Web Application Penetration Test

Final Report

Prepared for: OWASP Juice Shop

April 22, 2020

Reference: S-200809042

TABLE OF CONTENTS

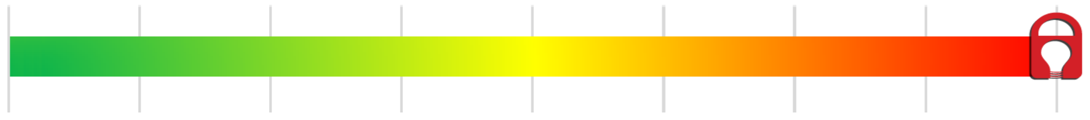
EXECUTIVE SUMMARY

Secure Ideas performed a penetration test of OWASP Juice Shop's web application. The scope of this assessment, as provided by Juice Shop, was http://juice-shop.wtf.

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

Critical	High	Medium	Low
2	1	1	1

Based on the findings in this report, Secure Ideas has evaluated the overall risk to 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 Juice Shop web application. These weaknesses are very concerning, and if leveraged, could decrease the security, usability, and functionality of the application.

One of the most critical issues Secure Ideas found is that the application was not resistant to injection-based attacks, which is considered one of the most dangerous attack vectors for applications. One example showcasing the severity of this vulnerability was in our ability to craft a simple SQL injection (SQLi) string to log in as the administrator of the application, without any prior knowledge of the username or credentials. There were also several instances of Cross-Site Scripting (XSS) flaws throughout the application. Both the SQLi and XSS flaws can easily be remediated through consistent input sanitization and output encoding and are discussed further in the finding section below.

Another significant issue discovered is an authorization bypass flaw. Secure Ideas found that an attacker can use the API to create a new user with any role, including administrator access. This is due to a lack of 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 Juice Shop to help improve its security posture.

NARRATIVE AND ACTIVITY LOG

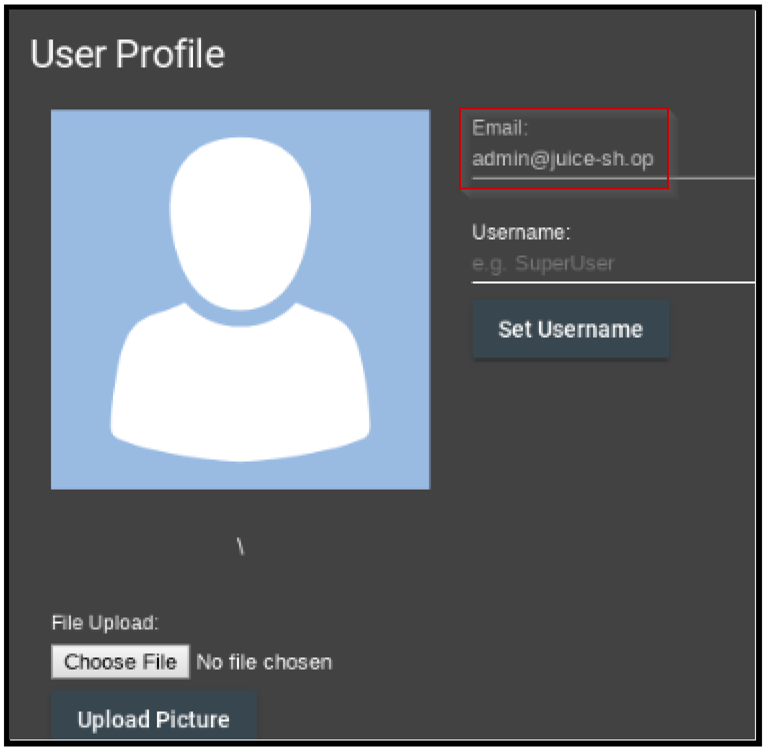
Secure Ideas began by working through our standard methodology of Recon → Mapping → Discovery → Exploitation. Since this assessment was not a black box assessment, the team skipped the initial Recon phase, starting with Mapping and Discovery. Throughout the engagement, we conducted several types of activities on each of the web interfaces within the 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

As part of the mapping phase, we explored all of the available functionality within the application using each account role provided. Starting as a normal user, we began building a map of the application features and potentially vulnerable areas, such as login forms, user profile pages, 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. By thoroughly mapping the application and roles, we had developed a good idea of the functionality and features used within the application, as well as how the application was intended to behave.

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:



Email: admin@juice-sh.op

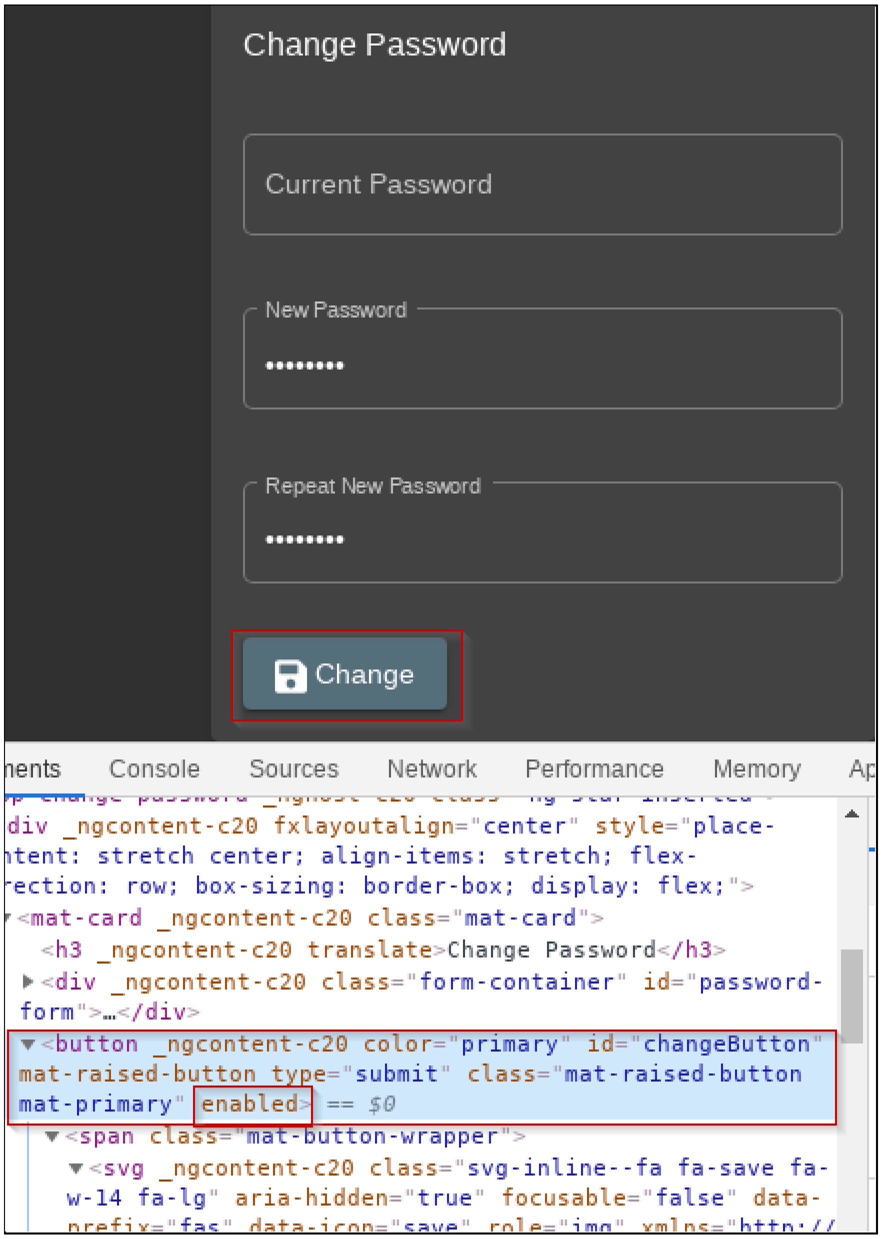
Username: e.g. SuperUser

No file chosen

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 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 Juice Shop application.

While waiting for 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.



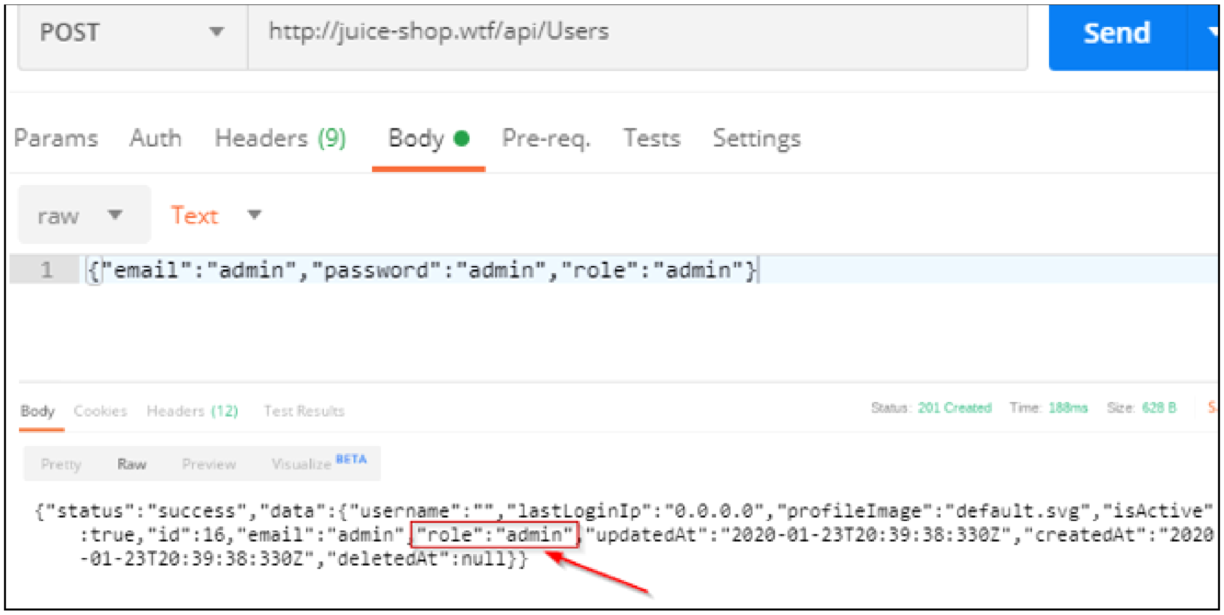
```
div _ngcontent-c20 fxlayoutalign="center" style="place-
ntent: stretch center; align-items: stretch; flex-
rection: row; box-sizing: border-box; display: flex;">
<mat-card _ngcontent-c20 class="mat-card">
  <h3 _ngcontent-c20 translate>Change Password</h3>
 <div _ngcontent-c20 class="form-container" id="password-
form">…</div>
 <button _ngcontent-c20 color="primary" id="changeButton"
mat-raised-button type="submit" class="mat-raised-button
mat-primary" enabled> == $0
   <span class="mat-button-wrapper">
     <svg _ngcontent-c20 class="svg-inline--fa fa-save fa-
     w-14 fa-lg" aria-hidden="true" focusable="false" data-
```

Continuing 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. An example request can be seen in the code block shown below:

```
POST /api/Users HTTP/1.1
Host: juice-shop.wtf
Content-Type: application/json
Connection: close
Content-Length: 82

{"email":"admin","password":"admin","role":"admin"}
```

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.



```
{"email":"admin","password":"admin","role":"admin"}
```

```
{"status":"success","data":{"username":"","lastLoginIp":"0.0.0.0","profileImage":"default.svg","isActive"
    :true,"id":16,"email":"admin","role":"admin","updatedAt":"2020-01-23T20:39:38:330Z","createdAt":"2020
    -01-23T20:39:38:330Z","deletedAt":null}}
```

Due to the critical nature of this authorization bypass, another email was sent to 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 Juice Shop's technical team through the critical findings discovered, and discussed various options related to the risk and remediation of these items. 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 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. 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. Inadequate Security Standards for Password Storage	Medium
5. Weak Password Complexity Requirements	Low

CRITICAL RISK FINDINGS

1. SQL Injection Flaws

Industry Standards	
OWASP Top 10	*A1: Injection*
NIST 800-53	*SI-10: Information Input Validation*

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:

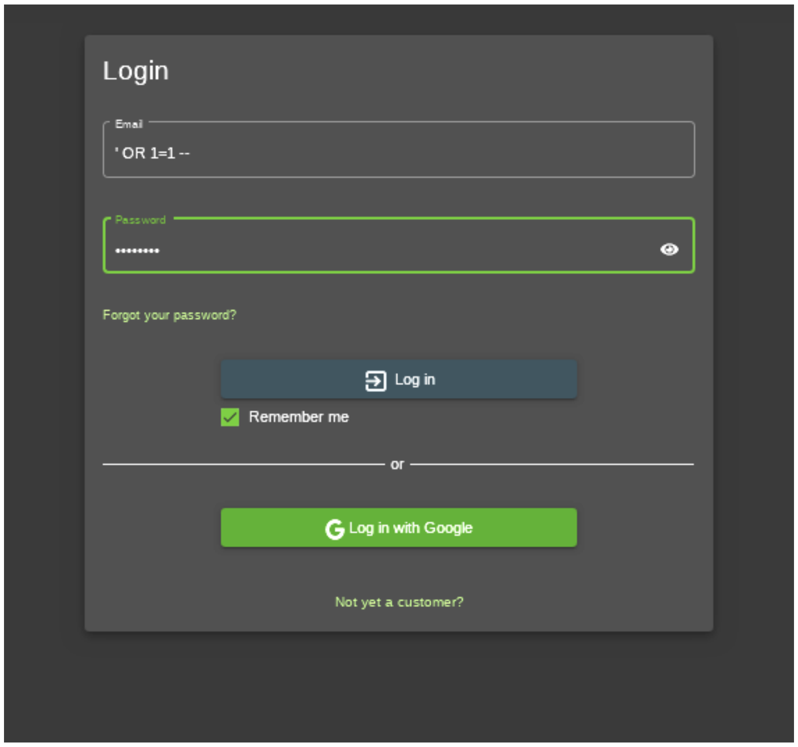
- Loss of sensitive or confidential data
- Altered sensitive or confidential data
- Bypass of authentication
- Bypass of authorization
- Access to underlying Operating System
- Further attacks against users of the application (XSS, CSRF)

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 SQL Injection. This is due to the use of unsanitized user supplied input. 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.



Recommendations

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

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 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	*A5: 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

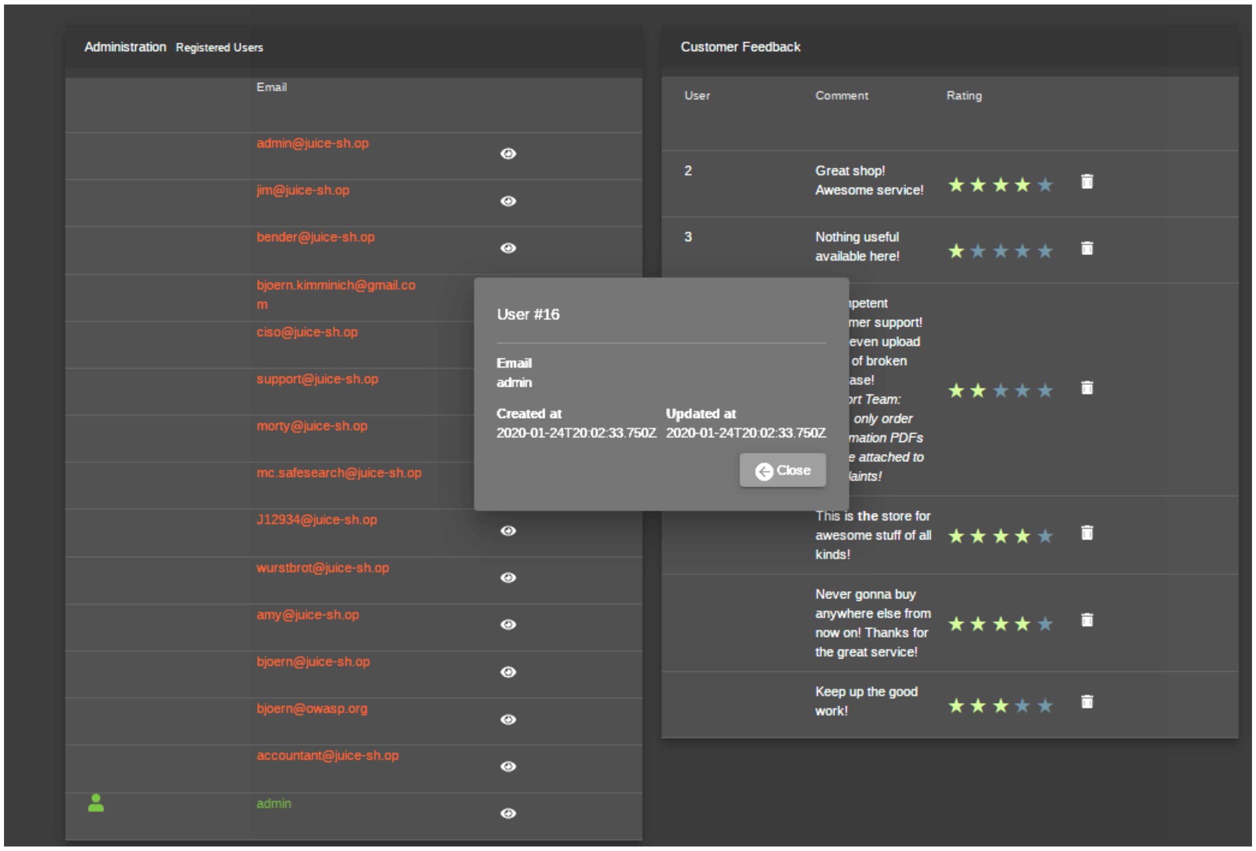
Secure Ideas has found that the Juice shop application contains an authorization bypass flaw. During the testing Secure Ideas was able to create an admin account with an unauthenticated session.

In the 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://juice-shop.wtf/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:



Administration Registered Users

	Email	
	admin@juice-sh.op	
	jim@juice-sh.op	
	bender@juice-sh.op	
	bjoern.kimminich@gmail.com	
	ciso@juice-sh.op	
	support@juice-sh.op	
	morty@juice-sh.op	
	mc.safesearch@juice-sh.op	
	J12934@juice-sh.op	
	wurstbrot@juice-sh.op	
	amy@juice-sh.op	
	bjoern@juice-sh.op	
	bjoern@owasp.org	
	accountant@juice-sh.op	
	admin	

User #16

Email
admin

Created at
2020-01-24T20:02:33.750Z

Updated at
2020-01-24T20:02:33.750Z

Close

Customer Feedback

User	Comment	Rating
2	Great shop! Awesome service!	★★★★☆
3	Nothing useful available here!	★☆☆☆☆
	petent mer support! even upload of broken ase! rt Team: only order mation PDFs e attached to laints!	★★☆☆☆
	This is the store for awesome stuff of all kinds!	★★★★☆
	Never gonna buy anywhere else from now on! Thanks for the great service!	★★★★☆
	Keep up the good work!	★★★☆☆

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. 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 Juice Shop support staff. Juice Shop could also modify the application to send messages to a central monitoring solution, if one has been implemented within the Juice Shop infrastructure. If modifying the application in this way is not preferred, 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 Juice Shop staff.

HIGH RISK FINDINGS

3. Cross-Site Scripting Flaws

Industry Standards	
OWASP Top 10	*A5: 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.

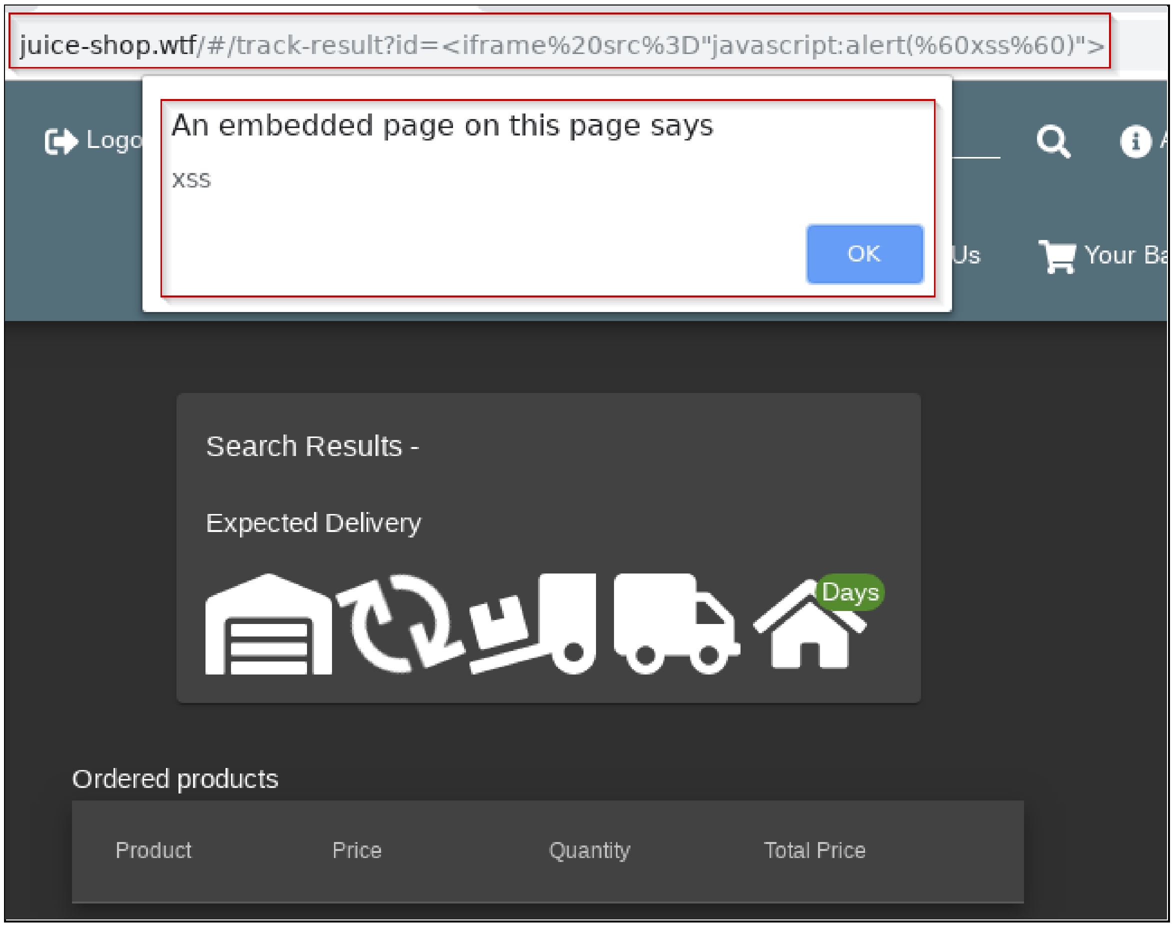
In all cases, the malicious scripts are executed in a context that appears to have originated from the targeted site. This gives the attacker full access to the document retrieved, providing almost unlimited control over the victim's experience using the application. A wide variety of options are available 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 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 example of an XSS flaw is within the user profile page of the Juice Shop application. An attacker can replace the *track-result id* with a JavaScript iframe payload. When a payload, such as *<iframe src="javascript:alert(`xss`)">* is submitted in the browser address bar, it causes the application to incorporate the attack within the resulting web page. The screenshot below is what the victim browser would see.


juice-shop.wtf/#/track-result?id=<iframe%20src%3D"javascript:alert(%60xss%60)">

An embedded page on this page says

xss

Recommendations

Secure Ideas recommends that 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 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.

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 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

MEDIUM RISK FINDINGS

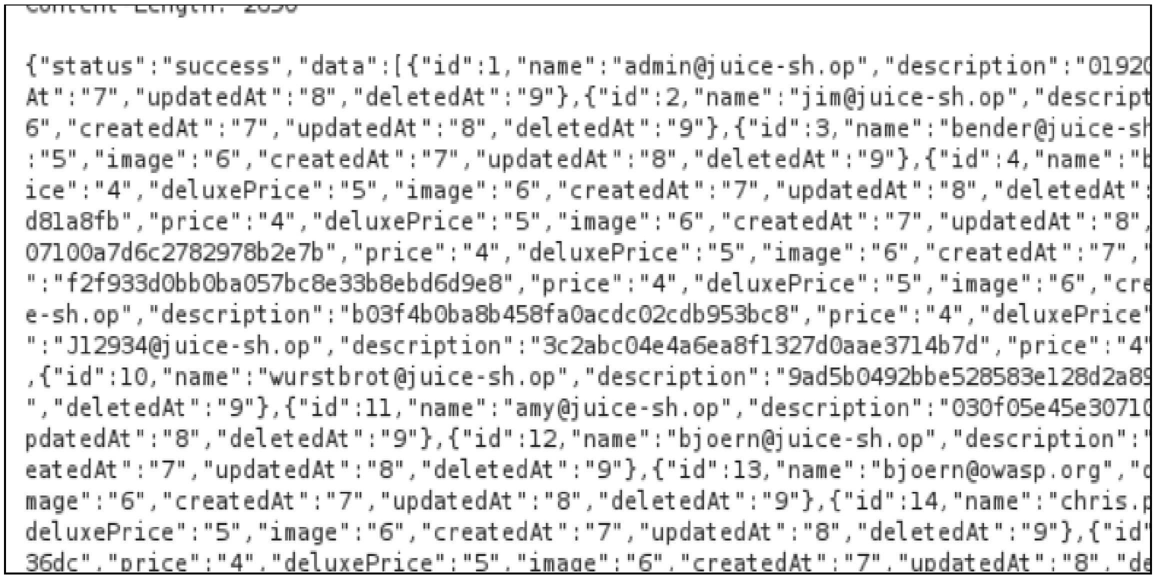
4. Inadequate Security Standards for Password Storage

Industry Standards	
OWASP Top 10	*A6: Security Misconfiguration*
NIST 800-53	*SC-28 Protection of Information at Rest* *IA-2 Identification and Authentication*

Summary
Password storage is a key point in securing business assets. If passwords are stored or transmitted via poor algorithms or worse, in clear text, then the business's entire software system is vulnerable to exploitation if the password table is accessed by an attacker.

If passwords are not regularly changed, if the algorithm is not salted (ensuring that no two encoded passwords are the same), and if the encryption is inadequate, then a business faces the risk that an attacker will access every account in the database for a period of time until the attack is discovered.

Finding
Due to the SQL injection flaw above, Secure Ideas found that the *Users* of the Juice Shop stores passwords. Analysis of the password table revealed that passwords are stored as a MD5 hash of the user's original password. The following screenshot shows a sample from the data.



```
Content-Length: [illegible]

{"status":"success","data":[{"id":1,"name":"admin@juice-sh.op","description":"0192
At":"7","updatedAt":"8","deletedAt":"9"},{"id":2,"name":"jim@juice-sh.op","descript
6","createdAt":"7","updatedAt":"8","deletedAt":"9"},{"id":3,"name":"bender@juice-sh
:"5","image":"6","createdAt":"7","updatedAt":"8","deletedAt":"9"},{"id":4,"name":"b
ice":"4","deluxePrice":"5","image":"6","createdAt":"7","updatedAt":"8","deletedAt":
d81a8fb","price":"4","deluxePrice":"5","image":"6","createdAt":"7","updatedAt":"8",
07100a7d6c2782978b2e7b","price":"4","deluxePrice":"5","image":"6","createdAt":"7","
":"f2f933d0bb0ba057bc8e33b8ebd6d9e8","price":"4","deluxePrice":"5","image":"6","cre
e-sh.op","description":"b03f4b0ba8b458fa0acdc02cdb953bc8","price":"4","deluxePrice"
":"J12934@juice-sh.op","description":"3c2abc04e4a6ea8f1327d0aae3714b7d","price":"4"
,{"id":10,"name":"wurstbrot@juice-sh.op","description":"9ad5b0492bbe528583e128d2a89
","deletedAt":"9"},{"id":11,"name":"amy@juice-sh.op","description":"030f05e45e3071
pdatedAt":"8","deletedAt":"9"},{"id":12,"name":"bjoern@juice-sh.op","description":"
eatedAt":"7","updatedAt":"8","deletedAt":"9"},{"id":13,"name":"bjoern@owasp.org","d
mage":"6","createdAt":"7","updatedAt":"8","deletedAt":"9"},{"id":14,"name":"chris.p
deluxePrice":"5","image":"6","createdAt":"7","updatedAt":"8","deletedAt":"9"},{"id"
36dc","price":"4","deluxePrice":"5","image":"6","createdAt":"7","updatedAt":"8","de
```

MD5 is a hashing algorithm that is known to have problems that allow for cryptographic collisions, meaning that two different pieces of data can produce the same MD5 hash. MD5 also lends itself to brute force attacks due to the relatively low computational power it takes to generate an MD5 hash.

In addition, the MD5 hashes that Secure Ideas found were not salted. This is evident by the fact that several of the password hashes in the previous screenshot are identical. Password salts make it much harder for an attacker to crack a password hash as it requires the attacker to know the salt value in order to start cracking the hashes.

Recommendations

Secure Ideas recommends that Juice Shop store passwords in the database with a secure one-way hashing algorithm as well as a salt.

A unique salt per user helps ensure that every password hash appears different even if the same password was selected by more than one user. Ensuring each hash is different is a measure that increases the complexity, and therefore the necessary time, of cracking the hashes. When the application needs to authenticate a user, it should hash the supplied password, with the salt, and compare the hashes.

The following OWASP resource provides more information on the value of salts and correct implementation:

https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

Additionally, Secure Ideas recommends that Juice Shop use the PBKDF2 hashing function for storing passwords. If PBKDF2 is not possible for some reason, then Juice Shop should defer to either the "scrypt" or "bcrypt" functions. All of these hashing solutions are designed to be computationally cumbersome so that an attacker with a stolen list of hashes will require significant resources to crack them. Each of these algorithms is also adaptive over time, meaning they can be configured to become more computationally cumbersome as technology becomes faster, thus building a degree of "future proof" in the algorithms.

LOW RISK FINDINGS

5. Weak Password Complexity Requirements

Industry Standards	
OWASP Top 10	*A6: Security Misconfiguration* *A5: Broken Access Control* *A2: Broken Authentication*
NIST 800-53	*AC-3: Access Enforcement*

Summary

One of the aspects tested during the penetration test, was the password complexity requirement of the 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 while the application attempts to enforce the use of complex passwords, the password complexity requirements are weaker than recommended for this type of application.

Secure Ideas found that the application tested allowed passwords such as *admin123* and *password123*. These types of passwords are commonly found in widely-accessible dictionaries. As a matter of fact, Secure Ideas commonly uses the *Password123* string against systems that implement account lockout due to it commonly being found as the password for accounts in web applications.

Recommendations

Juice Shop should strengthen the password requirements within the application. While it does perform some complexity checking, Juice Shop should increase these checks based on industry standards. One source of an industry standard is from the SANS Institute. SANS publishes a free guide to password complexity, which is available at:
https://www.sans.org/security-resources/policies/general/pdf/password-construction-guidelines

According to this guide, passwords should contain at least fourteen characters, with preference given to passphrase. Additionally, passwords should avoid any of the following flaws:

- *Contain eight characters or less*
- *Contain personal information such as birthdates, addresses, phone numbers, or names of family members, pets, friends, and fantasy characters*
- *Contain number patterns such as aaabbb, qwerty, zyxwvuts, or 123321*

- *Are some version of* Welcome123, Password123, *or* Changeme123

Secure Ideas also recommends that 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, 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, 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.

STRATEGIC GUIDANCE

Secure Ideas performed a web application penetration test for Juice Shop. Through testing this application Secure Ideas was able to gather a general sense of Juice Shop's security posture and would like to make the following strategic considerations available to 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 Juice Shop's developers are not properly trained to avoid common mistakes. 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.)