GROUP 6

FINAL PROJECT OLC CYBERSECURITY CLASS 2024

Group Members :

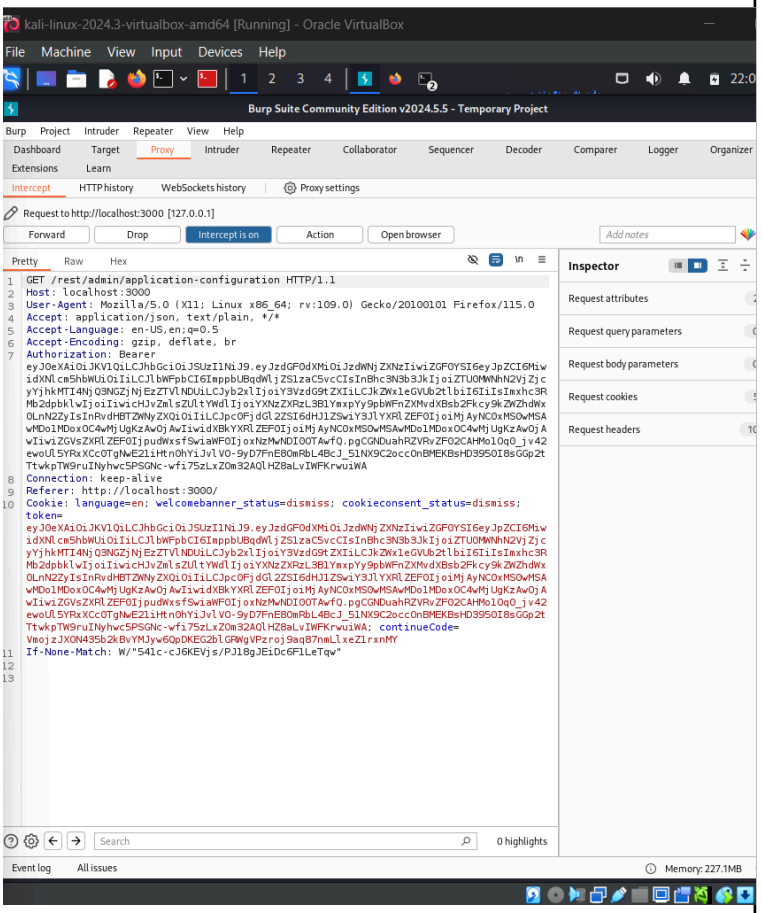
- Bayu Putra Ibana (24/536830/PA/22776)
- Muhammad Ahsan Zaki Wiryawan (24/532772/PA/22534)
- Klara Ahinta Daniswara (24/532751/PA/22531)
- Daffa M. Siddiq (24/533358/PA/22569)

Description	**SQL Injection** SQL Injection is a type of attack where an attacker inserts or "injects" malicious SQL code into a query. This can manipulate the database to reveal information, modify data, or even execute administrative operations. In OWASP Juice Shop, challenges like "Order the Christmas special offer of 2014" or "Log in as the administrator" often involve exploiting SQL Injection vulnerabilities. The goal is to manipulate the SQL queries to gain unauthorized access or extract sensitive data. **Broken Access Control** Broken Access Control occurs when users can access resources or perform actions they shouldn't be able to. This can lead to unauthorized access to sensitive data, administrative functions, or other users' accounts. In OWASP Juice Shop, challenges like "Access the administration section of the store" or "Post some feedback in another user's name" involve exploiting these vulnerabilities. The goal is to bypass security controls and perform actions reserved for higher-privileged users.
Machine	**Owasp Juice Shop**
General Issue	Broken access control: 1. JTW(administrator page) 2. SSFR SQLi: 1. Login Admin 2. Login Bender 3. Login Jim 4. User Credentials

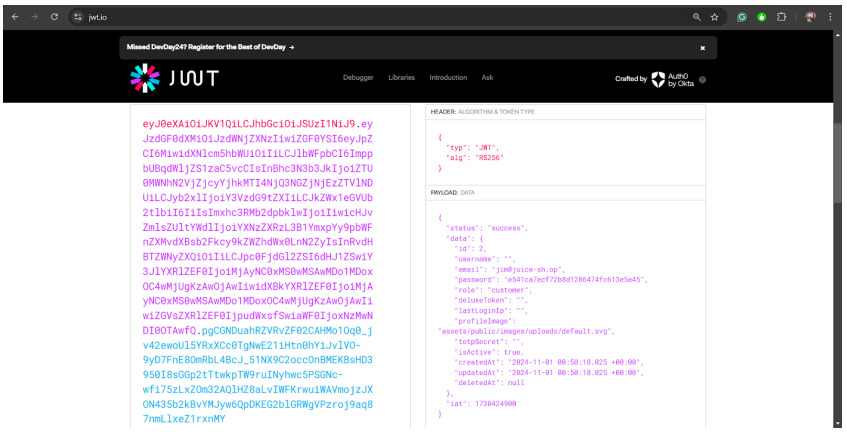
General Impact	1. Session Hijacking: Stealing session cookies can allow attackers to impersonate users. 2. Unauthorized Actions: Exploiting CSRF (Cross-Site Request Forgery) to make unauthorized changes or transactions. 3. Data Theft: Extracting sensitive data like credentials or personal information from JavaScript variables. 4. Malware Distribution: Prompting the user to download malicious files. 5. Phishing: Displaying fake login forms to capture user inputs. 6. Keylogging: Logging keystrokes to capture passwords and other sensitive inputs. 7. Denial of Service (DoS): Causing the browser or application to crash by exploiting vulnerabilities. Impacts of Other Vulnerabilities: Broken Access Control: 1. JTW (administrator page): Unauthorized users accessing admin functionalities can lead to data breaches, unauthorized changes, and system control. 2. Tampering Product Descriptions: Can be used by attackers to tamper and change product descriptions, making them confusing and misleading SQL Injection (SQLi): 1. Login Admin / Bender / Jim: Unauthorized access to user or admin accounts, particularly privileged accounts, which can lead to data theft or modification, and system manipulation. 2. User Credentials: Stealing user credentials can lead to account takeover and unauthorized access to sensitive information. These vulnerabilities can have cascading effects, jeopardizing user data integrity, application functionality, and overall system security.
Reference	https://pwning.owasp-juice.shop/companion-guide/latest/part2/xss.html https://owasp.org/www-community/Types_of_Cross-Site_Scripting

	https://pwning.owasp-juice.shop/companion-guide/latest/part2/injection.html https://pwning.owasp-juice.shop/companion-guide/latest/part2/broken-access-control.html
Issues : 1	Broken access control: JTW(administrator page)
Impact	If someone has access to a JWT (JSON Web Token) key, it means they can potentially create valid tokens, allowing them to: 1. **Impersonate Users**: They can act as other users by generating fake tokens, accessing user-specific data, and performing actions as that user. 2. **Privilege Escalation**: They can create tokens with elevated roles, like admin privileges, gaining unauthorized access to sensitive operations or data. 3. **Bypass Security**: The JWT signature validates the authenticity of a token. With the key, an attacker can sign any token, bypassing security checks, leading to unauthorized actions within the system. The impact is severe as it undermines the entire authentication mechanism, making it possible for an attacker to take over accounts, steal information, or modify critical system functions.
Replication of Steps	1. The first thing you do in terms of trying to obtain a JWT key is to first intercept any responses from OWASP Juice

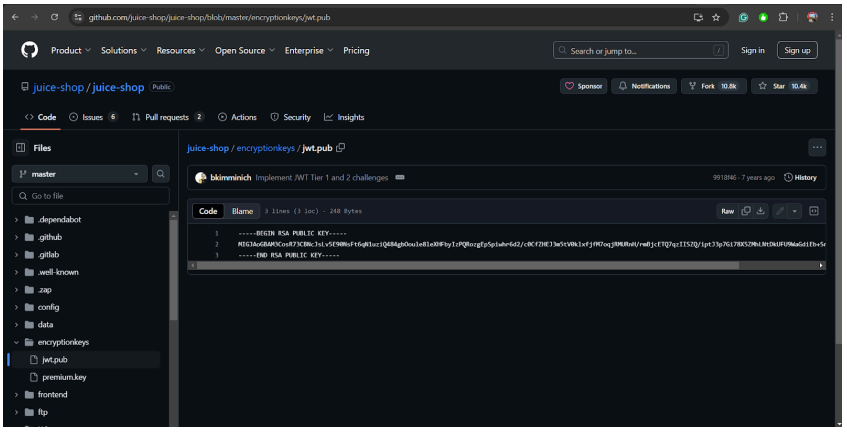
Shop itself.



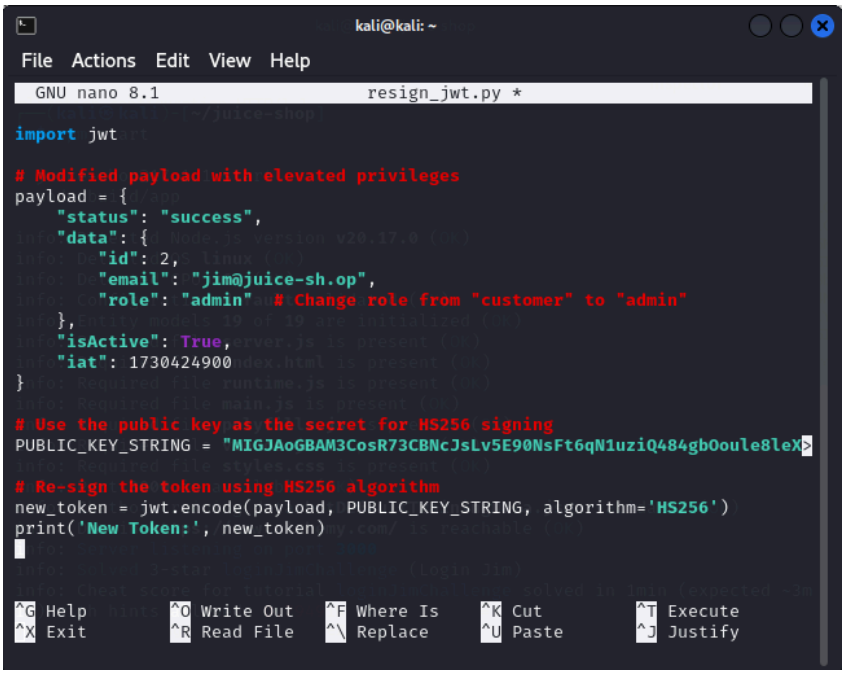
2. up is copying down that token that you see and decoding it through jwt.io to find out it's contents.



3. After that we obtain the RSA Public Key from the github repository accessible to everyone through the internet.

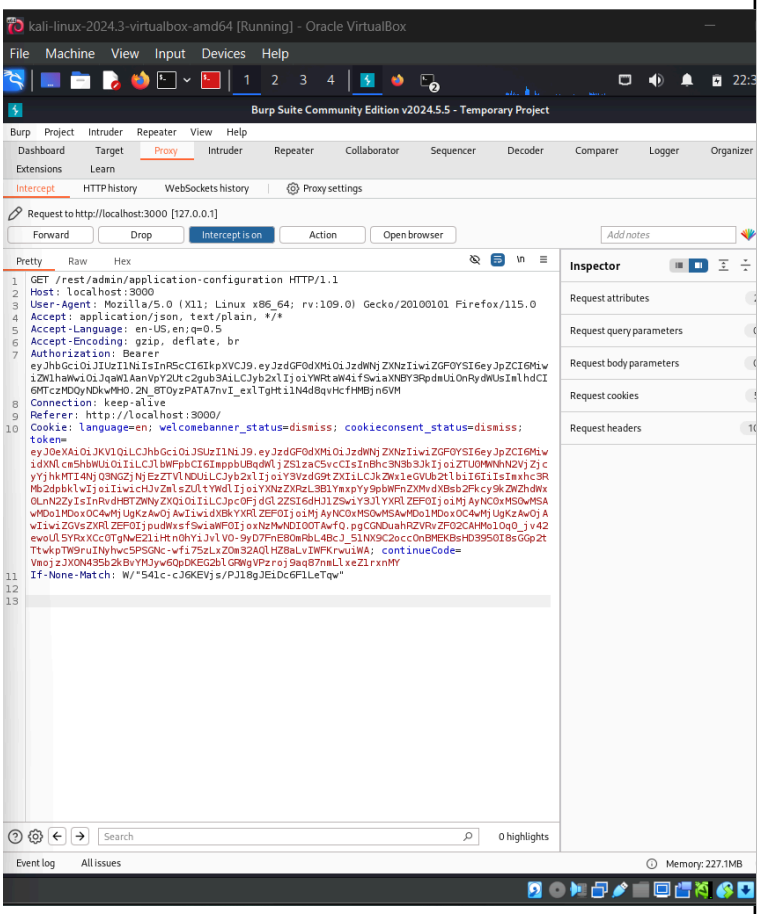


4. Then after that, in a Python nano script we change our previous token, giving us admin role and resigning the token using the HS256 algorithm.

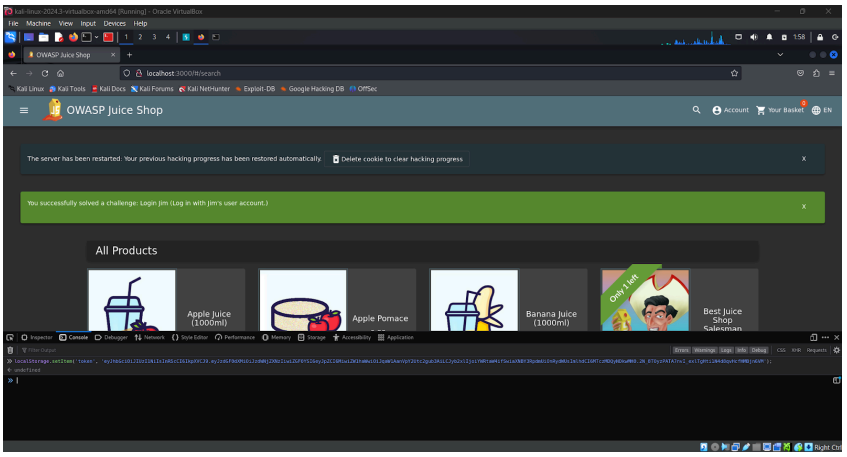


5. After we finish up the script, we finally obtain our very own custom made JTW key.

6. After this we input that JTW key into our authorization: bearer and proceed.

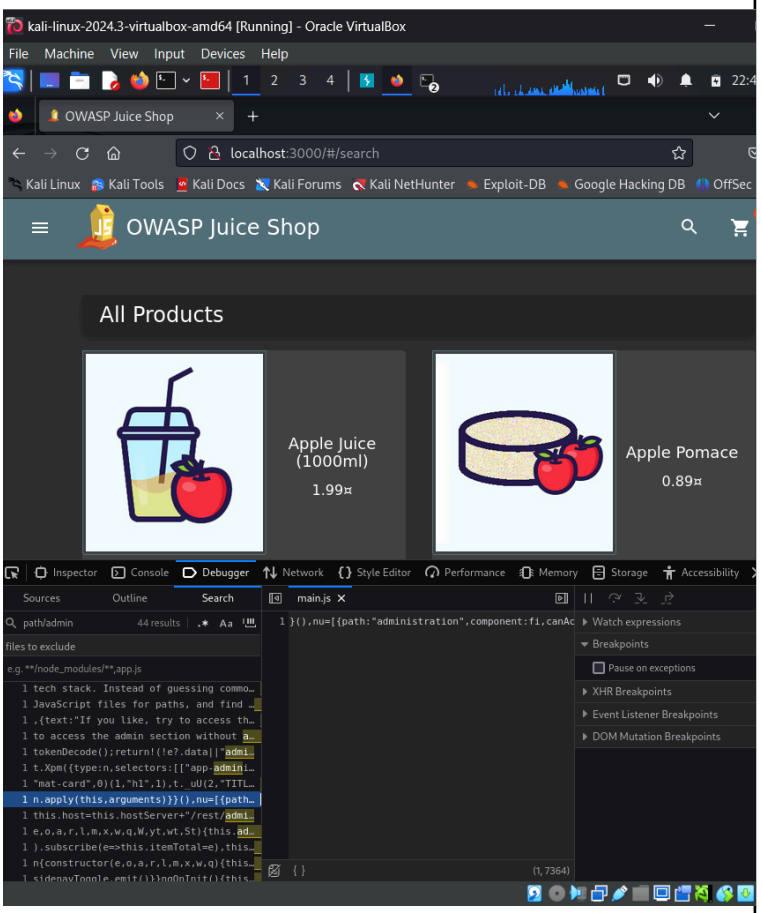


7. To finalize having access of the JWT key make sure to store it in the local storage from your console in the developer tools section.

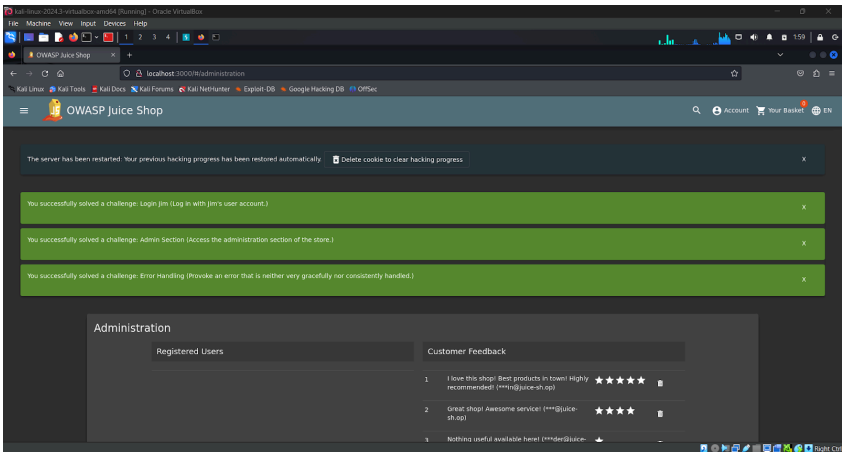


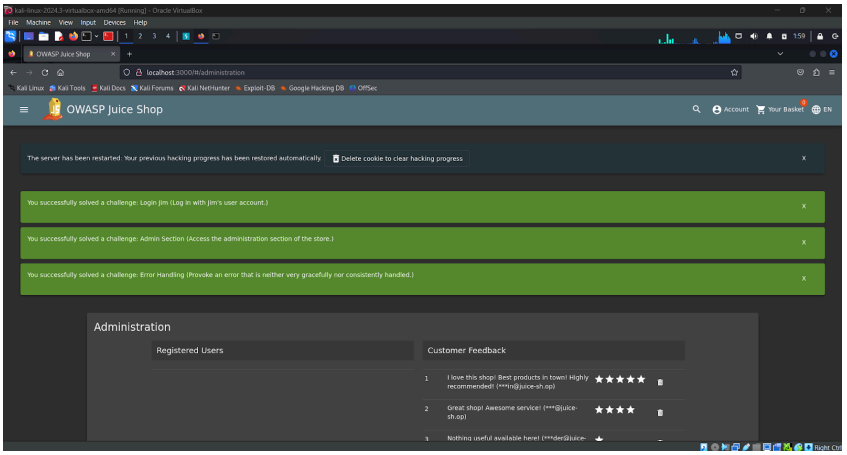
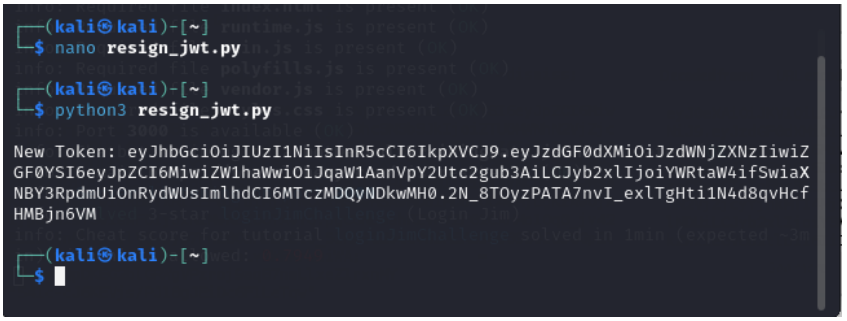
8. With the JWT key, you now have authorization to enter a lot of admin pages you otherwise would not have access to. One example of how you can find admin pages is by browsing through the developer tools and finding paths

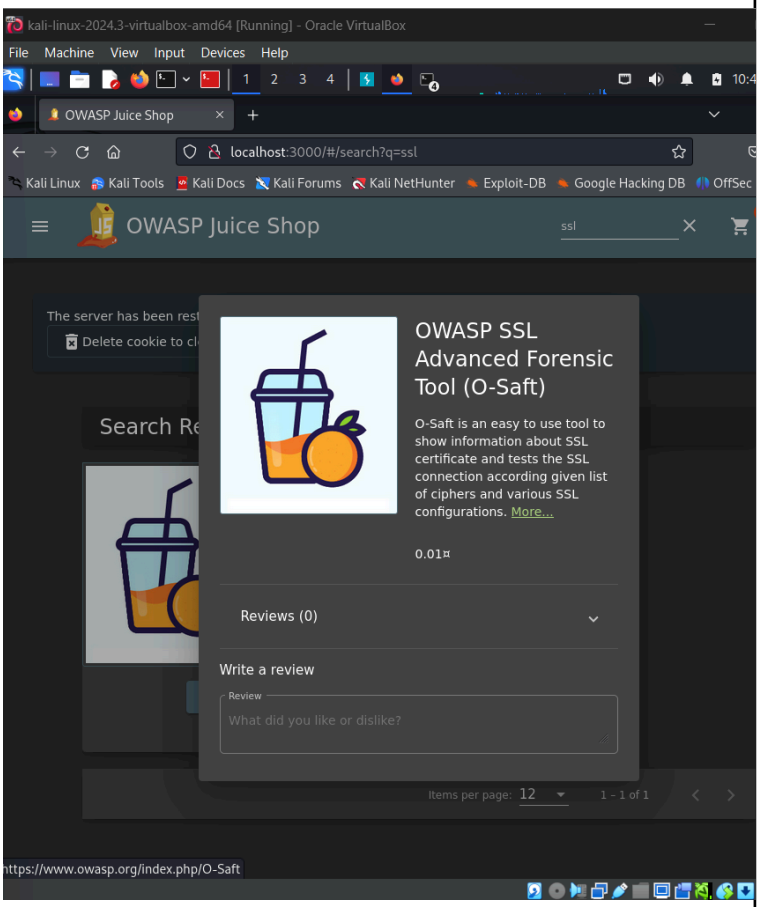
that lead to admin related things.



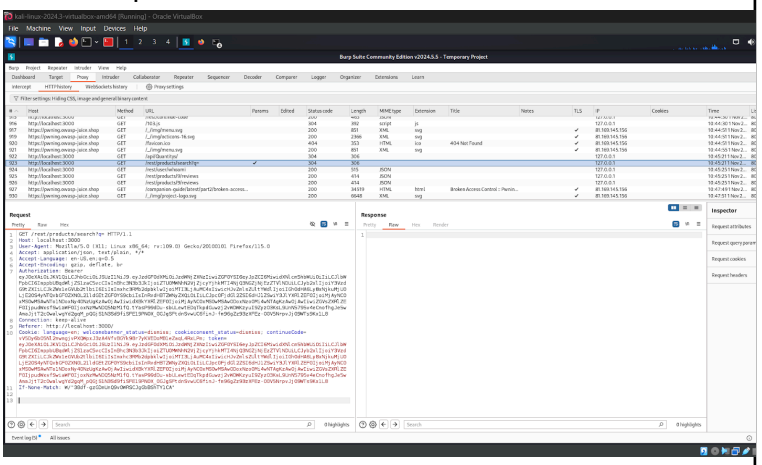
9. After that, you just input it into the spacebar with your newly acquired JWT token and now you can enter freely.



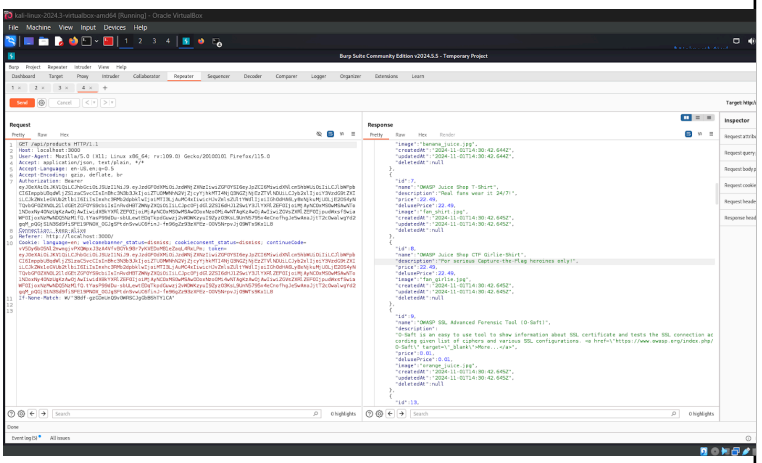
Proof Of Concept	 
Recomendation	**Use Strong Algorithms**: Use `HS256` or `RS256`. Avoid "none". **Protect Signing Key**: Secure your JWT secret keys, never hard-code them. **Short Expiry Times**: Use short-lived tokens, and refresh securely. **Verify Properly**: Validate JWT signature, `iss`, `aud`, `exp`. **Use HTTPS**: Always transmit JWTs over HTTPS. **Blacklist on Logout**: Track and invalidate tokens after logout. **Rotate Keys**: Periodically change signing keys. **Secure Refresh Tokens**: Store refresh tokens safely, like in HTTP-only cookies.
Reference	https://jwt.io/
Issues : 2	Broken Access Control : Tampering Product Descriptions

Impact	The various impacts of random users having the ability to freely change and tamper with product descriptions as they wish are among the following: **Misinformation**: Users may be misled about product features, prices, or availability, which can erode trust. **Financial Loss**: Manipulated descriptions could display incorrect pricing, leading to unintentional sales at a loss. **Brand Reputation Damage**: False claims or inappropriate content can harm a brand's credibility and customer loyalty.
Replication of Steps	1. The first thing we do is lock onto a specific product that we want to tamper with, and we explore all the ins and outs of the product, triggering a lot of responses.  2. With all those responses collected through Burp Suite we can check each one until we find one that's similar to

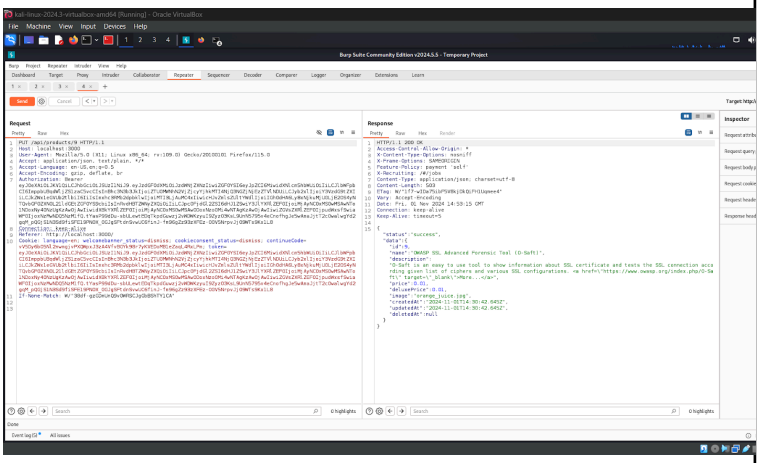
GET /rest/products/…



3. After we've found it, we can then change it into api instead of the rest. We found this out through browsing other similar data streams in Burp Suite, then we can view the whole list of products and their various descriptions.

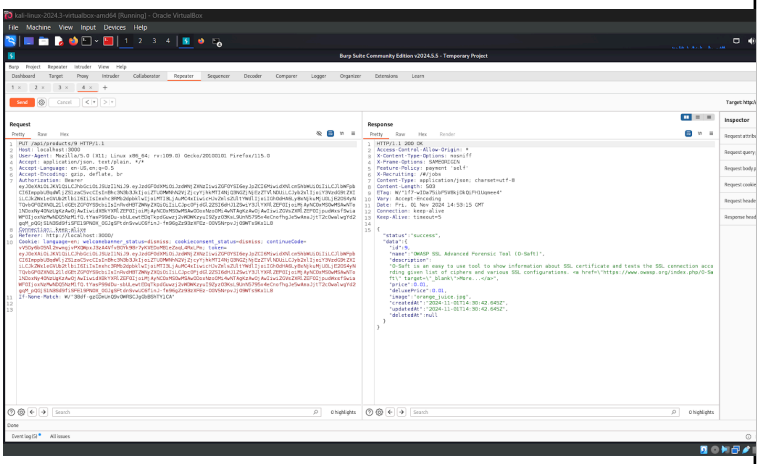


4. After this, we focus in on our specific target, which is product number 9.

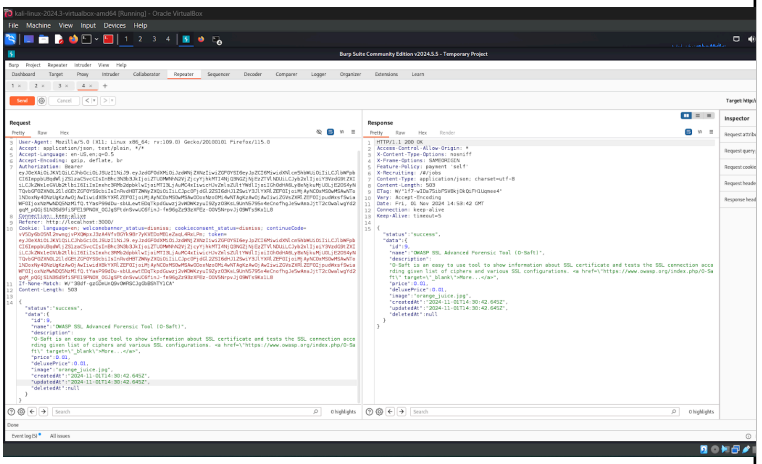


5. After we've focused on our target, we change the GET request to PUT since we now focus on changing the

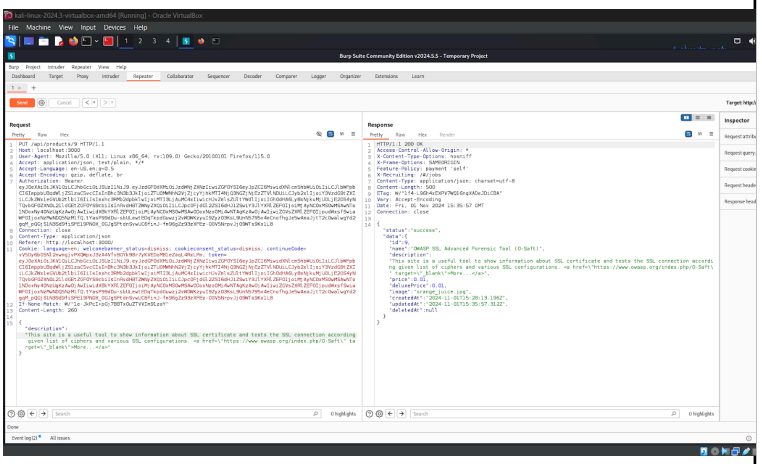
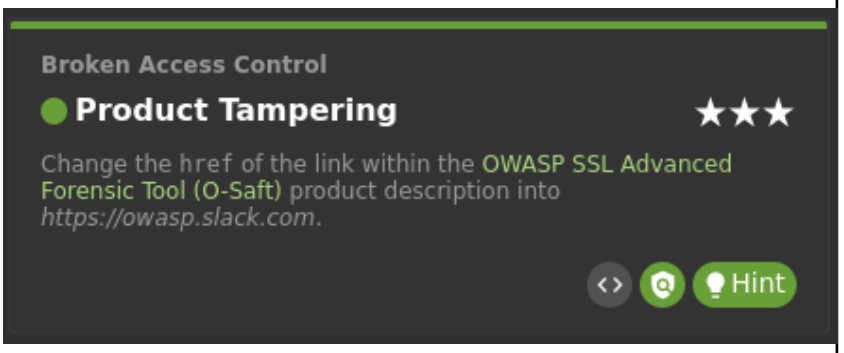
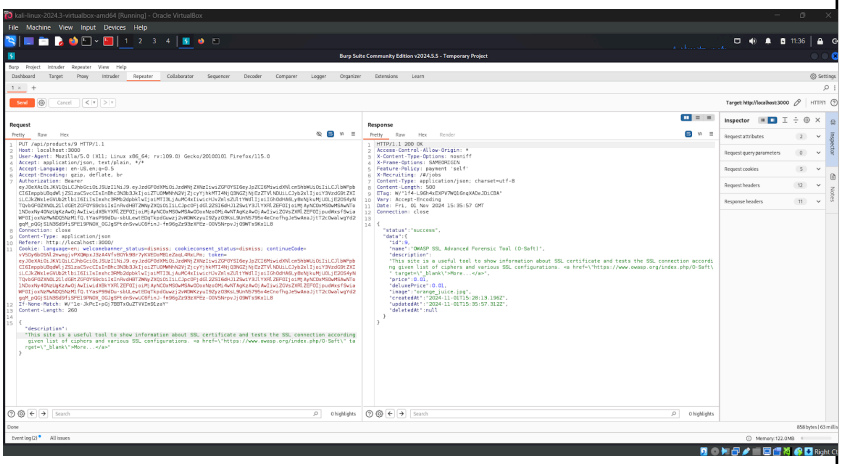
description based on what we request.

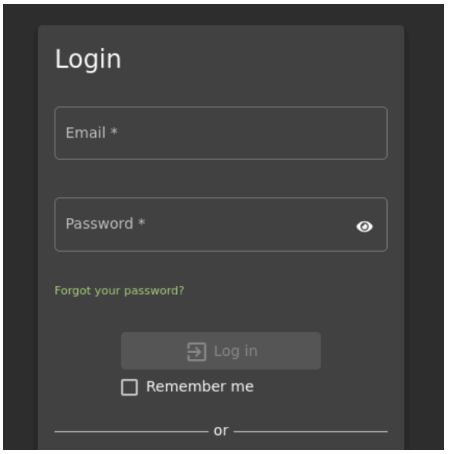


6. After this, we copy over the whole product listing from the response and test it out. It comes back with no errors so we proceed.



7. Lastly, we add in an additional header named Content-Type, which informs the client what type of data is being sent and allows us to properly handle it. Also, we remove everything other than the description, as we're not focused on that at the moment. After all that, we've finally succeeded in stamping with the product

	description. 
Proof Of Concept	 
Recomendation	**Server-Side Validation**: Ensure all input from clients is validated and sanitized on the server-side. This prevents unauthorized changes by intercepting and modifying requests. **Authentication & Authorization**: Implement strong authentication and role-based access controls (RBAC). Only authorized users, such as admins, should be allowed to modify product data.

	Digital Signatures: Add digital signatures to product data to verify integrity. If the data is altered, the signature will not match, making tampering evident. **Audit Logs**: Keep logs of all changes to sensitive data, including who made changes and when. This makes tampering traceable and helps detect malicious behavior.
Reference	https://www.youtube.com/watch?v=G4UKdotkyu8&list=PL8j1j35M7wtKXpTBE6V1RIN_pBZ4StKZw&index=35&ab_channel=Hacksplained
Issues : 3	SQLi : Login Admin
Impact	Logging in with an Admin account as a user can lead to various problems, since admin accounts usually have more power and privileges on a certain website or domain, so there are even more ways the hacker can hack or extract sensitive data.
Replication of Steps	1.  go to the login page in the juice shop and enter in a value, for ex email admin, password admin 2.

Request

Pretty Raw Hex

```
POST /rest/user/login HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
Firefox/115.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 36
Origin: http://localhost:3000
Connection: keep-alive
Referer: http://localhost:3000/
Cookie: language=en; welcomebanner_status=dismiss; continueCode=
kOe74JaD1lPQEYLRrZ8oMgOWAD5fQku38Tjydk3bjwxNq52ByvnmzKVX96p8;
cookieconsent_status=dismiss

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

Click log in, it will return invalid account. go to burpsuite and find the post request, and send it to the repeater. you can see the email and password that we entered.

3. We use an SQL injection method in the email parameter, in this case we insert “' OR '1'='1' -- ”

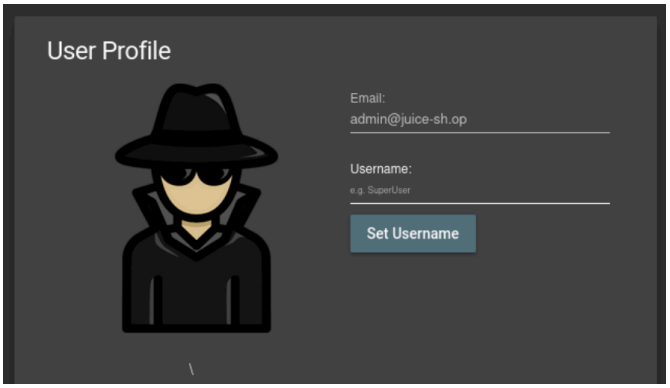
```
{
  "email":"' OR '1'='1' -- ",
  "password":"admin"
}
```

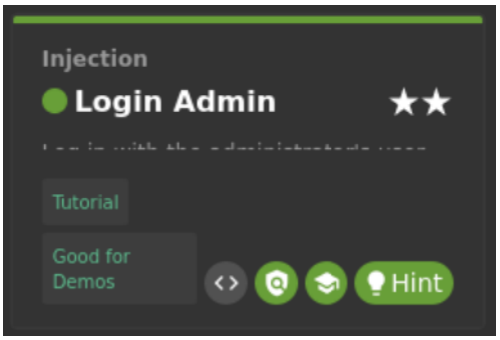
4. The result: granted access.

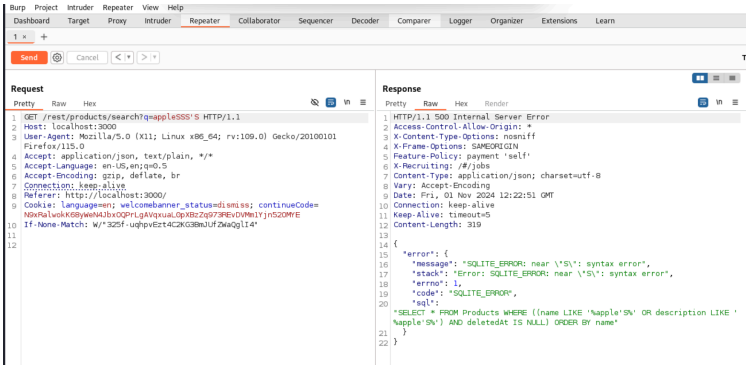

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 799
ETag: W/"31f-8juK+D+BKLNR617NL715jiSd3l0"
Vary: Accept-Encoding
Date: Fri, 01 Nov 2024 13:53:20 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "authentication":{
    "token":
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI
    6eyJpZCI6MSwidXNlcmShbWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFz
    c3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pb
    iIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW1hZ2UiOiJhc3
    NldHMvcHVibGljL2ltYWdlcy9icGxvYWRzL2RlZmFlbHRBZGlpbiSwbmciLCJ0b3RwUZvjcmV
    0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjQtMTEtMDEgMDk6MDQ6Mzcu
    MzI3ICswMDowMCIsInVwZGF0ZWRBdCI6IjIwMjQtMTEtMDEgMDk6MDQ6MzcuMzI3ICswMDowM
    CIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTczMDQ2OTIwMX0.yLj0PSAtCUhD6Cn3HiCmwa
    6GpTQMLRCRUMsxV_JrpEYpUUMj4-HPSuDw2qzMRfZLigD9b5jPzHH8KsTuNUQnp3IrzYs2Qa5
    7gHRZOVn6lvoDrpG3z8UGpzu3TRHdM9LPeziSgxRgWDXzS6rhCALWIlQKy_Wj6mLh7IOyXIjQ
    zbY",
    "bid":1,
    "umail":"admin@juice-sh.op"
  }
}
```


Proof Of Concept

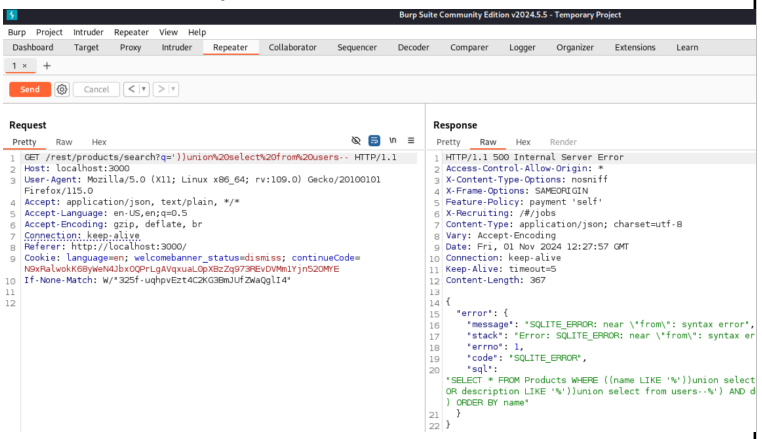




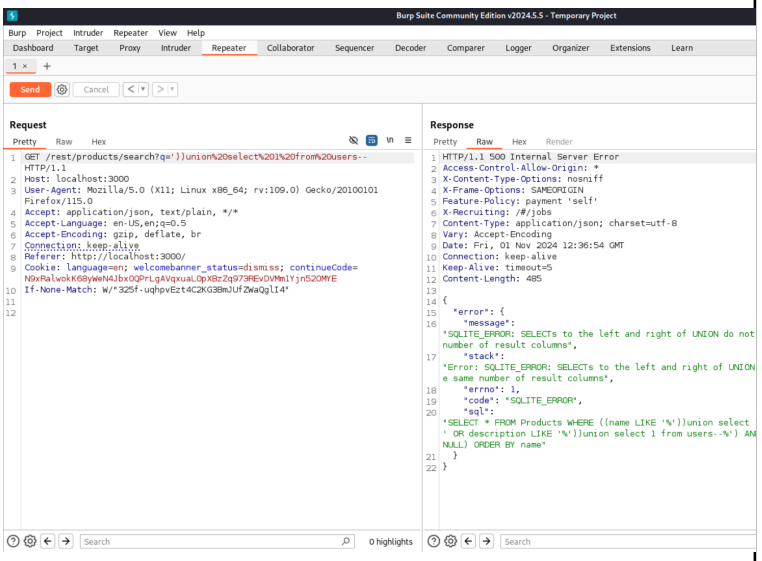
Recomendation	1. Use Parameterized Queries 2. Use Stored Procedures 3. Allow List input Validation
Reference	1. https://cwe.mitre.org/data/definitions/89.html https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html 2. https://cwe.mitre.org/data/definitions/89.html
Reference	https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html https://cwe.mitre.org/data/definitions/89.html https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.html
Issues : 4	SQLi : User Credentials
Impact	Being able to view all user credentials is very deadly and can lead to many problems, such as exposing usernames and emails increases the risk of phishing attacks and credential stuffing. The fact that you could access this data suggests a lack of proper access control. There could be a lot of data in a user credential database and in a real situation this would cause a massive data leak which would endanger many people's private and sensitive information.
Replication of Steps	1. Go to the search bar, and try to intercept the request. We can play around with the q= we know from the database schema that searching with a character like ' will return in products not being found/ no results found. So we try to use a common SQLi payload which is union select, since we are looking for a database. 

2. The UNION SELECT payload is used in SQL injection attacks primarily when the goal is to retrieve data from a database table different from the one the original query is intended to access. It allows you to access sensitive information like usernames, passwords, emails, etc., by appending a query that selects fields from a sensitive table, potentially bypassing access controls.

3. So when we use UNION SELECT, we can kind of guess the name of the table and columns. In this case, let's assume the name of the database for user data is "users" because it is a common name for storing user data. Next we will have to test different column numbers or even the datatype names from the "users" database.

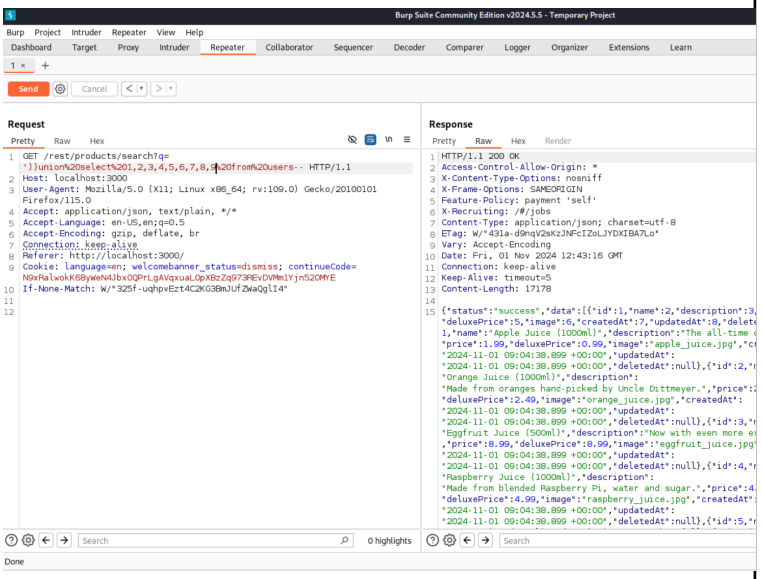


In this photo, i am guessing that in the database the columns with user data is named "users"

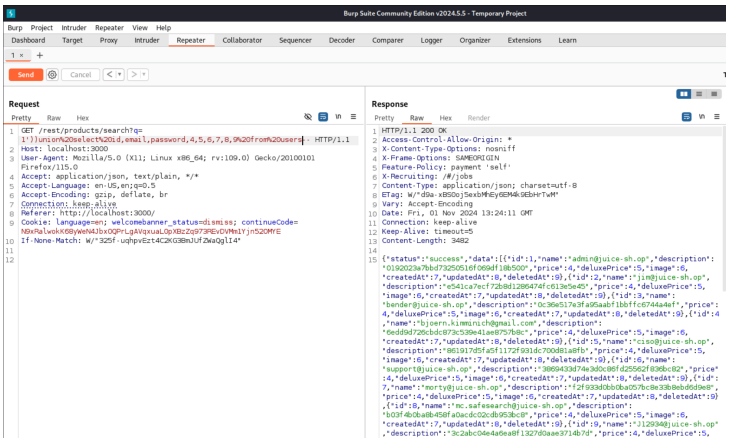


4. There is a different error message there but this time we enter the users column, indicating that the database for the users might be potentially available to access, we

just need to get the right number of columns according to the error.

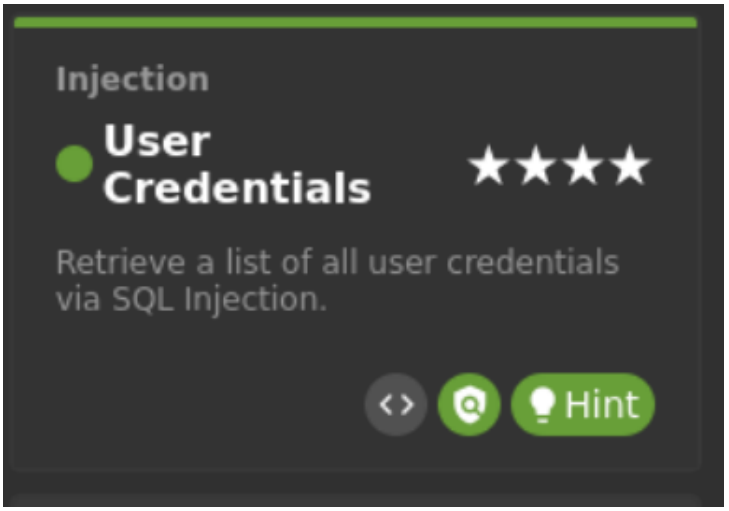
5. 

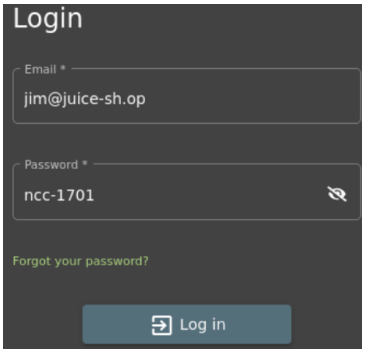
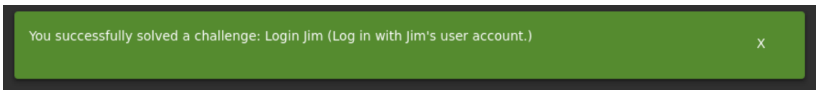
After trial and error, we finally got an OK after typing out that there are 9 columns. However, since we are specifically finding user data, we are going to change the values of the column

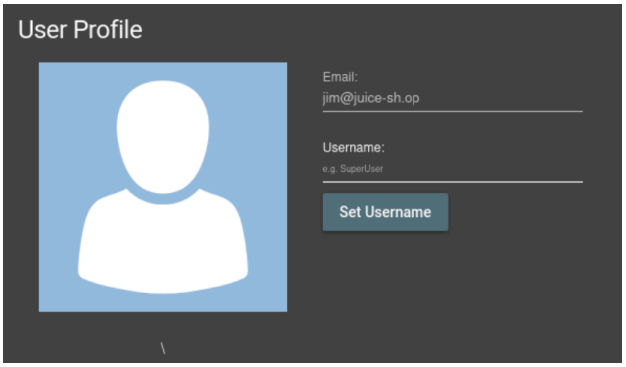
6. 

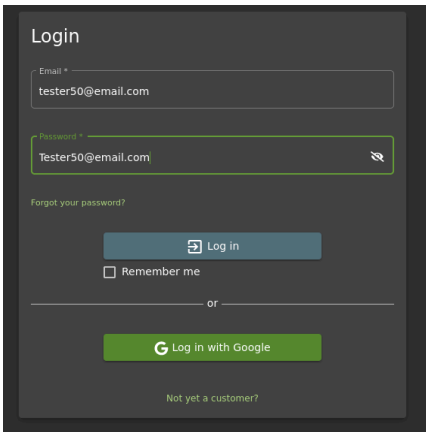
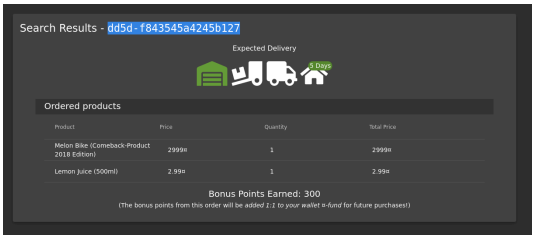
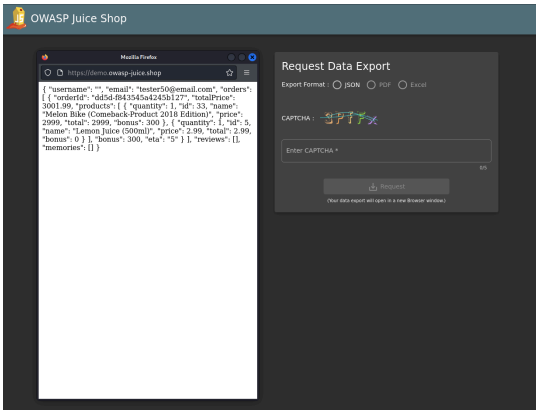
Then we finally got the user data that is available in the database. You can also add any character or number after the q= for more specific results. In this case I only typed ‘1’.

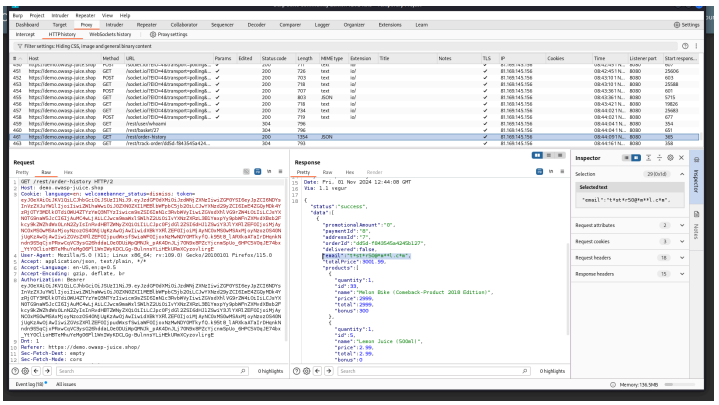
1'))union%20select%20id,email,password,4,5,6,7,8,9%20from%20users--

Proof Of Concept	 {"status":"success","data":[{"id":1,"name":"admin@juice-sh.op","description":"0192023a7bbd73250516f069df18b500","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":2,"name":"jim@juice-sh.op","description":"e541ca7ecf72b8d1286474fc613e5e45","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":3,"name":"bender@juice-sh.op","description":"0c36e517e3fa95aabf1bbffc6744a4ef","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":4,"name":"bjoern.kimminich@gmail.com","description":"6edd9d726cbdc873c539e41ae8757b8c","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":5,"name":"ciso@juice-sh.op","description":"861917d5fa5f1172f931dc700d81a8fb","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":6,"name":"support@juice-sh.op","description":"3869433d74e3d0c86fd25562f836bc82","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":7,"name":"morty@juice-sh.op","description":"f2f933d0bb0ba057bc8e33b8ebd6d9e8","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":8,"name":"mc.safesearch@juice-sh.op","description":"b03f4b0ba8b458fa0acdc02cdb953bc8","price":4,"deluxePrice":5,"image":6,"createdAt":7,"updatedAt":8,"deletedAt":9},{"id":9,"name":"J12934@juice-sh.op","description":"3c2abc04e4a6ea8f1327d0aae3714b7d","price":4,"deluxePrice":5,
Recommendation	1. Encrypt sensitive data 2. Install and implement proper access controls 3. Limit data exposure 4. Use API Authentication and Authorization:
Reference	https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/ https://owasp.org/www-project-api-security/
Issues : 5	SQLi : Login Jim

Impact	This is also a simulation to logging into another user's account but this time it highlights more on password security, as it is vulnerable to password guessing or encrypting. Without any strong password policies,iit can lead to easy access to user accounts which can lead to leaked user data, perform unauthorised user actions and many more problems.
Replication of Steps	1. From the user credentials problem we delved into earlier, we were able to access the user data, let's look for the data that belongs to jim's account. "createdAt":7,"updatedAt":8,"deletedAt":9},{"id":2,"name":"jim@juice-sh.op", "description":"e541ca7ecf72b8d1286474fc613e5e45","price":4,"deluxePrice":5, 2. From there we are able to see in the "description" there is a long string which could possibly be his password. 3. By the looks of it, it is a hash, so I will try to use a password cracking tool, Crackstation. 4. Paste in the hash from the database to the website tool Hash: e541ca7ecf72b8d1286474fc613e5e45 Type: md5 Result: ncc-1701 Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found. 5. From the result we get, let's try to login with Jim's account.  Press log in, then you can log inside of Jim's account.
Proof Of Concept	

	
Recommendation	1. Enforce Strong Password Policies, Require complex passwords with a mix of uppercase, lowercase, numbers, and special characters. 2. Implement Multi Factor Authentication to provide an additional layer of security. Even if the hacker guesses or obtains Jim's password, they would still need access to a second factor. 3. Log all login attempts and failed authentications. Implement alerts for unusual login activity. And limit login attempts.
Reference	https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html https://pages.nist.gov/800-63-3/sp800-63b.html
Issue 6	**GDPR Data Theft without Injection**
Impact	User data exposed
Replication of Steps	1. Initialise the rigged Firefox along with Burpsuite 2. Access the juice-shop website 3. Take a look at reviews and target-user's account name 4. Make an account with the same name as the target-user's account name, differing only in **vowels** 5. Account > Privacy & Security > Request Data Export > Complete Captcha > Done 6. Enjoy user data

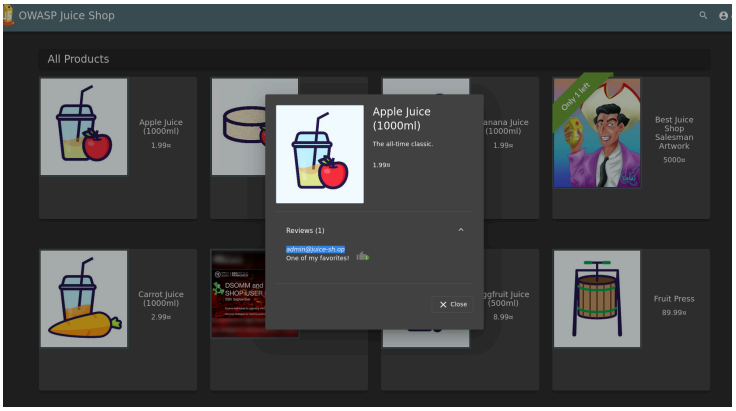
Proof of Concept	Initialise the rigged Firefox along with Burpsuite. Access the juice-shop website and create an account. Then, buy some items, order it, track the order, and download a .json file.   The .json file should look like this.  Opening burp to confirm the files.



Confirmed.

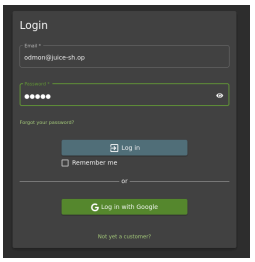
Notice something weird in the email? Showing asterisks instead of its real characters, that is something we can exploit.

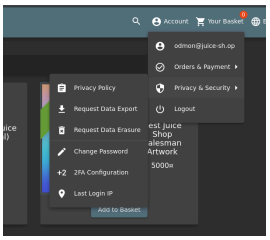
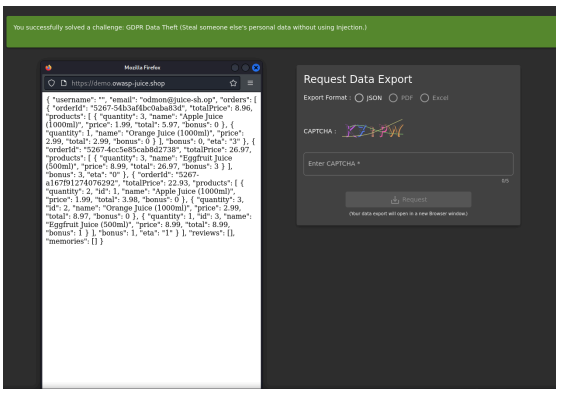
Here we look at the reviews left by people in the products, shoutout to my man “admin” for reviewing the apple juice.

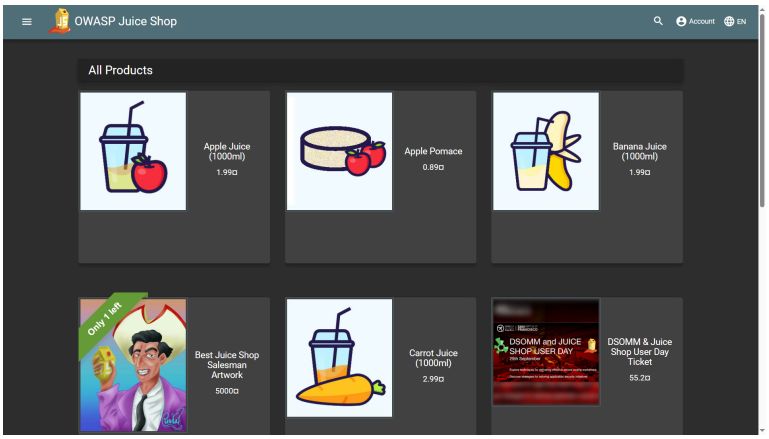
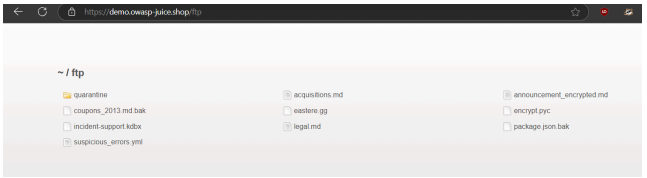


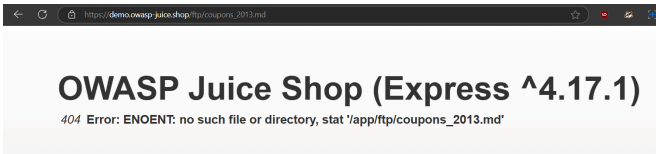
Yeah admin, sorry, we be hackin’ you, please wait.

Remember the asterisks? This might mean that we can create an account with a similar name to a user, and possibly get their data. Let us try that.



	Here we used the name ‘odmon’ instead of ‘admin’. Note that we **have not** done anything just yet.  Since I am curious, I would like to check the new account’s data.  100000000% not our data, remember that we **have not** done anything, so whose data is this? Yeah, it is admin’s data. Sorry admin. Mission accomplished.
Recommendation	• Improve account saving method security by eliminating asterisks, instead preserving the vowel character • Simply not making an account with a vowel
Reference	https://curiositykillscolby.com/2020/12/20/pwning-owasps-juice-shop-pt-55-gdpr-data-theft/
Issue : 7	**Sensitive Data Exposure :** Obtaining “Forgotten” Developer and Salesman Backup Data

Replication of Steps	1. Access the website 2. Add /ftp at the end of the website 3. Add %2500 followed by .md at the end of the target-file 4. Enjoy the file
Proof of Concept	As always, we access the website (we need not Kali nor Burpsuite this time).  Add /ftp in the website's URL  See coupons_2013.md.bak and package.json.bak? Those are our targets. But we can not download them directly, as it would result in a denied access error (the web has got some protections eh). Not that easy, unfortunately :(Also, downloading the .md file does not work, as it does not exist, you can see here.



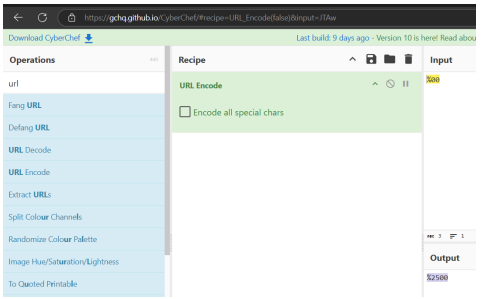
We will have to improvise by using …. Yes, you are correct, *null byte injection*. What is *null byte injection* you might ask, “Null byte injection is a sophisticated exploitation technique used in web security to bypass sanity checking filters in web applications. It involves injecting URL-encoded null byte characters, such as %00 or 0x00 in hexadecimal, into user-supplied data.”

Ok, let us try inserting %00 *after* the .bak and followed by an .md to *trick* the website to download an .md file, as trying to download .bak files results in an error.

Wait what? What went wrong?

“Null byte injection is a sophisticated exploitation technique used in web security to bypass sanity checking filters in web applications. It involves injecting **URL-encoded** null byte characters, such as %00 or 0x00 in hexadecimal, into user-supplied data.”

Ah I see, we will have to encode the null byte first, in this case I will be using a utility called CyberChef to do an URL encoding.



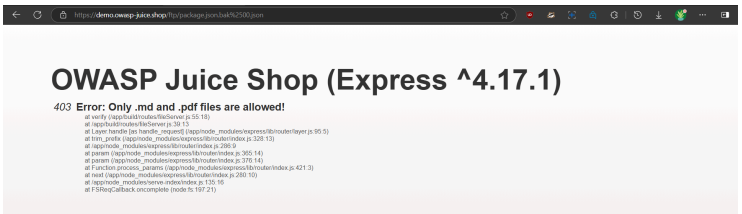
When we input %00, we receive the output %2500.

We will replace the added %00 to %2500.

```
n<MibgC7sn
mNYS#gC7sn
o*IVigC7sn
k#pDlgC7sn
o*I]pgC7sn
n(XRvgC7sn
n(XLtgC7sn
k#*AfgC7sn
q:<IqgC7sn
pEw8ogC7sn
pes[BgC7sn
l}6D$gC7ss
```

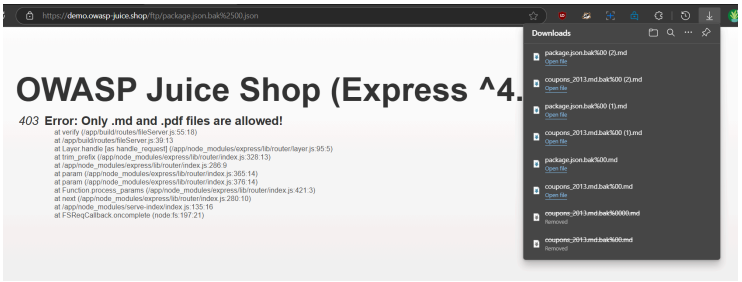
Enjoy the file.

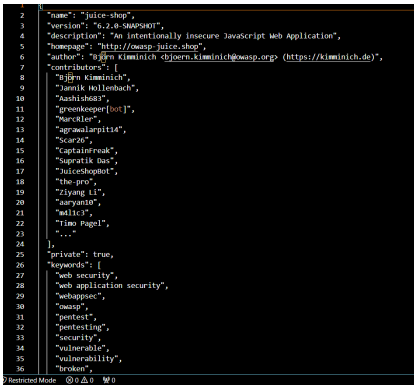
Wait, what about the .json one? Well we just add .bak followed by %2500 then .json, just like last time (except that this time the file type is .json)



No way! Error. *Hmm … only .md and .pdf files*.

Well, let us try to replace .json with .md, since that file is allowed to be downloaded. We can always rename the .md extension back to .json later on.



	 That worked :)
Recommendation	• Always double-check your directory for old .bak files especially those which might contain sensitive data • Do maintenance for your directory!
Reference	https://pwning.owasp-juice.shop/companion-guide/latest/part2/sensitive-data-exposure.html


```
{
  "name": "juice-shop",
  "version": "6.2.0-SNAPSHOT",
  "description": "An intentionally insecure JavaScript Web Application",
  "homepage": "http://owasp-juice.shop",
  "author": "Björn Kimminich <bjoern.kimminich@owasp.org> (https://kimminich.de)",
  "contributors": [
    "Björn Kimminich",
    "Jannik Hollenbach",
    "Aashish683",
    "greenkeeper[bot]",
    "MarcRler",
    "agrawalarpit14",
    "Scar26",
    "CaptainFreak",
    "Supratik Das",
    "Juiceshopbot",
    "the-pro",
    "Ziyang Li",
    "aaryan10",
    "m4lic3",
    "Timo Pagel",
    "..."
  ],
  "private": true,
  "keywords": [
    "web security",
    "web application security",
    "webappsec",
    "owasp",
    "pentest",
    "pentesting",
    "security",
    "vulnerable",
    "vulnerability",
    "broken",
```


THANK YOU!