return res.status(401).json({
58       error: 'Too many failed attempts. Account locked for 15 minutes.'
59     });
60
61   await user.save();
62   return res.status(401).json({ error: 'Invalid credentials' });
63 }
64
65 // Reset login attempts on successful login
66 user.loginAttempts = 0;
67 user.lockUntil = null;
68 await user.save();
69
70 const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {
71   expiresIn: '7d'
72 });
73
74 res.json({ user, token });
75 }
```

1 of 4 ▾ Revert file Ctrl+Q | Accept file Ctrl+D | Review next file >

Attack Save

2. Intruder attack of http://localhost:5000

Results Positions

Capture filter: Capturing all items View filter: Showing all items Apply capture filter

Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
0		401	221			756	
1	000000	401	151			756	
2	100000	401	180			756	
3	200000	401	179			793	
4	300000	401	8			787	
5	400000	401	6			787	
6	500000	401	6			787	
7	600000	401	6			787	
8	700000	401	9			787	

Request Response

Pretty Raw Hex Render

```

1 HTTP/1.1 401 Unauthorized
2 Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'; img-src 'self' data: https: font-src 'self' data: connect-src 'self'
3 X-Content-Type-Options: DEFLATE
4 X-XSS-Protection: 1; mode=block
5 X-Content-Type-Options: nosniff
6 Access-Control-Allow-Origin: http://localhost:3000
7 Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
8 Access-Control-Allow-Headers: Content-Type, Authorization
9 Access-Control-Allow-Credentials: true
10 Content-Type: application/json; charset=utf-8
11 Content-Length: 62
12 ETag: W/3e-9HcxdemHt2LcQ8lvcbs5yvKU
13 Date: Thu, 03 Apr 2025 17:06:26 GMT
14 Connection: keep-alive
15 Keep-Alive: timeout=5
16
17 {
  "error": "Account is locked. Please try again in 15 minutes."
}

```

Paused Search 0 highlights

localhost:3000/login

BlogApp Home Login Register

Sign in to your account

bala@bala.com

Account is locked. Please try again in 5 minutes.

Sign In

3. Token Validation: JWT tokens are used to manage sessions. The token's payload includes an expiry timestamp (verified by decoding on jwt.io and checking via a Unix timestamp converter). Below attached is the screenshot for the same.  
login and from local storage (in console) copy the JWT code for the login-ed user.

Cache Storage Cookies Indexed DB Local Storage Session Storage

Value

all-auth... {"username": "Muhammad", "password": "123456", "remember\_me": false, "secure": false, "client\_id": "1", "token": "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJsb2dpbiIsInVzZXJfaWQiOjEwLCJleHAiOjE2NjUxOTMwNTk3LCJ0eXAiOiJKV1QiLCJ4eXAiOiJsb2dpbiJ9.1gXfZCmDqYRzPQdXWzLqjyqjgkWtV1HmHjQyJdM4C1DwWWjMMyqgwaJWf8yfslUpw", "token\_type": "bearer", "exp": 1766105000, "iat": 1766098000}

tokens

\_proto\_\_Array

Copy this token and paste into jwt.io to decode, under “exp” you will get unix time stamp for token expiry.

The screenshot shows the jwt.io interface. At the top, there is a pink banner with the text "Get an exclusive look at jwt.io v2 and help us shape its final form with your feedback. →". Below the banner, the JUUT logo is on the left, followed by links for Debugger, Libraries, Introduction, and Ask. To the right, it says "Crafted by auth0". The main area has tabs for Algorithm (set to HS256) and Encoded/Decoded. The Encoded tab shows a long base64 string:  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3ZThmZGMzMDFkOGM2MWE2ZD1iODU30SISImIhdCI6MTc0MzkxMDkzNiwiZXhwIjoxNzQ0NTE1NzM2fQ.u4tt4AizC11bw7tkHissAKz9ywQLwJnFF8s95ohLPbc

The Decoded tab shows the JSON structure of the token:

```
HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "HS256",
  "typ": "JWT"
}

PAYLOAD: DATA
{
  "id": "67e8fdc301d8c61a6d9b8579",
  "iat": 1743910936,
  "exp": 1744515736
}

VERIFY SIGNATURE
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

Below the tabs, there is a red error message: "⊗ Invalid Signature". On the right, there is a blue button labeled "SHARE JWT".

Copy the value of “exp” which is “1744515736” in our case and paste it on <https://www.unixtimestamp.com/> to get the expiry of the token

The screenshot shows the unixtimestamp.com website. At the top, there is a navigation bar with categories: Dan's Tools, Web Dev, Conversion, Encoders / Decoders, Formatters, Internet, and English. The main content area is titled "The Current Epoch Unix Timestamp". It has a text input field containing "1744515736" and a large digital clock displaying "1743911642 SECONDS SINCE JAN 01 1970. (UTC)" with the time "6:54:03 AM". Below the digital clock, there is a "Copy" button. A table below shows timestamp representations in different formats:

Format	Seconds
GMT	Sun Apr 13 2025 03:42:16 GMT+0000
Your Time Zone	Sun Apr 13 2025 06:42:16 GMT+0300 (Eastern European Summer Time)
Relative	in 7 days

Below the table, there is a section to "Enter a Date & Time" with fields for Year (2025), Month (04), Day (06), Hour (03), Minutes (52), and Seconds (35). A "Convert →" button is located below these fields. At the bottom, a note states "The current epoch translates to" followed by a redacted URL.

4. Token tampering: Copy the same JWT token from previous step and now using Postman hit <http://localhost:5000/api/users/profile> api using get and Authorization: Bearer <JWT\_token>, as we have injected correct token it should validate the login session.

The screenshot shows the Postman interface with a successful API call. The URL is <http://localhost:5000/api/users/profile>. The response status is 200 OK, and the body contains the following JSON data:

```

1  {
2   "id": "67e0fd301d0c61a6d9b0579",
3   "username": "admin",
4   "email": "bala@bala.com",
5   "password": "$2a$10$ynePAwRwK/OGmzxGsyKeII1MpHnysRfx001Wm0fFTx8cVA.EwdBG",
6   "isAdmin": true,
7   "createdAt": "2025-03-30T08:16:03.476Z",
8   "updatedAt": "2025-04-06T03:42:16.922Z",
9   "_v": 0,
10  "loginAttempts": 0,
11  "lockUntil": null
12 }

```

However, if we try to tamper the token and try to login it fails.

The screenshot shows the Postman interface with an unauthorized API call. The URL is <http://localhost:5000/api/users/profile>. The response status is 401 Unauthorized, and the body contains the following JSON data:

```

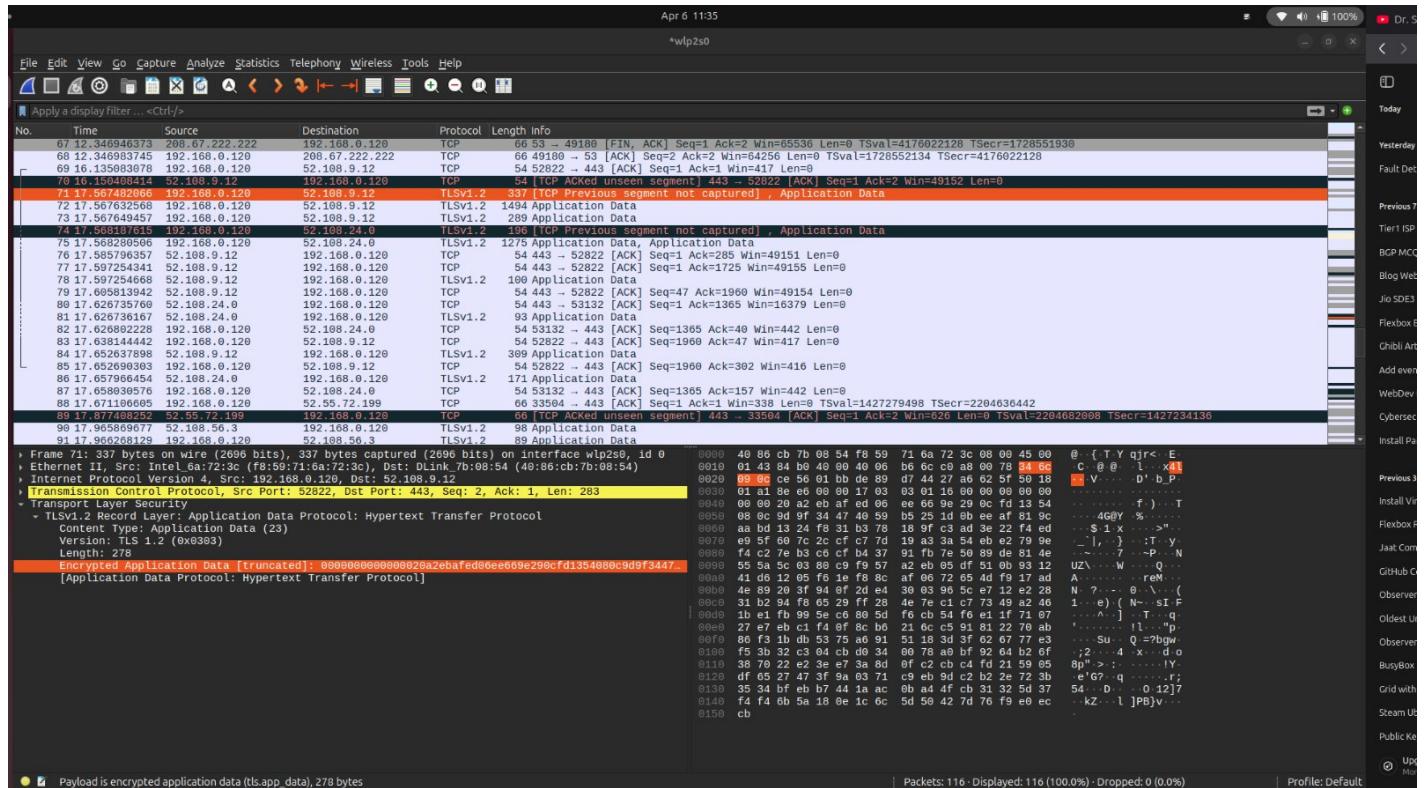
1  {
2   "error": "Please authenticate."
3 }

```

code for JWT creation and validation is below:

```
ckend /src /middleware / auth.js ...  
1  /**  
2   * Authentication middleware to verify JWT tokens and user authentication  
3   * This middleware is used to protect routes that require authentication  
4   */  
5  const jwt = require('jsonwebtoken');  
6  const User = require('../models/User');  
7  
8  /**  
9   * Middleware to verify JWT token and authenticate user  
10  * @param {Object} req - Express request object  
11  * @param {Object} res - Express response object  
12  * @param {Function} next - Express next middleware function  
13  */  
14 const auth = async (req, res, next) => {  
15  try {  
16    // Extract token from Authorization header  
17    const token = req.header('Authorization')?.replace('Bearer ', '');  
18  
19    // Check if token exists  
20    if (!token) {  
21      throw new Error();  
22    }  
23  
24    // Verify token and decode payload  
25    const decoded = jwt.verify(token, process.env.JWT_SECRET);  
26  
27    // Find user by ID from token payload  
28    const user = await User.findOne({ _id: decoded.id });  
29  
30    // Check if user exists  
31    if (!user) {  
32      throw new Error();  
33    }  
34  
35    // Attach user and token to request object for use in routes  
36    req.user = user;  
37    req.token = token;  
38    next();  
39  } catch (error) {  
40    // Return 401 Unauthorized if authentication fails  
41    res.status(401).json({ error: 'Please authenticate.' });  
42  }  
43};  
44
```

5. Sensitive Data Exposure: Even though requests are sent using HTTP, TLSv1.2 is enforced to guarantee that data remains encrypted during transmission.



6. Broken Access Control: Role-based access control is implemented to ensure that normal users cannot modify or delete administrative posts. Attempts to modify privileged content results in an error response. However, admin having privileged rights can edit and delete any user's post.

In Postman hit (Delete) <http://localhost:5000/api/posts/<admins-Post-id>>

with Authorization: Bearer <JWT\_token from prev steps>

Body	Cookies	Headers (14)	Test Results
<pre>{   "error": "Not authorized to delete this post" }</pre>			403 Forbidden   8 ms   768 B

Thus, as a user we can't delete the admin post as mentioned in JSON ("error": "Not authorized to delete this post").

Similarly, we can try to edit an admins post as a normal user and evident in Screenshot below it throws error.

The screenshot shows the Postman interface. On the left, there's a sidebar with 'My Workspace' containing a collection named 'Your collection'. The main area shows a PATCH request to `http://localhost:5000/api/posts/67e8fe0801d8c61a6d9b85a5`. The 'Body' tab is selected, showing the following JSON:

```

1 | {
2 |   "title": "Updated by normal user",
3 |   "content": "Updated content of the post"
4 |

```

The response section shows a 403 Forbidden status with the following JSON body:

```

1 | {
2 |   "error": "Not authorized to edit this post"
3 |

```

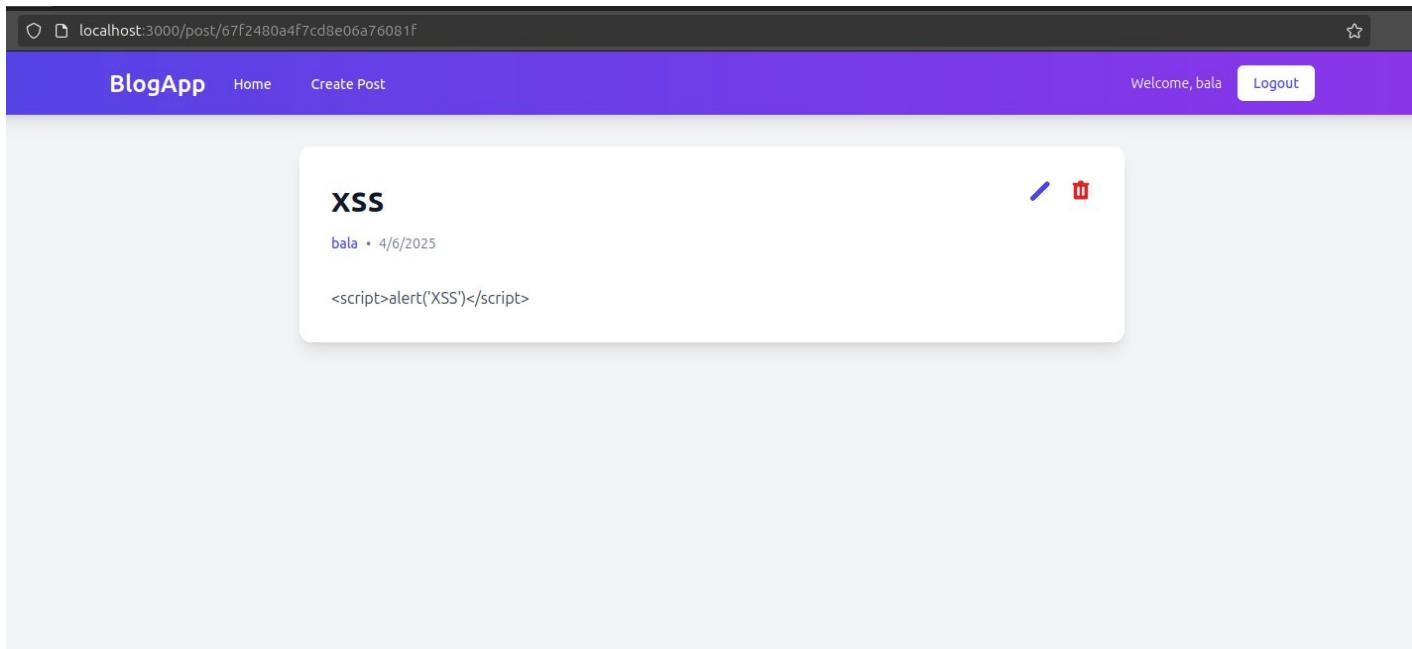
- XSS and Click-Jacking Prevention: The website has been secured against XSS and click-jacking by applying restrictive content security policies (CSP) and using the X-XSS-Protection header. An attempted stored XSS attack is demonstrated with accompanying code and screenshots. We have used inbuilt libraries.

```

un Terminal Help securityjs - experiment - Cursor
JS User.js JS security.js X JS server.js JS userRoutes.js JS postRoutes.js ...
backend > src > middleware > JS security.js > securityMiddleware
5 const securityMiddleware = (req, res, next) => {
13 // data: - Allow data URIs for images and fonts
14 res.setHeader('Content-Security-Policy',
15   "default-src 'self'; " + // Default: only allow from same origin
16   "script-src 'self' 'unsafe-inline' 'unsafe-eval'; " + // Scripts: allow inline and eval
17   "style-src 'self' 'unsafe-inline'; " + // Styles: allow inline
18   "img-src 'self' data: https:; " + // Images: allow from same origin, data URIs, and HTTPS
19   "font-src 'self' data:; " + // Fonts: allow from same origin and data URIs
20   "connect-src 'self'" // AJAX/WebSocket: only allow to same origin
21 );
22
23 // X-Frame-Options header
24 // Prevents clickjacking attacks by disabling iframe embedding
25 res.setHeader('X-Frame-Options', 'DENY');
26
27 // X-XSS-Protection header
28 // Enables browser's built-in XSS filtering
29 res.setHeader('X-XSS-Protection', '1; mode=block');
30
31 // X-Content-Type-Options header
32 // Prevents MIME-type sniffing which could lead to security vulnerabilities
33 res.setHeader('X-Content-Type-Options', 'nosniff');
34
35 // Cross-Origin Resource Sharing (CORS) headers
36 // Controls which domains can access the API
37 res.setHeader('Access-Control-Allow-Origin', 'http://localhost:3000'); // Only allow requests from
38 res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS'); // Allowed HTTP
39 res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization'); // Allowed request
40 res.setHeader('Access-Control-Allow-Credentials', 'true'); // Allow credentials (cookies, authorization)

```

stored XSS below throws no popup.



8. XML External Entities (XXE) vulnerabilities are inherently avoided due to the use of JSON parsing rather than XML processing.

## Testing and Quality Assurance

Security Testing Approach:

- All testing was conducted manually with a focus on security aspects.
- Testing involved both standard functionality tests as well as security-specific attempts (e.g., brute force, token tampering, and XSS).

Testing Outcomes:

- Vulnerabilities such as potential brute force attacks and token tampering were identified during testing and addressed using account lockout mechanisms and strict JWT validation.
- Code-level tests confirmed that attempts to modify or bypass authentication failed as intended.

Additional Notes: Although no automated or DevSecOps pipeline was integrated, the manual testing results were thoroughly documented and included code snippets and screenshots for validation.

## Evolution of the Project

- The Secure Blog App was developed entirely from scratch, allowing full control over both functionality and security enhancements.
- The project did not evolve from earlier work; instead, it represents a fresh and comprehensive implementation aimed at solidifying secure programming practices.

## **Known Issues and Future Improvements**

### **Unimplemented Features:**

- Two-Factor Authentication (2FA): The implementation of 2FA was considered but not completed. The idea was to limit OTP delivery exclusively to Gmail accounts, due to complexities and costs associated with supporting additional email providers.

### **Potential Enhancements:**

- Future iterations could implement comprehensive 2FA to broaden user authentication methods.
- Other potential features include a blog post sharing functionality and further integration of security tools.

### **Known Vulnerabilities:**

- As of the latest testing and evaluation, no residual security vulnerabilities have been detected.

### **Repository:**

- The complete source code is available on GitHub at [Secure Blog App](#).

### **Documentation Style:**

- The source code is well-documented with inline comments, and vital modules are explained through the code itself.

### **User and Developer Guidelines:**

- Basic setup instructions include running “npm install” in both the frontend and backend directories followed by the appropriate run commands.

*Note\* AI-powered writing assistance such as ChatGPT and GROK was used to refine documentation and this final report, ensuring clarity and completeness. While AI did not write the core logic or make security decisions, it played a helpful role in accelerating development and supporting best practices.*