API Security Assessment & Hardening Using Node.js

Author: Wayne Howlett

Date: May 2026

Skill Area: Application Security / API Security

Tools: Node.js, Express, JWT, Zod, Express Rate Limit, Thunder Client

Attack Type: Broken Authentication / Authorization / API Abuse

Environment: Local Development (Node.js API)

Outcome: Successfully identified and mitigated common API security vulnerabilities by implementing authentication, role-based access control, rate limiting, input validation, and security logging.

1. Objective

This project was designed to demonstrate the identification, exploitation, and mitigation of common API security vulnerabilities within a modern web application environment. Rather than focusing solely on building a functional API, the objective was to analyze how insecure design decisions can expose sensitive data and then apply layered security controls to protect against real-world attack scenarios.

The project aimed to provide hands-on experience with authentication mechanisms, access control, input validation, and abuse prevention techniques, while also reinforcing the importance of monitoring and logging in application security.

---

2. Scenario

APIs are a critical component of modern applications, serving as the primary interface between clients and backend systems. Because of this, they are frequently targeted by attackers seeking to exploit weak authentication, improper authorization, or poorly validated inputs.

In many real-world scenarios, APIs may initially be deployed without sufficient security controls, allowing unauthorized users to access sensitive endpoints or perform restricted actions. These vulnerabilities can lead to data exposure, privilege escalation, or service abuse.

In this scenario, a REST API was intentionally developed with minimal security controls to simulate a vulnerable environment. The goal was to identify weaknesses such as exposed endpoints and unrestricted access and then implement security measures to mitigate these risks.

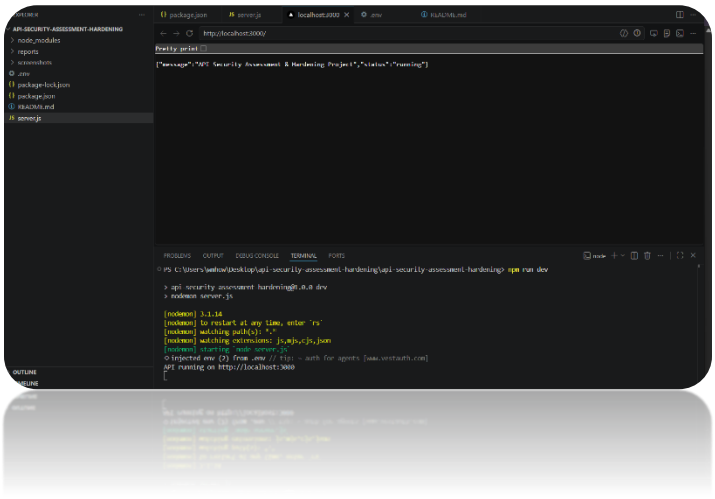
---

3. Environment

- Node.js / Express API
- Local development environment
- Thunder Client (API testing)
- JWT authentication system

Add:

Figure A1. API environment and testing setup



---

4. Tools Used

This project utilized several tools to simulate API interaction, test vulnerabilities, and validate security controls. Node.js and Express were used to build the API, providing a lightweight and flexible backend environment.

Thunder Client was used to simulate client requests and test API behavior, including authentication, authorization, and error handling. JSON Web Tokens (JWT) were implemented to manage authentication, while Zod was used to enforce input validation rules.

Express Rate Limit was used to simulate brute-force protection by limiting repeated login attempts. Together, these tools created a controlled environment for both vulnerability testing and security implementation.

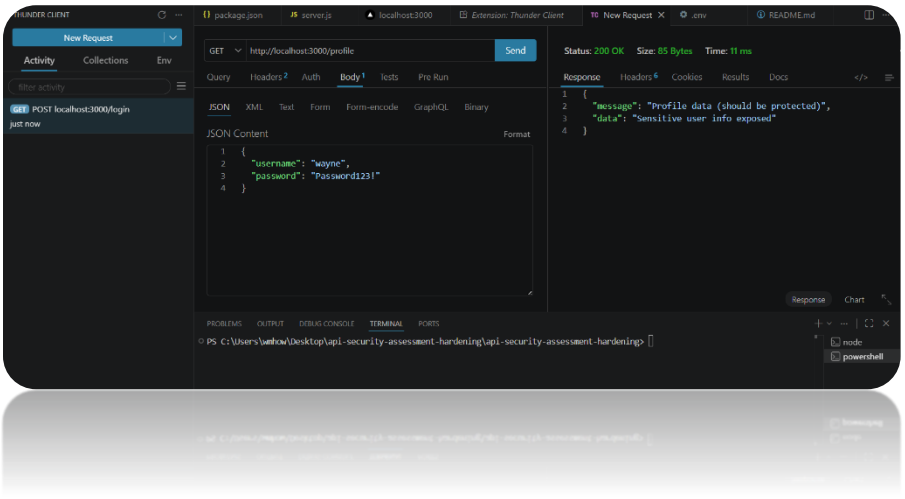
---

5. Vulnerability Simulation / Activity

During the initial phase, the API was tested without security controls to simulate common vulnerabilities found in poorly secured applications. A key issue identified was the exposure of a sensitive endpoint (/profile), which was accessible without authentication.

By sending direct requests to this endpoint, it was possible to retrieve sensitive data without any form of verification. This behavior reflects a common real-world vulnerability where APIs fail to enforce authentication and authorization checks.

Figure A2. Unauthorized access to /profile endpoint

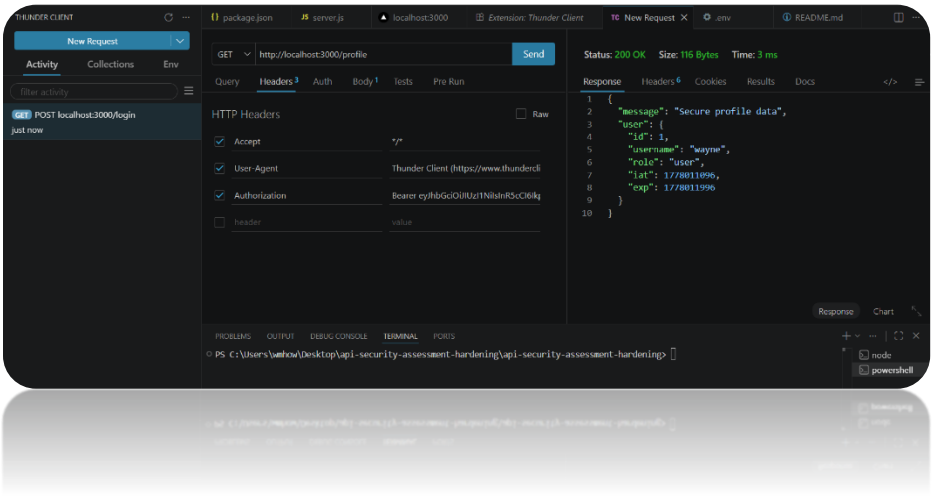


6. Authentication & Access Control

To address the identified vulnerabilities, authentication was implemented using JSON Web Tokens (JWT). Upon successful login, users are issued a token that must be included in subsequent requests to access protected endpoints.

This ensures that only authenticated users can interact with sensitive API routes. The /profile endpoint was updated to require a valid token, effectively preventing unauthorized access.

Figure A3. Authorized access using JWT token



7. Authorization (Role-Based Access Control)

Following authentication, role-based access control was introduced to restrict access to administrative functionality. A new endpoint (/admin) was created and configured to allow access only to users with an admin role.

Testing confirmed that normal users were denied access, while users with administrative privileges were able to successfully access the endpoint. This demonstrates the importance of enforcing role-based restrictions to prevent privilege escalation.

Figure A4. Admin access denied for standard user

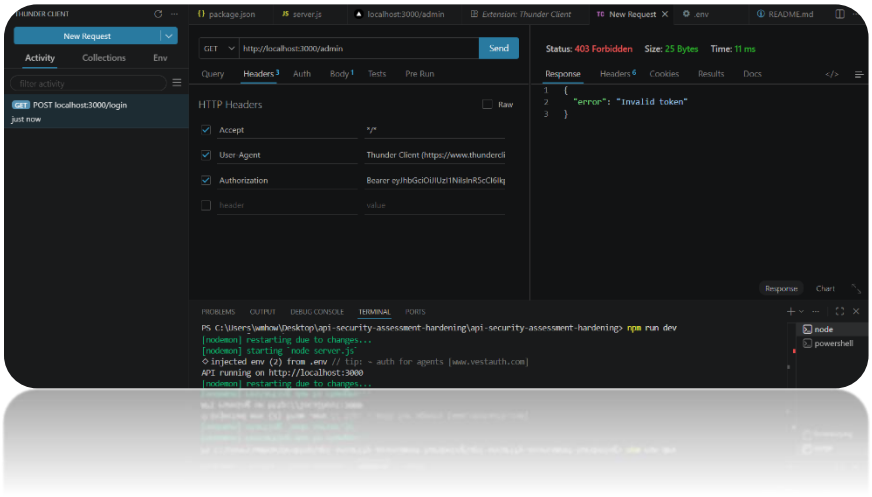
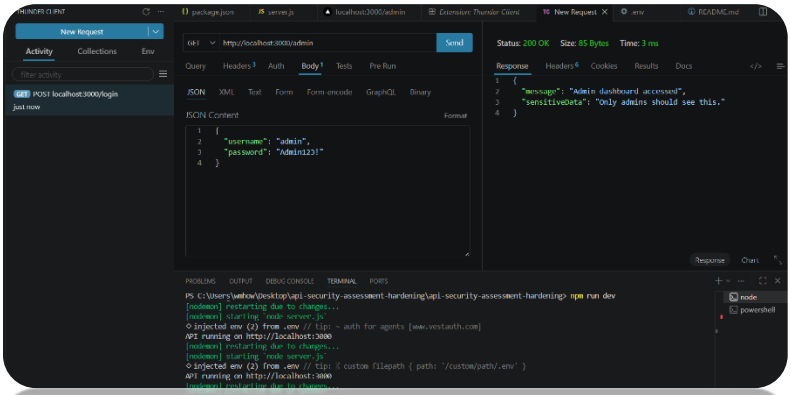


Figure A5. Admin access granted for authorized user

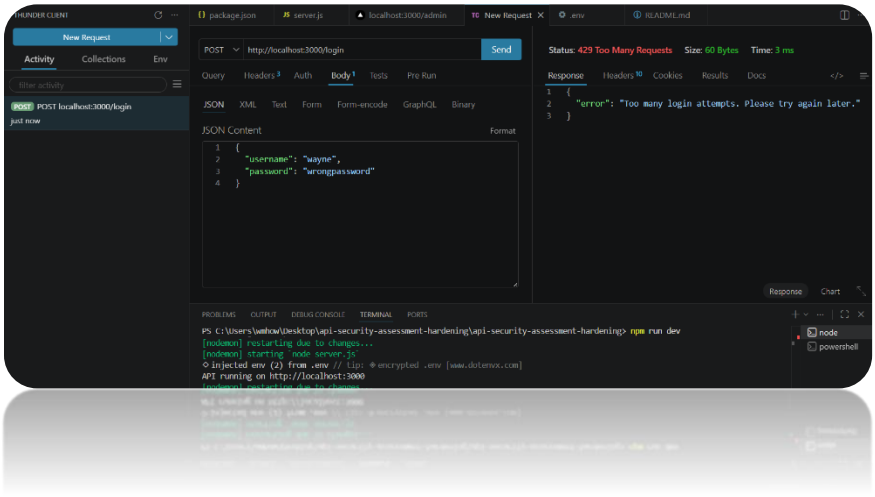


8. Abuse Prevention (Rate Limiting)

To mitigate brute-force login attempts, rate limiting was applied to the authentication endpoint. This restricts the number of login attempts within a defined time window, preventing attackers from rapidly testing multiple credential combinations.

During testing, repeated failed login attempts triggered the rate limiter, resulting in blocked requests. This demonstrates how rate limiting can effectively reduce the risk of automated attacks.

Figure A6. Rate limiting triggered after repeated login attempts

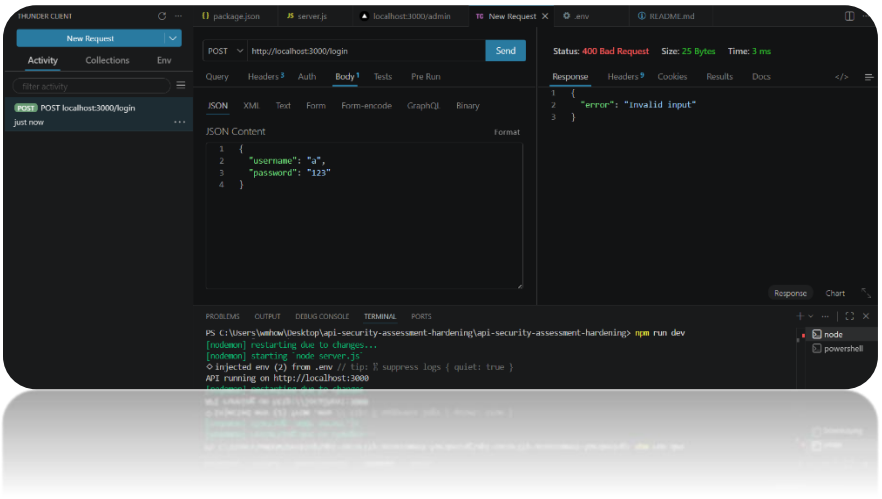


9. Input Validation

Input validation was implemented using Zod to enforce strict data requirements for incoming requests. This prevents malformed or weak inputs from being processed by the API.

Testing confirmed that invalid requests were rejected, ensuring that only properly structured data is accepted. This helps reduce the risk of injection attacks and unexpected application behavior.

Figure A7. Input validation error response

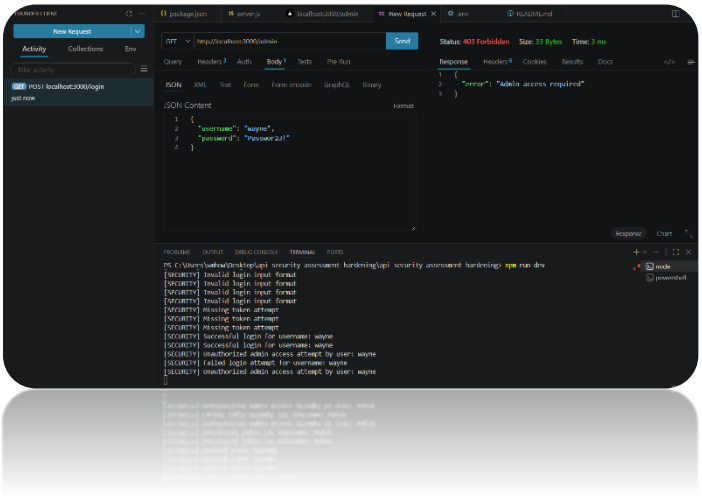


10. Security Logging

Security logging was implemented to capture key events such as failed login attempts, invalid input submissions, and unauthorized access attempts. These logs provide visibility into potentially malicious behavior and support monitoring and incident response.

The logs generated during testing demonstrate how security events can be tracked and analyzed in real time.

Figure A8. Security event logging in terminal



11. Validation

Testing confirmed that all security controls were functioning as intended. Unauthorized access was blocked, authentication was enforced, role-based restrictions were applied correctly, and rate limiting successfully prevented repeated abuse.

The results demonstrate that the API evolved from an insecure implementation into a layered and controlled system capable of defending against common attack scenarios.

---

12. Conclusion

This project demonstrates the importance of implementing layered security controls in modern API design. While functionality is critical, security must be integrated at every stage to prevent unauthorized access and abuse.

By applying authentication, authorization, validation, and monitoring, it is possible to significantly reduce the attack surface and improve overall system resilience.

---

13. What Could Be Improved

Future improvements could include integrating this API with a SIEM platform to monitor security events in real time, implementing automated response mechanisms such as IP blocking, and expanding validation rules to cover additional input scenarios.

Additional enhancements may include deploying the API in a cloud environment and applying security controls such as API gateways, WAF protection, and centralized logging.

---

14. Resume Bullet

Designed and secured a Node.js API by identifying vulnerabilities and implementing JWT authentication, role-based access control, rate limiting, input validation, and security logging to protect against real-world API attacks.