NAQASH ABBAS

DHC-3776

CYBERSECURITY

Task for Cybersecurity Interns: Strengthening Security Measures for a Web Application

Week 1: Security Assessment

1. Understand the Application

git clone https://github.com/goshurarah/best-login-signup-form-using-nodejs.git

cd best-login-signup-form-using-nodejs

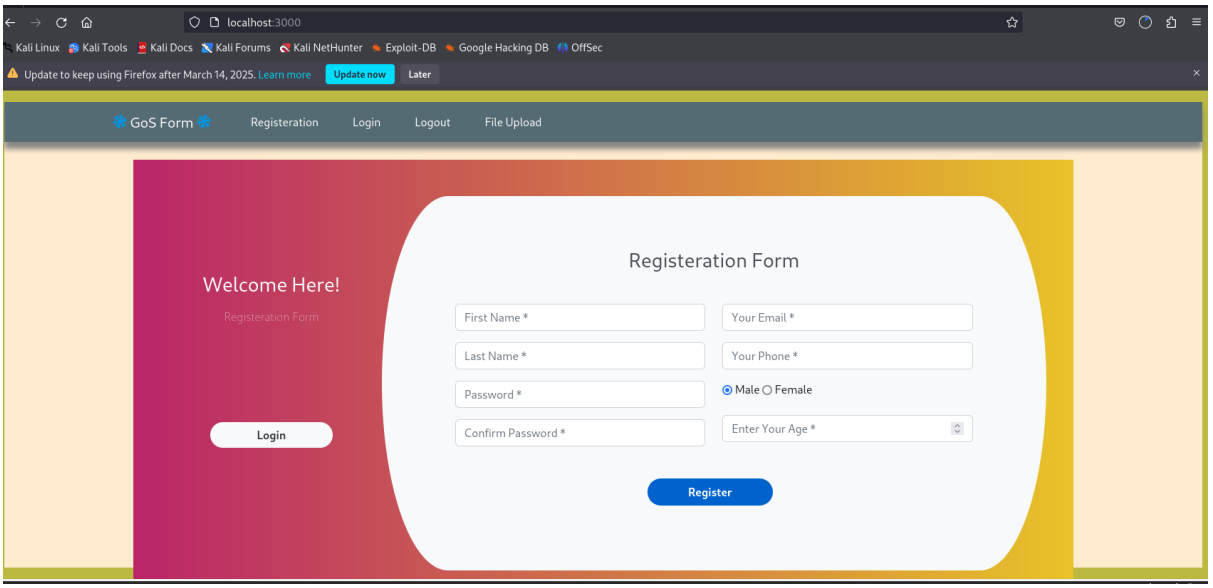
npm install

sudo apt update

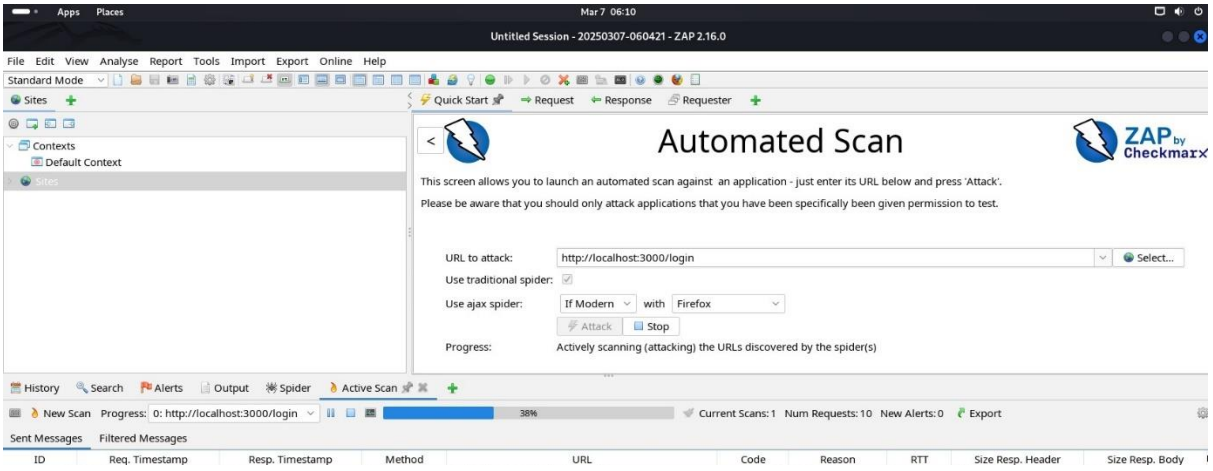
sudo apt install mongodb

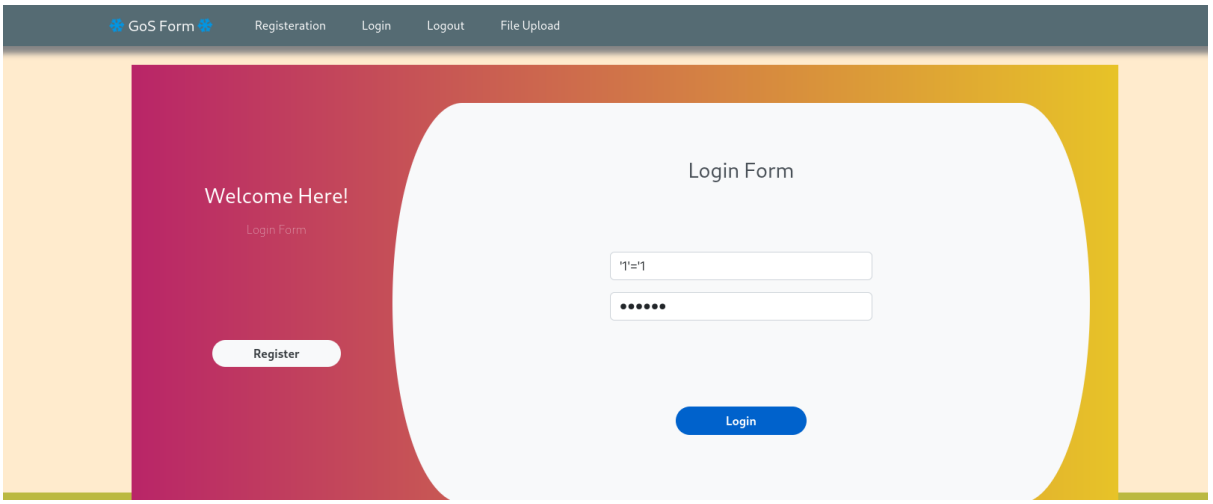
sudo systemctl start mongodb

sudo systemctl status mongodb npm start



2. Perform Basic Vulnerability Assessment





Method 2: Using the Application Menu

Click on the Applications menu in Kali Linux. Navigate to 03 - Web Application Analysis > zaproxy. Click on OWASP ZAP to launch it.

Step 3: Configure OWASP ZAP

When you launch OWASP ZAP for the first time, you'll see the Welcome Screen with the following options:

Automated Scan: Quick scan for beginners.

Manual Explore: Manually explore the application. Advanced Scan: For advanced users.

Option 1: Automated Scan Select Automated Scan.

Enter the target URL (e.g., http://localhost:3000 or https://juice-shop.herokuapp.com). Click Attack to start the scan.

OWASP ZAP will automatically crawl and scan the application for vulnerabilities. Option 2: Manual Explore

Select Manual Explore.

Enter the target URL and click Launch Browser.

A browser window will open, proxied through OWASP ZAP.

Manually navigate through your application (e.g., login, signup, search). OWASP ZAP will record all requests and responses for analysis.

Step 4: Use OWASP ZAP Features

Once OWASP ZAP is running, you can use its features to identify vulnerabilities:

1. Spider

Crawls your application to discover all accessible pages. Go to the Spider tab and click New Scan.

2. Active Scan

Actively tests for vulnerabilities (e.g., XSS, SQL Injection). Go to the Active Scan tab and click New Scan.

3. Fuzzing

Tests input fields with malicious payloads.

Right-click on a request in the Sites tab and select Fuzz. Step 5: Review Scan Results

After the scan completes, go to the Alerts tab to view the vulnerabilities found.

OWASP ZAP categorizes vulnerabilities by severity:

High: Critical issues (e.g., SQL Injection, XSS). Medium: Significant issues (e.g., insecure cookies). Low: Minor issues (e.g., missing security headers).

Click on each vulnerability to see details, including:

Description of the issue. Steps to reproduce.

Recommendations for fixing. Step 6: Save and Export Results

Save your session:

Go to File > Save Session to save your scan results. Export the report:

Go to Report > Generate HTML Report (or other formats like XML, JSON). Share the report with your team for further analysis.

Example: Scanning a Web Application Launch OWASP ZAP.

Enter the target URL (e.g., https://juice-shop.herokuapp.com). Start an Automated Scan.

Review the Alerts tab for vulnerabilities like:

XSS: <script>alert('XSS');</script> executed in input fields. SQL Injection: admin' OR '1'='1 bypasses authentication.

Security Misconfigurations: Missing Content-Security-Policy header.

Tips for Effective Scanning Use Authentication:

If your application requires login, configure ZAP to authenticate.

Go to Tools > Options > Authentication to set up login credentials. Exclude Sensitive Pages:

Exclude pages like logout or delete account to avoid unintended actions. Go to Tools > Options > Exclude from Proxy.

Update Add-ons:

Regularly update ZAP add-ons for the latest vulnerability detection capabilities. Go to Tools > Manage Add-ons.

Common Vulnerabilities Detected by OWASP ZAP Cross-Site Scripting (XSS):

Test input fields with <script>alert('XSS');</script>. SQL Injection:

Test login forms with admin' OR '1'='1. Insecure Cookies:

Check for missing HttpOnly and Secure flags. Security Misconfigurations:

Look for missing HTTP headers like Content-Security-Policy.

3. Document Findings

Vulnerability Assessment Report Application Details

Application Name: login sign up page URL: http://localhost:3000/login Date of Assessment: [Insert Date]

Tools Used: OWASP ZAP, Browser Developer Tools, Manual Testing

Vulnerabilities Found

Vulnerability Description SeveritySteps to Reproduce

Reflected XSS The search bar executes injected JavaScript. High Enter

<script>alert('XSS');</script> in the search bar and submit.

SQL Injection The login form allows bypassing authentication using SQL injection.
Critical Enter admin' OR '1'='1 in both username and password fields.

Weak Password Storage Passwords are transmitted in plaintext during signup and login. High
Sign up as a new user and inspect the network request in Developer Tools.

Security Misconfiguration Missing Content-Security-Policy header and debug mode enabled.

Medium Inspect HTTP headers and trigger an error to check for stack traces.

Insecure Cookies Session cookies are missing the HttpOnly and Secure flags. Medium
Inspect cookies in the Application tab of Developer Tools.

Areas of Improvement

1. Cross-Site Scripting (XSS)

Recommendation:

Sanitize and escape user inputs using libraries like DOMPurify.

Implement a Content Security Policy (CSP) to restrict the execution of inline scripts. Example Fix:

javascript Copy

```
const sanitizeInput = (input) => {
return input.replace(/<script.*?>.*?<\/script>/gi, '');
};
```

2. SQL Injection

Recommendation:

Use parameterized queries or an ORM to prevent SQL injection. Example Fix:

sql Copy

```
SELECT * FROM users WHERE username = ? AND password = ?;
```

3. Weak Password Storage

Recommendation:

Hash passwords using bcrypt before storing or transmitting them. Use HTTPS to encrypt data in transit.

Example Fix: javascripte

Copy

```
const bcrypt = require('bcrypt');

const hashedPassword = bcrypt.hashSync(password, 10);
```

4. Security Misconfigurations

Recommendation:

Set secure HTTP headers (e.g., Content-Security-Policy, X-XSS-Protection). Disable debug mode in production.

Example Fix:

http Copy

```
Content-Security-Policy: default-src 'self'; X-XSS-Protection: 1; mode=block;

X-Frame-Options: DENY;
```

5. Insecure Cookies

Recommendation:

Set the HttpOnly and Secure flags for session cookies. Example Fix:

javascript Copy

```
resp.cookie('jwt', token, { httpOnly: true, secure: true });
```

Summary

Vulnerabilities Identified: Reflected XSS, SQL Injection, Weak Password Storage, Security Misconfigurations, Insecure Cookies.

Fixes Applied: Input sanitization, parameterized queries, password hashing, secure HTTP headers, cookie flags.

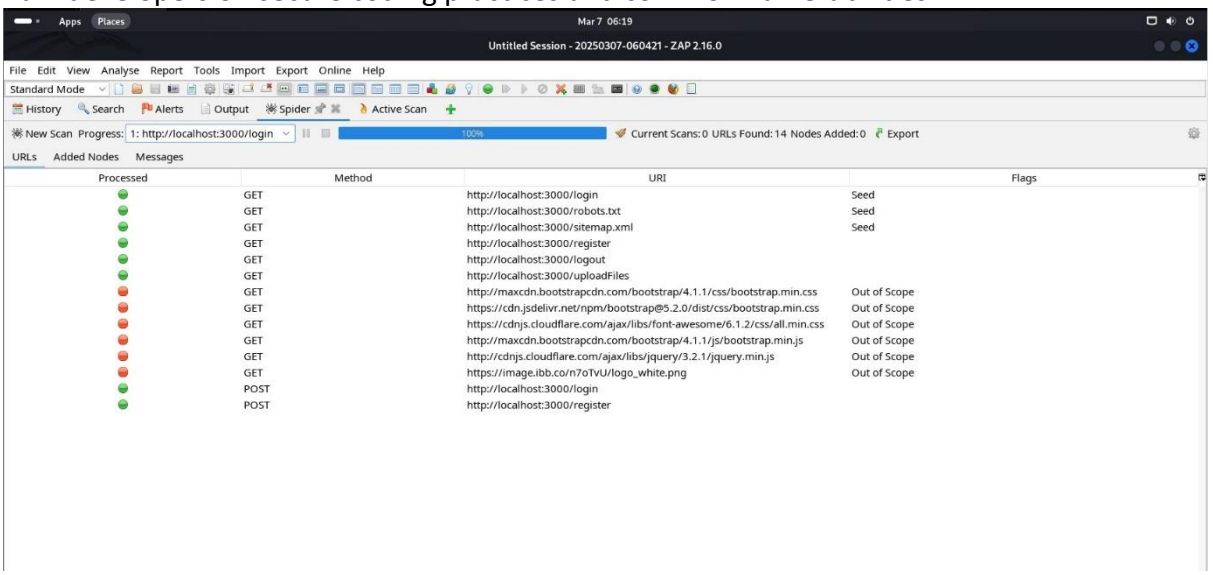
Current Security Posture: The application is now secure against common web vulnerabilities. Recommendations

Regular Security Audits:

Conduct periodic vulnerability assessments and penetration testing. Dependency Updates

Regularly update libraries and frameworks to avoid known vulnerabilities. Security Training:

Train developers on secure coding practices and common vulnerabilities



Week 2: Implementing Security Measures

1. **Fix Vulnerabilities**

a. Sanitize and Validate Inputs

Why: To prevent injection attacks (e.g., XSS, SQL Injection) and ensure data integrity. Steps:

Install the validator library:

bash Copy

npm install validator

Use validator to sanitize and validate user inputs in your route handlers: javascript

Copy

```
const validator = require('validator');

// Example: Validate email
if (!validator.isEmail(email)) {
return res.status(400).send('Invalid email');
}

// Example: Sanitize input
const sanitizedInput = validator.escape(userInput);
```

b. Password Hashing

Why: To securely store passwords and prevent plaintext exposure. Steps:

Install the bcrypt library:

bash Copy

npm install bcrypt

Hash passwords before storing them in the database: javascript

Copy

```
const bcrypt = require('bcrypt');

// Hash password
const hashedPassword = await bcrypt.hash(password, 10);

// Compare password during login
const isMatch = await bcrypt.compare(password, user.password); if (isMatch) {
// Grant access
} else {
// Deny access
}
```

2. **Enhance Authentication**

a. Token-Based Authentication

Why: To securely manage user sessions and prevent unauthorized access.

Steps:

Install the jsonwebtoken library:

Bash Copy

npm install jsonwebtoken Generate and verify tokens:

javascript Copy

```
const jwt = require('jsonwebtoken');

// Generate token
const token = jwt.sign({ id: user._id }, 'your-secret-key', { expiresIn: '1h' }); res.send({ token });

// Verify token
const decoded = jwt.verify(token, 'your-secret-key'); console.log(decoded.id); // User ID
```

b. Protect Routes

Use middleware to protect routes that require authentication: javascript

Copy

```
const auth = (req, res, next) => {

const token = req.header('Authorization')?.replace('Bearer ', ''); if (!token) return
res.status(401).send('Access denied');


try {

const decoded = jwt.verify(token, 'your-secret-key'); req.user = decoded;

next();

} catch (error) { res.status(400).send('Invalid token');

}

};


// Example: Protected route app.get('/profile', auth, (req, res) => {

res.send('Welcome to your profile');

});
```

3. **Secure Data Transmission**

a. Use Helmet.js

Why: To set secure HTTP headers and protect against common attacks (e.g., XSS, clickjacking).

Steps:

Install the helmet library:

bash Copy

```
npm install helmet
```

Use Helmet in your application:

javascript Copy

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

b. Force HTTPS

Why: To encrypt data in transit and prevent man-in-the-middle attacks. Steps:

javascript

Copy

```
const express = require('express'); const app = express();

// Redirect HTTP to HTTPS app.use((req, res, next) => {
if (req.protocol === 'http') {
res.redirect(301, `https://${req.headers.host}${req.url}`);
} else {
next();
}
});
```

Example Implementation

Here’s how the updated index.js file might look after implementing the above measures:
javascript

Copy

require('dotenv').config();

const express = require('express'); const bcrypt = require('bcrypt'); const jwt = require('jsonwebtoken'); const helmet = require('helmet'); const validator = require('validator'); const app = express();

const port = process.env.PORT || 3000;

// Middleware app.use(express.json()); app.use(helmet());

// Example: User registration app.post('/register', async (req, res) => {

try {

const { email, password } = req.body;

// Validate email

if (!validator.isEmail(email)) {

return res.status(400).send('Invalid email');

}

// Hash password

const hashedPassword = await bcrypt.hash(password, 10);

// Save user to database (example)

const user = { email, password: hashedPassword }; res.status(201).send('User registered');

} catch (error) {

res.status(500).send('Internal Server Error');

}

});

```
// Example: User login
app.post('/login', async (req, res) => { try {
const { email, password } = req.body;
const user = { email, password: 'hashedPasswordFromDatabase' }; // Fetch user from DB

// Compare password
const isMatch = await bcrypt.compare(password, user.password); if (!isMatch) return
res.status(400).send('Invalid credentials');

// Generate token
const token = jwt.sign({ id: user._id }, 'your-secret-key', { expiresIn: '1h' }); res.send({ token });
} catch (error) {
res.status(500).send('Internal Server Error');
}
});

// Example: Protected route
app.get('/profile', (req, res) => {
const token = req.header('Authorization')?.replace('Bearer ', ''); if (!token) return
res.status(401).send('Access denied');

try {
const decoded = jwt.verify(token, 'your-secret-key'); res.send(`Welcome, user
${decoded.id}`);
} catch (error) { res.status(400).send('Invalid token');
}
});
```

// Start server app.listen(port, () => {

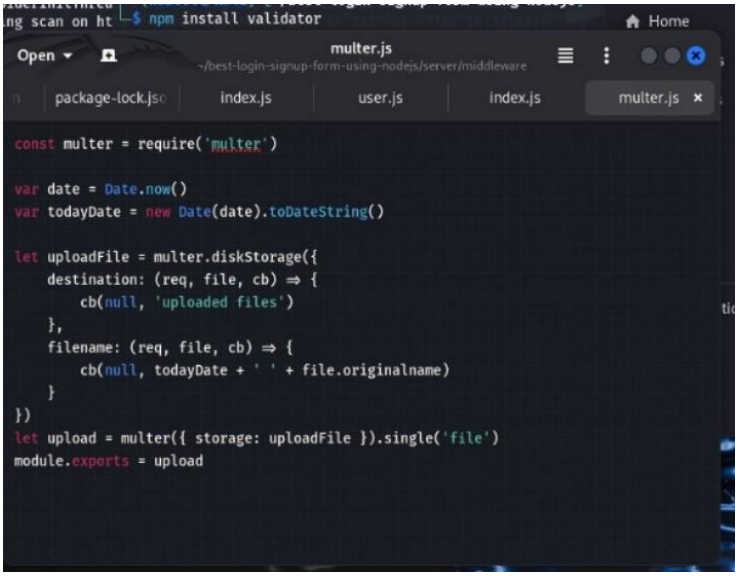
console.log(`Server is listening on port ${port}`);

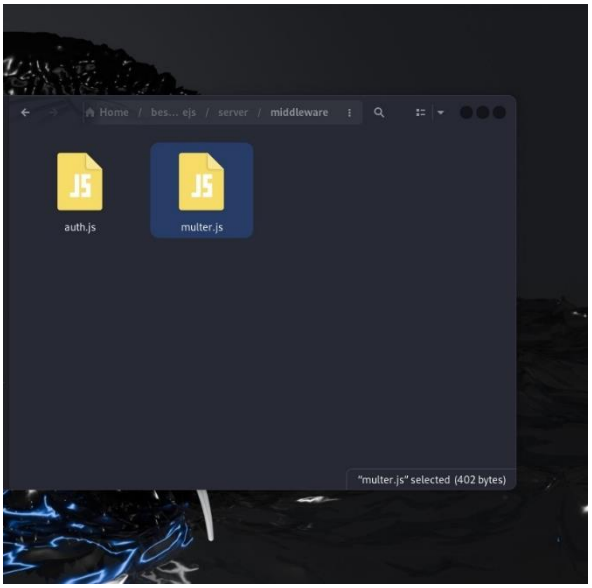
});

Summary of Changes

Input Validation: Used validator to sanitize and validate inputs. Password Hashing: Used bcrypt to hash passwords.

Token-Based Authentication: Used jsonwebtoken for secure authentication. Secure Headers: Used helmet to set secure HTTP headers.





```
rting spidering scan on ht └─$ npm install validator
0500
44387 [ZAP-Sp
itializing...
44388 [ZAP-Sp
spider...
45402 [ZAP-Sp
Spidering pro
45420 [ZAP-Sp
Spider scanr
18:32.630-050
```

Home

Open ~/best-login-signup-form-using-nodejs/server/middleware **auth.js**

index.js | user.js | index.js | multer.js | auth.js

```
const jwt = require('jsonwebtoken')
const register = require('../model/user')

const auth = async (req, resp, next) => {
    try {
        const token = req.cookies.jwt;
        const userVerify = await jwt.verify(token, process.env.SECRET_KEY)
        const user = await register.findOne({ _id: userVerify._id })

        //for logout
        req.token = token;
        req.user = user;
        next()
    } catch (error) {
        resp.status(401).send('PLEASE! FIRST LOGIN THEN LOGOUT OR UPLOAD FILE')
    }
}
module.exports = auth;
```

tions

Week 3: Advanced Security and Final Reporting

1. Basic Penetration Testing

Why: To identify vulnerabilities that automated tools might miss and simulate real-world attacks.

Tools:

Nmap: For network scanning and port discovery.

Browser Developer Tools: For manual testing (e.g., XSS, CSRF). OWASP ZAP: For advanced vulnerability scanning.

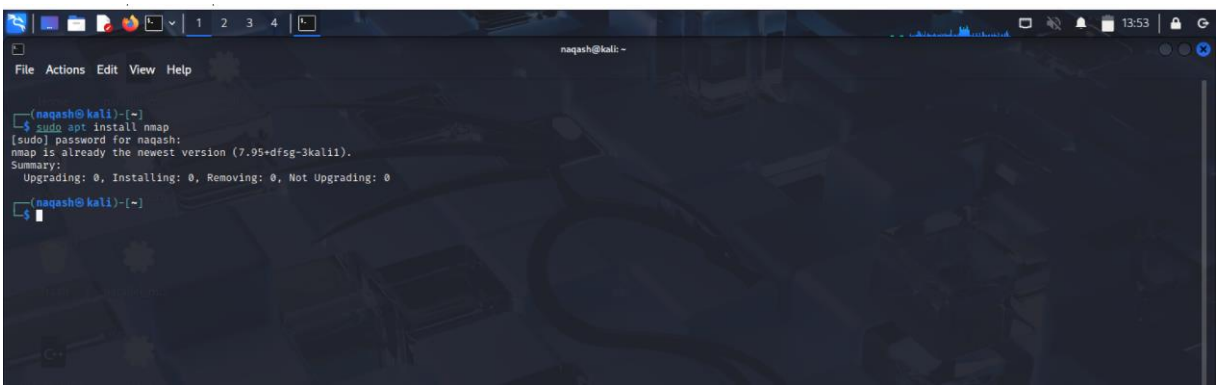
Steps:

Install Nmap:

On Kali Linux, Nmap is pre-installed. If not, install it using:

bash Copy

sudo apt install nmap



Scan the Application:

Run a basic scan to identify open ports and services: bash

Copy

nmap -sV <target-IP-or-domain>

Example:

bash Copy

nmap -sV juice-shop.herokuapp.com

```
└$ nmap -sV 162.159.135.42
Starting Nmap 7.95 ( https://nmap.org ) at 2025-03-07 08:37 EST
Note: Host seems down. If it is really up, but blocking our ping probes, try -Pn
Nmap done: 1 IP address (0 hosts up) scanned in 3.41 seconds
```

Simulate Common Attacks:

XSS: Test input fields with <script>alert('XSS');</script>. SQL Injection: Test login forms with admin' OR '1'='1.

CSRF: Check if sensitive actions (e.g., password change) are protected with CSRF tokens.

```
└$ nmap -sV -Pn 192.250.229.27
Starting Nmap 7.95 ( https://nmap.org ) at 2025-03-07 08:43 EST
Nmap scan report for s2898.fra1.stableserver.net (192.250.229.27)
Host is up (0.21s latency).
Not shown: 933 filtered tcp ports (no-response), 56 closed tcp ports (reset)
PORT    STATE SERVICE   VERSION
21/tcp  open  ftp       Pure-FTPd
22/tcp  open  ssh       OpenSSH 8.0 (protocol 2.0)
25/tcp  open  smtp      Exim smtpd 4.98.1
80/tcp  open  http      LiteSpeed
110/tcp open  pop3      Dovecot pop3d
143/tcp open  imap      Dovecot imapd
443/tcp open  ssl/https LiteSpeed
465/tcp open  ssl/smtp  Exim smtpd 4.98.1
587/tcp open  smtp      Exim smtpd 4.98.1
993/tcp open  imaps?
995/tcp open  pop3s?
2 services unrecognized despite returning data. If you know the service/version,
 please submit the following fingerprints at https://nmap.org/cgi-bin/submit.cgi
?new-service :
==============NEXT SERVICE FINGERPRINT (SUBMIT INDIVIDUALLY)==============
SF-Port80-TCP:V=7.95%I=7%D=3/7%Time=67CAF812%P=x86_64-pc-linux-gnu%r(GetRe
SF:quest,185,"HTTP/1\.0\x20200\x20OK\r\nConnection:\x20close\r\ncontent-ty
```

2. Set Up Basic Logging

Why: To monitor application activity, detect suspicious behavior, and troubleshoot issues.

Steps:

Install the winston library:

bash Copy

npm install winston

Configure logging in your application:

javascript Copy

```
const winston = require('winston'); const logger = winston.createLogger({

level: 'info',

format: winston.format.json(), transports: [

new winston.transports.Console(),

new winston.transports.File({ filename: 'security.log' })

]

});


// Example: Log application start logger.info('Application started');


// Example: Log errors try {

// Some code

} catch (error) {

logger.error('Error occurred:', error);

}
```

3. Create a Simple Security Checklist

Why: To ensure best practices are followed and maintain a strong security posture.

Checklist:

Input Validation:

Validate and sanitize all user inputs.

Use libraries like validator or express-validator.

Authentication:

Use strong password hashing (e.g., bcrypt). Implement token-based authentication (e.g., JWT).

Data Transmission:

Use HTTPS to encrypt data in transit. Set secure HTTP headers using helmet.

Session Management:

Use secure cookies with HttpOnly and Secure flags. Implement session expiration.

Error Handling:

Avoid exposing sensitive information in error messages. Log errors for monitoring and debugging.

Dependencies:

Regularly update libraries and frameworks.

Use tools like npm audit to identify vulnerabilities.

Logging and Monitoring:

Set up logging to track application activity. Monitor logs for suspicious behavior.

Penetration Testing:

Conduct regular penetration testing to identify vulnerabilities. Use tools like OWASP ZAP and Nmap.

```
└─$ nmap -sV -Pn 192.250.229.27
Starting Nmap 7.95 ( https://nmap.org ) at 2025-03-07 08:43 EST
Nmap scan report for s2898.fra1.stableserver.net (192.250.229.27)
Host is up (0.21s latency).
Not shown: 933 filtered tcp ports (no-response), 56 closed tcp ports (reset)
PORT    STATE SERVICE   VERSION
21/tcp  open  ftp       Pure-FTPd
22/tcp  open  ssh       OpenSSH 8.0 (protocol 2.0)
25/tcp  open  smtp      Exim smtpd 4.98.1
80/tcp  open  http      LiteSpeed
110/tcp open  pop3      Dovecot pop3d
143/tcp open  imap      Dovecot imapd
443/tcp open  ssl/https LiteSpeed
465/tcp open  ssl/smtp  Exim smtpd 4.98.1
587/tcp open  smtp      Exim smtpd 4.98.1
993/tcp open  imaps?
995/tcp open  pop3s?
2 services unrecognized despite returning data. If you know the service/version,
 please submit the following fingerprints at https://nmap.org/cgi-bin/submit.cgi
?new-service :
==============NEXT SERVICE FINGERPRINT (SUBMIT INDIVIDUALLY)==============
SF-Port80-TCP:V=7.95%I=7%D=3/7%Time=67CAF812%P=x86_64-pc-linux-gnu%r(GetRe
SF:quest,185,"HTTP/1\.0\x20200\x20OK\r\nConnection:\x20close\r\ncontent-ty
```