COMP.SEC.300-2024-2025-1

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:

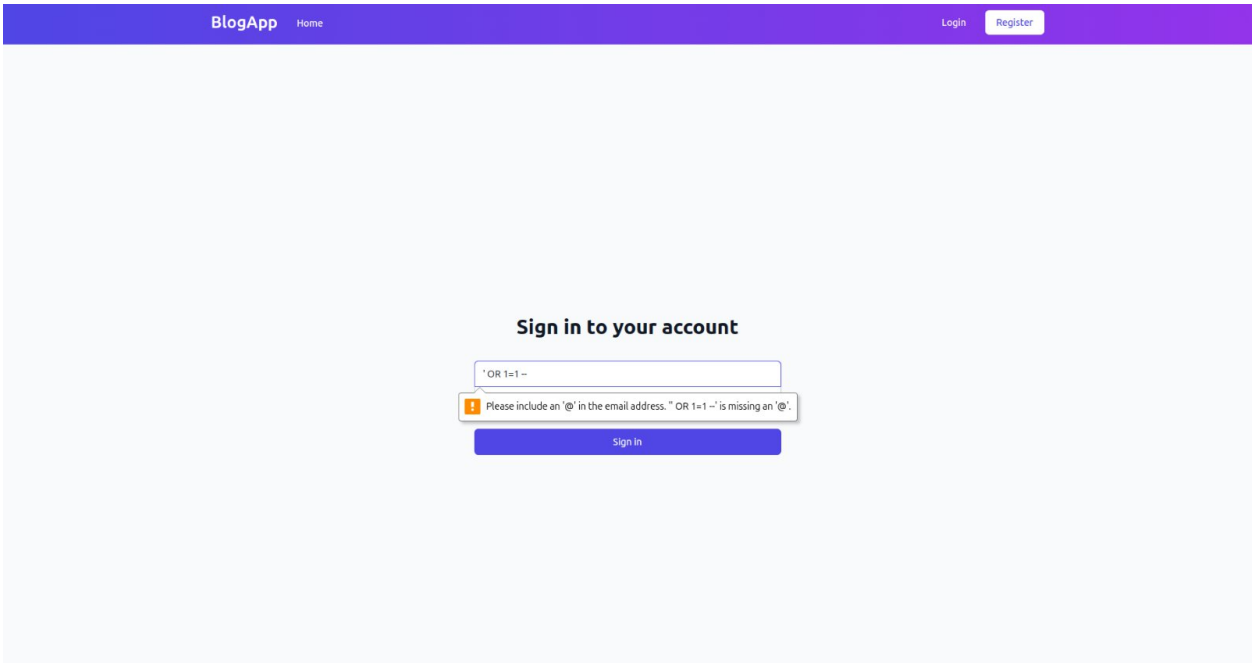
- Passwords are encrypted using bcryptjs.
- Data exchanges are secured using TLSv1.2 even when HTTP is used, ensuring data is encrypted.

Authentication and Token Security:

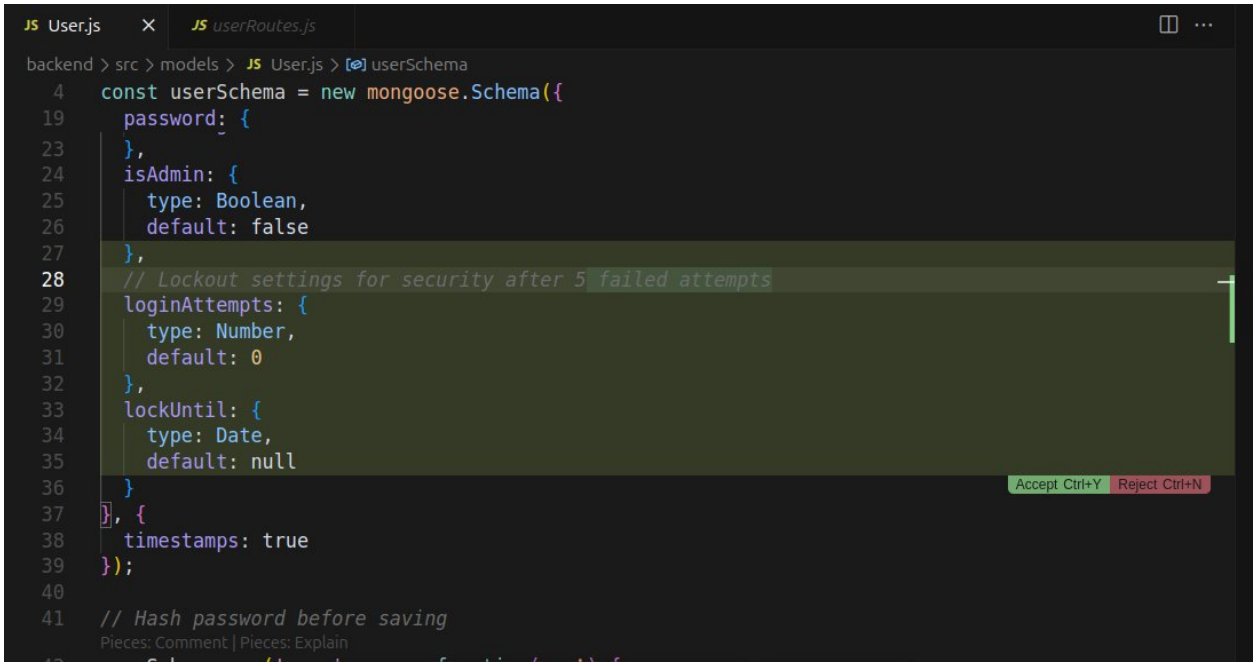
- JWT tokens are used to manage sessions. The token's payload includes an expiry timestamp (verified by decoding on jwt.io and checking via a Unix timestamp converter).
- The application enforces token validation to prevent token tampering; any modification of the token results in a failed authentication.

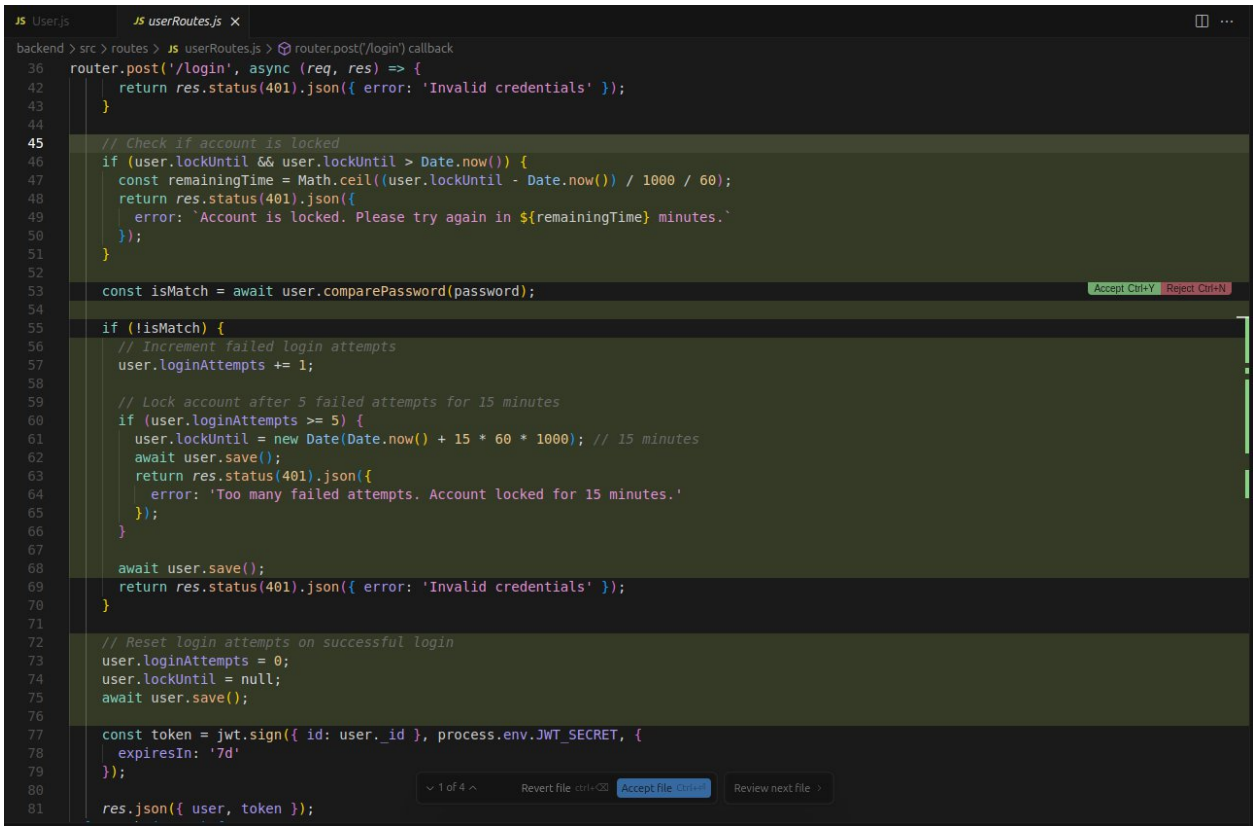
Adherence to Best Practices Using OWASP Top 10

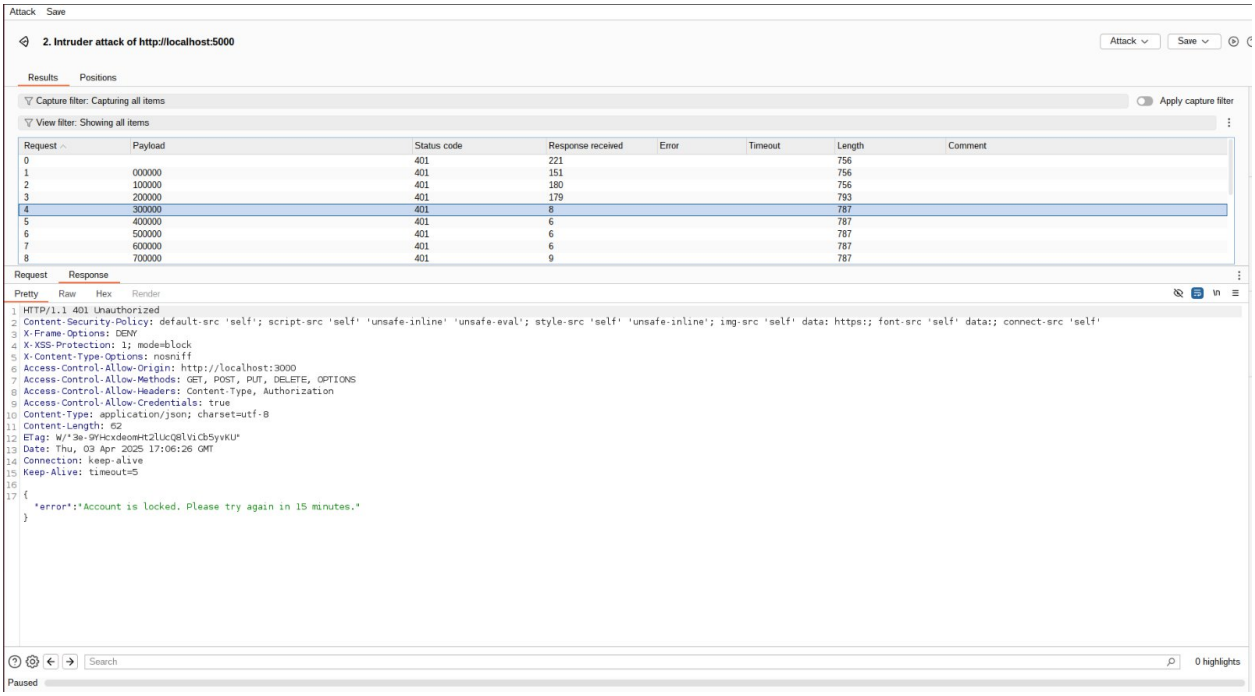
1. Injection Attacks: The use of a NoSQL database (MongoDB) reduces traditional SQL injection risks. Input is validated and sanitized even though the risk is less prominent with NoSQL databases.

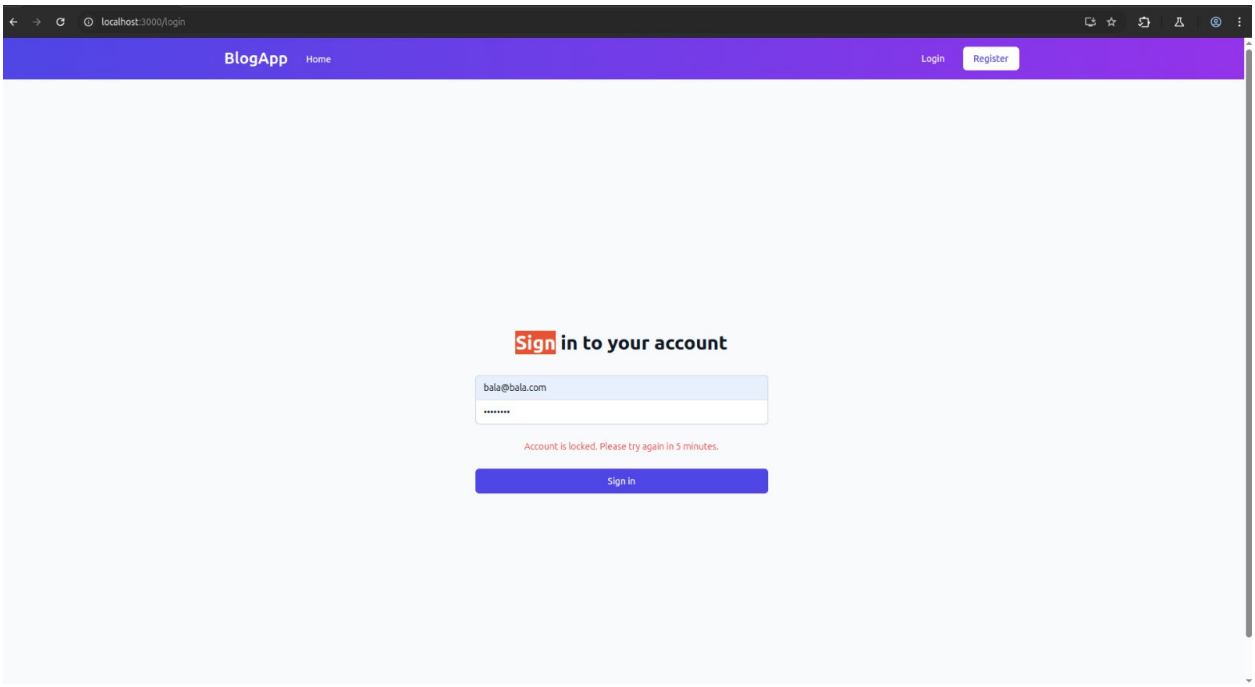


2. Broken Authentication: To mitigate brute force attacks, the login system enforces a maximum of five unsuccessful attempts, locking the account for 15 minutes upon reaching this limit. Code snippets and BurpSuite screenshots below document this implementation.

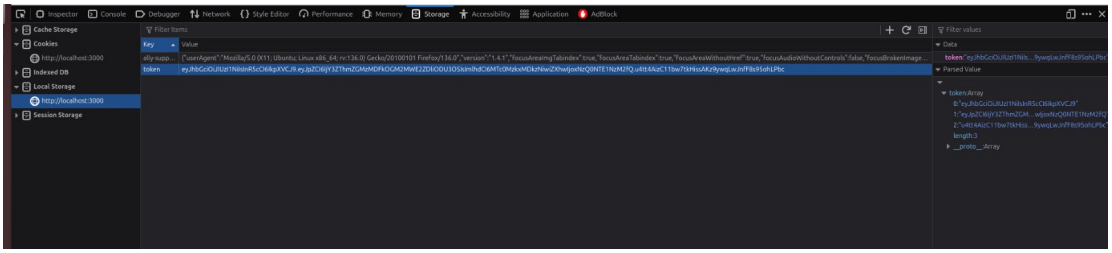

User.js
userRoutes.js
backend > src > models > User.js > userSchema
const userSchema = new mongoose.Schema({
password: {
},
isAdmin: {
type: Boolean,
default: false
},
// Lockout settings for security after 5 failed attempts
loginAttempts: {
type: Number,
default: 0
},
lockUntil: {
type: Date,
default: null
}
}, {
timestamps: true
});
// Hash password before saving
Pieces: Comment | Pieces: Explain
Accept Ctrl+Y
Reject Ctrl+N


User.js
userRoutes.js
backend > src > routes > userRoutes.js > router.post('/login') callback
router.post('/login', async (req, res) => {
return res.status(401).json({ error: 'Invalid credentials' });
}
// Check if account is locked
if (user.lockUntil && user.lockUntil > Date.now()) {
const remainingTime = Math.ceil((user.lockUntil - Date.now()) / 1000 / 60);
return res.status(401).json({
error: `Account is locked. Please try again in ${remainingTime} minutes.`
});
}
const isMatch = await user.comparePassword(password);
Accept Ctrl+Y
Reject Ctrl+N
if (!isMatch) {
// Increment failed login attempts
user.loginAttempts += 1;
// Lock account after 5 failed attempts for 15 minutes
if (user.loginAttempts >= 5) {
user.lockUntil = new Date(Date.now() + 15 * 60 * 1000); // 15 minutes
await user.save();
return res.status(401).json({
error: 'Too many failed attempts. Account locked for 15 minutes.'
});
}
await user.save();
return res.status(401).json({ error: 'Invalid credentials' });
}
// Reset login attempts on successful login
user.loginAttempts = 0;
user.lockUntil = null;
await user.save();
const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {
expiresIn: '7d'
});
1 of 4
Revert file
Accept file
Review next file
res.json({ user, token });

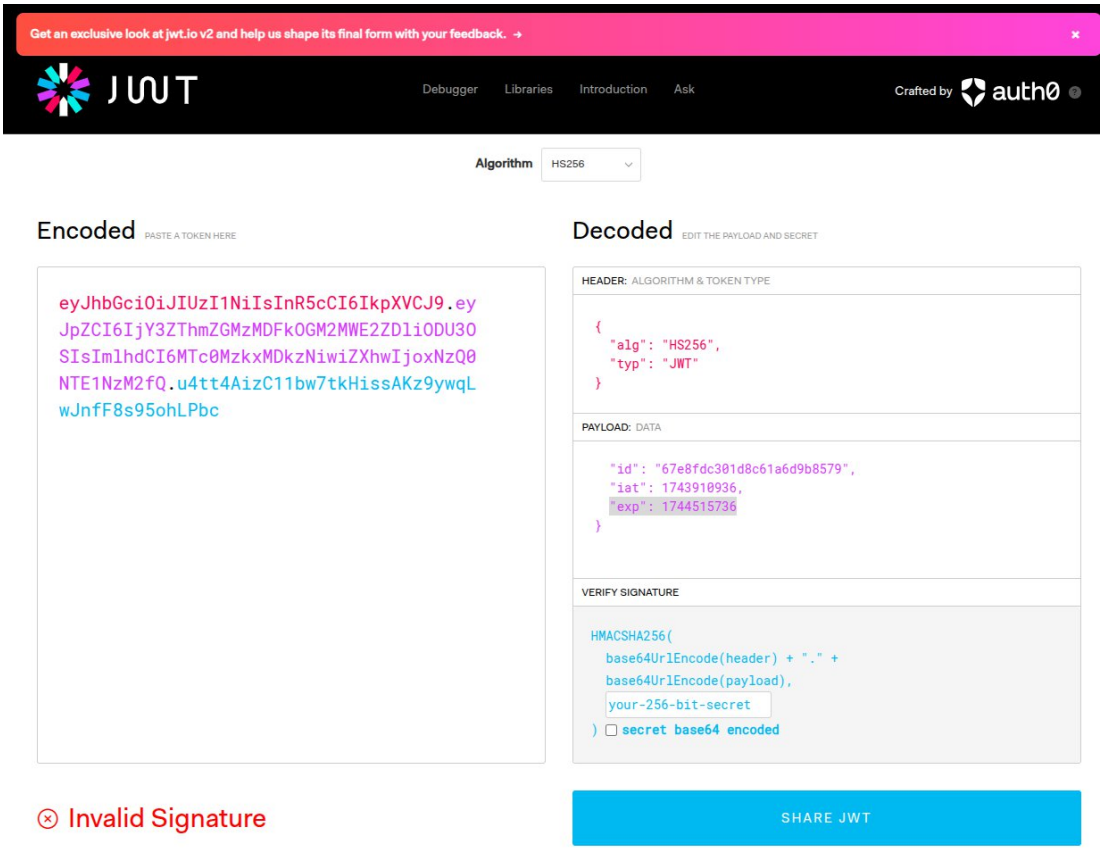




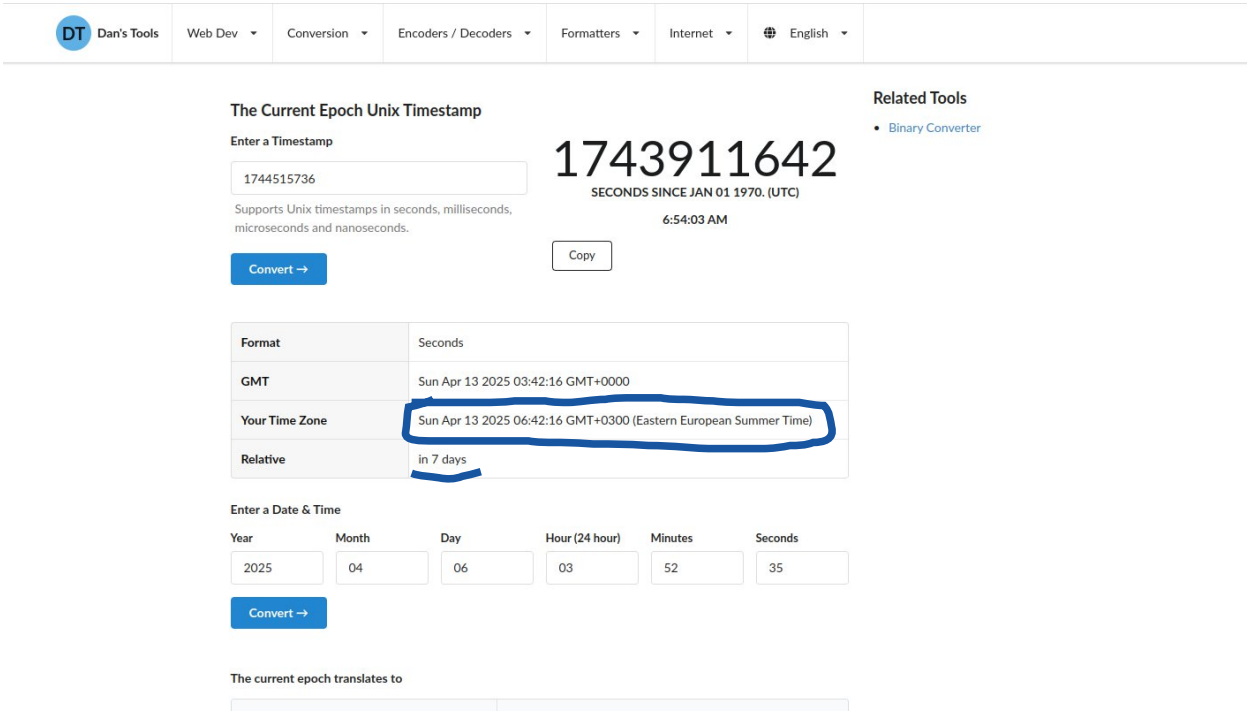
3. Token Validation: JWT tokens are used to manage sessions. The token's payload includes an expiry timestamp (verified by decoding on jwt.io and checking via a Unix timestamp converter). Below attached is the screenshot for the same.
login and from local storage (in console) copy the JWT code for the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



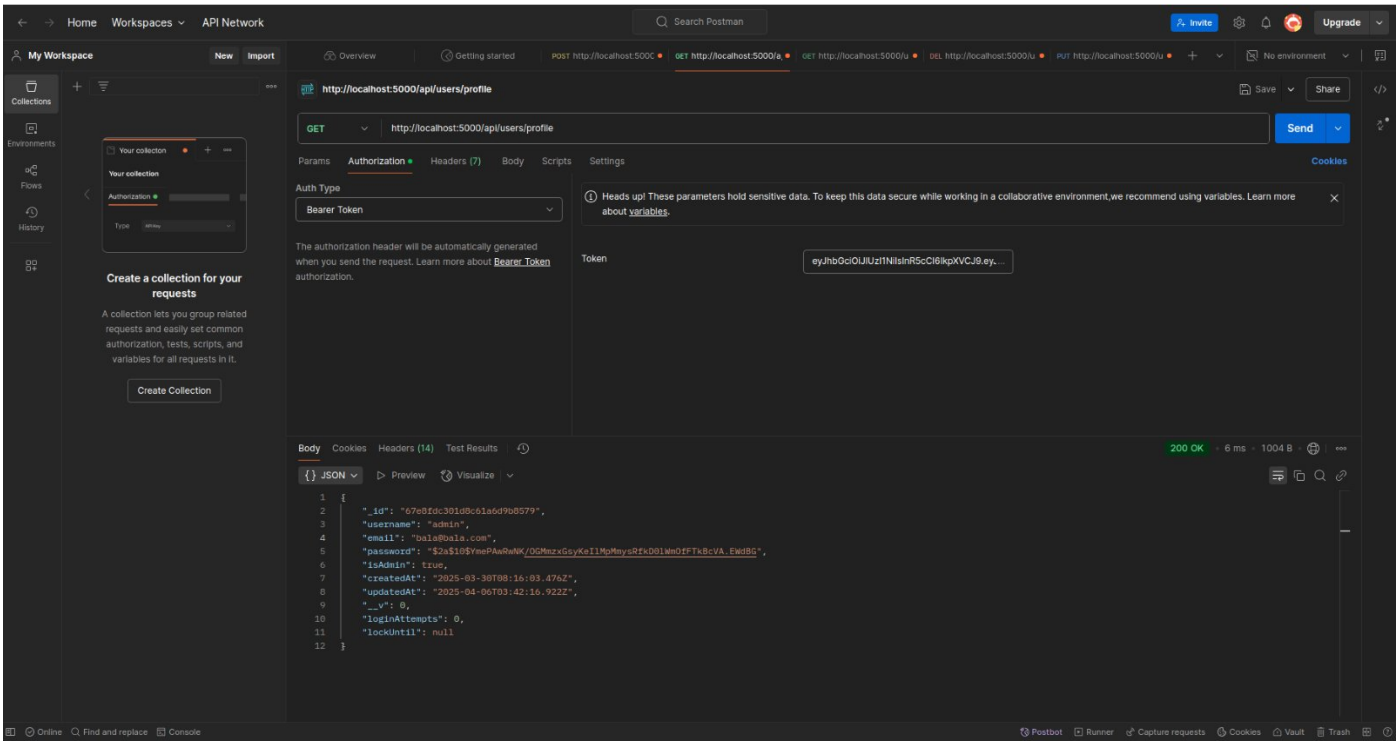
Copy this token and paste into jwt.io to decode, under “exp” you will get unix time stamp for token expiry.



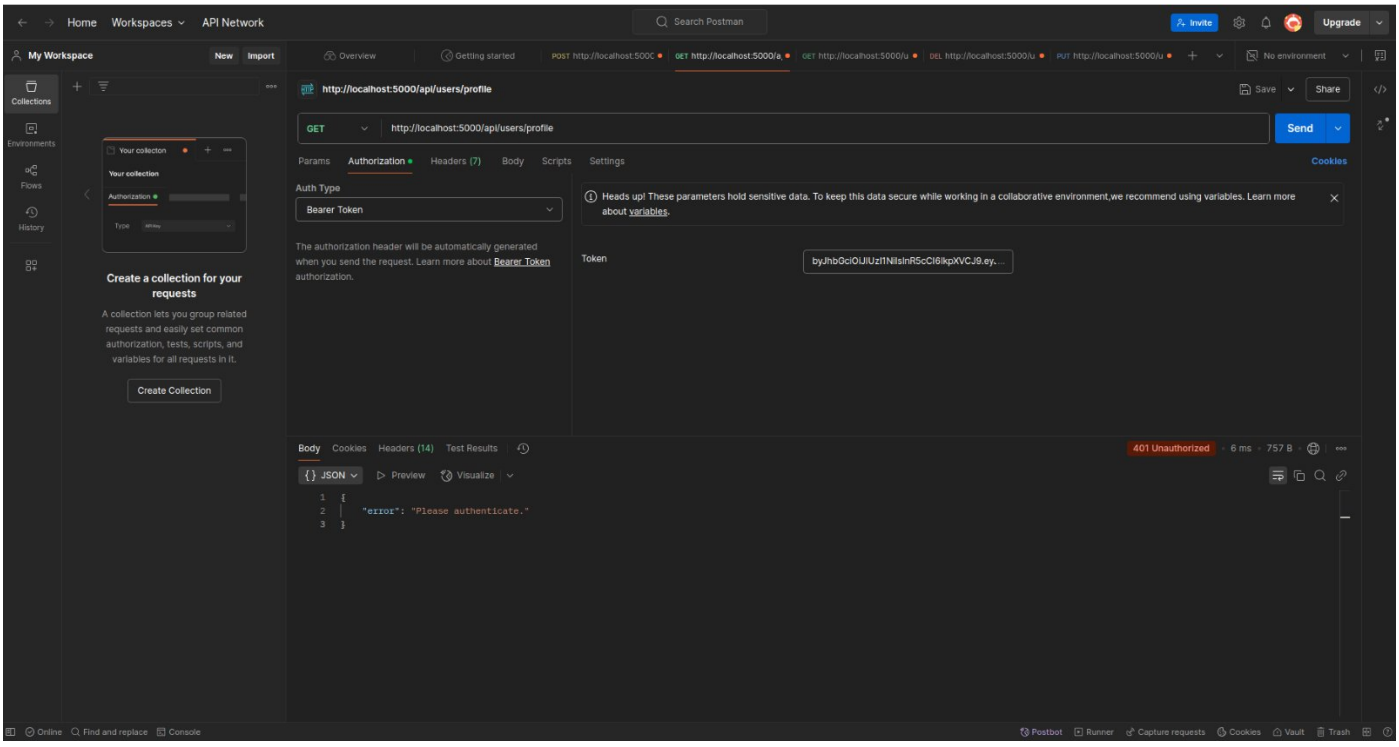
Copy the value of “exp” which is “1744515736” in our case and paste it on https://www.unixtimestamp.com/ to get the expiry of the token



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.



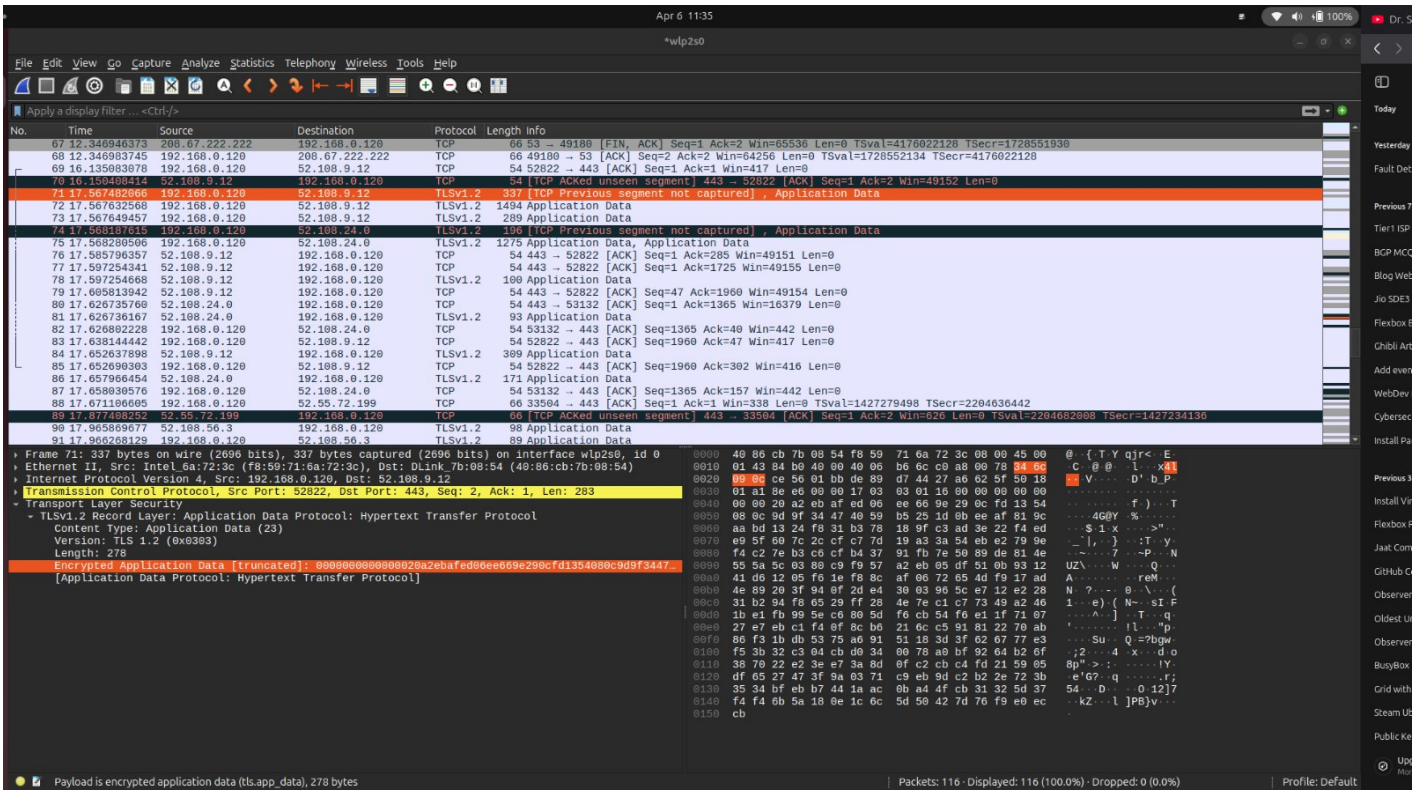
However, if we try to tamper the token and try to login it fails.



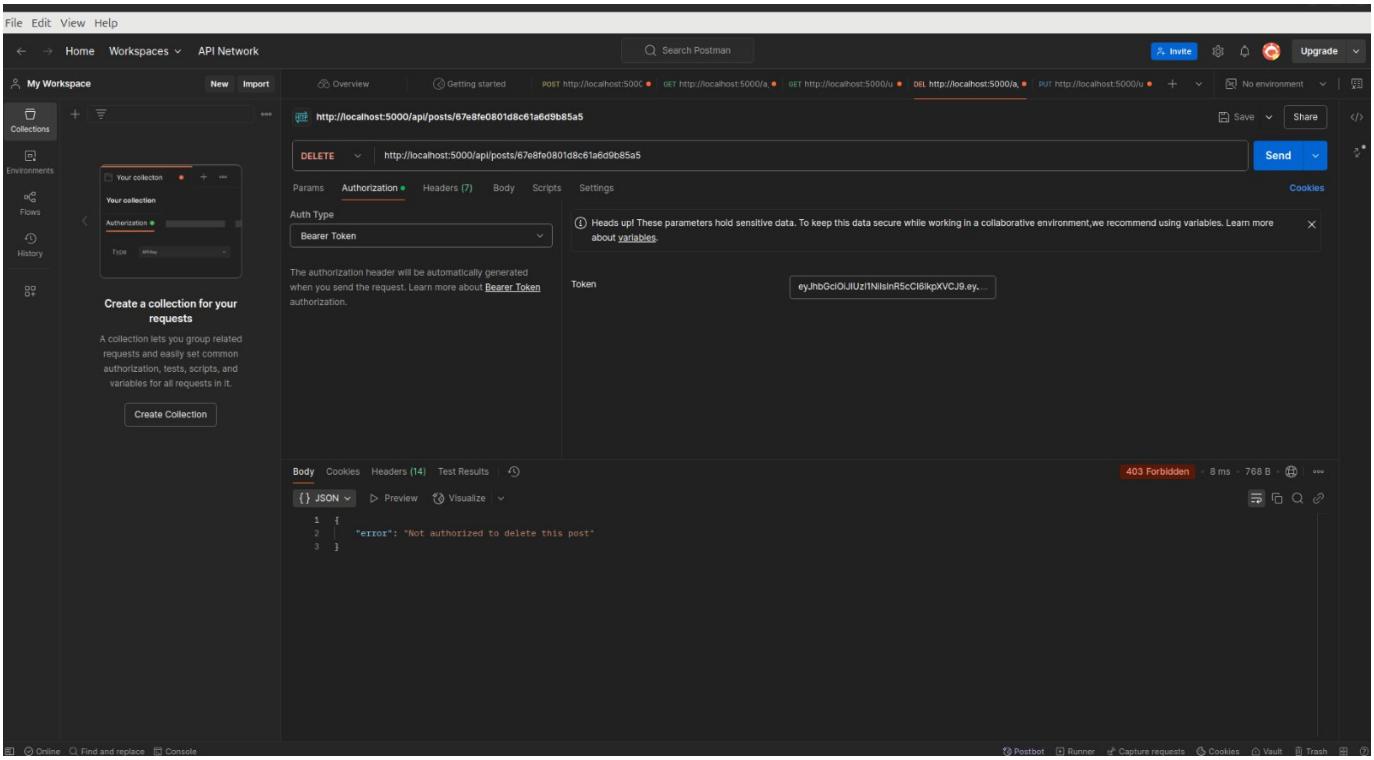
code for JWT creation and validation is below:

```
/**
 * Authentication middleware to verify JWT tokens and user authentication
 * This middleware is used to protect routes that require authentication
 */
const jwt = require('jsonwebtoken');
const User = require('../models/User');

/**
 * Middleware to verify JWT token and authenticate user
 * @param {Object} req - Express request object
 * @param {Object} res - Express response object
 * @param {Function} next - Express next middleware function
 */
const auth = async (req, res, next) => {
  try {
    // Extract token from Authorization header
    const token = req.header('Authorization')?.replace('Bearer ', '');

    // Check if token exists
    if (!token) {
      throw new Error();
    }

    // Verify token and decode payload
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    // Find user by ID from token payload
    const user = await User.findOne({ _id: decoded.id });

    // Check if user exists
    if (!user) {
      throw new Error();
    }

    // Attach user and token to request object for use in routes
    req.user = user;
    req.token = token;
    next();
  } catch (error) {
    // Return 401 Unauthorized if authentication fails
    res.status(401).json({ error: 'Please authenticate.' });
  }
};

```

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

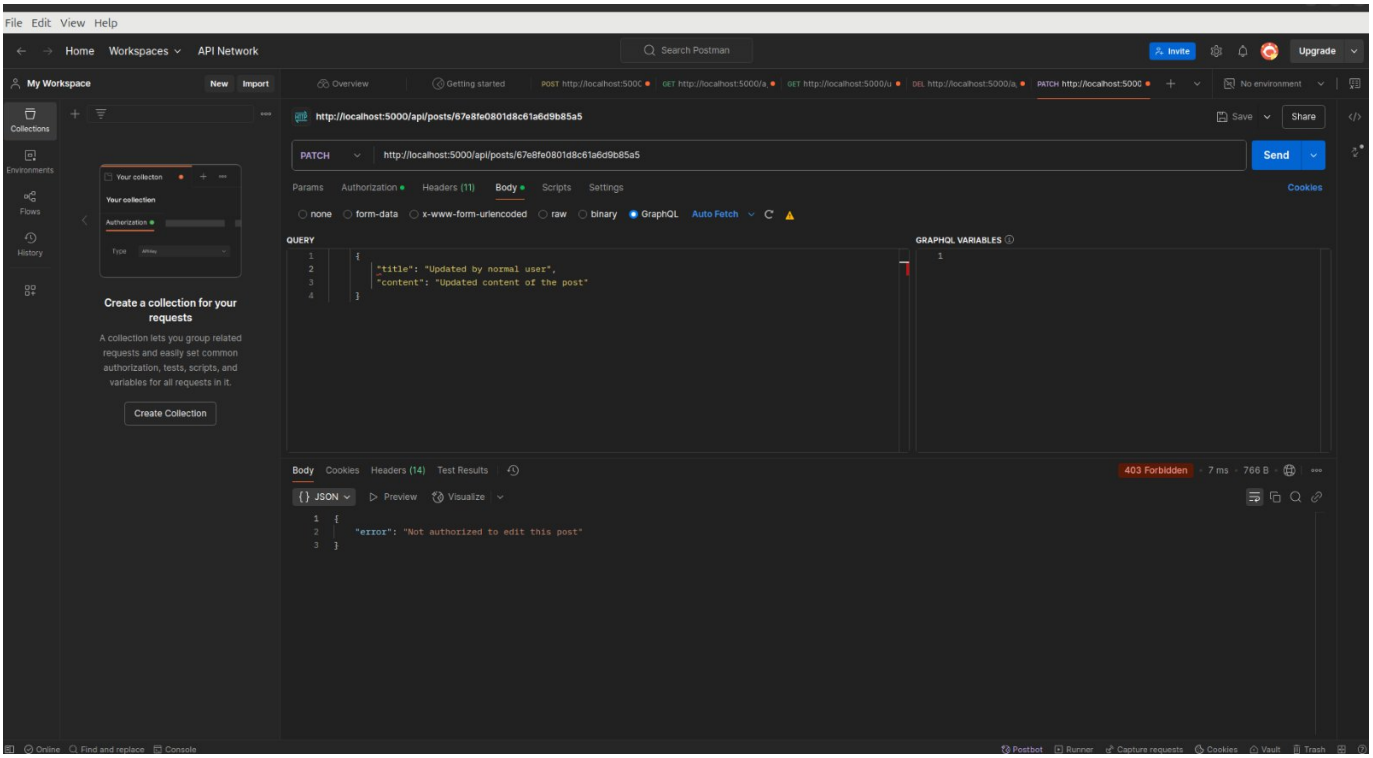


6. Broken Access Control: Role-based access control is implemented to ensure that normal users cannot modify or delete administrative posts. Attempts to modify privileged content results in an error response. However, admin having privileged rights can edit and delete any users post.
In postman hit (Delete) http://localhost:5000/api/posts/<admins-Post-id>
with Authorization: Bearer <JWT_token from prev steps>

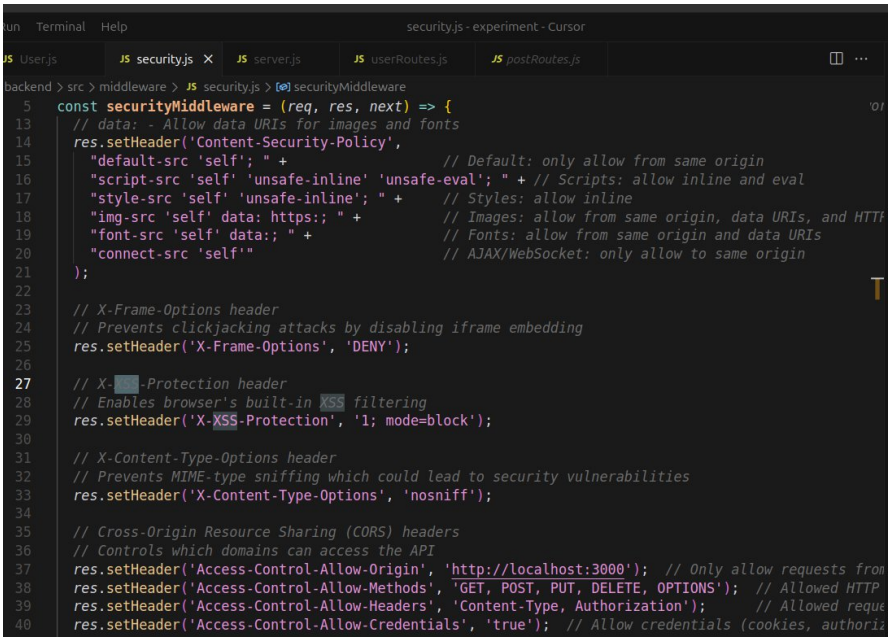


Thus, as a user we can't delete the admin post as mentioned in JSON ("error": "Not authorized to delete this post").

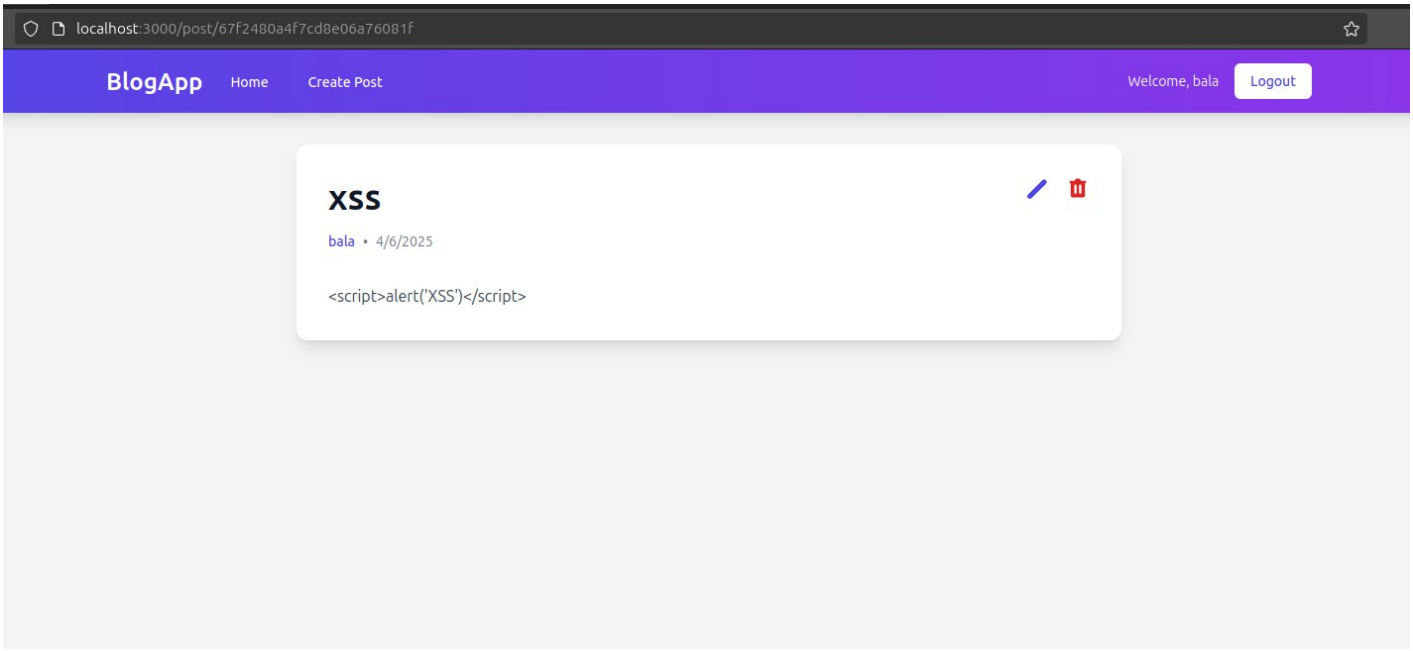
Similarly, we can try to edit an admins post as a normal user and evident in Screenshot below it throws error.



7. 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.



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.*