



Web Application Penetration Test

Final Report

Prepared for: OWASP Juice Shop

June 16th, 2023

Reference: S-230616904

TABLE OF CONTENTS

TABLE OF CONTENTS	1
EXECUTIVE SUMMARY	2
NARRATIVE AND ACTIVITY LOG	3
FINDINGS AND RECOMMENDATIONS	10
RISK RATINGS	10
FINDINGS SUMMARY	11
CRITICAL RISK FINDINGS	11
1. SQL Injection Flaws	11
2. Authorization Bypass	15
HIGH RISK FINDINGS	17
3. Cross-Site Scripting Flaws	17
4. XML External Entity (XXE)	19
5. Improper Validation and Handling	22
6. Sensitive Information Disclosure	26
MEDIUM RISK FINDINGS	28
7. Weak Password Complexity Requirements	28
8. Username Harvesting	31
LOW RISK FINDINGS	34
9. Lack of Javascript Library Patching	34
STRATEGIC GUIDANCE	36
Provide Secure Coding Training for Developers	36
Consider Multi-Factor Authentication	36
Formalize Application Security Practices and Requirements	36

EXECUTIVE SUMMARY

Secure Ideas performed a penetration test of OWASP Juice Shop's web application. The scope of this assessment, as provided by OWASP Juice Shop, was `http://localhost:3000`.

The following chart shows the count of findings by risk for this report:

Critical	High	Medium	Low
2	4	2	1

Based on the findings in this report, Secure Ideas has evaluated the overall risk to OWASP Juice Shop as it pertains to the scope of this engagement is Very High:



Secure Ideas found multiple critical and high-rated vulnerabilities in the OWASP Juice Shop web application. These weaknesses are very concerning and, if leveraged, would decrease the security and usability of the application.

One of the most significant vulnerabilities Secure Ideas uncovered was the application's susceptibility to various injection-based attacks. To illustrate this vulnerability's severity, a simple SQL injection string was used, enabling Secure Ideas to log in as the application's administrator account without knowing the username or password. Other examples of injection flaws are the instances of Cross-Site Scripting vulnerabilities identified throughout the application. Cross-Site Scripting allows attackers to have malicious code run in the browsers of OWASP Juice Shop users.

Another significant issue discovered is in the form of an authorization bypass flaw. This security flaw allows an attacker to exploit the OWASP Juice Shop API to create a new user with any role, including ones having administrative privileges. This vulnerability stems from a lack of consistent authorization checking within the API. Consistency is important in authorization validation, and the application must enforce it across all interfaces to prevent resources from being unprotected.

These, and the other issues found are outlined in the report that follows. Secure Ideas appreciates the opportunity to work with OWASP Juice Shop to help improve its security posture.

NARRATIVE AND ACTIVITY LOG


Secure Ideas began by working through our standard methodology of Recon → Mapping → Discovery → Exploitation. Since this assessment was not a black box assessment, the team skipped the initial Recon phase, starting with Mapping and Discovery. Throughout the engagement, we conducted several types of activities on each of the web interfaces within the OWASP Juice Shop application. The following list details the high-level activities and considerations carried out during the engagement. This list is not inclusive of every test performed.

- Conducted mapping of the in-scope application
- Evaluated for common web flaws such as:
 - Authentication and session management flaws
 - Authorization bypasses
 - JSON Web Token (JWT) manipulation
 - Cross-Origin Resource Sharing (CORS) misconfigurations
 - Cross-Site Request Forgery (CSRF)
 - Testing for Server-Side Request Forgery (SSRF)
 - Ineffective / misconfigured security controls
 - Injection flaws such as Cross-Site Scripting (XSS) and SQL Injection (SQLi)
 - Fuzzing of HTTP header values
 - Testing for HTTP Desync and Cache poisoning flaws
 - Fuzzing of query and body parameters
 - Client side JavaScript static and dynamic analysis
- Testing for other high-risk items including all testable vulnerabilities listed in the OWASP Top-10

This web-based storefront is designed for selling juice products and presents various functionalities that are common in e-commerce platforms. During the initial mapping phase, the scope of the application has been explored and several key components of interest were identified, allowing for a detailed exploration of the application's functionalities, expected behaviors, and underlying technology stack.

As part of the mapping phase, we explored all of the available functionality within the application using each account role provided. We started by creating a user account with our test email *SampleReport@SecureIdeas.com* and building our profile, which is seen below.

User Profile



Email:
SampleReport@SecureIdeas.com

Username:
H4cker4Hire

Set Username

H4cker4Hire

File Upload:

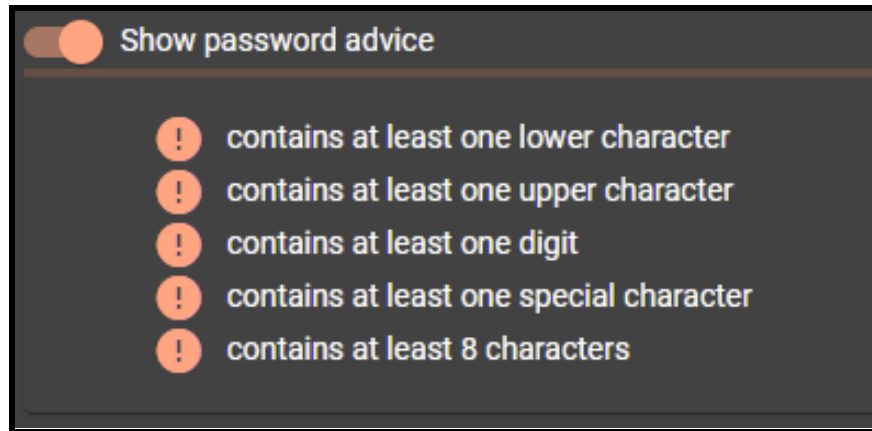
Choose File No file chosen

Upload Picture

_____ or _____

Here we began mapping the application features and potentially vulnerable areas, such as login forms, user profile pages, file uploads or input fields for sensitive information. This process was also repeated under an administrative account, and the differences in roles and access permissions were noted.

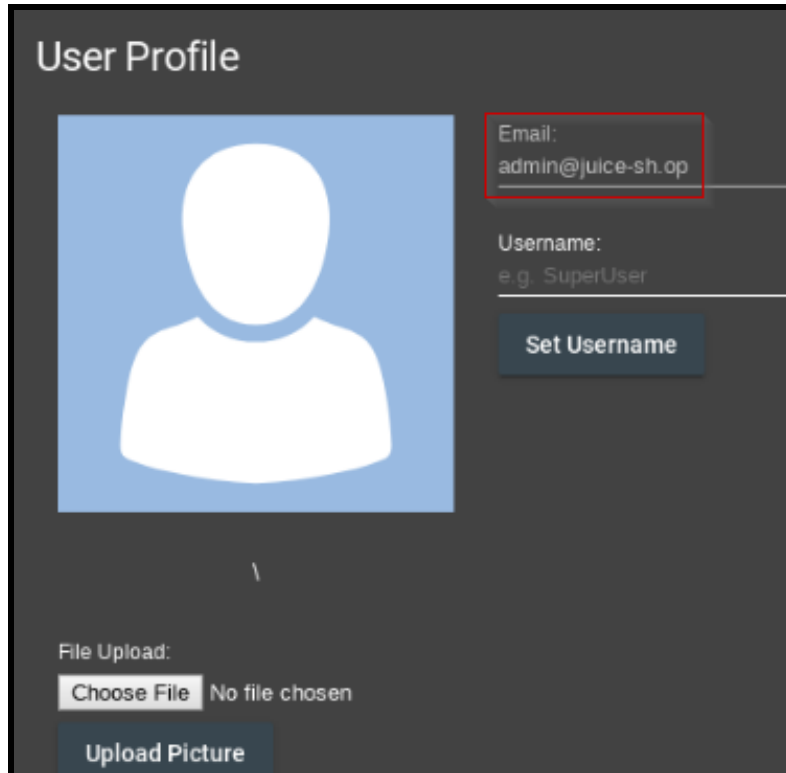
An observation made rather quickly was that a potential security concern existed when creating a user account. The application provides guidelines for password creation, but these recommendations are not only optional to view, they are also not enforced during the account creation process. Despite offering reasonable advice on how to establish a robust password, the application does not mandate compliance with these guidelines. The password advice can be seen here:



As evidence of this lax password policy, we successfully created a user account with the overly simplistic and highly insecure password 12345. The lack of enforced strong password criteria can be a significant risk factor for the application's security, and this finding is further outlined in the *Findings* section of this report.

Next, we walked back through the application again, but this time from the perspective of a malicious user or attacker. Instead of considering the expected actions from normal application usage, we applied various techniques related to intercepting/manipulating outgoing requests or incoming responses, passing malformed data to input fields, and attempting to generate unusual responses from the application.

When examining the login form, we noticed that single quotes could be used to cause errors on the page, which is typically indicative of poor input handling. Additional probing showed that the *Email* field was susceptible to SQL injection. By entering the string `' or 1=1--` in the *Email* field, along with any value in the *Password* field, caused the application to evaluate the login condition as *true*. This authenticated us as the first entry in the Users database table, which is the administrator account. The results can be seen in the screenshot shown below that depicts the profile of the current logged on user:

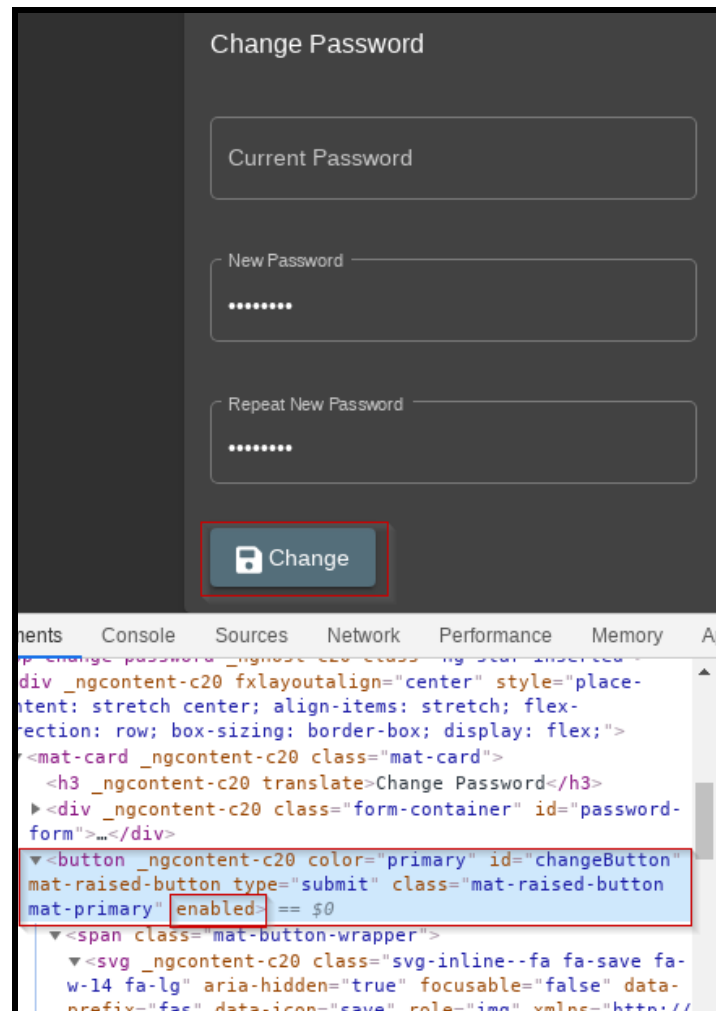
A screenshot of a 'User Profile' form. On the left is a large blue square placeholder for a profile picture. To its right, the 'Email' field contains 'admin@juice-sh.op' and is highlighted with a red rectangle. Below the email field is the 'Username' field with the placeholder text 'e.g. SuperUser' and a 'Set Username' button. At the bottom left, there is a 'File Upload' section with a 'Choose File' button (showing 'No file chosen') and an 'Upload Picture' button.

After observing that this field can be used to inject various types of SQL commands, we began experimenting with different queries to see what other information could be gained in this manner. We discovered that by applying a slight modification to the query, we were able to log into the next account in the application. Using a query such as, ' or 1=1 and email not like('%admin%');-- , we were able to filter out the *admin* account, moving the 'login pointer' to the next account in the database which didn't contain the string *admin*. Using some creativity, and this query as a base template, an attacker could eventually enumerate the *Users* table to harvest every username it contained.

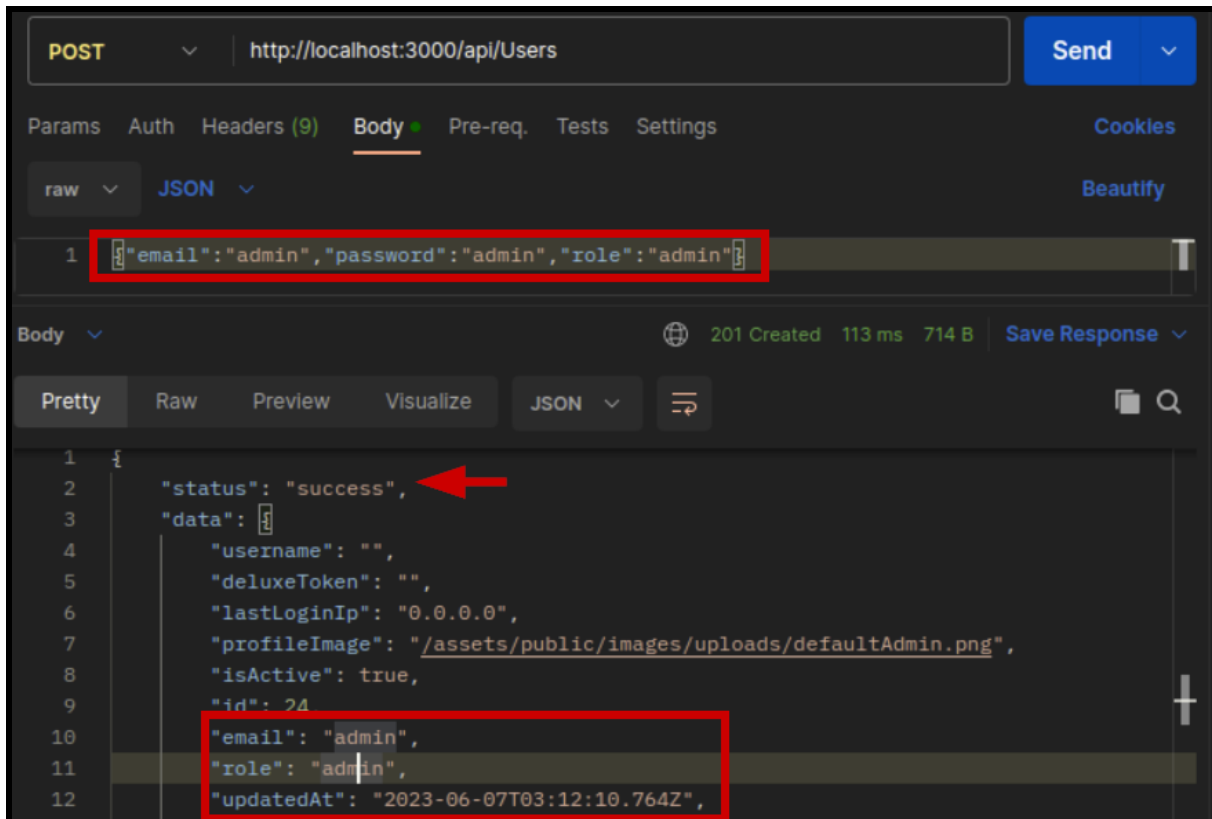
Due to the criticality of the SQLi, which allows an unauthenticated user to bypass the login authorization process, Secure Ideas quickly reached out to the OWASP Juice Shop point of contact. A brief explanation of the flaw was provided, sanitizing any specific information that shouldn't be sent over unsecure email, and a meeting was requested to review the findings discovered within the OWASP Juice Shop application.

While waiting for OWASP Juice Shop to respond, we continued our testing of the application. During this time, another significant issue was discovered, which compounds the risk associated with the SQLi flaw noted above. After logging into the account of the next user in the database, we took some time to inspect the *Change Password* functionality. By default, the *Change* button used to update a user's password is only enabled when the *Current Password*, *New Password*, and *Repeat New Password* fields are populated correctly. However, we discovered that by using the browser's f12 developer tools, a user's password can be updated without knowing the current password.

This was accomplished by manipulating the webpage *submit* action. As seen below, when changing the *mat-raised-button mat-primary* from *disabled* to *enabled*, the password change can be processed by the application, bypassing the *Current Password* field requirements.



Moving on to the API's tested in this engagement, we found that an unauthenticated user could submit a simple POST request, and create users with administrative privileges. In this request, no authentication or tokens are provided, and no cookie values are given. The only requirements found for the creation of an administrative account are the *Content-Type: application/json* header, and a few basic account flags added into the body of the request. As shown in the following screenshot, this POST request was used to successfully create an admin user account, giving it the *admin* role.



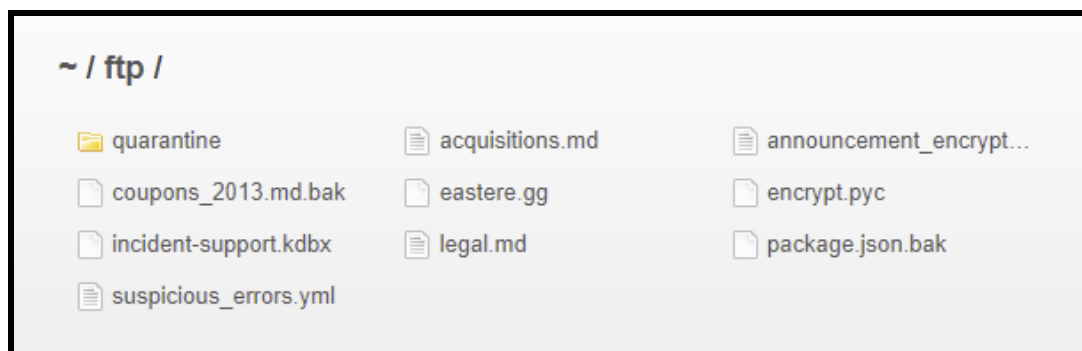
Due to the critical nature of this authorization bypass, another email was sent to OWASP Juice Shop. A meeting was set up to go over these findings immediately as well as get additional direction on considerations for the remainder of this test. Secure Ideas then walked OWASP Juice Shop's technical team through the critical findings discovered, and discussed various options related to the risk and remediation of these items. OWASP Juice Shop's technical team was quick to respond to the input provided, and have already begun a root cause analysis to determine how to best implement fixes in accordance with OWASP Juice Shop's internal policies and processes. Additional effort is going to be spent on reviewing, and determining the best way to address security concerns throughout the application's development process.

Continuing with the testing, we utilized Burp Suite, a widely used proxy tool, to examine the requests and responses while interacting with the application. A peculiar observation was made while investigating the payment options for the deluxe membership offered by the OWASP Juice Shop. Upon scrutinizing the developer tools on the respective page, we found that the *payment* button attribute was marked as *"disabled=true"*. This indicated that the application was appropriately blocking payment attempts due to insufficient wallet balance. Our attempt to purchase a deluxe membership with insufficient funds in the wallet is displayed below.



However, after experimenting with the code and attempting various payloads, we successfully bypassed the payment processing logic. This manipulation allowed us to upgrade our user account to a deluxe membership, despite having no funds in the wallet. This exploit represents a significant vulnerability in the application's payment security, potentially enabling unauthorized premium access. More information regarding this exploit can be found in the *Findings* section of this report.

Next, we embarked on a mission to enumerate the OWASP Juice Shop application's directories using a command-line tool known as *dirb*. Leveraging an extensive wordlist, we probed over 20,000 potential directory names. Among these, eight were accessible for browsing. One notable finding was the directory `http://localhost:3000/ftp`, which provided us with access to a file server. The files visible within this server are illustrated in the screenshot provided below.



A number of these files were easily accessible, and our investigation revealed that some did in fact contain sensitive data. Additional insights concerning this finding are detailed in the *Sensitive Information Disclosure* section of this report.

These, and the other issues discovered are outlined in the report that follows.

FINDINGS AND RECOMMENDATIONS

This report outlines the findings Secure Ideas collected from the testing, as well as Secure Ideas' recommendations that will assist OWASP Juice Shop in reducing its risks and helping remove the vulnerabilities found.

RISK RATINGS

Each finding is classified as a Critical, High, Medium, or Low risk based on Secure Ideas' professional judgment and experience providing consulting services to organizations of various sizes and industries. In determining risk, Secure Ideas considers each of the following aspects:

- **Potential Threats:** This includes an assessment of potential threat actors and the level of expertise
- **Likelihood of Attack:** Considerations include attacker motivations, complexity of the attack vector, and potentially mitigating security controls
- **Possible Impact:** For each finding, Secure Ideas considers the potential damage to the organization resulting from a successful attack

Each of these factors is assessed individually and in combination to determine the overall risk designation. These assessments are based on Secure Ideas' professional judgment and experience providing consulting services to enterprises across the country. The following risk level descriptions demonstrate the types of vulnerabilities designated in each category.

Critical

Vulnerabilities found that are being actively exploited in the wild and are known to lead to remote exploitation by external attackers. These security flaws are likely to be targeted and can have a significant impact on the business. These require immediate attention in the form of a workaround or temporary protection. When discovered, Secure Ideas immediately stops all testing and contacts the client for further instructions. Examples of this may include external-facing systems with known remote code execution exploits or remote access interfaces with weak or default credentials.

High

Vulnerabilities found that could lead to exploitation by internal or remote attackers. These security flaws are likely to be targeted and can have a significant impact on the business. These flaws may require immediate attention for temporary protection, but often require more systemic changes in security controls. Some examples include command injection flaws, use of end-of-life software, and default credentials.

Medium

Vulnerabilities or services found that could indirectly contribute to a more major incident; or that are directly exploitable to an extent that is somewhat limited in terms of availability and/or impact. This class of vulnerability is unlikely to lead to a significant compromise on its own, however can pose a substantial danger when combined with others. Some examples include weak transport layer security on a sensitive transaction, insufficient network segmentation, or the use of vulnerable software libraries.

Low

Vulnerabilities or services that, when found alone, are not directly exploitable and present little risk, but may provide information that facilitate the discovery or successful exploitation of other flaws. Examples include disclosure of server software versions and debugging messages.

FINDINGS SUMMARY

The following table summarizes the findings. Each finding is broken out in detail by risk immediately after the summary table.

Finding	Risk
1. SQL Injection Flaws	Critical
2. Authorization Bypass	Critical
3. Cross-Site Scripting Flaws	High
4. XML External Entity (XXE)	High
5. Improper Validation and Handling	High
6. Sensitive Information Disclosure	High
7. Weak Password Complexity Requirements	Medium
8. Username Harvesting	Medium
9. Lack of Javascript Library Patching	Low

CRITICAL RISK FINDINGS

1. SQL Injection Flaws

Industry Standards

OWASP Top 10	<i>A3:2021: Injection</i>
NIST 800-53	<i>SI-10: Information Input Validation</i>

Summary

When data enters a web application without being properly sanitized, it may expose the application to several categories of vulnerabilities. One of these categories is the injection of Structured Query Language (SQL) by a third party. This type of attack is commonly referred to as SQL injection.

SQL injection occurs when data is inserted or appended into an application input parameter, and that input is used to dynamically construct a SQL query. When a web application fails to properly sanitize data, which is passed on to dynamically create SQL statements, it is possible for an attacker to alter the construction of back-end SQL statements.

Some of the potential risks include:

- Loss of sensitive or confidential data
- Altered sensitive or confidential data
- Bypass of authentication
- Bypass of authorization
- Access to underlying Operating System
- Further attacks against users of the application (XSS, CSRF)

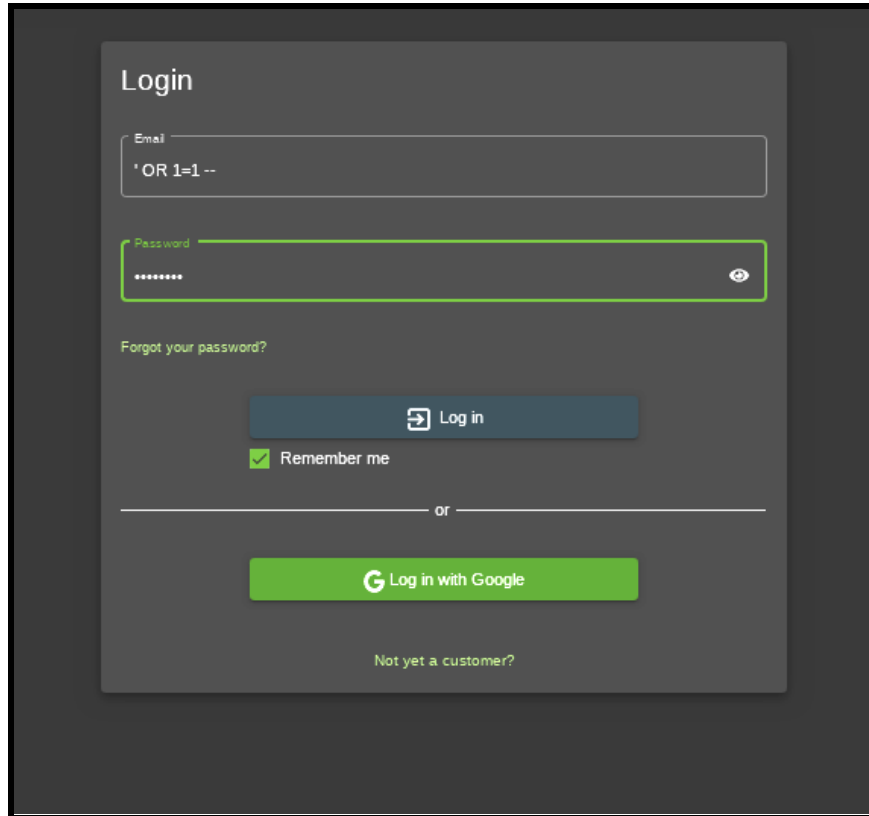
One way to exploit this type of vulnerability is via Blind SQL Injection. Blind SQL injection is identical to a standard SQL Injection attack, except that when an attacker attempts to exploit an application, rather than getting a useful error message, the attacker instead gets a generic page specified by the developer. This makes exploiting a potential SQL Injection attack more difficult but not impossible. An attacker can still gain access to data by asking a series of True and False questions through SQL statements.

Finding

Secure Ideas discovered that the login page of the Juice-shop application is vulnerable to a classic form of SQL Injection as well as Blind SQL Injection. This is due to the use of unsanitized user supplied input.

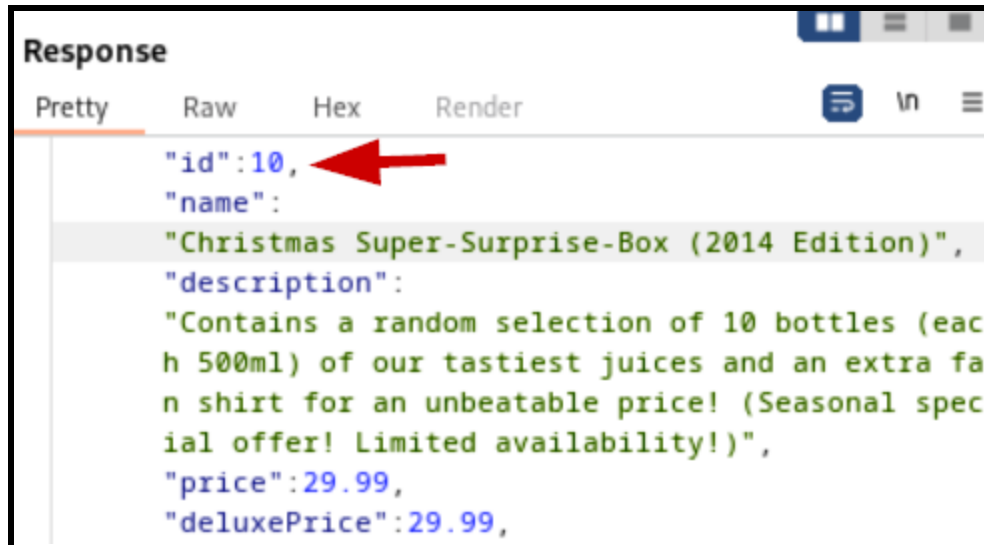
In the first example, using the parameters ' = OR 1=1-- , as the username and any password, Secure Ideas was able to login as the *Admin* account. Considering Admin was the first user listed in the application, it was therefore used due to the exploit payload.

As shown in the following screenshots, *the admin* account was the first account listed in the application. Additional accounts could be accessed by using ' or 1=1 and email not like('%admin%');-- and so on.

A screenshot of a web application's login page. The page has a dark gray background. At the top, the word "Login" is displayed in white. Below it, there is a white input field for "Email" containing the text "' OR 1=1 --". Underneath the email field is a white input field for "Password" with a green border and a green eye icon on the right. Below the password field is a link that says "Forgot your password?". There is a blue "Log in" button with a white arrow icon. Below the button is a checked checkbox labeled "Remember me". A horizontal line with the word "or" in the center separates this section from the Google login option. Below the line is a green "Log in with Google" button with a white "G" icon. At the bottom, there is a link that says "Not yet a customer?".

Next, Secure Ideas was able to perform a Blind SQL Injection by inserting "')-- into the query parameter of a GET request within the product search page of the application. This effectively manipulated the application's SQL command to ignore the original filtering conditions, thereby returning all the products from the database.

This resulted in a *Christmas Super-Surprise-Box (2014 Edition)* being revealed as part of the available products, as seen in the Response below.



The product, in this case the *Christmas Super-Surprise-Box*, is not readily accessible through the application's available product options. Secure Ideas was then able to use the intercept functionality in Burp Suite to add a Green Smoothie to the shopping basket. Intercepted requests were forwarded until the POST request responsible for adding items to the basket is reached. At this point, Secure Ideas was able to manipulate the Javascript in the request, replacing the ProductId of the "Green Smoothie" with the ProductId of the *Christmas Super-Surprise-Box*, which is '10'. Upon forwarding the modified request and revisiting the basket, Secure Ideas confirmed that the shopping cart had now added the unavailable item.

Recommendations

Secure Ideas recommends that OWASP Juice Shop use parameterized queries when interacting with a database backend. Parameterized queries are a method where the query is created within the application code without the values needed. Placeholders are used and during execution replaced with the values from the user or the transaction. Currently parameterized queries are the strongest protection from SQL injection attacks.

If for some reason, parameterized queries are not possible, Secure Ideas recommends that OWASP Juice Shop perform input validation to prevent this form of attack. Developers should ensure that the application validates that the input from the user is of exactly the type that the developer intends. For example, if OWASP Juice Shop only expects alphanumeric characters in the input, then the application should perform input filtering to reject anything else. This is considered a whitelist approach.

Further, Secure Ideas recommends that OWASP Juice Shop properly handle all SQL statements, and commands within the code so that DBMS error messages are not returned directly to the browser.

OWASP Juice Shop developers can also use a common security library to perform input filtering and output encoding. Implementations should follow OWASP best practices for preventing this vulnerability,

regardless of whether or not OWASP Juice Shop chooses to use a library for these tasks.

https://owasp.org/www-project-cheat-sheets/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

2. Authorization Bypass

Industry Standards	
OWASP Top 10	A1:2021: Broken Access Control
NIST 800-53	AC-3: Access Enforcement

Summary

Authorization bypass is a flaw in software or a hole in security planning where a user or an attacker is able to access data or functionality for which the user is not authorized. This vulnerability does not require a malicious attacker to cause increased risk to a business; the mere fact that unauthorized users have access to a business infrastructure increases risks to the company. The core issue in authorization bypass is a lack of validation within the application. When the web application accepts input from a user and uses that input to retrieve data or provide access, it is critical that the application validate that the user actually has permission to perform that action. When this validation does not happen, or is able to be fooled, the application is vulnerable to attack.

Risks businesses face from an authorization bypass include the introduction of bugs to code via users' mistakes, an attacker gaining access to administrative portions of the application, or loss of important business-related data to a data thief.

Finding

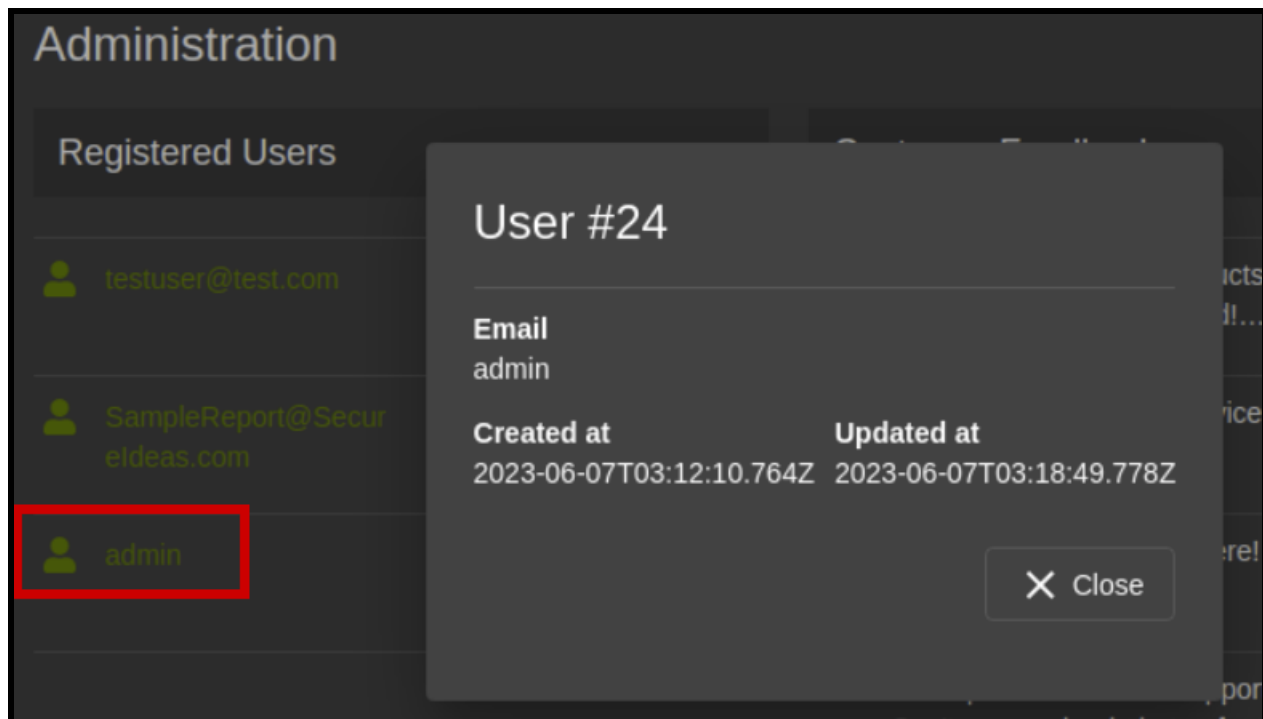
Secure Ideas has found that the OWASP Juice Shop application contains an authorization bypass flaw. During the testing Secure Ideas was able to create an admin account with an unauthenticated session.

In the OWASP Juice Shop API, Secure Ideas discovered that an attacker or malicious user could create a new user with the role of admin.

The following description explains how Secure Ideas was able to perform this attack.

1. Create a Post request in Postman API testing tool to *https://localhost:3000/api/Users*
2. Add a line in the Body of the request using the following statement
`{"email":"admin","password":"admin","role","admin"}`
3. Send Request to the api endpoint
4. Visit login page to login using new admin account

As shown below, the new user has been created with administrative privileges:



Recommendations

Secure Ideas recommends the authorization bypass flaw be remediated immediately due to the exposure of administrative access via the API.

The first step in remediating this flaw involves changing the application to validate authorization information. OWASP Juice Shop must modify the code of the application to verify that a user is allowed to view the information before returning it to the browser. If the user is not authorized, the application should return an error message instead of the information requested.

The second step is to never trust user supplied input, or expect that the client side code is protected from manipulation. Every authorization should be validated by backend services, and exposure of this validation process should be hidden as much as possible to any client side process. This will help ensure that any user input is handled safely.

The next step is to include a logging and monitoring system within the application to detect attempts to access other members' information. These logs can then be reviewed to determine if someone is attempting to attack the application.

Any time a user attempts to access information or functionality that is restricted from them, the application can alert staff members of the attempt. This can be performed in a number of ways. The simplest is the application sends an email or SMS message to OWASP Juice Shop support staff. OWASP

Juice Shop could also modify the application to send messages to a central monitoring solution, if one has been implemented within the OWASP Juice Shop infrastructure. If modifying the application in this way is not preferred, OWASP Juice Shop can also use a script that parses the log files for messages of exploitation attempts. The script can then perform the action chosen to alert the OWASP Juice Shop staff.

HIGH RISK FINDINGS

3. Cross-Site Scripting Flaws

Industry Standards	
OWASP Top 10	A1:2021: Broken Access Control
NIST 800-53	AC-3: Access Enforcement

Summary

Not filtering untrusted user-supplied input may expose a web application to several categories of vulnerabilities. One of these categories is the injection of HTML or JavaScript code by a third party. This type of attack has been generally referred to as “Cross-Site Scripting” or XSS.

One common way of exploiting this is with a social-engineering attack vector and a crafted link. This would exploit a flaw in one or more parameters in the URL and query string. When the target user follows the link, the malicious code executes in the target’s browser, within the context of the vulnerable page.

Cross-site scripting flaws are typically classified by two attributes: whether they are persisted and whether they are reflected. When a persisted exploit is used, the payload is stored, and executes again on subsequent visits to the vulnerable page. The classic example is server-side persistence in the database. Because the data in the database may be shared between users, it is possible for an attacker to simply add the payload through a shared data field in order to circumvent the need for social engineering. This is predicated on the attacker being able to add the payload from either a legitimate account or an unauthenticated context. Even when social engineering is necessary to introduce the payload, if it is in shared data it can still reach other users in addition to the original target. Persistence is not necessarily always on the server, however, and could instead be stored in cookies set by JavaScript. In more modern applications, the *localStorage* and *indexedDB* client-side APIs may be used as well.

The other attribute used for classification is whether it is a reflected flaw. If it is reflected, the flaw is in the handling of input that is sent to the server and returns in a response. The database-persisted example does this, and could therefore be considered both reflected and

persisted. An unpersisted example would be an error message returned from the server that unsafely includes a value from the input.

In all cases, the malicious scripts are executed in a context that appears to have originated from the targeted site. This gives the attacker full access to the document retrieved, providing almost unlimited control over the victim's experience using the application. A wide variety of options are available for crafting an effective exploit, which may incorporate some of the following:

- Sending application data to a server controlled by the attacker
- Using the victim's session to access additional data or functionality
- Stealing cookies that are not protected with the *httponly* flag
- Manipulating the view presented to the victim for a social engineering purpose, such as faking a session timeout to prompt for a login or convincing the user to install something
- Stealing data from sensitive input boxes, such as account credentials or credit card information
- Launching attacks against or harvesting data from other applications open to interaction with the current domain through a cross-origin resource sharing (CORS) policy, potentially using the victim's cookie-stored credentials
- Changing links on the page to include the cross-site scripting payload in other pages as the user navigates the site

Finding

Secure Ideas discovered that OWASP Juice Shop's applications are vulnerable to cross-site scripting (XSS) due to the application's use of input within the response to the user. Many of the flaws identified were persisted through the database, and many could be exploited by an unauthenticated attacker without relying on a direct social engineering attack such as phishing.

One instance of a Cross-Site Scripting (XSS) vulnerability that Secure Ideas observed can be found in the application's search functionality, which is susceptible to these types of attacks. An attacker can execute JavaScript in the context of the application's webpage by injecting script code through the search bar.

A payload `<iframe src="javascript:alert('XSS_SecureIdeas')">` was entered into the search bar of the OWASP Juice Shop application. Upon submission of the input, the JavaScript payload was executed, demonstrating that the user input was not properly sanitized or escaped before being processed by the application. The successful execution of the payload indicates a Document Object Model (DOM) XSS vulnerability. The screenshot below is what the victim browser would see.



Recommendations

Secure Ideas recommends that OWASP Juice Shop perform both input validation, and output encoding to prevent this form of attack. Developers should ensure that the application validates that the input from the user is of exactly the type that the developer intends. For example, if OWASP Juice Shop only expects alphanumeric characters in the input, then the application should perform input filtering to reject anything else. Output encoding provides additional protection by ensuring that hostile data, such as JavaScript, will not be sent to the browser. This way if an attack gets past the input filtering, it would be defanged or made non-malicious by the output encoding.

OWASP Juice Shop developers can use a common security library to perform this input filtering and output encoding. Implementations should follow OWASP best practices for preventing this vulnerability, regardless of whether or not OWASP Juice Shop chooses to use a library for these tasks. These recommendations can be found at:

https://owasp.org/www-project-cheat-sheets/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

4. XML External Entity (XXE)

Industry Standards	
OWASP Top 10	A5:2021: Security Misconfiguration
NIST 800-53	SI-15: Information Output Filtering

Summary

The software processes an XML document that can contain XML entities with URIs that resolve to documents outside of the intended sphere of control, causing the product to embed incorrect documents into its output. Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or

integrations. These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks. The business impact depends on the protection needs of all affected applications and data.

Finding

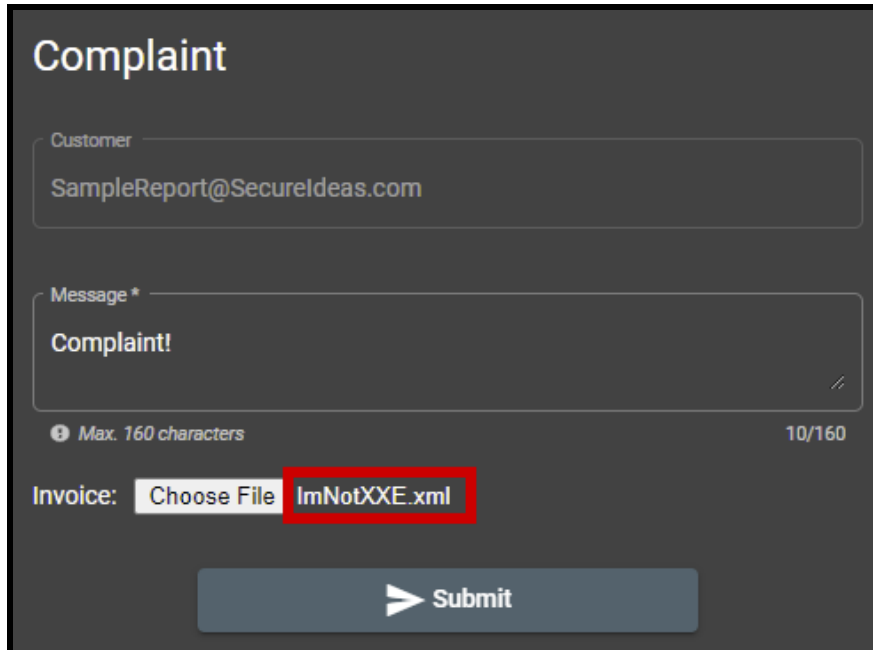
Secure Ideas found that the OWASP Juice Shop application is receiving untrusted XML and may be vulnerable to XML External Entity (XXE) attacks. An XML payload was generated to exploit an XML External Entity (XXE) vulnerability, which allowed access to and extraction of data from the system.ini file on the server. The POC payload is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo [<!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///C:/Windows/system.ini" >]>

<trades>
  <metadata>
    <name>Secure Ideas</name>
    <trader>
      <foo>&xxe;</foo>
      <name>K. Johnson</name>
    </trader>
    <units>24</units>
    <price>13.99</price>
    <name>Picanha</name>
    <trader>
      <name>Jason Gillam</name>
    </trader>
    <units>CAD</units>
    <price>buck</price>
  </metadata>
</trades>
```

This was accomplished by uploading the crafted XML file at <http://localhost:3000/#/complain> through the *Complaint* dialog box, using the *Invoice: Choose File* feature. This is demonstrated in the screenshot below.



Upon submitting the complaint, the server processed the XML payload, and the response inadvertently disclosed the content of the `system.ini` file. This demonstrates the server's susceptibility to XXE attacks, as it allowed the XML parser to process external entities defined within the XML document, thus revealing internal system information.

The potential for a security risk arises from the fact that the XML External Entity (XXE) attack was able to access and extract information from the `system.ini` file. If the XXE attack could access this file, it might also be able to access other files that contain more sensitive data.

The `system.ini` file is a configuration file in Windows that is primarily used to manage the system settings in the Windows 3.x and Windows 9x series of operating systems. The security risk here is less about the specific information in these visible sections of the `system.ini` file, and more about the broader file access capabilities that the successful XXE attack demonstrates. The extracted data is shown below.

```

Response
Pretty Raw Hex Render
1 HTTP/1.1 410 Gone
2 Access-Control-Allow-Origin: *
3 X-Content-Type-Options: nosniff
4 X-Frame-Options: SAMEORIGIN
5 Feature-Policy: payment 'self'
6 X-Recruiting: /#/jobs
7 Content-Type: text/html; charset=utf-8
8 Vary: Accept-Encoding
9 Date: Thu, 22 Jun 2023 04:05:07 GMT
10 Connection: close
11 Content-Length: 5385
12
13 <html>
14   <head>
15     <meta charset='utf-8'>
16     <title>
      Error: B2B customer complaints via file upload have been deprecated for
      security reasons: &lt;?xml version='1.0'>
      encoding='UTF-8'>&lt;!DOCTYPE foo [&lt;!ELEMENT foo
      ANY&gt;&lt;!ENTITY xxe SYSTEM
      &quot;file:///C:/Windows/system.ini&quot;&gt;&lt;trades&gt;&lt;metad
      ata&gt;&lt;name&gt;Secure Ideas&lt;/name&gt;&lt;trader&gt;&lt;foo&gt;;
      for 16-bit app
      support[386Enh]woafont=dosapp.fonEGA80WOA.FON=EGA80WOA.FONEGA40WOA.FON=EG
      A40WOA.FONCGA80WOA.FON=CGA80WOA.FONCGA40WOA.FON=CGA40WOA.FON[drivers]wave
      =mmdrv.dlltimer=timer.drv[mci]&lt;/foo&gt;&lt;name&gt;K. Johnson&lt;/n...
      (ImNotXXE.xml)
    </title>
  </head>
  <body>
  </body>
</html>

```

Recommendations

Secure Ideas recommends that OWASP Juice Shop Implement positive (“whitelisting”) server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes. Disabling both XML external entity and DTD processing and in all XML parsers in the application is also recommended. Developers should also verify that XML or XSL file upload functionality validates incoming XML using XSD validation or replace the processor with a library that does . For python, we recommend defusedxml (<https://pypi.org/project/defusedxml/>)

5. Improper Validation and Handling

Industry Standards	
OWASP Top 10	A04:2021: Insecure Design
NIST 800-53	SI-10: Information Input Validation

Summary

Improper validation and handling is a flaw where an application doesn't adequately check or manage the data it's working with. This can relate to user input, system processes, or interaction with resources. Essentially, it's a failure of the application to ensure data or actions are valid and appropriate, which can lead to a range of security issues.

Improper validation and handling can allow both accidental and intentional unauthorized activity. Even without malicious intent, the simple occurrence of unvalidated or improperly handled data can pose significant risks to a business. The root cause of such vulnerabilities is typically a lack of adequate validation within the application's functionality.

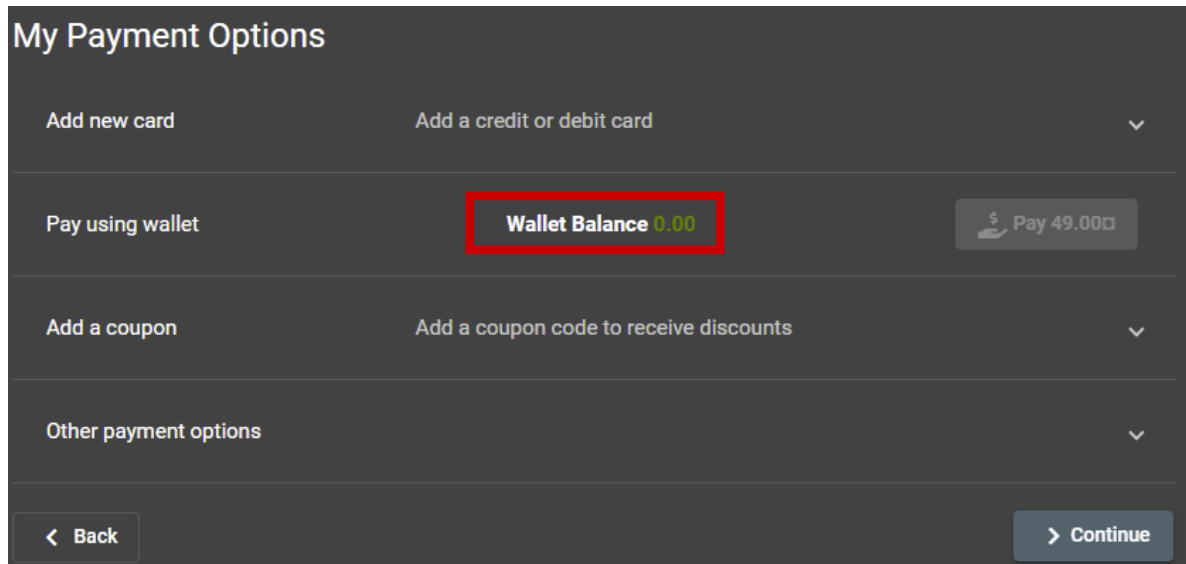
When an application takes input from a user to perform actions or retrieve data, it's crucial that the application confirms the input and the action are legitimate and safe. If the application fails to conduct these checks, or if the checks can be bypassed, it becomes vulnerable.

Potential risks from improper validation and handling include inadvertent introduction of bugs into the system, an attacker manipulating the application's functionality in unintended ways, or exposure of sensitive data to unauthorized individuals. These risks underline the importance of thorough validation and error handling mechanisms in maintaining the security and integrity of an application.

Finding

Secure Ideas has found that the OWASP Juice Shop application contains an improper validation and handling flaw. During testing, Secure Ideas found that a Deluxe Membership upgrade was possible without payment.

Initially, when browsing to <http://localhost:3000/#/payment/deluxe> the payment options were displayed correctly, with a noted account balance of \$0 in the wallet, as seen in the accompanying screenshot.



Through an examination of the webpage using developer tools, Secure Ideas observed that the payment button attribute was set to "disabled=true", suggesting that the system was correctly disallowing payment attempts due to the absence of funds in the wallet. The relevant code snippet and the corresponding screenshot are included below.



Next, the "disabled" attribute from the payment button was manually removed, which enabled the payment button to be clicked. As a result, a POST request was initiated, which predictably returned an error message, "Insufficient funds in wallet". This behavior is appropriate as the "paymentMode" was set to "wallet".

Subsequently, the request was captured using the Burp Suite Repeater tool. In this intercepted request, the "paymentMode" parameter was manipulated by replacing "wallet" with an empty string, and then the request was resent.

This resulted in a successful upgrade to the deluxe membership, confirmed by the application's response - "Congratulations! You are now a deluxe member!" Despite the wallet balance being

6. Sensitive Information Disclosure

Industry Standards	
OWASP Top 10	A5:2021: Security Misconfiguration
NIST 800-53	PE-19: Information Leakage

Summary

Many systems deal with various degrees of sensitive information, such as usernames, passwords, email addresses, phone numbers, confidential documents, and much more. What is considered sensitive information is typically contextual to the system's purpose. However, in cases in which non-public information can be accessed by unauthorized users, there is risk present to the company and its users. Such information can be very useful to an attacker in numerous ways, in some cases allowing attackers to gain a foothold within a company's network, cloud infrastructure, or other resources depending upon the nature of the information disclosed.

Sensitive information disclosure does not typically require an attacker to have bypassed some security control, instead it is typically caused by some mishandling of the information by the system, or a misconfiguration that would allow the information to be accessed in a context in which it should not have been.

The following is a non-exhaustive list of some common examples of sensitive information disclosure:

- Social Security Numbers
- Sensitive documents in a public file share
- Employee identification numbers and other types of identifiers
- Database connection strings or any other types of credentials
- Telephone numbers

Finding

During testing, Secure Ideas effectively employed *dirb* to probe for concealed directories. This effort led to the discovery of several directories that were not initially evident during the application mapping stage, as illustrated in the subsequent screenshot.

```

URL_BASE: http://172.23.48.1:3000/
WORDLIST_FILES: /usr/share/dirb/wordlists/big.txt

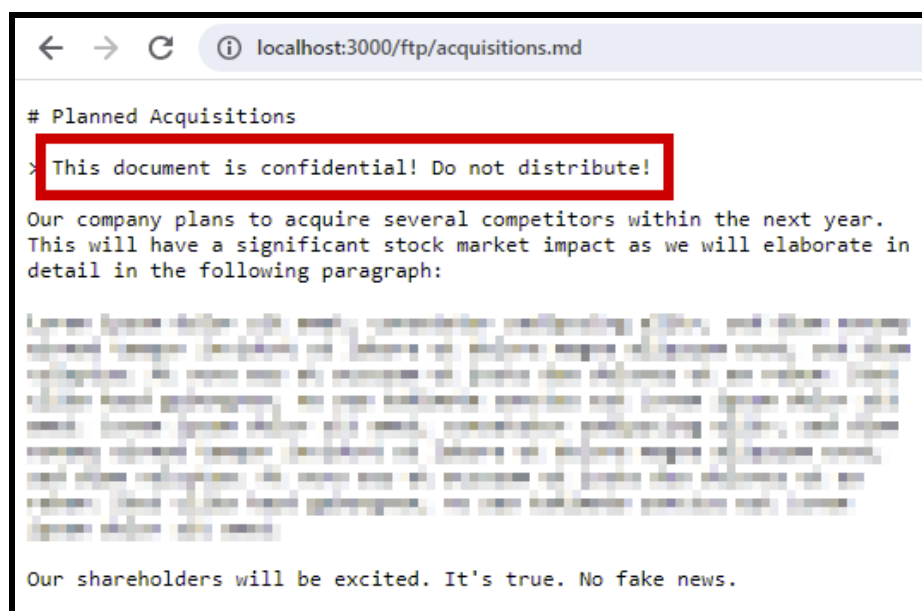
_____ acquisitions.md
easere gg

GENERATED WORDS: 20459


— Scanning URL: http://172.23.48.1:3000/ —
+ http://172.23.48.1:3000/assets (CODE:301|SIZE:179)
+ http://172.23.48.1:3000/ftp (CODE:200|SIZE:11072)
+ http://172.23.48.1:3000/metrics (CODE:200|SIZE:22797)
+ http://172.23.48.1:3000/profile (CODE:500|SIZE:1218)
+ http://172.23.48.1:3000/promotion (CODE:200|SIZE:6690)
+ http://172.23.48.1:3000/redirect (CODE:500|SIZE:4329)
+ http://172.23.48.1:3000/robots.txt (CODE:200|SIZE:28)
+ http://172.23.48.1:3000/secci (CODE:400|SIZE:4286)
+ http://172.23.48.1:3000/snippets (CODE:200|SIZE:707)
+ http://172.23.48.1:3000/support/logs (CODE:200|SIZE:7780)
+ http://172.23.48.1:3000/video (CODE:200|SIZE:10075518)
+ http://172.23.48.1:3000/Video (CODE:200|SIZE:10075518)

```

When browsing to `http://localhost:3000/ftp`, it was discovered that there was visibility into several of the files on this server. The file labeled `acquisitions.md` contains sensitive information about upcoming acquisitions that are clearly confidential, as evident in the screenshot provided below. Secure Ideas has obfuscated the sensitive information to maintain privacy.



Additionally, Secure Ideas found a directory at `http://localhost:3000/support/logs` which contains the `access.log` file corresponding to today's activity. This log provides a comprehensive record of all the requests made to the server. Each request contains potentially sensitive information, including client IP addresses and user agent data. It's important to note that such information could be leveraged maliciously by potential attackers, underscoring the need for proper handling and protection of these log files. The accessible directory and `access.log` file is displayed below.

~ / support / logs		
Name	Size	Modified
 access.log.2023-06-21	...8615	PM 10:09:09 6/21/2023

Recommendations

Secure Ideas recommends that OWASP Juice Shop examine these files thoroughly and evaluate whether all of the disclosed information is indeed sensitive and determine which elements should rightfully be present. Consider restricting the access to said information by avoiding coding practices that might expose file system details. For example, avoid displaying directory listings, and avoid using predictable file and directory names.

MEDIUM RISK FINDINGS

7. Weak Password Complexity Requirements

Industry Standards	
OWASP Top 10	A1:2021: Broken Access Control A5:2021: Security Misconfiguration A7:2021: Identification and Authentication Failures
NIST 800-53	AC-3: Access Enforcement

Summary

One of the aspects tested during the penetration test, was the password complexity requirement of the OWASP Juice Shop applications. For most applications, the password is the single factor of authentication that grants access to all other information. For this reason, it is imperative that users create strong passwords that are difficult to attack. Unfortunately, most users do not understand the importance of strong passwords or how to create them. Application developers

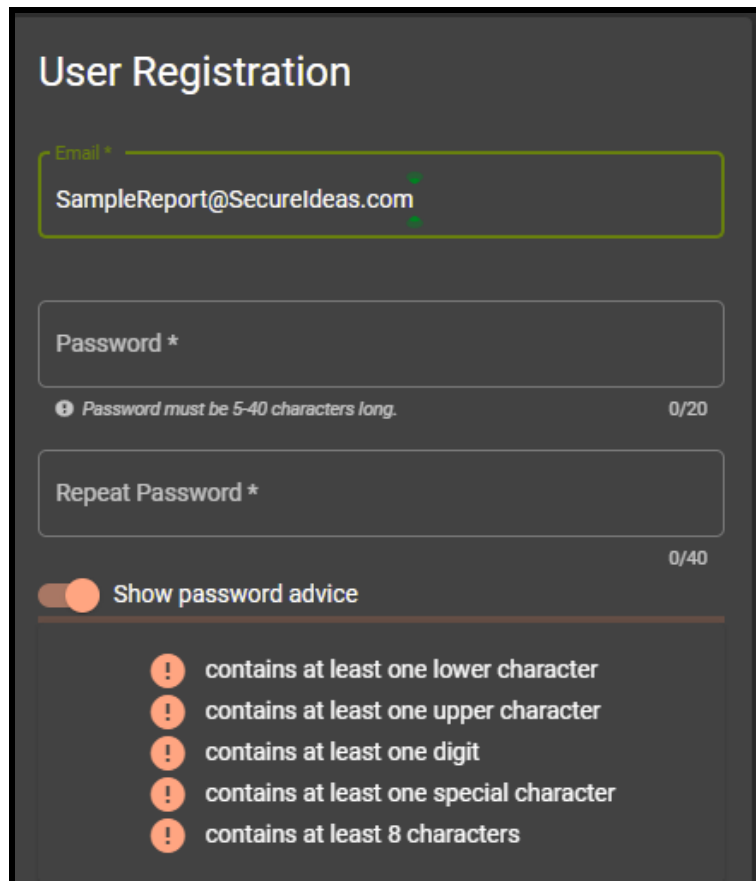
must take the responsibility to develop applications in such a way that requires users to create passwords that can withstand common password-guessing attacks.

There are three common types of password guessing attacks. The first is a brute-force attack in which attackers try every combination of every letter in order to eventually find the correct password. Dictionary attacks utilize a list of common passwords such as *Password1* and *abc123*. The third type of attack is a hybrid attack in which the attacker uses common passwords that have been mangled with brute-force techniques. For instance, the attacker might try the word *Secret* followed by every possible 2-digit numeral and symbol combination. This can be successful when users tack on numbers and symbols to the end of their password to comply with password requirements.

Finding

Secure Ideas found that although the OWASP Juice Shop application does attempt to provide guidelines for secure passwords, it does not adequately enforce the use of complex password configurations. The password complexity criteria are weaker than recommended for an application of this type.

The *User Registration* page shown below displays password advice, however, Secure Ideas found that the application gives this advice as optional, and does not enforce this advice.



User Registration

Email *
SampleReport@SecureIdeas.com

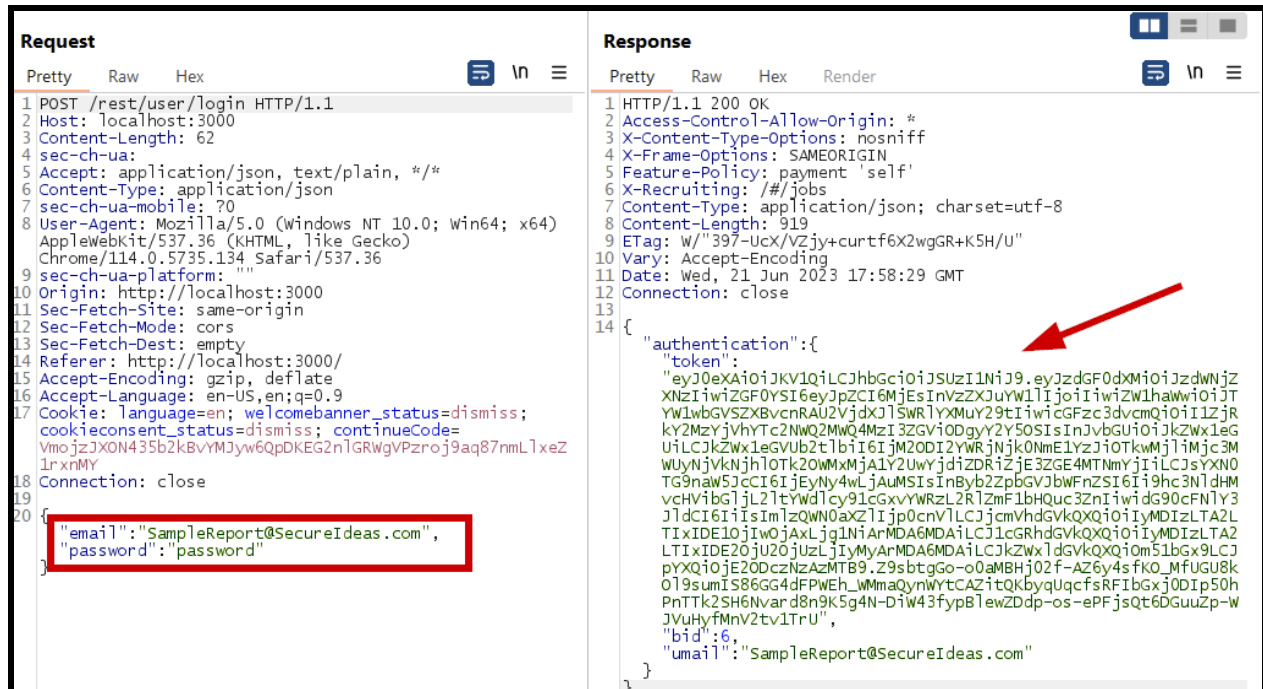
Password *
Password must be 5-40 characters long. 0/20

Repeat Password *
0/40

☒ Show password advice

- contains at least one lower character
- contains at least one upper character
- contains at least one digit
- contains at least one special character
- contains at least 8 characters

Further investigation revealed that the application permits the use of simple passwords such as "password" and "12345". Passwords of this nature are typically included in readily available dictionaries. It is worth noting that such simple password strings are often tested against systems with account lockout mechanisms due to their frequent occurrence as account passwords in web applications. The screenshot provided below displays a generated authentication token and successful login for the user *SampleReport@SecureIdeas.com*, whose password has been entered as *password*.



Recommendations

OWASP Juice Shop should strengthen the password requirements within the application. While it does perform some complexity checking, OWASP Juice Shop should increase these checks based on industry standards. Passwords should contain at least fifteen alphanumeric characters, with preference given to passphrase. Additionally, passwords should avoid any of the following flaws:

- Contains less than fifteen characters.
- The password is a word found in a dictionary (English or foreign).
- Contain personal information such as birth dates, addresses, phone numbers, or names of family members, pets, friends, and fantasy characters.
- Contain common simple patterns such as `aaabbb`, `qwerty`, `zyxwvuts`, or `123321`.
- Are some version of `Welcome123`, `Password123`, or `Changeme123`.
- Any of the above spelled backwards.
- Any of the above preceded or followed by a digit (e.g., `secret1`, `1secret`).

Secure Ideas also recommends that OWASP Juice Shop review options for adding multi-factor authentication to the application. Traditional user credentials are easily reused by attackers once stolen. By applying a second form of authentication, such as something the user knows, something the user is, or something the user has, OWASP Juice Shop can ensure that user accounts are much more difficult to compromise. In addition, less complex passwords pose less risk when additional authentication factors are required.

Finally, OWASP Juice Shop should review the application's logging to confirm that failed login attempts are recorded into an error log. Logs should be reviewed daily to detect user accounts with a higher than normal amount of failed login attempts.

8. Username Harvesting

Industry Standards	
OWASP Top 10	A5:2021: Security Misconfiguration
NIST 800-53	CA-6: Security Authorization

Summary

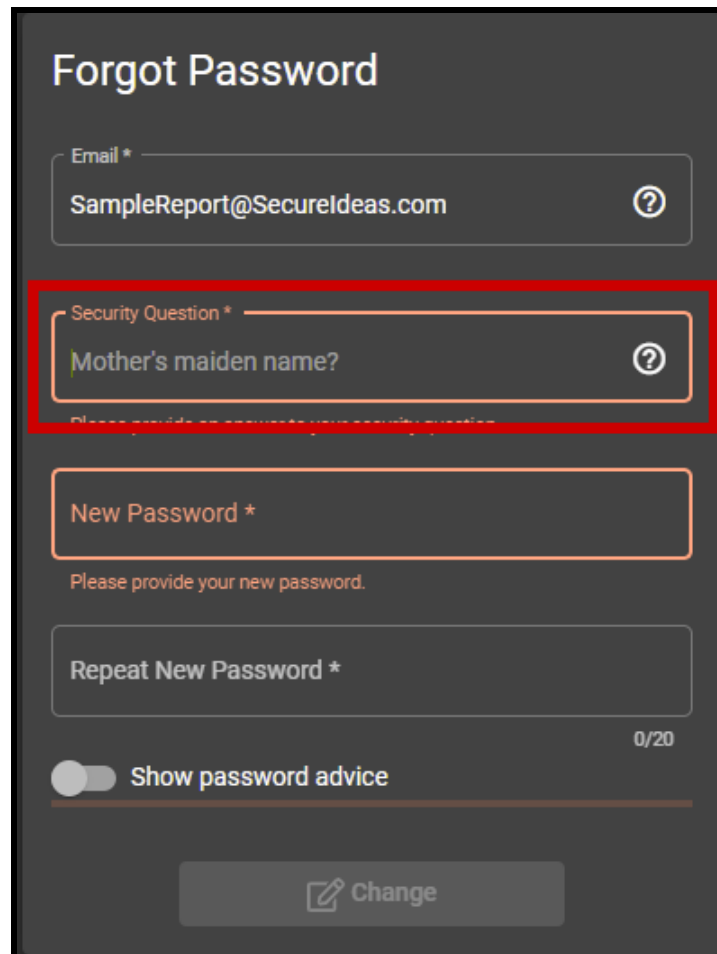
Username harvesting is a flaw that allows an attacker to verify that a username is valid and in use within the system. This is often found on forms such as login, registration, and forgot password. It is caused by the system reacting somehow differently for a valid username when compared to an invalid username. Attackers look for these differences in handling when attacking systems. Often the difference is an obvious error message that tells the user they have entered an invalid or valid username. Other times the flaw can arise through subtleties in the way that a site processes the submission and returns the response.

Once an attacker discovers this flaw, they can use various tools and scripts to harvest large lists of valid usernames. With a list of usernames, the attacker has several options for attacking OWASP Juice Shop's systems and users. One option is to attempt logging into these harvested accounts using common passwords or brute force techniques. Another option is to use the harvested usernames to lock-out large swaths of users, in effect performing a denial of service attack. Attackers could also leverage the usernames in a social engineering attack. Usernames provide a valid piece of information when attempting to social engineer either customers or staff members of OWASP Juice Shop systems. This is even more of a risk when the username is also the user's email address.

Finding

During the testing process, Secure Ideas discovered a Username Harvesting vulnerability was discovered in OWASP Juice Shop. The application was found to be providing different responses when attempting to reset passwords for existing users versus non-existing users.

When a password reset was initiated for an existing user, the application responded by allowing the input for a security question, in this case Mothers maiden name, as seen in the screenshot below.



Forgot Password

Email *
SampleReport@SecureIdeas.com

Security Question *
Mother's maiden name?

New Password *

Repeat New Password *

0/20

Show password advice

Change

Conversely, when attempting the same action with a non-existent user, the OWASP Juice Shop application does not prompt for any security questions and does not provide any input fields to enter such information.

Analysis of the related requests and responses using the Burp Suite proxy tool reveals this disparity. The application responds by prompting for a security question when a request is made with the valid email address SampleReport@SecureIdeas.com. In contrast, when a request is associated with a non-existent user, such as fakeuser@test.com, the response received comprises an empty body, indicating that no data is present. This difference in responses can be exploited by an attacker to validate the existence of a specific username within the system. A screenshot of these differences is shown below.

Request	Response
<pre> 1 GET /rest/user/security-question?email= SampleReport@SecureIdeas.com HTTP/1.1 Host: localhost:3000 2 3 sec-ch-ua: 4 Accept: application/json, text/plain, */* 5 sec-ch-ua-mobile: ?0 6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.134 Safari/537.36 7 sec-ch-ua-platform: "" 8 Sec-Fetch-Site: same-origin 9 Sec-Fetch-Mode: cors 10 Sec-Fetch-Dest: empty 11 Referer: http://localhost:3000/ 12 Accept-Encoding: gzip, deflate 13 Accept-Language: en-US,en;q=0.9 14 Cookie: language=en; welcomebanner_status= dismiss; cookieconsent_status=dismiss; continueCode= VmojzJXON435b2kBVYmJyw6QpDKEG2nIGRWgVPzroj9a q87nmLlxeZ1rxnMY 15 Connection: close 16 </pre>	<pre> 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 X-Recruiting: /#/jobs 7 Content-Type: application/json; charset=utf-8 8 Content-Length: 134 9 ETag: W/"86-t44KL9+SmYvURfc8UpGdncLOkp0" 10 Vary: Accept-Encoding 11 Date: Wed, 21 Jun 2023 16:58:16 GMT 12 Connection: close 13 14 { "question":{ "id":2, "question":"Mother's maiden name?", "createdAt":"2023-06-21T15:13:43.937Z", "updatedAt":"2023-06-21T15:13:43.937Z" } } </pre>

Valid User SampleReport@SecureIdeas.com Request and Response

Request	Response
<pre> 1 GET /rest/user/security-question?email= fakeuser@test.com HTTP/1.1 Host: localhost:3000 2 3 sec-ch-ua: 4 Accept: application/json, text/plain, */* 5 sec-ch-ua-mobile: ?0 6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.134 Safari/537.36 7 sec-ch-ua-platform: "" 8 Sec-Fetch-Site: same-origin 9 Sec-Fetch-Mode: cors 10 Sec-Fetch-Dest: empty 11 Referer: http://localhost:3000/ 12 Accept-Encoding: gzip, deflate 13 Accept-Language: en-US,en;q=0.9 14 Cookie: language=en; welcomebanner_status= dismiss; cookieconsent_status=dismiss; continueCode= VmojzJXON435b2kBVYmJyw6QpDKEG2nIGRWgVPzroj9a q87nmLlxeZ1rxnMY 15 Connection: close 16 17 </pre>	<pre> 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 X-Recruiting: /#/jobs 7 Content-Type: application/json; charset=utf-8 8 Content-Length: 2 9 ETag: W/"2-vyGp6PvFo4RvsFtPoIWeCReyIC8" 10 Vary: Accept-Encoding 11 Date: Wed, 21 Jun 2023 17:00:11 GMT 12 Connection: close 13 14 { } </pre> <p style="text-align: center; color: red; font-weight: bold;">Empty</p>

Non-Existent User fakeuser@test.com Request and Response

Recommendations

Secure Ideas recommends that whenever possible, OWASP Juice Shop should return the same response whether or not a username exists within the authentication system. In situations such as account registration, when the user needs to be notified that a username is already in use, the application should employ other defensive techniques such as rate-limiting.

LOW RISK FINDINGS

9. Lack of Javascript Library Patching

Industry Standards	
OWASP Top 10	<i>A6:2021: Vulnerable and Outdated Components</i>
NIST 800-53	<i>SI-2: Flaw Remediation SA-15 (7): Automated Vulnerability Analysis SA-22: Unsupported System Components</i>

Summary

Patches are software changes that close loopholes and vulnerabilities in the applications code. Whether it is an application, a programming library, or a management program, businesses need to keep patches up to date to prevent known threats from being used in a successful attack against them.

Finding

While performing web application penetration tests on the OWASP Juice Shop application, Secure Ideas attempts to determine what version of common Javascript libraries are running. Determining the specific version of libraries allows the tester to find vulnerable and exploitable versions which may result in further access to the system. Furthermore, the jQuery project team has indicated that jQuery 1.x and 2.x major versions are end-of-life as of July 2018. Some of these versions have known vulnerabilities, including/such as cross-site scripting flaws.

During the testing, Secure Ideas determined that the OWASP Juice Shop application is running several older versions of Javascript libraries that have known vulnerabilities.

Some examples are:

- **jquery version 2.2.4**
 - <http://localhost:3000/score-board/socket.io/>
 - <http://localhost:3000/support>
 - <http://localhost:3000/assets/public/images/uploads/%F0%9F%98%BC->
- **jquery version 3.3.1**
 - <http://localhost:3000/profile>

Many of the library patches released are built to fix security issues found in the product. By not applying these patches, OWASP Juice Shop exposes the systems to attack from malicious users or external attackers. In the case of these unpatched Javascript libraries, there are published

vulnerabilities in each of them; however, OWASP Juice Shop is only truly at risk if using the vulnerable part of the library.

Recommendations

Secure Ideas recommends that OWASP Juice Shop deploy application library security patches as soon as possible. In addition Secure Ideas recommends that OWASP Juice Shop develop a process to monitor which applications are using which Javascript libraries and track these libraries on a regular basis so that they are kept up-to-date. This may be facilitated by retire.js, a free and open source tool for cross-referencing Javascript libraries with known vulnerabilities. This tool is available in several flavors (command line, browser extension, proxy plugin, etc...) and can be downloaded here: <https://retirejs.github.io/retire.js/>.

STRATEGIC GUIDANCE

Secure Ideas performed a web application penetration test for OWASP Juice Shop. Through testing this application Secure Ideas was able to gather a general sense of OWASP Juice Shop's security posture and would like to make the following strategic considerations available to OWASP Juice Shop:

Provide Secure Coding Training for Developers

Finding and remediating flaws after the fact is the most expensive way for organizations to handle security vulnerabilities. It takes a considerable amount of time and effort for developers to consider the discovered issues, review the code, make the appropriate modifications, work through quality assurance testing, and then roll out the changes. Alternatively, training developers to understand security flaws and avoid vulnerabilities during the development process is much more efficient and effective. Unfortunately most developer training venues do not adequately teach secure coding. During this assessment Secure Ideas found evidence that suggests many of OWASP Juice Shop's developers are not properly trained to avoid common mistakes. OWASP Juice Shop should consider providing secure coding training to all developers on a regular basis.

Consider Multi-Factor Authentication

Multi-Factor authentication is recommended for employees to use when accessing sensitive systems such as VPN, domain controllers and other critical or sensitive resources. The *Factors* of the Multi-Factor Authentication mechanism fall into three categories: knowledge (something they know), possession (something they have) and inherence (something they are). A wide range of systems exist that can be implemented directly into the Windows Server authentication systems as well as Linux servers and applications. One common system found in corporate environments is the RSA SecurID solution. Another popular system in smaller-scale environments is available from Duo Security or Google Authenticator. Whichever solution is chosen, the most important aspect of implementing Multi-Factor Authentication is to ensure that at least two different *Factors* are required to gain access and not simply the same *Factor* required multiple times (i.e., password plus fingerprint instead of multiple passwords.)

Formalize Application Security Practices and Requirements

In many organizations, applications make up a substantial portion of the exposed threat surface. Application teams within the organization have varying levels of security training, experience, and proficiency. This presents a challenge in ensuring that all applications consistently implement security controls to a similar standard. Any significant variation should be driven by the individual business cases, with necessary exceptions and risk acceptance being deliberately discussed and documented. To address this challenge, an organization may develop or adopt a comprehensive software security model to ensure that consistent processes and controls are in place to protect the applications against attackers. Some common models are the Microsoft Security Development

Lifecycle (MS-SDL)¹ and the OWASP Software Assurance Maturity Model (SAMM)². While practices like these inherently add overhead to the architectural design and development processes, they also empower development teams to take ownership of the application security, and to take a more active role in maturing the organization's security posture.

Secure Ideas found that OWASP Juice Shop is not consistently adhering to common best practices. This suggests that OWASP Juice Shop's current application security requirements are not sufficiently defined or are not comprehensive enough. OWASP Juice Shop should consider formalizing a comprehensive set of processes and requirements to ensure consistent adherence to best practices.