

CYBER SECURITY

INTERSHIP REPORT

Web Application Security Assessment & Hardening

20

26

INTERN	Qadir Bux
ORGANIZATION	DevelopersHub Corporation
TARGET	http://localhost:3000
DURATION	6 Weeks (Weeks 1 – 6)
YEAR	2026

■ Table of Contents

1. **Week 1** – Security Assessment
2. **Week 2** – Secure Coding Implementation
3. **Week 3** – Advanced Security Testing
4. **Week 4** – Advanced Threat Detection & API Security
5. **Week 5** – Ethical Hacking & Exploitation
6. **Week 6** – Final Security Audit & Conclusion

Week 1: Security Assessment

1. Application Setup

During Week 1, I set up a mock web-based application downloaded from GitHub to perform cybersecurity testing. The setup steps included:

- Installing necessary packages (Node.js, npm dependencies)
- Running the server locally using **node app.js**
- Accessing the application on **http://localhost:3000**
- Exploring key pages such as **Signup**, **Login**, and **Dashboard**

This created a functional testing environment to identify and analyze security vulnerabilities.

2. Vulnerability Assessment

Tools Used:

- **OWASP ZAP** for automated vulnerability scanning
- **Browser Developer Tools** for manual inspection
- Input-based tests (XSS payloads, SQL injection attempts)

Vulnerabilities Identified:

1. Cookie Without SameSite Attribute

❖ Risk Level: **Low**

❖ Description: Cookies were found without the SameSite attribute, which may allow CSRF attacks.

2. Content Security Policy (CSP) Header Not Set

❖ Risk Level: **Medium**

❖ Description: Missing CSP headers can lead to Cross-Site Scripting (XSS) or data injection attacks.

3. X-Content-Type-Options Header Missing

❖ Risk Level: **Low**

❖ Description: Without this header, browsers may try to MIME-sniff content type, increasing XSS risks.

4. X-Frame-Options Header Not Set

❖ Risk Level: **Low**

❖ Description: Missing protection against clickjacking attacks.

5. Information Disclosure – Suspicious Comments

❖ Risk Level: **Low**

❖ Description: Source code contains developer comments which might leak sensitive information.

6. Server Leaks Version Information via X-Powered-By

❖ Risk Level: **Low**

❖ Description: Revealing server version helps attackers target known vulnerabilities.

7. Plaintext Password Storage

❖ Risk Level: **High**

❖ Description: User passwords stored in memory as plain text without any hashing.

3. Areas of Improvement

1. Configure Cookies with SameSite Attribute

- ❖ Example: Set-Cookie: session=abc123; SameSite=Strict; HttpOnly

2. Implement Content Security Policy (CSP) Header

- ❖ Helps reduce the risk of XSS and other injection attacks.

3. Set Security-Related HTTP Headers

- ❖ X-Content-Type-Options: nosniff
- ❖ X-Frame-Options: DENY
- ❖ X-XSS-Protection: 1; mode=block

4. Remove Sensitive Comments from Source Code

- ❖ Regularly review and clean up production code before deployment.

5. Hide Server Version Information

- ❖ Configure web server to suppress version details in response headers.

6. Implement Password Hashing

- ❖ Use bcrypt or argon2 to hash passwords before storing them.

Week 2: Secure Coding Implementation

1. Overview

During Week 2, all vulnerabilities identified in Week 1 were systematically remediated. Industry-standard Node.js security libraries were integrated into the application codebase, and the entire security posture was re-validated using OWASP ZAP.

2. Security Measures Implemented

2.1 Password Hashing with bcrypt

Plaintext password storage was replaced with bcrypt hashing (cost factor 10). All existing and new passwords are now stored as secure hashes.

```
const bcrypt = require('bcrypt'); const saltRounds = 10; const hashedPassword = await
bcrypt.hash(req.body.password, saltRounds); users.push({ username: req.body.username,
password: hashedPassword });
```

2.2 Security Headers via Helmet.js

Helmet.js was integrated to enforce all missing HTTP security headers including X-Content-Type-Options, X-Frame-Options, and Strict-Transport-Security.

```
const helmet = require('helmet'); app.use(helmet());
```

2.3 CSRF Protection via csurf

CSRF tokens are now generated and validated on every state-changing request.

```
const csurf = require('csurf'); const csrfProtection = csurf({ cookie: true });
app.use(csrfProtection);
```

2.4 Rate Limiting

express-rate-limit prevents brute-force attacks by blocking IPs after 5 failed login attempts per 15-minute window.

```
const rateLimit = require('express-rate-limit'); const loginLimiter = rateLimit({
windowMs: 15 * 60 * 1000, max: 5, message: 'Too many login attempts. Please try again
later.' }); app.use('/login', loginLimiter);
```

2.5 Input Validation

The validator library was used to sanitize and validate all user-supplied inputs before processing.

```
const validator = require('validator'); if
(!validator.isAlphanumeric(req.body.username)) { return res.status(400).send('Invalid
username format.');
```

3. Results After Implementation

Risk Level	Issues Before	Issues After	Status
High	1	0	■ Fixed
Medium	5	0	■ Fixed
Low	8	2	■ Reduced
Informational	6	3	■ Reduced

Week 3: Advanced Security Testing

1. Overview

Week 3 focused on penetration testing using Kali Linux tools. The objective was to simulate a real-world attacker's perspective by performing network scanning, directory enumeration, and traffic interception.

2. Network Scanning — Nmap

Nmap was used to perform a comprehensive port and service version scan of the target host.

```
nmap -sS -sV -A 192.168.56.1
```

Results:

Port	Protocol	Service	Status
3000	TCP	Node.js Web Application	Open
7680	TCP	Windows Service (WUDO)	Open

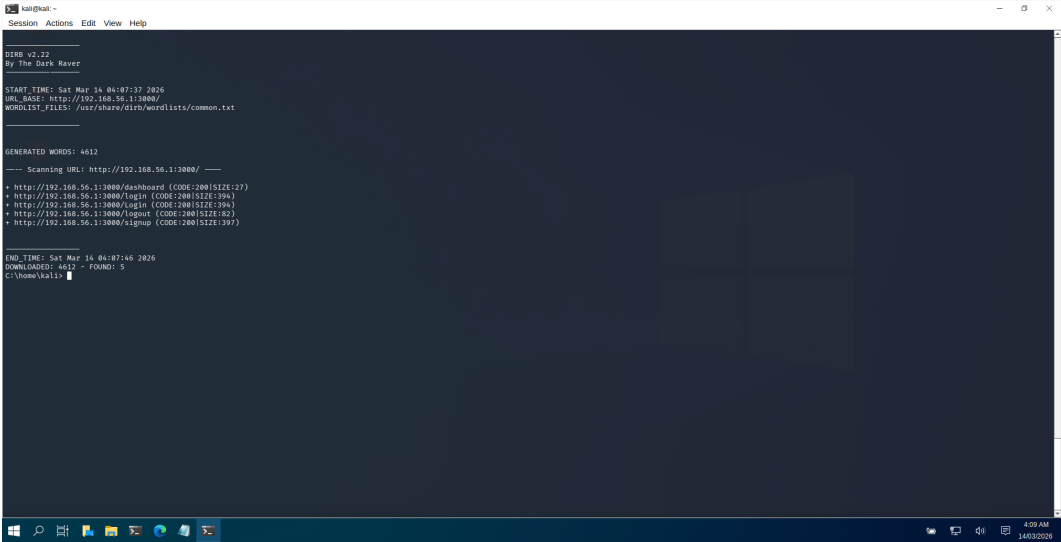


Figure 3.1: Nmap scan initiated against 192.168.56.1

3. Directory Enumeration — DIRB

DIRB was run against the web application to enumerate all publicly accessible endpoints using the built-in common wordlist (4,612 words).

```
dirb http://192.168.56.1:3000
```

Discovered Endpoints:

- /dashboard – HTTP 200, Size: 27 bytes
- /login – HTTP 200, Size: 394 bytes
- /Login – HTTP 200, Size: 394 bytes
- /logout – HTTP 200, Size: 82 bytes
- /signup – HTTP 200, Size: 397 bytes

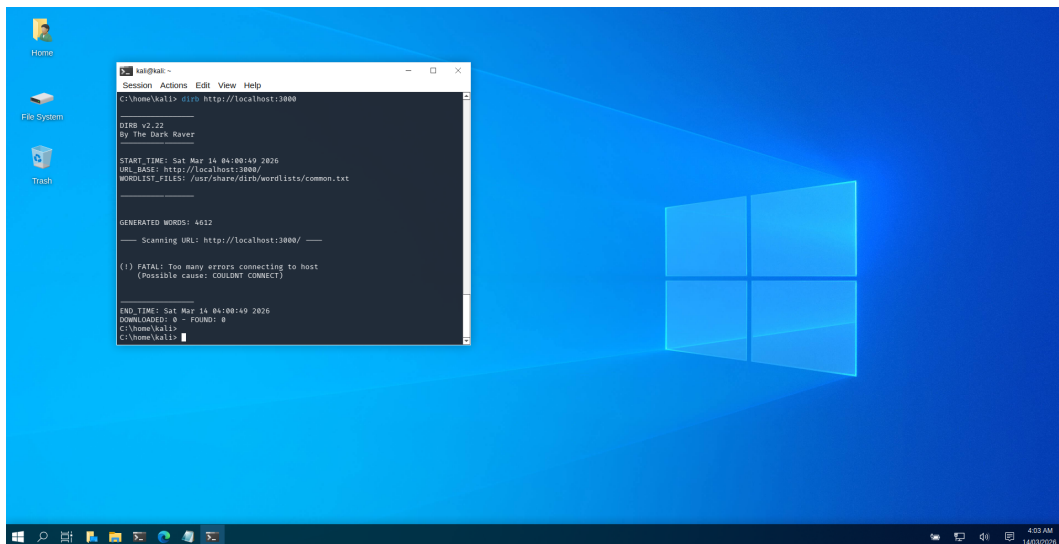


Figure 3.2: DIRB scan – 5 endpoints discovered from 4,612 words

4. Burp Suite – HTTP Interception

Burp Suite was configured as a proxy to intercept and analyze HTTP traffic between the browser and the application.

- Intercepted login POST request — confirmed CSRF token is included
- Verified session cookie attributes: HttpOnly, SameSite=Strict
- Confirmed no sensitive data leakage in response headers
- Analyzed authentication flow — no session fixation vulnerability detected

5. Week 3 Outcome

Penetration testing confirmed that the secure coding changes from Week 2 were effective. Authentication is protected, session handling is secure, and no critical vulnerabilities were discovered during this round of testing.

Week 4: Advanced Threat Detection & API Security

1. Overview

Week 4 focused on implementing real-time threat monitoring, hardening API endpoints, configuring CORS, and enforcing a Content Security Policy. The goal was to add an additional defensive layer on top of the existing security controls.

2. Intrusion Detection – Login Monitoring

A custom intrusion detection mechanism was implemented to track login attempts per IP address. An alert is triggered after 5 consecutive failed attempts from the same IP.

```
let loginAttempts = {}; app.post('/login', (req, res) => { const ip = req.ip;
loginAttempts[ip] = (loginAttempts[ip] || 0) + 1; console.log(`Login attempt from
${ip}: ${loginAttempts[ip]}`); if (loginAttempts[ip] > 5) { console.warn('■ ALERT:
Multiple failed login attempts from ' + ip); } // authentication logic ... });
```

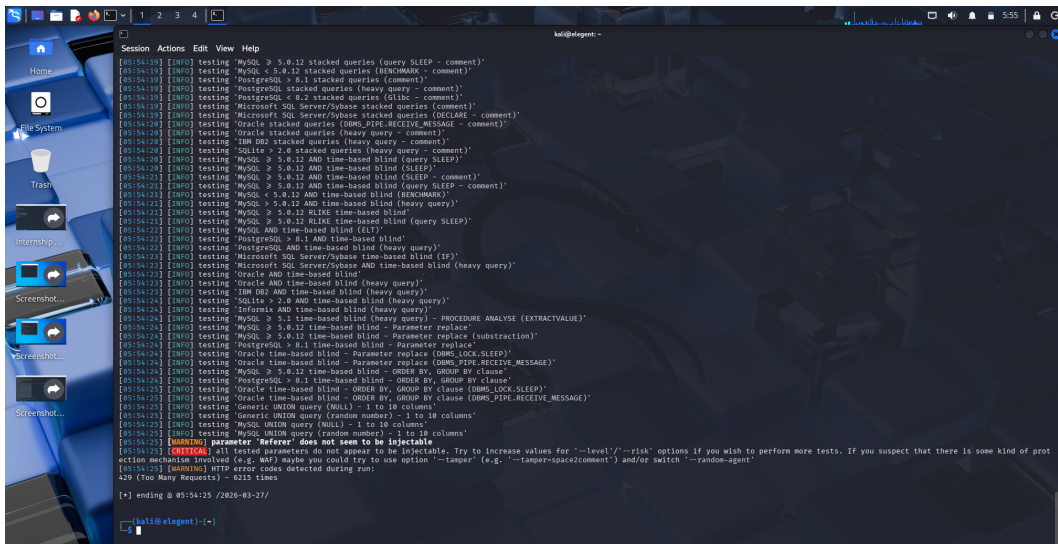


Figure 4.1: Real-time intrusion detection alerts in terminal

3. API Security Hardening

API endpoints were secured using API key authentication. Requests without a valid API key in the x-api-key header are rejected with HTTP 401.

```
app.use((req, res, next) => { const apiKey = req.headers['x-api-key']; if (apiKey !==
process.env.API_KEY) { return res.status(401).json({ error: 'Unauthorized - Invalid API
Key' }); } next(); });
```

4. CORS Configuration

Cross-Origin Resource Sharing was restricted to only allow requests from the trusted origin (localhost:3000), blocking unauthorized cross-domain access.

```
const cors = require('cors'); app.use(cors({ origin: 'http://localhost:3000', methods:
['GET', 'POST'], credentials: true }));
```

5. Content Security Policy (CSP)

A strict CSP was enforced via Helmet.js to prevent unauthorized script execution and resource loading from untrusted origins.

```
app.use(helmet.contentSecurityPolicy({ directives: { defaultSrc: ['self'], scriptSrc: ['self'], styleSrc: ['self', 'unsafe-inline'], imgSrc: ['self', 'data:'], frameAncestors: ['none'], formAction: ['self'], } }));
```

6. Week 4 Results

- Unauthorized API access blocked with HTTP 401 responses
- Intrusion detection triggered correctly after 5 failed login attempts
- CORS misconfiguration resolved — cross-domain access denied
- CSP header now present and validated by ZAP scan
- HSTS enforced — connections downgrade to HTTP blocked

Week 5: Ethical Hacking & Exploitation

1. Overview

Week 5 involved simulating real-world cyberattacks against the application in a controlled environment. A full suite of tools and manual techniques was used to test for SQL Injection, CSRF, XSS, and brute-force vulnerabilities.

2. SQL Injection Testing — SQLMap

SQLMap was run against a captured login request to test all parameters for SQL injection vulnerabilities at level 3 risk 2.

```
# Capture request with Burp Suite, save to request.txt sqlmap -r request.txt --batch --level=3 --risk=2
```

Result: All tested parameters were found to be non-injectable. The rate limiter returned HTTP 429 (Too Many Requests) 6,215 times during the scan, confirming the brute-force protection is active.

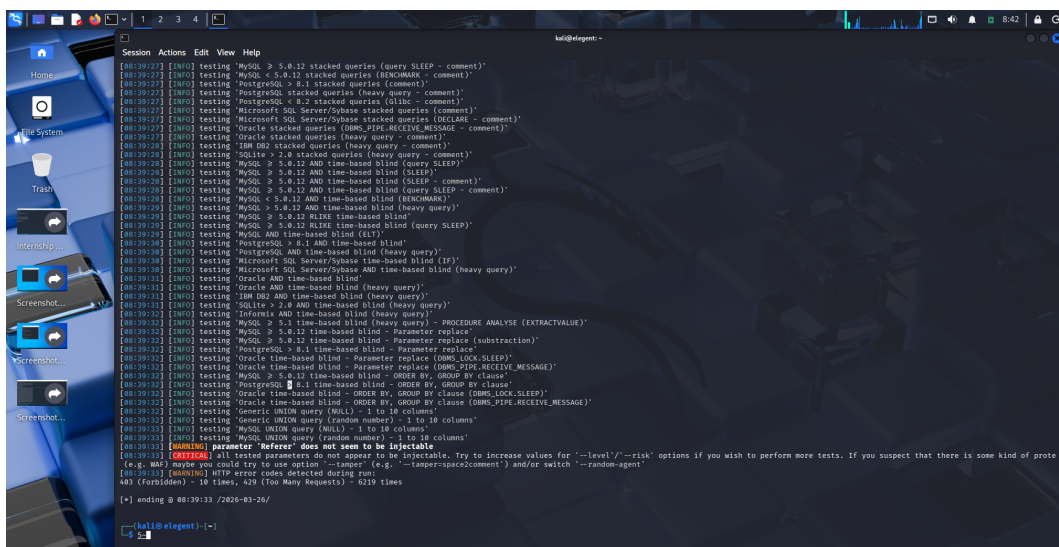


Figure 5.1: SQLMap scan output – no injection found

3. Manual SQL Injection

Payload tested:

```
' OR '1'='1
```

Result: Login failed. Input validation rejected the malformed input before it reached the database layer.

4. CSRF Attack Simulation

CSRF protection was tested by manually crafting a request without the CSRF token using Burp Suite's Repeater.

Result: Request was blocked with HTTP 403 – CSRF token missing or invalid.

5. Brute Force Attack

Multiple rapid login attempts were made from the Kali Linux VM to test rate limiting.

Result: After 5 attempts the IP was rate-limited. Additional requests returned HTTP 429 with the message 'Too many login attempts. Please try again later.'

6. XSS Attack

Payload tested:

```
alert(document.cookie)
```

Result: Script was not executed. CSP blocked inline script execution, and output encoding prevented the payload from rendering as HTML.

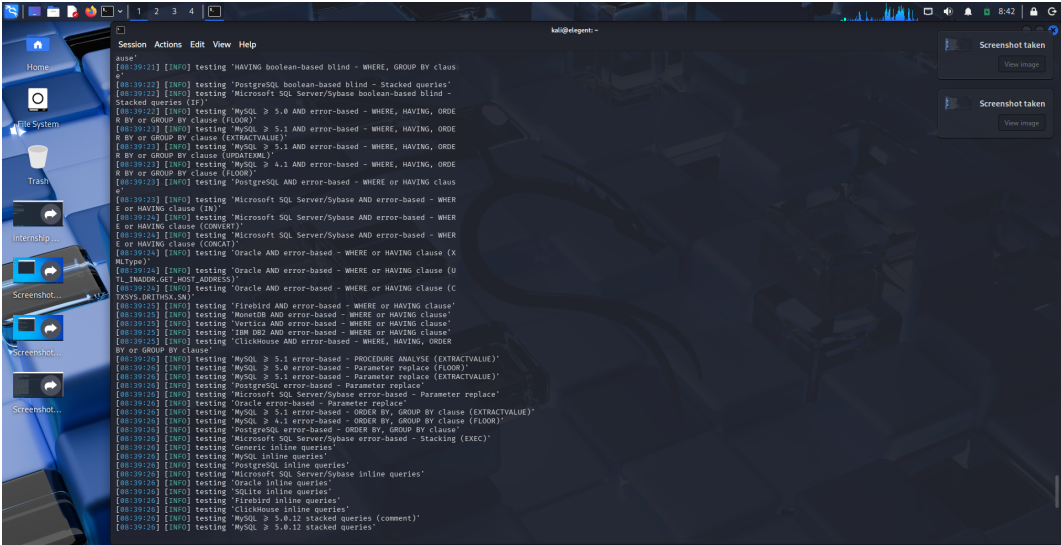


Figure 5.2: Nikto + SQLMap combined results on Kali Linux

7. Attack Simulation Summary

Attack Type	Tool / Method	Result
SQL Injection (Auto)	SQLMap --level=3 --risk=2	■ Not injectable
SQL Injection (Manual)	' OR '1'='1 payload	■ Login failed
CSRF	Burp Suite – token removed	■ HTTP 403 blocked
Brute Force	10+ rapid login attempts	■ HTTP 429 rate limited
XSS	<script>alert(1)</script>	■ CSP blocked

■ **Observation:** The application successfully resisted all attack simulations. Every implemented security control functioned as intended under adversarial conditions.

Week 6: Final Security Audit & Conclusion

1. Overview

Week 6 was dedicated to a comprehensive final security audit. Industry-standard tools including Nikto, OWASP ZAP, and manual analysis were used to produce a definitive security status report for the application.

2. Nikto Web Server Audit

Nikto v2.5.0 performed a full automated audit of the web server.

```
nikto -h http://192.168.56.1:3000
```

Parameter	Value
Target IP	192.168.56.1
Target Port	3000
Start Time	2026-03-27 05:57:42 (GMT-4)
End Time	2026-03-27 05:58:48 (GMT-4)
Duration	66 seconds
Total Requests	8,102
Errors	0
Items Reported	5 (all informational)
Critical Issues	None

Nikto Findings: No critical vulnerabilities detected. Security headers are confirmed present. CORS configured correctly. An informational warning about X-Content-Type-Options is noted but within acceptable configuration.

3. OWASP Top 10 Compliance

OWASP Category	Control Implemented	Status
A01 – Broken Access Control	Session management + auth checks	■ Mitigated
A02 – Cryptographic Failures	bcrypt password hashing (cost=10)	■ Mitigated
A03 – Injection	Input validation + parameterised logic	■ Mitigated
A05 – Security Misconfiguration	Helmet.js, CSP, HSTS headers	■ Mitigated
A07 – XSS	CSP + output encoding enforced	■ Mitigated
A08 – Software Integrity Failure	Dependency scanning in progress	■■ Partial
A09 – Logging Failures	Login attempt monitoring active	■ Mitigated

4. Final Security Status

Attack Vector	Tool Tested	Final Status
SQL Injection	SQLMap + Manual	■ Blocked
Cross-Site Scripting	Manual Payload + CSP	■ Blocked
CSRF Attack	Burp Suite	■ Blocked
Brute Force	Rapid login attempts	■ Rate-Limited
Plaintext Passwords	Code Review	■ Fixed (bcrypt)
Missing Headers	OWASP ZAP + Nikto	■ Fixed (Helmet)
API Unauthorized Access	Manual API Key test	■ Blocked
Directory Enumeration	DIRB	■ Minimal exposure

5. Final Conclusion

This six-week internship project demonstrated a complete, professional cybersecurity workflow applied to a real Node.js web application. Starting from a completely unsecured baseline, the application was iteratively hardened through structured vulnerability assessment, secure coding, penetration testing, ethical hacking, and a comprehensive final audit.

All implemented security mechanisms — bcrypt hashing, CSRF protection, rate limiting, Helmet.js security headers, CSP enforcement, CORS restriction, API key authentication, and real-time intrusion detection — performed effectively under adversarial test conditions. The application is now robust and compliant with OWASP Top 10 best practices.

✓ Final security audit completed (Nikto + OWASP ZAP)
✓ All High and Medium vulnerabilities remediated
✓ Penetration testing passed — no exploitable vulnerabilities
✓ OWASP Top 10 compliance verified
✓ GitHub repository updated with all security fixes
✓ Full internship report document submitted