|  |
| --- |
| OWASP Juice Shop |
|  |
| Muhammad adnan Ali  DHC-261  Cybersecurity Internship |



Security Improvement Report for OWASP Juice Shop

Task: Fix Vulnerabilities — Input Sanitization, Validation, and Password Hashing

Overview

In this task, we addressed two critical security concerns in the OWASP Juice Shop website related to user data handling:

1. Sanitize and Validate Inputs — Ensuring that user inputs, particularly email addresses, are properly validated to prevent injection and malformed data.
2. Password Hashing — Securing user passwords by hashing them with a strong hashing algorithm before storing them in the database.

Actions Taken

1. Input Validation and Sanitization

* Integrated the validator library to validate email addresses.
* In the User model, the email setter method now verifies the email format using validator.isEmail(email).
* If the email is invalid, the system throws an error and rejects the input immediately.
* Additionally, input sanitization is applied to prevent injection or XSS attacks by using secure sanitization functions.

typescript

CopyEdit

import validator from 'validator';

email: {

type: DataTypes.STRING,

unique: true,

set(email: string) {

if (!validator.isEmail(email)) {

throw new Error('Invalid email format');

}

// Additional sanitization code here

this.setDataValue('email', email);

}

}

2. Password Hashing

* Implemented password hashing with bcrypt to securely store user passwords.
* Added Sequelize hooks beforeCreate, beforeUpdate, and beforeSave to automatically hash passwords whenever they are created or updated.
* Used a salt rounds value of 10 for bcrypt, balancing security and performance.

typescript

CopyEdit

import bcrypt from 'bcrypt';

User.addHook('beforeCreate', async (user: User) => {

if (user.password) {

user.password = await bcrypt.hash(user.password, 10);

}

});

User.addHook('beforeUpdate', async (user: User) => {

if (user.changed('password')) {

user.password = await bcrypt.hash(user.password, 10);

}

});

User.addHook('beforeSave', async (user: User) => {

if (user.changed('password')) {

user.password = await bcrypt.hash(user.password, 10);

}

});

Outcome

* Email validation ensures only well-formed email addresses are accepted, reducing the risk of injection and invalid data.
* Password hashing guarantees that plaintext passwords are never stored, significantly improving user credential security.
* The combined improvements enhance overall system robustness against common web vulnerabilities such as injection attacks and password theft.

Conclusion

By integrating input validation with validator and enforcing secure password hashing using bcrypt, the OWASP Juice Shop project now adheres to essential security best practices, reducing the attack surface related to user authentication and data integrity.

##### Enhance Authentication

1. Introduction

This report documents the process of integrating JSON Web Token (JWT) authentication into the OWASP Juice Shop application.  
The purpose of this integration was to allow the backend to issue a signed token upon successful login and enable secure, token-based access to protected resources.

2. Objectives

* Modify the login function to generate JWT tokens after successful authentication.
* Return the token in the API response along with additional user details.
* Ensure the generated token can be used for authorization in future API calls.

3. Tools & Environment

* Platform: Node.js v22.17.1
* Framework: Express.js (within OWASP Juice Shop backend)
* Authentication Library: jsonwebtoken (JWT handling)
* Database Model: Sequelize (BasketModel)
* Testing Tool: Postman

4. Implementation Steps

4.1 Modifying the Login Function

The afterLogin() method was updated to:

1. Verify post-login security challenges.
2. Retrieve or create a basket for the authenticated user.
3. Generate a JWT token containing:
   * id: User ID
   * email: User email
4. Return the token, basket ID, and user email in the JSON response.

Updated Code Snippet:

javascript

CopyEdit

const jwt = require('jsonwebtoken');

function afterLogin(user, res, next) {

verifyPostLoginChallenges(user);

BasketModel.findOrCreate({ where: { UserId: user.data.id } })

.then(([basket]) => {

const token = jwt.sign(

{ id: user.data.id, email: user.data.email },

'your-secret-key',

{ expiresIn: '1h' }

);

user.bid = basket.id;

res.json({

authentication: {

token,

bid: basket.id,

umail: user.data.email

}

});

})

.catch((error) => next(error));

}

4.2 Token Verification Middleware

A reusable middleware (authMiddleware.js) was implemented to:

* Extract the token from the Authorization header.
* Verify it using the same secret key.
* Allow access if the token is valid; otherwise, return an error.

javascript

CopyEdit

const jwt = require('jsonwebtoken');

const SECRET\_KEY = 'your-secret-key';

function verifyToken(req, res, next) {

const authHeader = req.headers['authorization'];

const token = authHeader && authHeader.split(' ')[1];

if (!token) return res.status(401).json({ message: 'No token provided' });

jwt.verify(token, SECRET\_KEY, (err, decoded) => {

if (err) return res.status(403).json({ message: 'Invalid token' });

req.user = decoded;

next();

});

}

module.exports = verifyToken;

4.3 Securing an Example Route

Example of protecting a route with the middleware:

javascript

CopyEdit

const verifyToken = require('./authMiddleware');

app.get('/api/secret-data', verifyToken, (req, res) => {

res.json({

message: `Hello ${req.user.email}, here is your protected data.`,

basketId: req.user.bid || null

});

});

5. Testing Procedure

1. Login Request:
   * Endpoint: /rest/user/login
   * Input: Valid user credentials.
   * Output:

json

CopyEdit

{

"authentication": {

"token": "<JWT\_TOKEN>",

"bid": 6,

"umail": "muhammadadnan12@gmail.com"

}

}

1. Accessing Protected Route:
   * Endpoint: /api/secret-data
   * Header: Authorization: Bearer <JWT\_TOKEN>
   * Response:

json

CopyEdit

{

"message": "Hello muhammadadnan12@gmail.com, here is your protected data.",

"basketId": 6

}

6. Results

* The login flow successfully generates JWT tokens.
* Tokens are valid for 1 hour and contain user ID and email.
* Protected routes reject requests without a valid token.
* Postman testing confirmed secure access control.

7. Conclusion

The integration of JWT authentication in OWASP Juice Shop was successfully implemented.  
This change strengthens the application’s security by requiring valid tokens for sensitive API endpoints.  
Future enhancements could include:

* Storing the secret key in environment variables.
* Implementing refresh tokens for longer sessions.
* Encrypting additional payload data in the token.

## **Helmet.js Implementation Report – OWASP Juice Shop**

### ****1. Objective****

The goal of this task was to enhance the security of the OWASP Juice Shop application by implementing **Helmet.js** to set secure HTTP headers, thereby reducing the risk of certain well-known web vulnerabilities.

### ****2. What is Helmet.js?****

Helmet.js is an Express.js middleware that helps secure web applications by setting various HTTP headers.  
It provides protection against attacks such as:

* **Clickjacking**
* **MIME sniffing**
* **Cross-site scripting (XSS)**
* **Information disclosure via referrer data**

### ****3. Implementation Steps****

1. **Install Helmet.js**

bash

CopyEdit

npm install helmet

1. **Import Helmet in the server file**

ts

CopyEdit

import helmet from 'helmet';

1. **Enable Helmet for the Express app**

ts

CopyEdit

const app = express();

app.use(helmet());

1. **Restart the Juice Shop application**

bash

CopyEdit

npm start

### ****4. Verification****

After adding Helmet:

* Accessed the application in the browser and opened **Developer Tools → Network → Headers**.
* Verified the presence of new HTTP security headers.

**Observed Headers:**

pgsql

CopyEdit

x-content-type-options: nosniff

x-frame-options: SAMEORIGIN

x-dns-prefetch-control: off (if enabled)

referrer-policy: no-referrer (if enabled)

These headers were not all present before Helmet was implemented.

### ****5. Security Benefits****

| Header Added | Protection Offered |
| --- | --- |
| **X-Content-Type-Options: nosniff** | Prevents MIME type sniffing. |
| **X-Frame-Options: SAMEORIGIN** | Prevents clickjacking. |
| **X-DNS-Prefetch-Control** | Prevents browsers from performing DNS prefetching. |
| **Referrer-Policy** | Controls amount of referrer info sent to other sites. |
| **Strict-Transport-Security (HSTS)** | Forces HTTPS usage (works if app is served over HTTPS). |

### ****6. Conclusion****

The implementation of Helmet.js successfully applied additional HTTP security headers to the OWASP Juice Shop application. This increases the overall security posture by mitigating several common attack vectors and fulfills the **“Secure Data Transmission”** challenge in Juice Shop.