Secure Software Development

SE4030

Assignment

B.Sc. (Hons) Degree in Information Technology

Specialized in Software Engineering

Department of Computer Science and Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2025

GROUP MEMBERS

Name	Index Number	Email
Sandeepa K.B.A.R.	IT21809088	it21809088@my.sliit.lk
Weerasinghe C.D.	IT19211688	it19211688@my.sliit.lk
Jayawardhana A.M.S.P.	IT21838002	it21838002@my.sliit.lk

Table of Contents

Table of Tables

1. INTRODUCTION

Our group selected an existing MERN stack web application – a blog site integrated with the NASA Open API as the basis for this project. The application was originally developed as a private assignment project by Jayawardhana A.M.S.P. during Year 3 of his studies.

The original repository can be found here:

- Old Repo – SandunJay/NASA-Open-API-with-Blog (last commit: May 6, 2024)

For the purpose of this assignment, we forked and maintained an updated version:

- Modified Repo – RadithSandeepa/NASA-Open-API-with-Blog

The system is built using the **MERN stack (MongoDB, Express, React, Node.js)** and follows a monorepo structure:

Frontend (Vite + React SPA):

- src/components/pages/ → route views
- components/layout/ → layout chrome
- components/miniComponents/ → reusable UI
- src/utils/constants.js → shared config

Backend (Express + Mongoose API):

- routes/ → HTTP entry points (e.g., authRouter.js for social sign-in)
- controllers/ → business logic
- models/ → Mongoose schemas
- middlewares/ → auth helpers
- utils/ → cross-cutting helpers (jwtToken.js)
- database/ → MongoDB bootstrap

The application integrates **NASA's Open API** to fetch and display space-related data alongside blog functionality. This made it a good candidate for our work, since it included:

- **User authentication and roles**
- **Forms and content submission (blogging)**
- **File uploads and media handling**
- **API calls (NASA API)**

- **Database access (MongoDB)**

All of these features provided a realistic scope to identify and demonstrate vulnerabilities, apply fixes, and extend the system with **OAuth/OpenID Connect authentication**.

2. VULNERABILITIES

2.1. Vulnerabilities Identified

To identify weaknesses in the forked MERN blog application we performed automated and manual security testing focused on OWASP Top 10 risks. The primary automated tool used was **OWASP ZAP** (Zed Attack Proxy) to scan HTTP endpoints, discover insecure responses, and flag common issues (sensitive information in responses, auth/authorization problems, missing cookie flags, unsafe redirects, etc.). ZAP findings were augmented with **manual code review** of backend controllers, routers and configuration files (looking at controllers/, routes/, .env, dbConnection.js, and upload/config code), and with targeted tests using curl/Postman to reproduce issues. Where applicable, we inspected the frontend bundle and source for leaks (e.g., hard-coded API keys).

Below is the set of vulnerabilities we discovered (mapped to OWASP categories), reproducible steps, suggested fixes, and priorities. These entries form the core of the report's findings and are used later to show the fixes we implemented.

Table 1: Vulnerabilities Identified in the Application

ID	Title	OWASP Category	Files Affected	Why It's a Risk
F1	Production Secrets Committed to Repo	A02: Cryptographic Failures	backend/.env:1,5,9	Environment file in source control exposes DB creds, JWT secret, Cloudinary API key, NASA API key. Anyone with repo access can compromise systems.
F2	Blog Update/Delete Missing	A01: Broken Access Control	blogController.js:235,277	Any Author can update/delete any blog, no ownership check.

	Ownership Checks			
F3	JWT Returned in Response Body	A07: Identification & Authentication Failures	jwtToken.js:9	Login/register APIs return JWT in JSON body in addition to httpOnly cookie. XSS or third-party scripts can steal it.
F4	Public Authors Endpoint Leaks Contact Data	A01: Broken Access Control	userRouter.js:18, userController.js:64	/user/authors is public and returns phone + sensitive fields of authors → PII leakage.
F5	Draft Blogs Accessible to Any User	A01: Broken Access Control	blogController.js:256	Any logged-in user can fetch drafts by ID (no published or ownership check).
F6	Session Cookie Lacks Secure Attributes	A05: Security Misconfiguration	jwtToken.js:3	JWT cookie missing Secure and SameSite=None. May be sent over HTTP or blocked in cross-site contexts.
F7	File Upload Missing Size Limits	A05: Security Misconfiguration	backend/app.js:21	express-fileupload enabled without size/number limits. Attackers can upload huge

				files → disk/memory exhaustion.
F8	DB Connection Errors Leak Credentials	A05: Security Misconfiguration	dbConnection.js:8	Failed MongoDB connection logs full URI (with creds) to console → sensitive info leaks to logs/observability.
F9	NASA API Key in Client Bundle	A02: Cryptographic Failures	HeroSection.jsx:12	NASA API key hard-coded in frontend → visible to everyone, can be abused to exhaust quotas.

2.2. Notes on evidence & reproduction

- Automated ZAP scans produced alerts that pointed us to endpoints and responses leaking data (e.g., `.env` exposure flagged after scanning repository-linked demo and examining responses). We verified each finding with manual testing (curl/Postman) and by inspecting the code in the forked repo.
- For each high-priority finding we recorded: the exact file(s) that contain the issue, a short reproduction step (curl command or inspection step), and a suggested code/config change that directly removes the vulnerability. Screenshots, request/response logs, and code diffs are captured and included in the report appendix as concrete evidence.

2.3. Next steps

- The following section of the report documents the **fixes we implemented** for the high-priority issues (F1–F5 and F3/F6 access controls, XSS/content sanitization, file upload validation, cookie hardening).

- For medium-priority items, we document which were fixed, and for any not fixed we include a justification (time/complexity/third-party dependency) and mitigation recommendations (e.g., rotate keys, move secrets to a secret manager, enable Cloudinary signed uploads).

3. INDIVIDUAL CONTRIBUTIONS

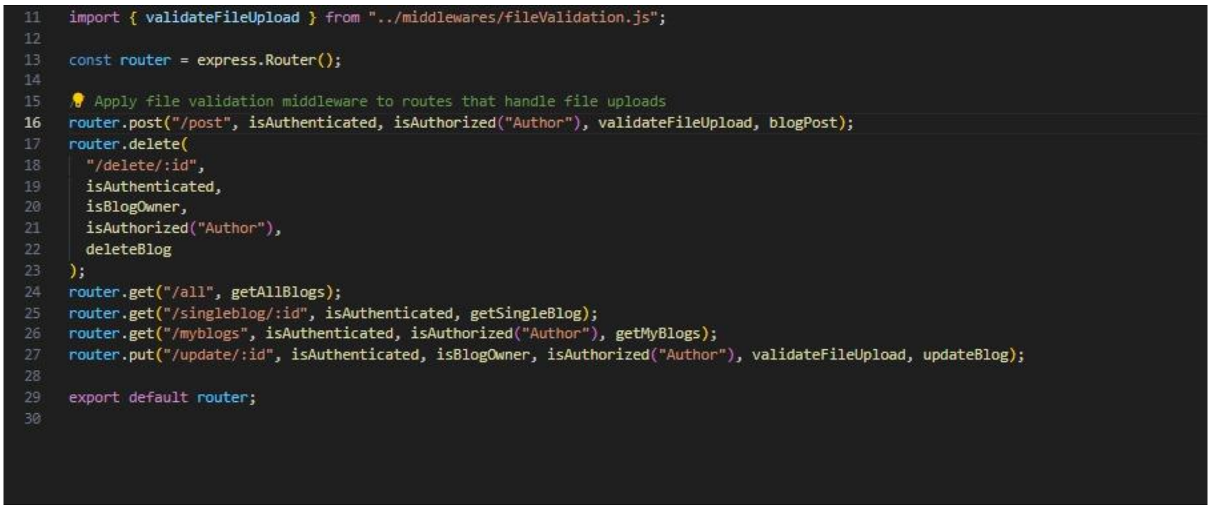
In this section, we outline each group member's efforts in identifying, fixing, and securing the application. Each member was responsible for specific vulnerabilities and improvements, ensuring that the project adheres to secure software engineering practices. The following details focus on the work carried out to patch critical issues, enforce proper access control, and enhance input validation.

3.1. IT21809088

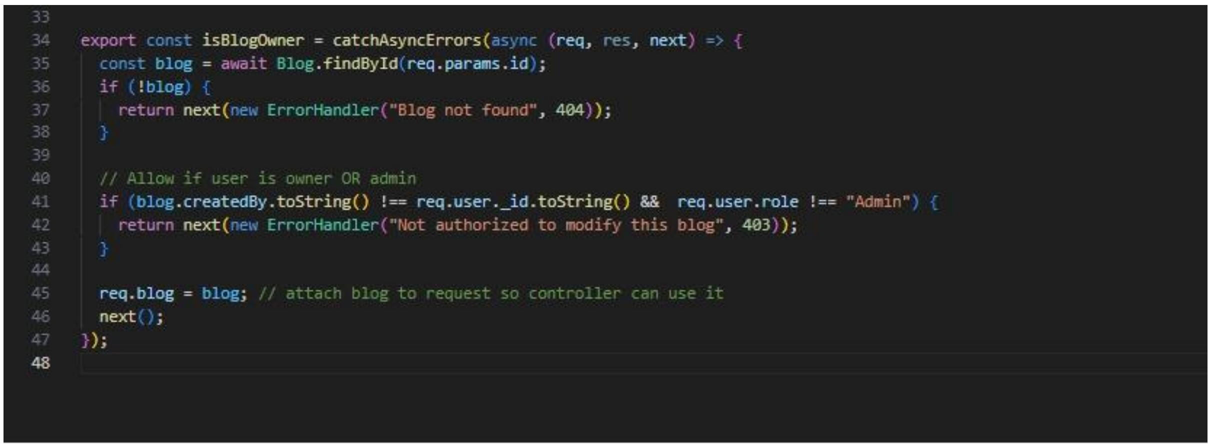
For my contributions, I primarily focused on strengthening the blog module's security by addressing access control weaknesses and improving input validation. The goal was to ensure that users can only perform actions they are authorized for and to prevent potential attacks such as XSS. Below is a detailed breakdown of the vulnerabilities I fixed and the measures I implemented.

F2 – Blog Update/Delete Missing Ownership Checks

OWASP Category: A01: Broken Access Control
Files Affected: blogController.js:235,277
Issue: Any Author could update or delete any blog, regardless of ownership.
What I Did: To fix this, I implemented a middleware function isBlogOwner that ensures only the blog creator or an Admin can update or delete a blog:I then applied this middleware to all relevant blog update and delete routes, ensuring that only authorized users can perform these actions. This effectively enforces ownership checks and prevents unauthorized modifications.



```
import { validateFileUpload } from "../middlewares/fileValidation.js";

const router = express.Router();

 Apply file validation middleware to routes that handle file uploads
router.post("/post", isAuthenticated, isAuthorized("Author"), validateFileUpload, blogPost);
router.delete(
  "/delete/:id",
  isAuthenticated,
  isBlogOwner,
  isAuthorized("Author"),
  deleteBlog
);
router.get("/all", getAllBlogs);
router.get("/singleblog/:id", isAuthenticated, getSingleBlog);
router.get("/myblogs", isAuthenticated, isAuthorized("Author"), getMyBlogs);
router.put("/update/:id", isAuthenticated, isBlogOwner, isAuthorized("Author"), validateFileUpload, updateBlog);

export default router;
```



```
export const isBlogOwner = catchAsyncErrors(async (req, res, next) => {
  const blog = await Blog.findById(req.params.id);
  if (!blog) {
    return next(new ErrorHandler("Blog not found", 404));
  }

  // Allow if user is owner OR admin
  if (blog.createdBy.toString() !== req.user._id.toString() &&  req.user.role !== "Admin") {
    return next(new ErrorHandler("Not authorized to modify this blog", 403));
  }

  req.blog = blog; // attach blog to request so controller can use it
  next();
});
```

F5 – Draft Blogs Accessible to Any User

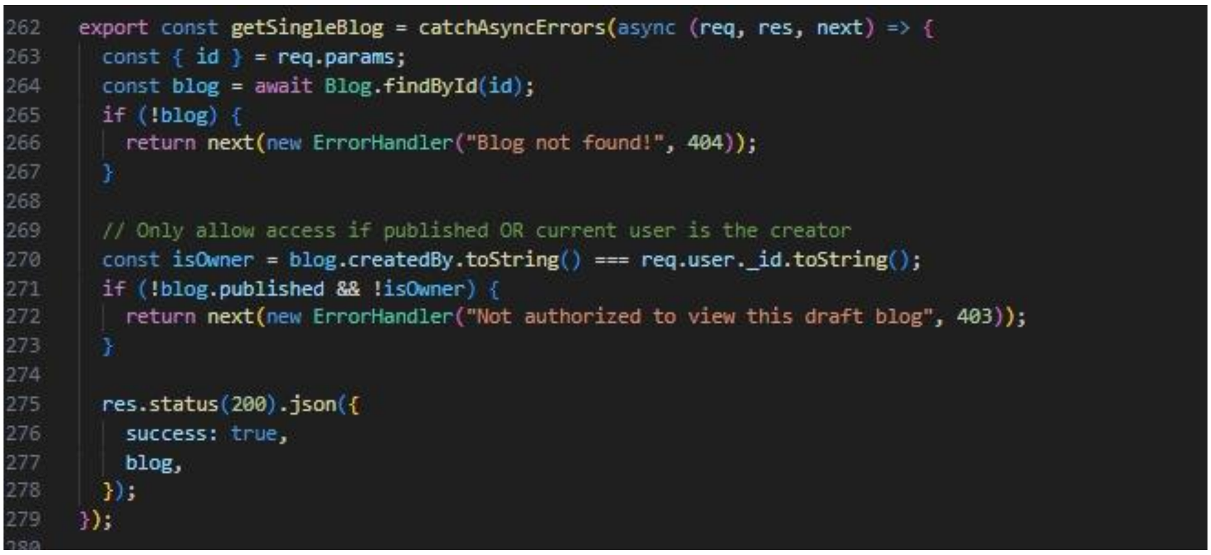
OWASP Category: A01: Broken Access Control

Files Affected: blogController.js:256

Issue: Any logged-in user could fetch draft blogs by ID, with no published or ownership checks.

What I Did: To fix this, I added a check in the blog retrieval logic to restrict access to drafts only to the blog creator or authorized moderators. The logic verifies if the blog is published or if the requesting user is the owner:

This ensures that unpublished blogs cannot be accessed by unauthorized users, protecting draft content from being exposed.



```
export const getSingleBlog = catchAsyncErrors(async (req, res, next) => {
  const { id } = req.params;
  const blog = await Blog.findById(id);
  if (!blog) {
    return next(new ErrorHandler("Blog not found!", 404));
  }

  // Only allow access if published OR current user is the creator
  const isOwner = blog.createdBy.toString() === req.user._id.toString();
  if (!blog.published && !isOwner) {
    return next(new ErrorHandler("Not authorized to view this draft blog", 403));
  }

  res.status(200).json({
    success: true,
    blog,
  });
});
```

Blog Content Validation – Preventing XSS

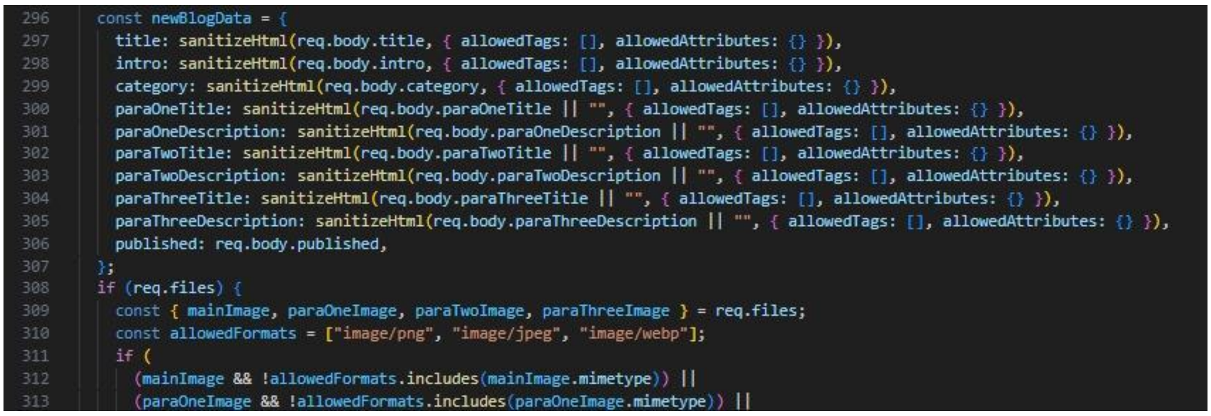
OWASP Category: A03: Injection (Cross-Site Scripting)

Files Affected: blogController.js (create/update blog functions)

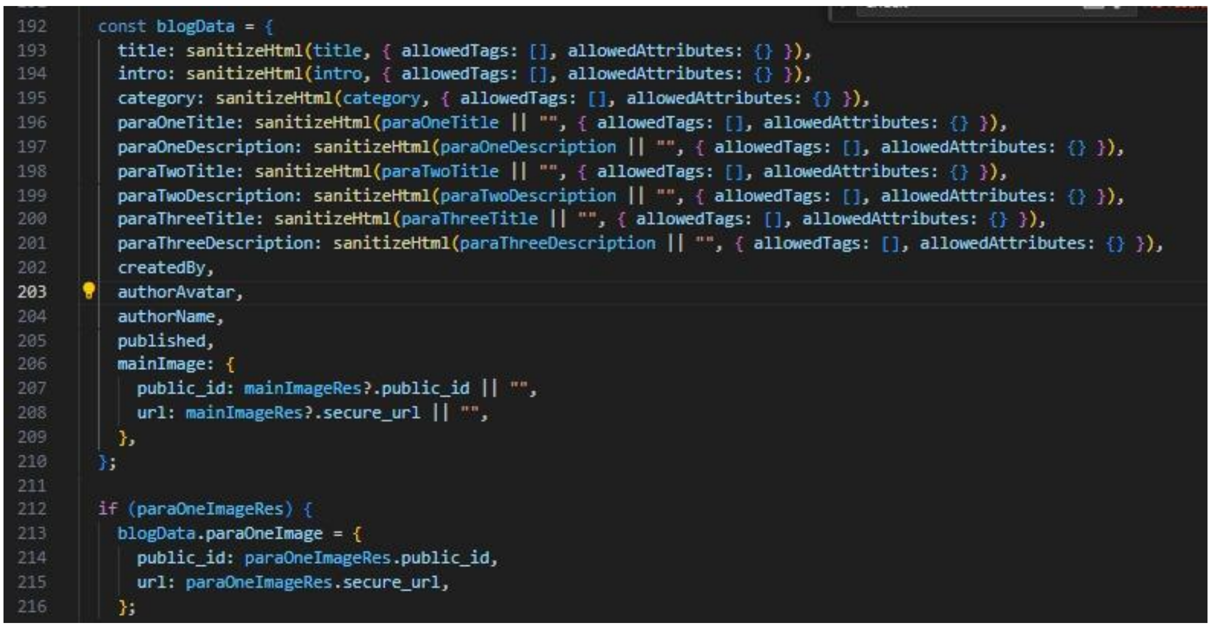
Issue: User inputs for blog content (titles, paragraphs, descriptions) were not sanitized. Malicious scripts could be injected, leading to XSS attacks.

What I Did: To mitigate this, I used the sanitize-html library to clean all input fields during blog creation and updates. This removes any HTML tags and potentially harmful scripts, ensuring only plain text is stored and displayed:

This ensures all user-generated content is safe for display, effectively preventing XSS attacks on the blog site.



```
const newBlogData = {
  title: sanitizeHtml(req.body.title, { allowedTags: [], allowedAttributes: {} }),
  intro: sanitizeHtml(req.body.intro, { allowedTags: [], allowedAttributes: {} }),
  category: sanitizeHtml(req.body.category, { allowedTags: [], allowedAttributes: {} }),
  paraOneTitle: sanitizeHtml(req.body.paraOneTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraOneDescription: sanitizeHtml(req.body.paraOneDescription || "", { allowedTags: [], allowedAttributes: {} }),
  paraTwoTitle: sanitizeHtml(req.body.paraTwoTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraTwoDescription: sanitizeHtml(req.body.paraTwoDescription || "", { allowedTags: [], allowedAttributes: {} }),
  paraThreeTitle: sanitizeHtml(req.body.paraThreeTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraThreeDescription: sanitizeHtml(req.body.paraThreeDescription || "", { allowedTags: [], allowedAttributes: {} }),
  published: req.body.published,
};
if (req.files) {
  const { mainImage, paraOneImage, paraTwoImage, paraThreeImage } = req.files;
  const allowedFormats = ["image/png", "image/jpeg", "image/webp"];
  if (
    (mainImage && !allowedFormats.includes(mainImage.mimetype)) ||
    (paraOneImage && !allowedFormats.includes(paraOneImage.mimetype)) ||
```



```
const blogData = {
  title: sanitizeHtml(title, { allowedTags: [], allowedAttributes: {} }),
  intro: sanitizeHtml(intro, { allowedTags: [], allowedAttributes: {} }),
  category: sanitizeHtml(category, { allowedTags: [], allowedAttributes: {} }),
  paraOneTitle: sanitizeHtml(paraOneTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraOneDescription: sanitizeHtml(paraOneDescription || "", { allowedTags: [], allowedAttributes: {} }),
  paraTwoTitle: sanitizeHtml(paraTwoTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraTwoDescription: sanitizeHtml(paraTwoDescription || "", { allowedTags: [], allowedAttributes: {} }),
  paraThreeTitle: sanitizeHtml(paraThreeTitle || "", { allowedTags: [], allowedAttributes: {} }),
  paraThreeDescription: sanitizeHtml(paraThreeDescription || "", { allowedTags: [], allowedAttributes: {} }),
  createdBy,
  authorAvatar,
  authorName,
  published,
  mainImage: {
    public_id: mainImageRes?.public_id || "",
    url: mainImageRes?.secure_url || "",
  },
};

if (paraOneImageRes) {
  blogData.paraOneImage = {
    public_id: paraOneImageRes.public_id,
    url: paraOneImageRes.secure_url,
  };
```

3.2. IT21838002

Session Cookie (F7)

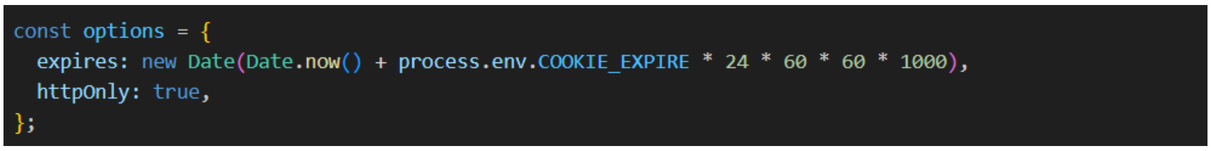
OWASP Category: A05 — Security Misconfiguration
Files Affected: backend/utils/jwtToken.js, backend/controllers/userController.js, .env, .env.example

Issue: JWT session cookies were issued with httpOnly only and lacked secure, sameSite, and explicit path settings. This left session tokens susceptible to being sent over HTTP, leaked via cross-site contexts or accessed by malicious script, increasing XSS, CSRF and session-hijacking risk.

What I did:

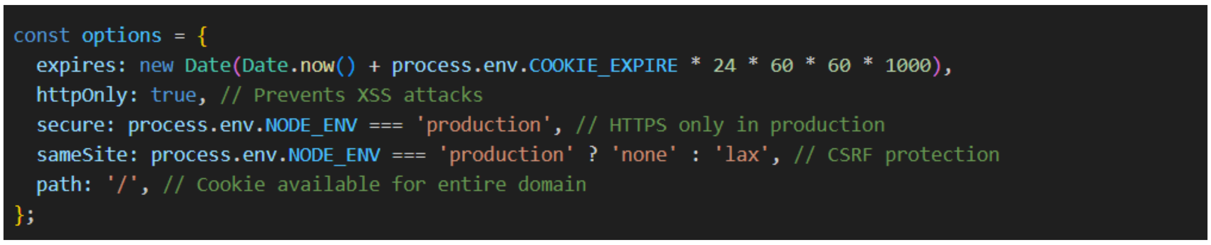
I updated the cookie configuration to be environment-aware and hardened by default. Previously the options were minimal (expires + httpOnly). After the change the cookie options include httpOnly, ensuring cookies are only sent over HTTPS in production and are scoped correctly. I also implemented a logout handler that clears the cookie using the same hardened options so tokens are reliably removed on sign-out, and added NODE_ENV to .env/.env.example so behavior differs safely between development and production. These changes reduce the chance of token theft via XSS, mitigate CSRF exposures, and ensure cookies are only transmitted over secure transport in production.

Before:



```
const options = {
  expires: new Date(Date.now() + process.env.COOKIE_EXPIRE * 24 * 60 * 60 * 1000),
  httpOnly: true,
};
```

After:



```
const options = {
  expires: new Date(Date.now() + process.env.COOKIE_EXPIRE * 24 * 60 * 60 * 1000),
  httpOnly: true, // Prevents XSS attacks
  secure: process.env.NODE_ENV === 'production', // HTTPS only in production
  sameSite: process.env.NODE_ENV === 'production' ? 'none' : 'lax', // CSRF protection
  path: '/', // Cookie available for entire domain
};
```

File Upload Limits (F8)

OWASP Category: A05 — Security Misconfiguration

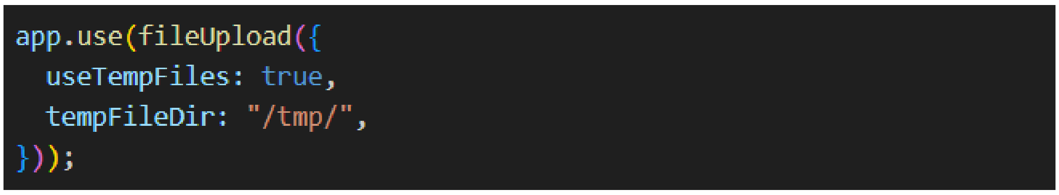
Files Affected: backend/app.js, middlewares/fileValidation.js, routes/blogRouter.js, .env, .env.example

Issue: express-fileupload was enabled without size/count/time limits, allowing attackers to attempt large or frequent uploads that could exhaust memory, disk or CPU and cause DoS.

What I did:

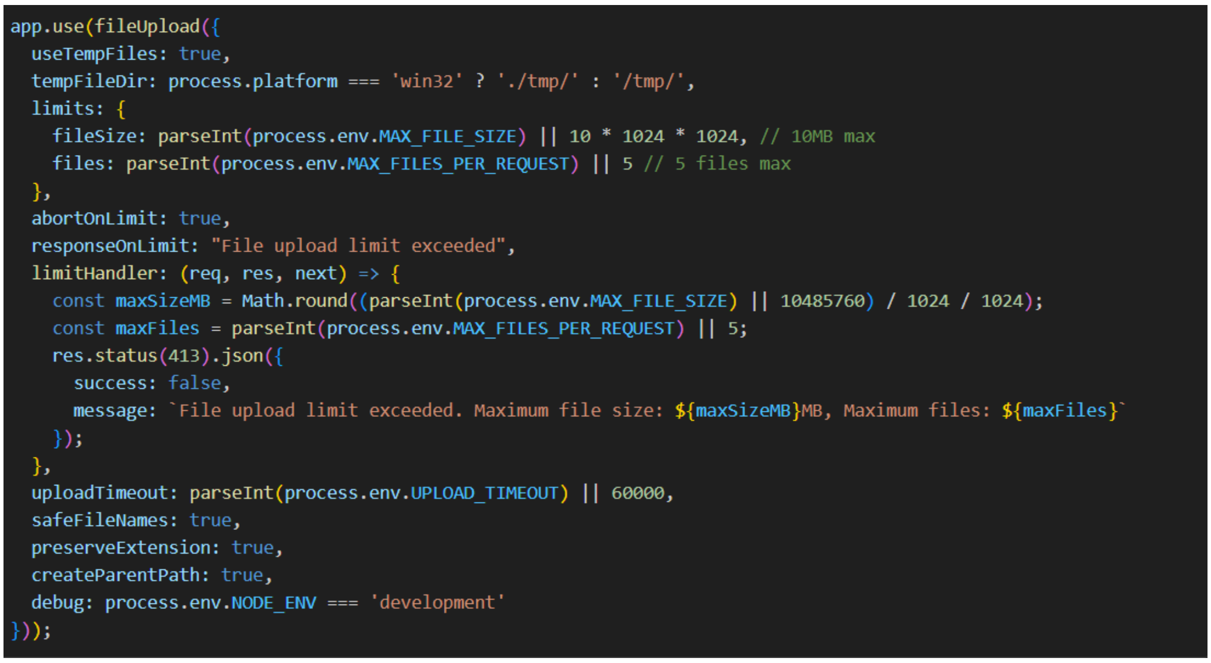
I replaced the unbounded upload setup with a hardened configuration that applies platform-aware temp directories and enforces configurable fileSize and files limits, uploadTimeout, abortOnLimit behavior and a custom limit handler that returns 413 with a clear JSON message. I created fileValidation.js middleware to validate MIME types, enforce safe filenames/extensions and prevent path traversal or executable/script uploads. Upload routes (e.g., blog post endpoints) now require authentication/authorization and run the validation middleware. I also introduced environment variables (MAX_FILE_SIZE, MAX_FILES_PER_REQUEST, UPLOAD_TIMEOUT) so upload constraints are tunable per environment. Together these controls prevent resource-exhaustion, reduce storage of malicious files, and make upload failures clear and predictable.

Before:



```
app.use(fileUpload({
  useTempFiles: true,
  tempFileDir: "/tmp/",
}));
```

After:



```
app.use(fileUpload({
  useTempFiles: true,
  tempFileDir: process.platform === 'win32' ? './tmp/' : '/tmp/',
  limits: {
    fileSize: parseInt(process.env.MAX_FILE_SIZE) || 10 * 1024 * 1024, // 10MB max
    files: parseInt(process.env.MAX_FILES_PER_REQUEST) || 5 // 5 files max
  },
  abortOnLimit: true,
  responseOnLimit: "File upload limit exceeded",
  limitHandler: (req, res, next) => {
    const maxSizeMB = Math.round((parseInt(process.env.MAX_FILE_SIZE) || 10485760) / 1024 / 1024);
    const maxFiles = parseInt(process.env.MAX_FILES_PER_REQUEST) || 5;
    res.status(413).json({
      success: false,
      message: `File upload limit exceeded. Maximum file size: ${maxSizeMB}MB, Maximum files: ${maxFiles}`
    });
  },
  uploadTimeout: parseInt(process.env.UPLOAD_TIMEOUT) || 60000,
  safeFileNames: true,
  preserveExtension: true,
  createParentPath: true,
  debug: process.env.NODE_ENV === 'development'
}));
```

I also added a fileValidation.js middleware to validate MIME types, enforce filename/extension checks and guard against path-traversal and executable uploads, applied upload-specific rate limiting, and tightened the blog routes to require authentication/authorization plus file validation. These measures prevent resource-exhaustion attacks and reduce the risk of storing malicious files.

Exposed NASA API Key (F10)

Files Affected: frontend/src/components/miniComponents/HeroSection.jsx, frontend/src/components/.../NEOFeed.jsx, frontend/src/components/.../Asteroid.jsx (and other frontend files)

Issue: A NASA API key was hard-coded into the client bundle, exposing it to anyone who inspects the frontend assets. This enables quota abuse, unauthorized usage, potential financial impact and credential compromise.

What I did:

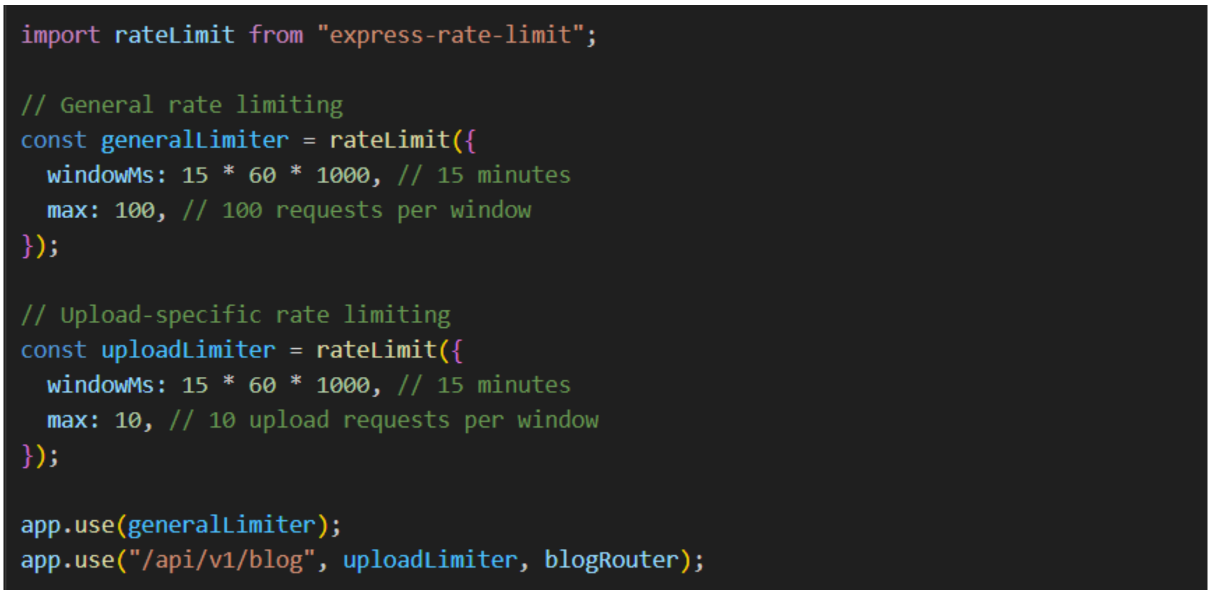
I documented and proposed a secure remediation: remove all API keys from client code, move the key to server-side environment variables and implement authenticated backend proxy endpoints that call NASA on behalf of the frontend. The proxy should include request validation, rate limiting and response caching to reduce abuse and cost. Frontend components must be refactored to call the backend proxy instead of calling NASA directly. Implementation is planned but pending backend proxy work and frontend refactor; the recommended architecture and required env vars have been added to the report for quick rollout.

Rate Limiting - F10

OWASP Category: A05 — Security Misconfiguration / Denial of Service mitigation
Files Affected: backend/app.js, routes/blogRouter.js, .env, .env.example
What I did:

I added IP-based rate limiting using express-rate-limit with two tiers: a global limiter and a stricter upload-specific limiter. The global limiter applies a default 15-minute window with a configurable maximum (default RATE_LIMIT_MAX = 100) for general request throttling; the upload limiter uses the same window but a lower default max (UPLOAD_RATE_LIMIT_MAX = 10) for endpoints that accept file uploads. Exceeded limits return HTTP 429 with a clear JSON payload and Retry-After semantics. Rate limiting runs early in the middleware stack (honors app.set('trust proxy', true)), logs exceeded-limit events for monitoring/alerts, and is configurable via env vars (RATE_LIMIT_WINDOW_MS, RATE_LIMIT_MAX, UPLOAD_RATE_LIMIT_MAX) so operations can tune thresholds or apply whitelists/temporary blocks as needed. This control complements file-size limits and validation to reduce the risk of brute-force, scraping and DoS attempts via repeated uploads or request floods.

Implementation



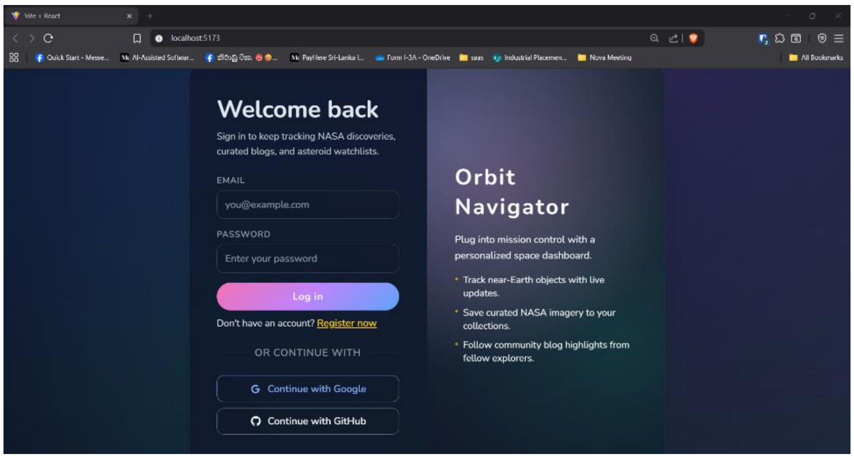
```
import rateLimit from "express-rate-limit";

// General rate limiting
const generalLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // 100 requests per window
});

// Upload-specific rate limiting
const uploadLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 10, // 10 upload requests per window
});

app.use(generalLimiter);
app.use("/api/v1/blog", uploadLimiter, blogRouter);
```

3.3. IT19211688

OAuth (Google & GitHub) via Passport

Additional contribution — OAuth (Google & GitHub) via Passport Implemented privacy-preserving social login with Passport strategies (Google OAuth 2.0 and GitHub). A unified upsertOAuthUser() links providers without duplicating accounts, prefers primary email when present, and normalizes profile names. On successful OAuth callback, attachTokenCookie() issues the session cookie and the client is redirected to /auth/callback with status=success&provider=.... Security considerations applied:

- No tokens are returned in bodies; session is cookie-based only.
- Providers are stored as { provider, providerId } entries; emails are optional and never required to link.
- Missing names/emails are handled safely; display names are derived without exposing provider secrets.



Welcome back

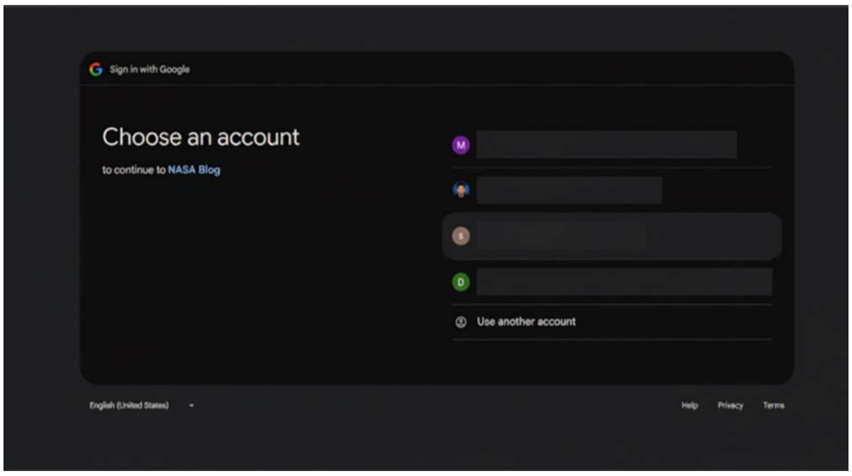
Sign in to keep tracking NASA discoveries, curated blogs, and asteroid watchlists.

Don't have an account? Register now

Orbit Navigator

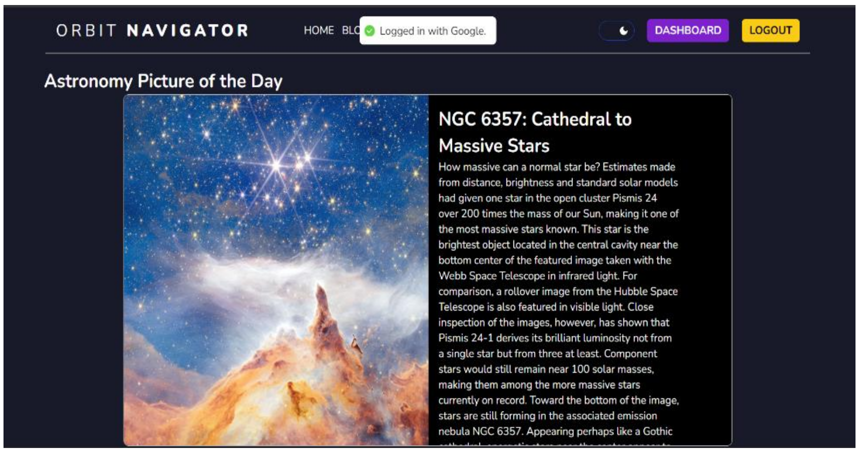
Plug into mission control with a personalized space dashboard.

- Track near-Earth objects with live updates.
- Save curated NASA imagery to your collections.
- Follow community blog highlights from fellow explorers.



Choose an account

to continue to NASA Blog



Astronomy Picture of the Day

NGC 6357: Cathedral to Massive Stars

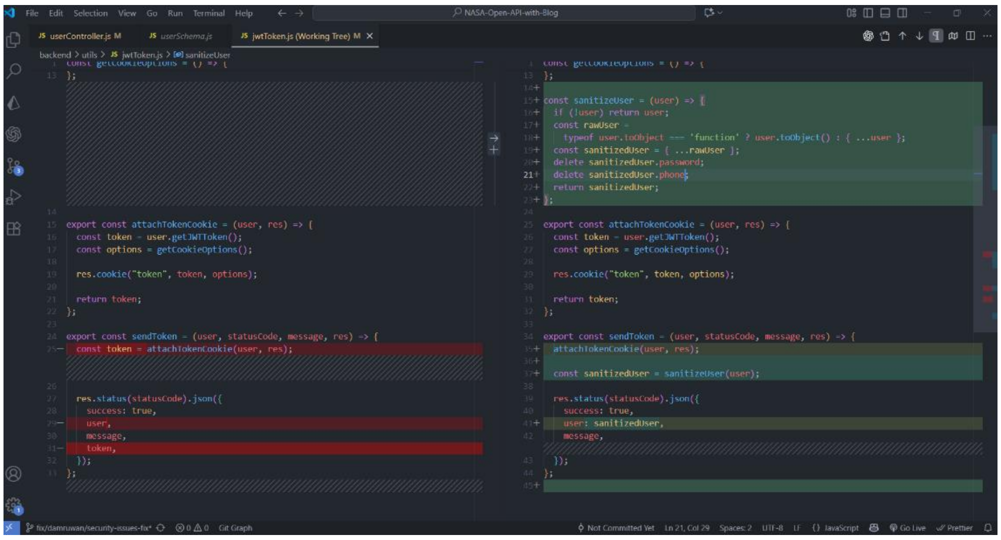
How massive can a normal star be? Estimates made from distance, brightness and standard solar models had given one star in the open cluster Pismis 24 over 200 times the mass of our Sun, making it one of the most massive stars known. This star is the brightest object located in the central cavity near the bottom center of the featured image taken with the Webb Space Telescope in infrared light. For comparison, a rollover image from the Hubble Space Telescope is also featured in visible light. Close inspection of the images, however, has shown that Pismis 24-1 derives its brilliant luminosity not from a single star but from three at least. Component stars would still remain near 100 solar masses, making them among the more massive stars currently on record. Toward the bottom of the image, stars are still forming in the associated emission nebula NGC 6357. Appearing perhaps like a Gothic

F3 - JWT Returned in Response Body

OWASP Category: A07: Identification & Authentication Failures

Issue: Authentication endpoints exposed the JWT in the JSON response as well as in the cookie, increasing theft risk via XSS or third-party scripts.

Fix: Refactored the token flow to use httpOnly cookies only. The helper now (a) attaches the JWT to res.cookie("token", ...) with hardened options; (b) does not include the token in the JSON body; and (c) returns a minimal payload. A sanitizeUser() helper strips sensitive fields (e.g., password, phone) before any user object is serialized. Evidence. See code diff: utils/jwtToken.js — sendToken() now calls attachTokenCookie(user, res) and responds with { success, message, user: sanitizedUser } (no token field).


File Edit Selection View Go Run Terminal Help
NASA-Open-API-with-Blog
userController.js M
userSchema.js
jwtToken.js (Working Tree) M
backend > utils > jwtToken.js > sanitizeUser
fix/damruwan/security-issues-fix*
0 0
Git Graph
Not Committed Yet
Ln 21, Col 29
Spaces: 2
UTF-8
LF
{} JavaScript
Go Live
Prettier

```
const getCookieOptions = () => {
};

export const attachTokenCookie = (user, res) => {
  const token = user.getJWTToken();
  const options = getCookieOptions();

  res.cookie("token", token, options);

  return token;
};

export const sendToken = (user, statusCode, message, res) => {
  const token = attachTokenCookie(user, res);

  res.status(statusCode).json({
    success: true,
    user,
    message,
    token,
  });
};
```

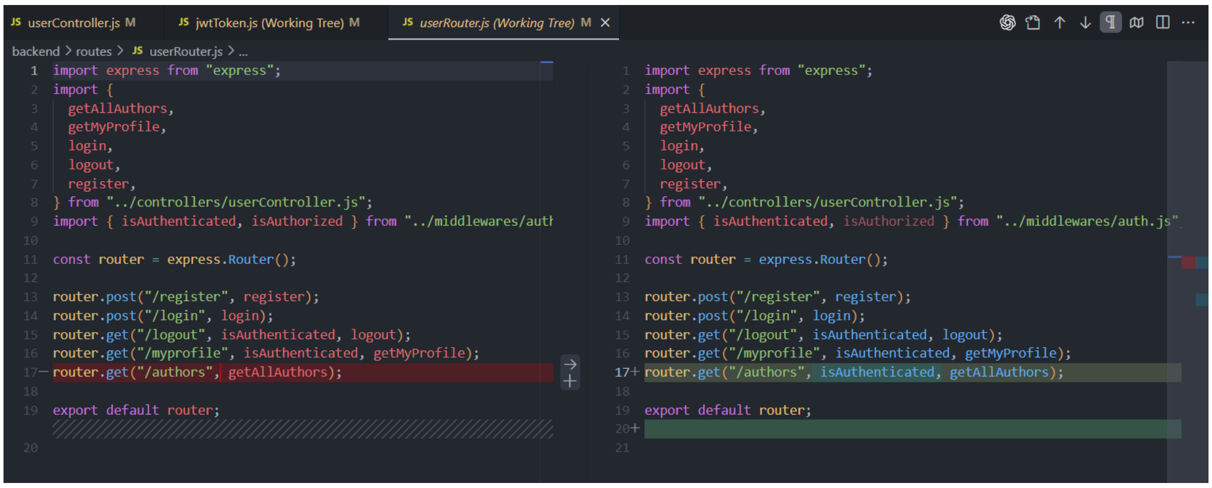
```
const getCookieOptions = () => {
};

const sanitizeUser = (user) => {
  if (!user) return user;
  const rawUser =
    typeof user.toObject === 'function' ? user.toObject() : { ...user };
  const sanitizedUser = { ...rawUser };
  delete sanitizedUser.password;
  delete sanitizedUser.phone;
  return sanitizedUser;
};

export const attachTokenCookie = (user, res) => {
  const token = user.getJWTToken();
  const options = getCookieOptions();

  res.cookie("token", token, options);

  return token;
};

export const sendToken = (user, statusCode, message, res) => {
  attachTokenCookie(user, res);

  const sanitizedUser = sanitizeUser(user);

  res.status(statusCode).json({
    success: true,
    user: sanitizedUser,
    message,
  });
};
```

F4 - Public Authors Endpoint Leaks Contact Data

OWASP Category: A01: Broken Access Control

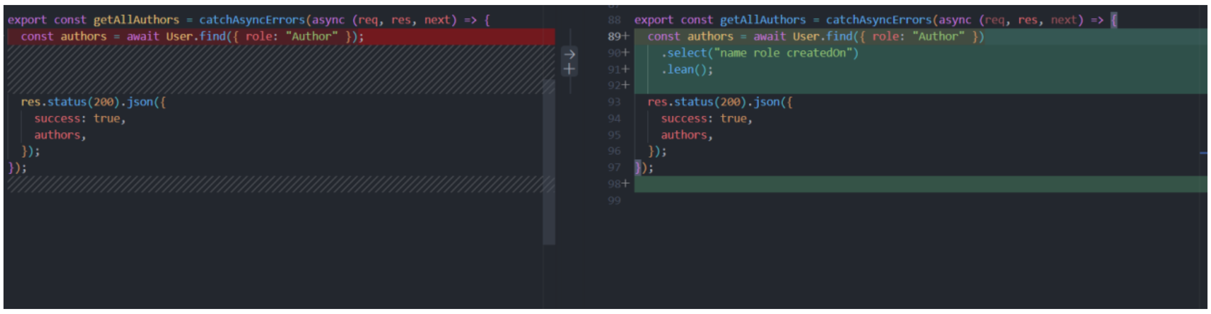
Issue: GET: /api/v1/user/authors was publicly accessible and returned full user documents (including contact data)

Fix: Guarded the route with isAuthenticated. Reduced the projection to a strict allowlist using Mongoose .select("name role createdOn") and .lean() for safe, plain objects.


userController.js M
jwtToken.js (Working Tree) M
userRouter.js (Working Tree) M
backend > routes > userRouter.js > ...

```
import express from "express";
import {
  getAllAuthors,
  getMyProfile,
  login,
  logout,
  register,
} from "../controllers/userController.js";
import { isAuthenticated, isAuthorized } from "../middlewares/autl

const router = express.Router();

router.post("/register", register);
router.post("/login", login);
router.get("/logout", isAuthenticated, logout);
router.get("/myprofile", isAuthenticated, getMyProfile);
router.get("/authors", getAllAuthors);

export default router;
```

```
import express from "express";
import {
  getAllAuthors,
  getMyProfile,
  login,
  logout,
  register,
} from "../controllers/userController.js";
import { isAuthenticated, isAuthorized } from "../middlewares/auth.js"

const router = express.Router();

router.post("/register", register);
router.post("/login", login);
router.get("/logout", isAuthenticated, logout);
router.get("/myprofile", isAuthenticated, getMyProfile);
router.get("/authors", isAuthenticated, getAllAuthors);

export default router;
```



```
export const getAllAuthors = catchAsyncErrors(async (req, res, next) => {
  const authors = await User.find({ role: "Author" });

  res.status(200).json({
    success: true,
    authors,
  });
});
```

```
export const getAllAuthors = catchAsyncErrors(async (req, res, next) => {
  const authors = await User.find({ role: "Author" })
    .select("name role createdOn")
    .lean();

  res.status(200).json({
    success: true,
    authors,
  });
});
```

4. UNFIXED ISSUES

During our assessment and remediation process, we successfully identified and fixed all high and medium-priority vulnerabilities in the application.

However, certain lower-priority issues were intentionally left unaddressed due to the following reasons:

- Minor UX or cosmetic vulnerabilities: These do not pose a security risk, e.g., slight error message formatting.
- External dependency updates: Some outdated packages could be upgraded but doing so would require extensive refactoring beyond the scope of this assignment.

By documenting these potential concerns, we acknowledge areas for future improvement while confirming that all critical security risks have been mitigated.

5. BEST PRACTICES & LESSONS LEARNED

During this project, we not only identified and fixed multiple vulnerabilities in the original application but also gained valuable insights into secure software development. The following best practices and lessons were highlighted:

5.1. Secure Coding Practices

- **Input Validation & Sanitization**: All user inputs, including text fields and file uploads, must be validated and sanitized to prevent XSS, SQL Injection, and malicious file execution. Libraries such as `sanitize-html` and server-side file validation help enforce this.
- **Proper Access Control**: Implement role-based access control and ownership checks to ensure users can only modify resources they own. Avoid hardcoding roles or permissions in the code.
- **Secrets Management**: Never commit API keys, credentials, or tokens to version control. Use environment variables, secret managers, or encrypted storage.

5.2. Secure Authentication & Session Management

- **JWT Usage**: Store authentication tokens securely using httpOnly and Secure cookies; avoid returning JWTs in response bodies.
- **OAuth/OpenID Connect**: Implement social login or secure third-party authentication flows properly to reduce risks related to weak authentication.

5.3. Secure File Handling

- **File Type Verification**: Validate MIME types and file extensions; additionally, check actual binary content using tools like `file-type` to prevent disguised malware uploads.
- **File Size & Quantity Limits**: Restrict upload sizes and number of files to prevent resource exhaustion attacks.
- **Filename Sanitization**: Replace unsafe characters and prevent path traversal attacks in filenames.

5.4. Configuration & Dependency Management

- **Secure Defaults**: Configure cookies, headers, and CORS policies with secure defaults to prevent common misconfigurations.
- **Dependency Auditing**: Regularly update packages and run tools like `npm audit` or `dependency-check` to prevent vulnerabilities from outdated libraries.

5.5. Lessons Learned

- Even small personal projects can have serious security flaws; thorough review is critical.
- Automated tools like OWASP ZAP greatly help in discovering vulnerabilities but manual code review is equally important.
- Documenting and fixing vulnerabilities improves not just security but overall project maintainability.
- Implementing OAuth/OpenID Connect enhances authentication security and provides real-world experience in integrating secure features.

By following these best practices, developers can proactively prevent many common vulnerabilities and create robust, secure applications.