

Vulnerable Web Application



Shashank Shekhar
01/07/2024

Project Scenario

You have been hired by a startup company, USociety, which has received reports from the well-known hacker group Fcity that their customer data was breached. They need you to identify how the attackers got into their system, extracted all of their customer's data, and identify any other security holes that their application might have. This security audit is considered the company's highest priority, and they need your help.

You will need to identify which OWASP Top 10 security issues the vulnerabilities belong to and their severity. You will also correctly match attack vectors to the OWASP Top 10 definition.

You will be given some static code analysis results and a web application. You will conduct a manual web application test to find all vulnerabilities and create write-up documentation to help the development team patch the code. The write-up documentation should clearly outline the steps needed to reproduce the security issue and best practices to support the development team in better understanding the issue.

At the end of this project, you will have the hands-on skills needed to tackle application security. You should be proud of this accomplishment.



Section One: Static Code Scan

Static Code Scan

In this part of the project, your task is to review the results of the static code scan performed on the application code. You need to identify and report comprehensive information for each security issue found in the scan. The identified issues are listed on the following pages.

For Each Issue:

- Set the Severity Type:
 - There are only Critical, High, or Medium vulnerabilities on the slides.
 - **There is one False Positive** among the issues.
 - Be diligent in your assessment because if everything is marked as Critical, then nothing is truly Critical.
- Associate with an OWASP TOP 10 Reference:
 - Link each issue to a relevant OWASP TOP 10 category.
 - You can refer to either the 2017 or 2021 OWASP TOP 10 lists.
- Provide a High-Level Recommendation:
 - Suggest a high-level solution or best practice to address and fix the issue effectively.



Issue: B106

>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: 'mysecurepassword'

Location: SampleCode/init_db.py:14

```
13     def open(self):
14         self.conn = psycopg2.connect(user = "webappuser",
15                                     password = "mysecurepassword",
16                                     host = "localhost",
17                                     port = "5432",
18                                     database = "website")
19         self.cursor = self.conn.cursor()
```

Severity:	Medium
OWASP TOP 10 reference:	A06:2017 – Security Misconfiguration
Remediation recommendation	
<p>We should never hardcode default passwords.</p> <ol style="list-style-type: none">1. Use environment variables to store sensitive information.2. Use configuration management tools <p>https://bandit.readthedocs.io/en/1.7.0/plugins/b106_hardcoded_password_funcarg.html</p>	



Issue: B108

>> Issue: [B108:hardcoded_tmp_directory] Use of a hardcoded temporary directory.

Location: SampleCode/temp_file.py:19

```
18 def createTempFile(data):
19     temp_path = "/tmp/tempfile.txt"
20     with open(temp_path, 'w') as temp_file:
21         temp_file.write(data)
22     return temp_path
```

Severity:	<i>Medium</i>
OWASP TOP 10 reference:	<i>A5:2017-Broken Access Control</i>
Remediation recommendation	
<i>Use secured temporary file creation methods</i> <ol style="list-style-type: none"><i>1. Use standard libraries like Python tempfile.</i><i>2. Avoid hardcoding paths.</i><i>3. Use secure randomness.</i><i>4. Ensure proper file permission.</i><i>5. Handle file cleanup.</i>	



Issue: B303

>> Issue: [B303:blacklist] Use of insecure MD2, MD4, MD5, or SHA1 hash function.

Location: SampleCode/create_customer.py:23

```
22         self.email = email
23         self.password =
hashlib.md5(password.encode('utf-8')).hexdigest()
24         self.banner = safestring.mark_safe(banner)
```

Severity:	<i>Medium</i>
OWASP TOP 10 reference:	<i>A03:2017 – Sensitive Data Exposure</i>
Remediation recommendation	
<p><i>MD5 should never be used in passwords, we need to migrate to another slow hashing such as bcrypt.</i></p> <ol style="list-style-type: none"><i>1. Replace insecure hash functions with strong, cryptographically secure hash functions designed for password hashing, such as bcrypt, Argon2, or PBKDF2.</i><i>2. Always use a salt (a random value) when hashing passwords. This ensures that even if two users have the same password, their hashes will be different.</i><i>3. Use multiple iterations of the hashing function to slow down brute-force attacks.</i> <p><i>https://bandit.readthedocs.io/en/1.6.2/blacklists/blacklist_calls.html#b303-md5</i></p>	



Issue: B311

>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.

Location: SampleCode/init_db.py:40

```
39         letters = string.ascii_lowercase
40         result_str = ''.join(random.choice(letters) for i in
range(length))
41         return result_str
```

Severity:	<i>False Positive</i>
OWASP TOP 10 reference:	<i>False Positive</i>
Remediation recommendation	
<i>FALSE POSITIVE, Ignore the warning</i>	
<i>https://docs.openstack.org/bandit/1.2.0/blacklists/blacklist_calls.html#b311-random</i>	



Issue: B320

>> Issue: [B320:blacklist] Using `lxml.etree.fromstring` to parse untrusted XML data is known to be vulnerable to XML attacks. Replace `lxml.etree.fromstring` with its `defusedxml` equivalent function.

Location: `SampleCode/fix_customer_orders.py:11`

```
10 def customerOrdersXML():
11     root = lxml.etree.fromstring(xmlString)
12     root = fromstring(xmlString)
```

Severity:	<i>Critical</i>
OWASP TOP 10 reference:	<i>OWASP: A04:2017 – XML External Entities (XXE)</i>
Remediation recommendation	
<ul style="list-style-type: none">• <i>Replace <code>lxml.etree.fromstring</code> with a secure XML parsing library designed to handle untrusted XML data, such as <code>defusedxml</code>.</i>• <i>Ensure that the XML parser is configured to disable the processing of external entities.</i>• <i>Perform validation and sanitization on the XML input to ensure it conforms to expected schemas and does not contain any harmful content.</i>	



Issue: B603

>> Issue: [B603:subprocess_without_shell_equals_true] subprocess call - check for execution of untrusted input.

Location: SampleCode/onLogin.py:8

```
7     def process(self, user, startupcmd):  
8         p = subprocess.Popen([startupcmd],  
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)  
9         r = p.communicate()[0]
```

Severity:

High

OWASP TOP 10 reference:

A09:2017-Using Components with Known Vulnerabilities

Remediation recommendation

The issue typically arises when using the subprocess module without setting shell=True, potentially leading to command injection vulnerabilities. To address this, ensure that you properly validate and sanitize any input before passing it to subprocess calls.

Avoid shell=True: Instead of setting shell=True, construct the command and arguments as a list and pass them directly to the subprocess module. This helps prevent shell injection vulnerabilities by avoiding shell interpretation of input.



Issue: B703

>> Issue: [B703:django_mark_safe] Potential XSS on mark_safe function.

Location: SampleCode/create_customer.py:24

```
23         self.password =  
hashlib.md5(password.encode('utf-8')).hexdigest()  
24         self.banner = safestring.mark_safe(banner)  
25
```

Severity:

Critical

OWASP TOP 10 reference:

A07: 2017 Cross-Site Scripting XSS

Remediation recommendation

- *Avoid using mark_safe unless you are absolutely sure the input is safe.*
- *Use Django's built-in functions and libraries to properly sanitize and escape user input before rendering it in templates.*
- *Implement strict validation rules for any input that is marked as safe. Ensure that it conforms to expected patterns and does not contain any malicious content.*
- *Utilize Django's templating engine, which automatically escapes variables by default, to prevent XSS vulnerabilities. Only use mark_safe when absolutely necessary and after proper sanitization.*



Section Two:

Assess the Web Application

Manually Assess the Web Application and Identify Vulnerabilities

There are 13 vulnerabilities present in the web application. You have to manually assess at least 5 of the 13 vulnerabilities present in the application to be able to complete your report. You need to find at least 1 of each type of the following security vulnerabilities:

- Broken Authentication
- XSS
- Broken Access
- Sensitive Data Exposure
- SQLi

You are not allowed to use any automated scanner, as you can't refer to them in your report.

There is no deliverable in this section, but you need to write a report about the vulnerabilities in the next section!

Tools you may need

We have provided some Python files available in the `/workspace/tools/` directory of the workspace that might be useful. You are only allowed to use these provided scripts because you are not allowed to refer to any other tool, like automated scanners, in the report.

- Use the `bruteforce.py` tool for brute force attacks.
- Use the `performbase64` file to encode/decode a value to Base64:
- Use the `hashid.py` and `checkhash.py` files to decode any hash values.
- If you need, you can also refer to the Hashing exercise's workspace present in one of the previous lessons to decode the hash value.
- You can also find the `test-password.txt` and `test-username.txt` in the folder; use these when you need a password or a username list.

Instructions

If you go to the **Web Application Environment page**, you will find a pre-setup environment for you to analyze the web application. Setting up your env is as simple as initiating the application and then using the “Start App” button. To view the website, you will need to click the “Open App” button.

VulnWebApp

Username	<input type="text" value="Username"/>
Password	<input type="password" value="Password"/>
Reset DB	Sign in

Tips

1. **Break into the application:** Start with breaking into the application using the brute force approach.
2. As soon as you break into the application, explore the different areas in the application to see what vulnerabilities you can find.
 - a. Profile
 - b. Customers
 - c. Users
3. **Profile Section:** Visit the profile section, and try various measures just like an attacker.
 - a. See if you can abuse the input fields.
 - b. Examine the Cookie to notice any sensitive data.
 - c. Try to elevate your role as an Admin. Remember, it becomes easier to make attacks such as SQLi only after becoming an Admin.
 - d. If you succeed to elevate your role to Admin, it will prove multiple other vulnerabilities.

Tips

4. **Customer Section:** Once you possess the Admin access to the Application, the next thing you would want to do is obtain the details of all the customers.
 - a. SQLi could be your preferred choice of attack to view customers' details.
 - b. If you can view any customer's details, then you'd look for any sensitive data, such as login credentials.
5. **Users:**
 - c. As an attacker, your next step would be to fetch sensitive data for all the users.
 - d. See if you can abuse the query string params.

Important

You might think that you don't need to log or track security items that are "similar". But make sure that you track all of them so that the developer/tester can fix/verify the issue properly.



Section Three: Security Report

Security Report

In the following slides, you need to provide a detailed report of six different types of vulnerabilities you found in the web application. The slides are named for each issue, so it is clear what to put where. For the walkthrough part, you can add as many slides as you see fit after the slide that identifies the issue.

Your report has to include at least one finding of these vulnerabilities: Broken Authentication, XSS, Broken Access, Sensitive Data Exposure, SQLi.

For each identified issue, include the following components:

- Severity: Assign a severity level to the issue.
- OWASP TOP 10 Reference: Specify the relevant OWASP TOP 10 issue reference (can be 2017 or 2021).
- Vulnerability Explanation: Clearly explain the specific vulnerability you discovered in the web application.
- Recommendation: Provide a high-level recommendation on how to fix the issue. This should not be a step-by-step guide but rather a strategic approach to remediation.
- Steps to Reproduce the Issue: Detail the process of replicating the vulnerability. Include:
 - A written walkthrough of each step.
 - Annotated screenshots to support your explanation.

Important Considerations

- **Accuracy in Severity and OWASP Reference:** Accurately identify the severity and corresponding OWASP TOP 10 reference. Common mistakes include incorrectly assigning severity levels and misidentifying the OWASP category. Remember, not every issue is Critical or High; prioritize accurately to help developers focus their efforts effectively.
- **Contextual Vulnerability Description:** Describe the vulnerability as it specifically pertains to the web application you are testing. Do not provide a generic description.
- **Detailed Reproduction Steps:** Ensure your steps to reproduce the issue are comprehensive. Use a combination of clear writing and annotated screenshots to guide the reader through the process. Assume the reader may not have advanced security or technical knowledge.
- You are only allowed to use the provided scripts. Using or referring to automated scanners is not an acceptable solution.
- The template slides are named, but they are not necessarily in order!
- There are a total of 13 vulnerabilities on the server. For a standout project, include more than the required 5 vulnerabilities in your report!
- There is an example slide that uses an imaginary SSRF vulnerability but without the step-by-step walkthrough.



Broken Authentication

Severity:	High
OWASP TOP 10 reference:	A02: 2017 Broken Authentication
Vulnerability Explanation	
<p>Brute force login vulnerabilities occur when an attacker can repeatedly attempt to guess usernames and passwords without sufficient limitations or deterrents. This vulnerability allows attackers to automate the process of guessing credentials and eventually gain unauthorized access to user accounts if they succeed. Common indicators of this vulnerability include the absence of rate limiting, lack of account lockout mechanisms, and insufficient monitoring of login attempts.</p>	
Remediation recommendation	
<ol style="list-style-type: none">1. <i>Rate Limiting:</i> Implement rate limiting on login attempts to restrict the number of login attempts from a single IP address or account within a specified timeframe.2. <i>Account Lockout Mechanism:</i> Introduce an account lockout mechanism that temporarily locks an account after a certain number of consecutive failed login attempts.3. <i>Captcha:</i> Use CAPTCHA or reCAPTCHA to distinguish between human and automated login attempts after a certain number of failed attempts.4. <i>Multi-Factor Authentication (MFA):</i> Enforce multi-factor authentication (MFA) to add an extra layer of security.	



Broken Authentication

Use the **bruteforce.py** python tool which uses a collection of usernames and passwords to brute force attack.

I have used the below command in the terminal to do the attack:

```
python bruteforce.py -U test-username.txt -P test-password.txt -d
username='USR':password='PWD' -m POST -f "Login Failed"
http://localhost:3000/login
```

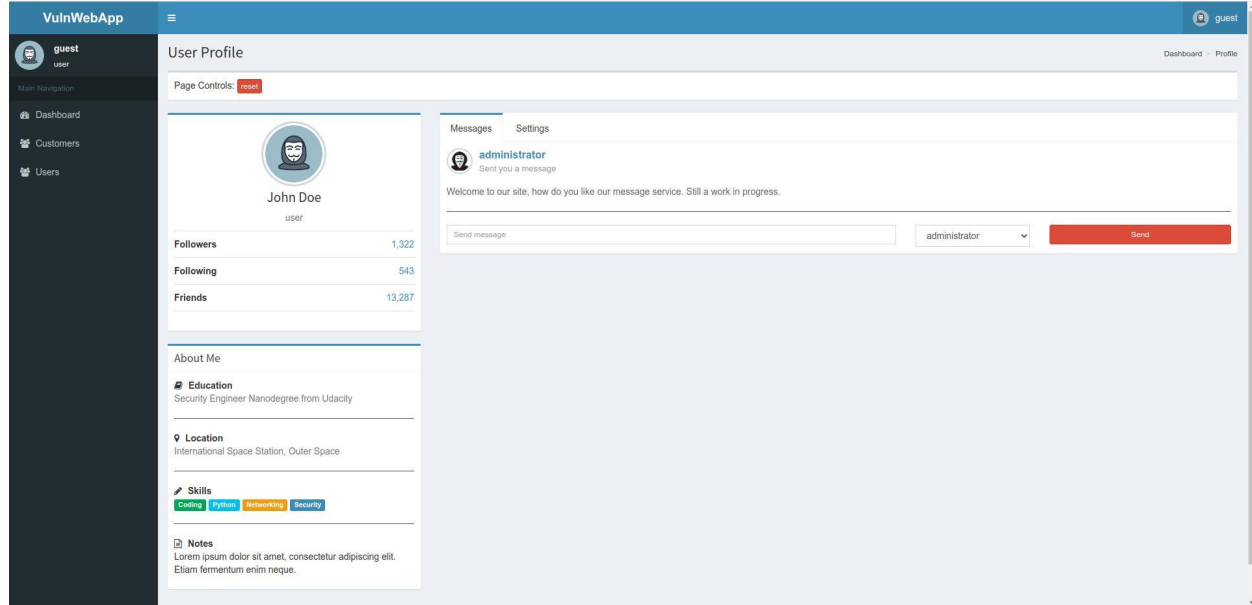
The screenshot shows a web application environment with a terminal window. The terminal displays a log of login attempts. The log shows multiple failed login attempts for various usernames and passwords. The final line of the log, highlighted with a green box, shows a successful login for the username 'guest' and password 'orange'. Below this line, the text 'This is a demo code used for this training.' and 'tpols roots' are visible.

```
[-] Login Failed! ('username': 'test', 'password': '2000')
[-] Login Failed! ('username': 'test', 'password': 'test')
[-] Login Failed! ('username': 'test', 'password': 'trustno1')
[-] Login Failed! ('username': 'test', 'password': 'thomas')
[-] Login Failed! ('username': 'test', 'password': '2020')
[-] Login Failed! ('username': 'test', 'password': 'tiger')
[-] Login Failed! ('username': 'test', 'password': 'asdfasdf')
[-] Login Failed! ('username': 'guest', 'password': '123456')
[-] Login Failed! ('username': 'guest', 'password': 'password')
[-] Login Failed! ('username': 'guest', 'password': '12345678')
[-] Login Failed! ('username': 'guest', 'password': '1234')
[-] Login Failed! ('username': 'guest', 'password': '12345')
[-] Login Failed! ('username': 'guest', 'password': 'dragon')
[-] Login Failed! ('username': 'guest', 'password': 'qwerty')
[-] Login Failed! ('username': 'guest', 'password': 'asdf')
[-] Login Failed! ('username': 'guest', 'password': 'mustang')
[-] Login Failed! ('username': 'guest', 'password': 'letmein')
[-] Login Failed! ('username': 'guest', 'password': 'baseball')
[-] Login Failed! ('username': 'guest', 'password': 'master')
[-] Login Failed! ('username': 'guest', 'password': 'michael')
[-] Login Failed! ('username': 'guest', 'password': 'football')
[-] Login Failed! ('username': 'guest', 'password': 'shadow')
[-] Login Failed! ('username': 'guest', 'password': 'qazwsx')
[-] Login Failed! ('username': 'guest', 'password': 'monkey')
[-] Login Failed! ('username': 'guest', 'password': 'abc123')
[-] Login Failed! ('username': 'guest', 'password': '1234567890')
[-] Login Failed! ('username': 'guest', 'password': 'pass')
[-] Login Failed! ('username': 'guest', 'password': 'computer')
[-] Login Failed! ('username': 'guest', 'password': 'liver!')
[-] Login Failed! ('username': 'guest', 'password': 'jennifer')
[-] Login Failed! ('username': 'guest', 'password': 'klaster')
[-] Login Failed! ('username': 'guest', 'password': 'mumassword')
[+] Login Found! ('username': 'guest', 'password': 'orange')
This is a demo code used for this training.
tpols roots
```

Running this command gives username and password that exists for that app to log in successfully.



Broken Authentication





Broken Authentication

Severity:	High
OWASP TOP 10 reference:	A02: 2017 Broken Authentication
Vulnerability Explanation	
<p><i>The cookie can be manipulated to gain admin rights in the web app.</i></p> <p><i>I have modified the cookie values in the web page and I was able to gain admin access to the web app.</i></p>	
Remediation recommendation	
<ol style="list-style-type: none"><i>1. Validate and Secure Cookies: Ensure that all cookies, especially those related to authentication and authorization, are properly validated on the server-side.</i><i>2. Use Secure Cookie Attributes: Set secure cookie attributes such as HttpOnly, Secure, and SameSite to enhance cookie security.</i><i>3. Implement Strong Authentication Controls: Use strong authentication mechanisms, including multi-factor authentication (MFA), to enhance security. Ensure that session tokens are properly generated, securely stored, and invalidated after logout or inactivity.</i>	



Broken Authentication

1. *Login Normally*
2. *Open developer tool using F12*
3. *Go to the Cookies tab*

The screenshot shows the VulnWebApp dashboard. The top navigation bar includes a 'guest' user profile and a 'Dashboard' link. The main content area displays a 'Monthly Recap Report' for the period 'Sales: 1 Jan, 2014 - 30 Jul, 2014'. Below the report, there are two progress bars: 'Add Products to Cart' at 160/200 and 'Complete Purchase' at 310/400. The Chrome DevTools 'Cookies' tab is open, showing a table of cookies. The first cookie is highlighted, with its name 'authinfo' and value 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMTQ1NiIsImNpdCI6ImJp1c2Vy'.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition Key	Priority
authinfo	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMTQ1NiIsImNpdCI6ImJp1c2Vy	eg4ds12ue7.prod.udacity-student-workspaces.com	/	Session	16			None		Medium

4. *Perform base64 decoding of the cookie value which is returned for the user.*

Web Application Environment

The screenshot shows a terminal window with the following commands and output:

```
tools root$ python performbase64.py -d Mjplc2Vy
2:user
tools root$
```



Broken Authentication

5. Encode **1:admin** via base64 to get the value to be replaced.

Web Application Environment

```
tools roots$ python performbase64.py 1:admin
MTphZG1pbG==
tools roots$
```

6. Replace the original cookie with the new encoded one and refresh the page.

The screenshot shows the VulnWebApp dashboard with the user logged in as 'administrator'. The dashboard includes a sidebar with navigation links (Dashboard, Customers, Users), a main content area with various widgets (CPU Traffic, Likes, Sales, Goal Completion), and a footer with a console and network tab. The browser's developer tools are open, showing the 'Cookies' tab with a list of cookies. The 'authinfo' cookie is highlighted, showing its value as 'MTphZG1pbG==', which is the base64-encoded value of '1:admin'.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition Key	Priority
authinfo	MTphZG1pbG==	eg4ds12ue7.prod.udacity-student-workspaces.com	/	Session	20	✓	✓	None		Medium
uworkspaces	eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMTQ1NiIsImF1dG8iOiJ1cm9kaW50IiwiaWF0IjoxNDUwMDAwMDAwfQ.	prod.udacity-student-workspaces.com	/	Session	1002	✓	✓	None		Medium



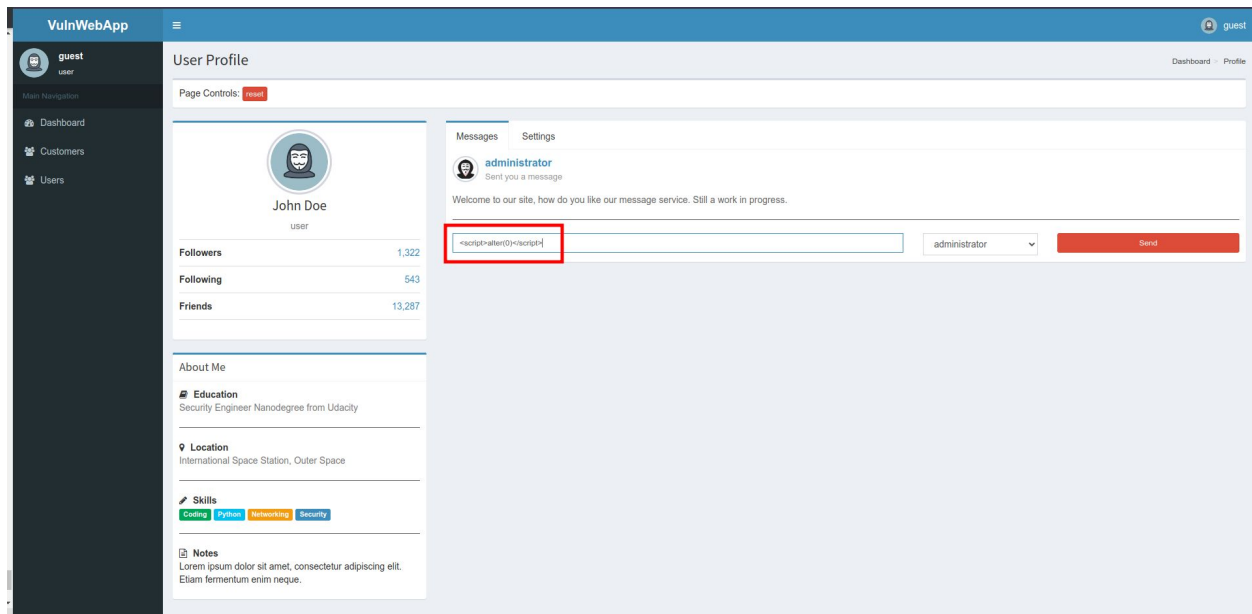
Cross-Site Scripting (XSS)

Severity:	Critical
OWASP TOP 10 reference:	A07: 2017 Cross-Site Scripting XSS
Vulnerability Explanation	
<p><i>Cross-site scripting (XSS) is a common web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. These scripts can steal sensitive information, hijack user sessions, or deface websites.</i></p> <p><i>In the internal chat section, the system is not doing any any sanitization and allowing scripts, or tags with javascript code on the client side. I tested by sending<script>alert(1)</script> in the internal chat box.</i></p>	
Remediation recommendation	
<ol style="list-style-type: none"><i>1. Preventing XSS requires the separation of untrusted data from active browser content. This can be achieved by:</i><i>2. Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, and React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases that are not covered.</i><i>3. Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The <u>OWASP Cheat Sheet 'XSS Prevention'</u> has details on the required data escaping techniques.</i><i>4. Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context-sensitive escaping techniques can be applied to browser APIs as described in the <u>OWASP Cheat Sheet 'DOM based XSS Prevention'</u>.</i>	

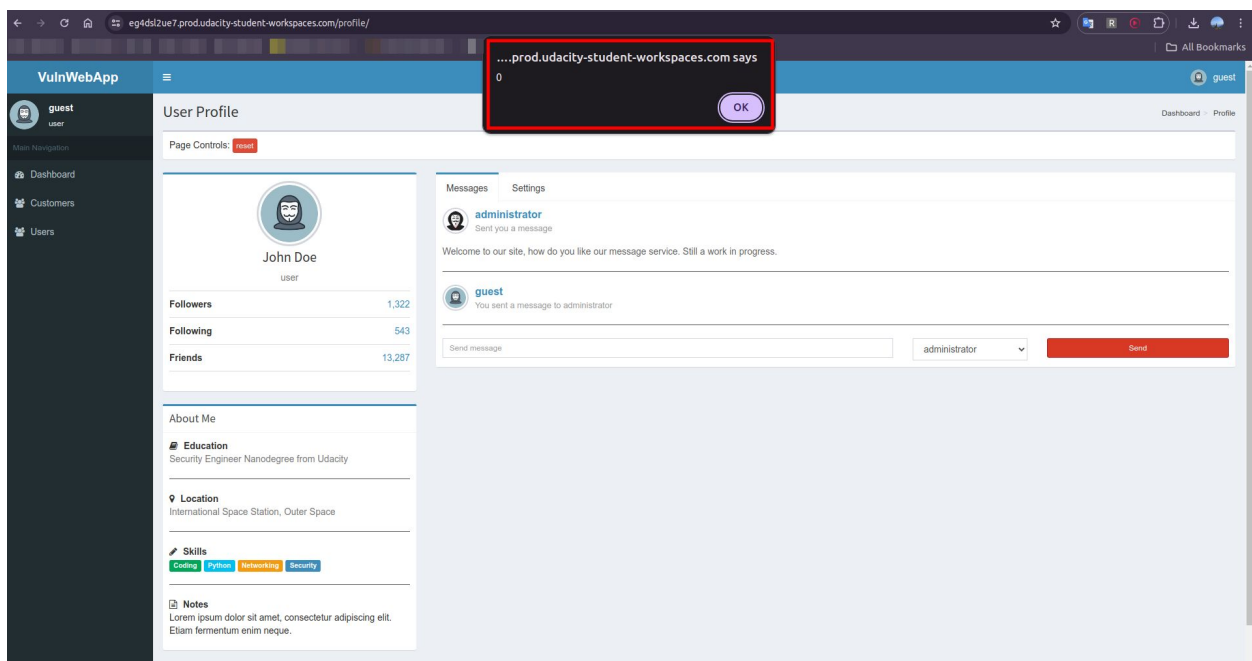


Cross-Site Scripting (XSS)

1. Login normally to the web app.
2. Goto user profile section and insert a script say `<script>alert(0)</script>` in the chat section.



3. With this, we are able to inject javascript in the chat section.





Broken Access

Severity:	High
OWASP TOP 10 reference:	A05: 2017 Broken Access Control
Vulnerability Explanation	
<p>One of the two request objects contains an async call to the server to retrieve the customer's current information.</p> <p>By altering the initial request, we can use the API to retrieve the data of the other customer without any valid authentication.</p>	
Remediation recommendation	
<ol style="list-style-type: none">1. Prevent users from scrapping all the data from the web application by rate limiting the access to the data on the site.2. Refuse access to the non-public pages and validation to access these pages.3. <u>OWASP Proactive Controls: Enforce Access Controls</u>	



Broken Access

1. Login Normally.
2. Gain admin access through cookie manipulation.
3. Go to the customer page.

ID	First Name	Last Name	Username	Options
1	paul	doe	pdoe	View
2	jake	doe	jdoo	View
3	dave	doe	ddoe	View
4	mike	doe	mdoe	View
5	nick	doe	ndoe	View

4. Hit the view button.
5. Go to the xhr file, then go to the response tab, and then open a new window by double-clicking it.

```
// 20240525004005
// https://eg4ds12ue7.prod.udacity-student-workspaces.com/customers/id/17_1716577662279

[
  1,
  "paul",
  "doe",
  "pdoe",
  "d8578edf8458ce06fbc5bb76a58c5ca4"
]
```



Broken Access

6. Here in the address bar customer ID can be modified to retrieve the information of other users.

```
// 20240525004216
// https://eg4dsl2ue7.prod.udacity-student-workspaces.com/customers/id/2?_=1716577662279

[
  {
    2,
    "jake",
    "doe",
    "jdoe",
    "5f4dcc3b5aa765d61d8327deb882cf99"
  }
]
```

```
// 20240525004246
// https://eg4dsl2ue7.prod.udacity-student-workspaces.com/customers/id/4?_=1716577662279

[
  {
    4,
    "mike",
    "doe",
    "mdoe",
    "8621ffdbc5698829397d97767ac13db3"
  }
]
```

Severity:**Critical****OWASP TOP 10 reference:****A01: 2017 Injection**

Vulnerability Explanation

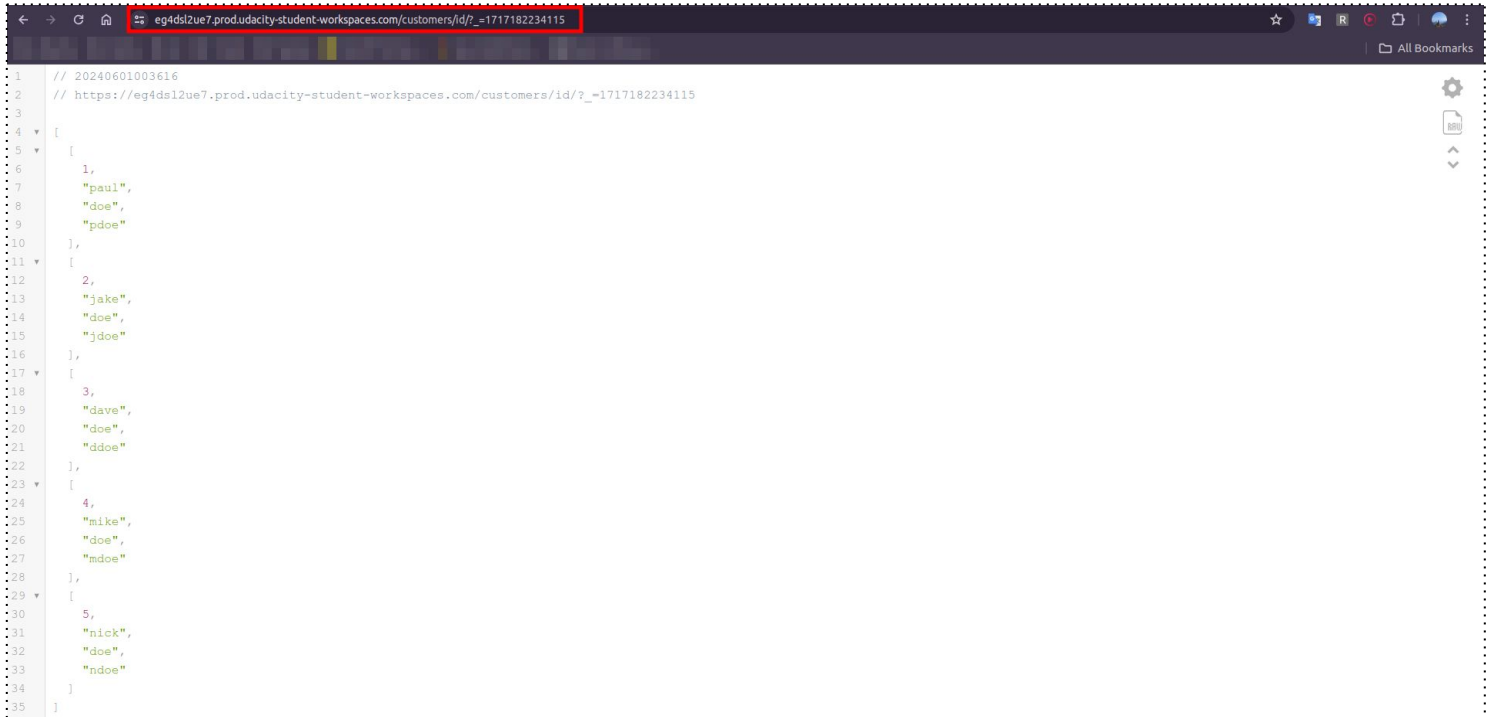
Inserting ' or 1 = '1 at the end of the url reveals all the user sensitive information. Ability to display the customer credentials with proper access/login using the sql injection in the url.

SQL injection (SQLi) is a type of security vulnerability that occurs when an attacker can manipulate SQL queries sent to a database through a web application. This manipulation can lead to unauthorized access to sensitive data, data manipulation, and in some cases, complete compromise of the underlying server.

Remediation recommendation

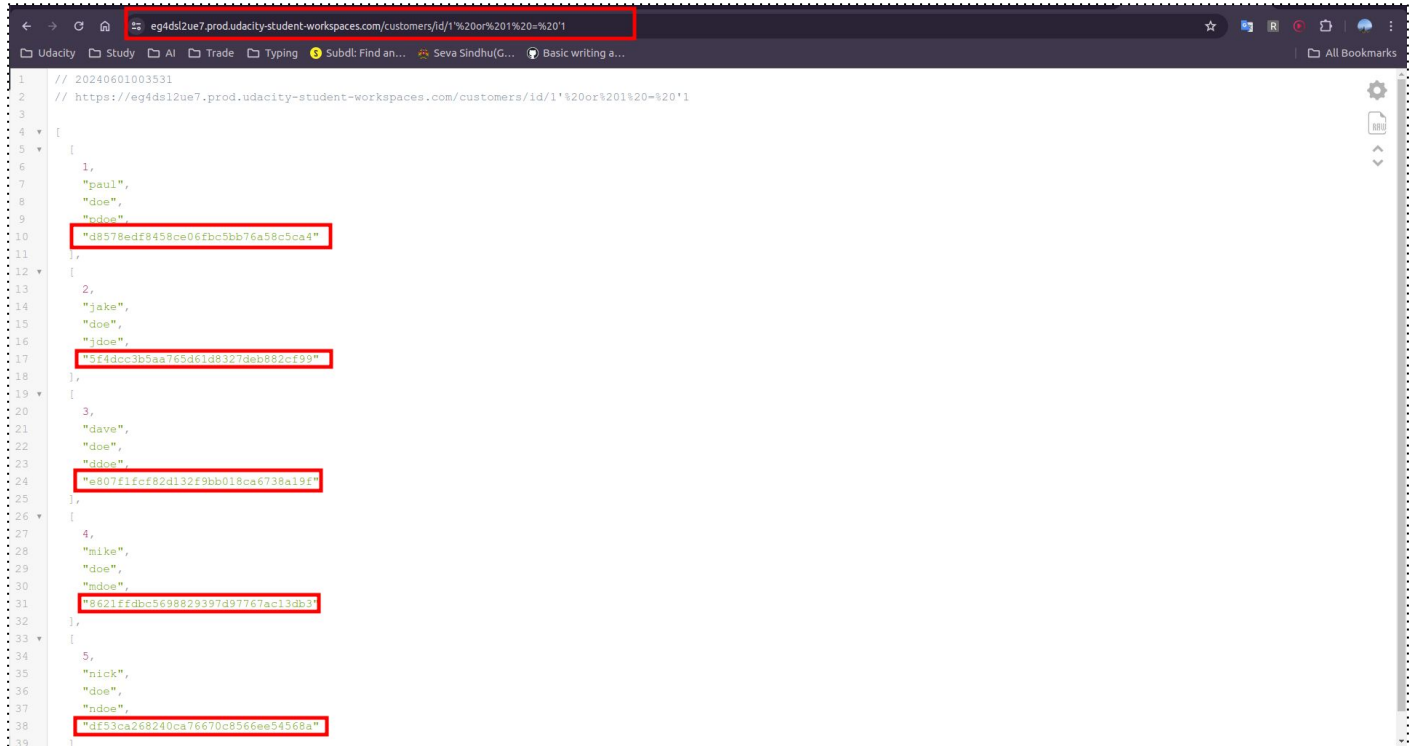
- 1. The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).*
- 2. Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().*
- 3. Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.*
- 4. For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.*
- 5. [OWASP Proactive Controls: Secure Database Access](#)*
- 6. https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html*

1. *Login to the web app*
2. *Gain admin privileges via cookie manipulation.*
3. *Go to the customers tab*



```
1 // 20240601003616
2 // https://eg4ds12ue7.prod.udacity-student-workspaces.com/customers/id/?_=1717182234115
3
4 {
5   [
6     1,
7     "paul",
8     "doe",
9     "pdoe"
10  ],
11   [
12     2,
13     "jake",
14     "doe",
15     "jdoe"
16  ],
17   [
18     3,
19     "dave",
20     "doe",
21     "ddoe"
22  ],
23   [
24     4,
25     "mike",
26     "doe",
27     "mdoe"
28  ],
29   [
30     5,
31     "nick",
32     "doe",
33     "ndoe"
34  ]
35  ]
```

4. Edit the URL and add ' or 1=' 1 towards the end of the url.



```
1 // 20240601003531
2 // https://eg4ds12ue7.prod.udacity-student-workspaces.com/customers/id/1'%20or%201%20=%20'1
3
4 [
5   [
6     1,
7     "paul",
8     "doe",
9     "evlca",
10    "d8578edf8458ce06fbc5bb76a58c5ca4"
11  ],
12   [
13     2,
14     "jake",
15     "doe",
16     "jdoe",
17     "5f4dccc3b5aa765d61d8327deb882cf99"
18  ],
19   [
20     3,
21     "dave",
22     "doe",
23     "ddoe",
24     "e807f1fcf82d132f9bb018ca6738a19f"
25  ],
26   [
27     4,
28     "mike",
29     "doe",
30     "mdoe",
31     "8621ffdbcf5698829397d97767ac13db0"
32  ],
33   [
34     5,
35     "nick",
36     "doe",
37     "ndoe",
38     "df53ca268240ca76670c8566ee54568a"
39  ]
40 ]
```

5. Use hashid.py and checkhash.py to crack the hashes.



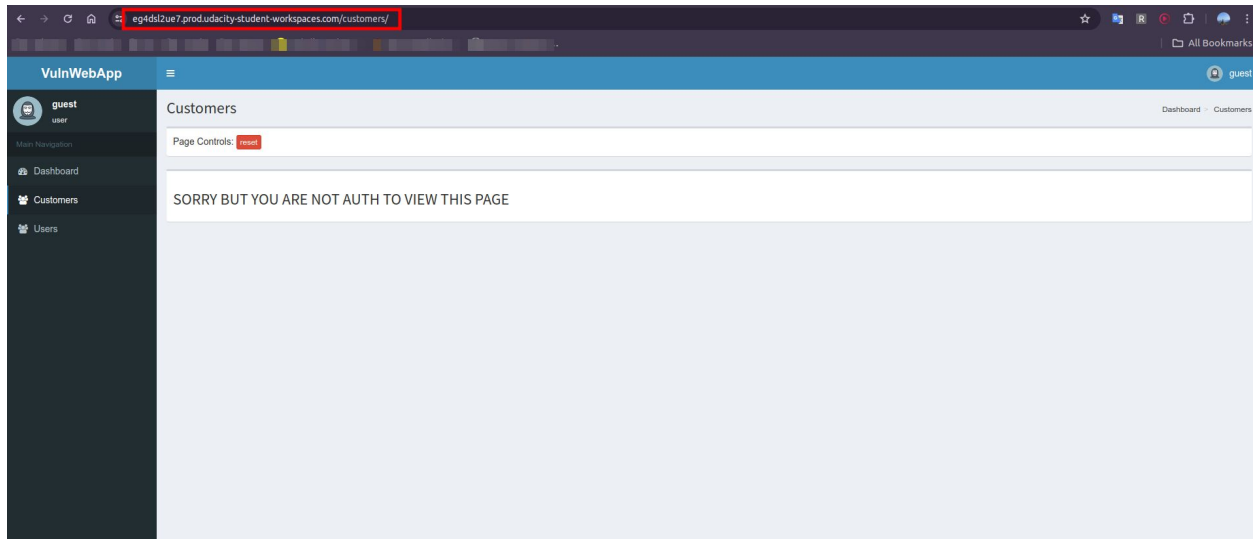
Sensitive Data Exposure

Severity:	High
OWASP TOP 10 reference:	A03: 2017 Sensitive Data Exposure
Vulnerability Explanation	
<p>Attacker can access the data that is not authorized to be accessed. By adding /id/ towards the end of the url the data which is not supposed to be visible can be seen.</p>	
Remediation recommendation	
<ol style="list-style-type: none">1. Use post instead of GET: Use HTTP POST requests to transmit sensitive data instead of GET requests.2. Implement strong Access Control3. Encrypt Sensitive Data4. User secure session management.	

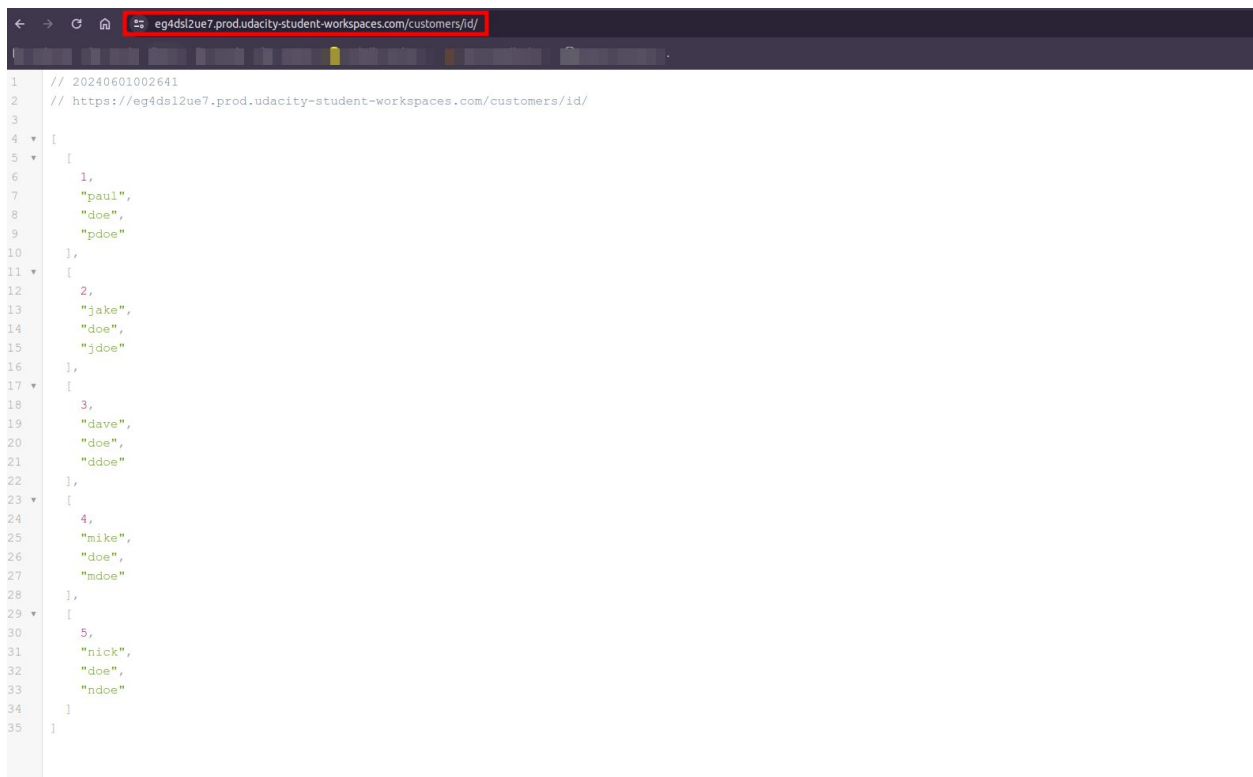


Sensitive Data Exposure

1. Login to the web app normally
2. Navigate to customer tab



3. Add `/id/` to the end of the URL





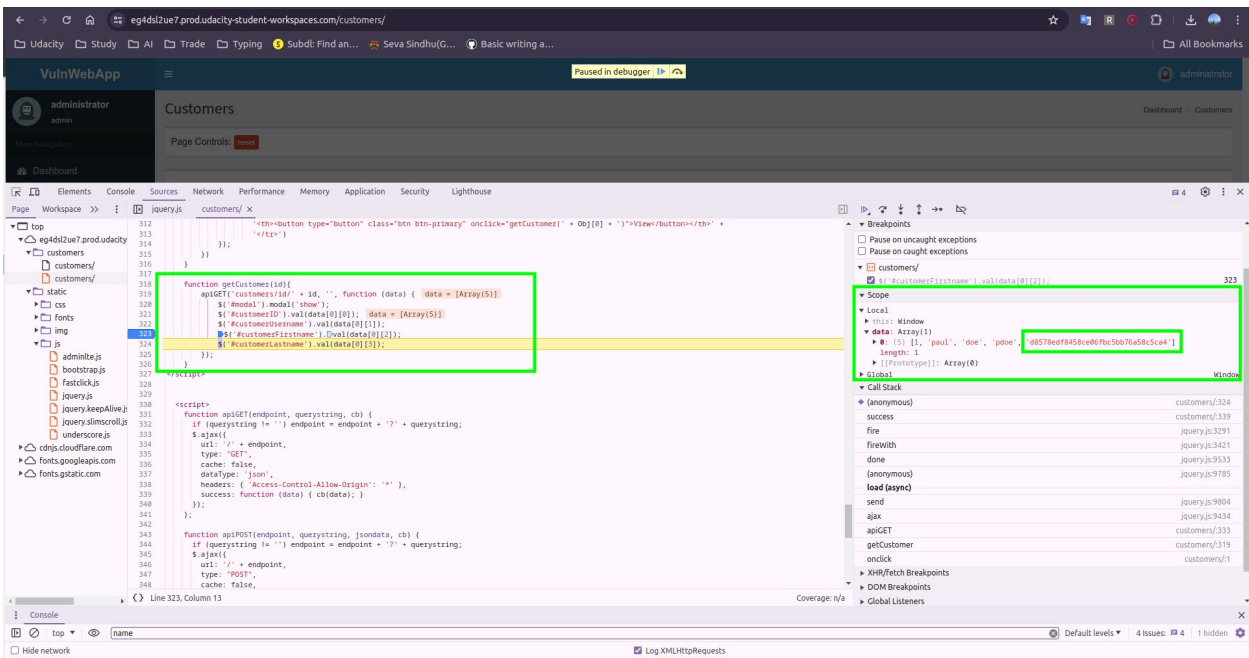
Sensitive Data Exposure

Severity:	High
OWASP TOP 10 reference:	A03: 2017 Sensitive Data Exposure
Vulnerability Explanation	
<p><i>Password hashes are visible to an administrator using a breakpoint in the code. This situation reveals a critical vulnerability which causes exposure to sensitive data.</i></p>	
Remediation recommendation	
<ol style="list-style-type: none"><i>1. Remove Sensitive Data from Code: Ensure that sensitive data such as password hashes are not exposed in code that can be easily accessed by breakpoints or logs.</i><i>2. Implement Proper Access Control: Ensure that only authorized personnel have access to sensitive data and that debugging tools and methods are restricted in production environments.</i><i>3. Sanitize Logs and Debugging Information: Ensure that logs and debugging information do not contain sensitive data. Mask or remove sensitive information before logging.</i>	

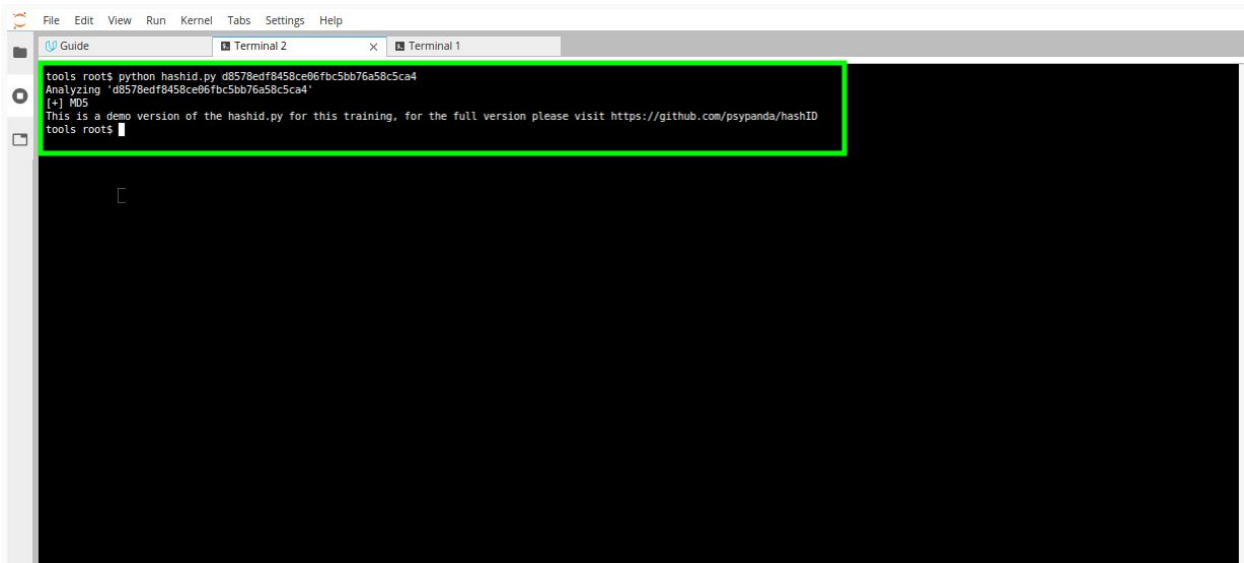


Sensitive Data Exposure

1. Login Normally
2. Gain Admin privilege through cookie manipulation as mentioned in the report.
3. Open developer tools using F12.
4. Go to the sources tab and open customers file.
5. Add breakpoint at line 323.
6. View any of the customer data and use jump button to step to the breakpoint.
7. In the array select the last item which looks like some encrypted data.



8. Use the hashid.py to know the hash type.





Sensitive Data Exposure

9. Use checkhash.py to crack the hash

```
tools roots$ python checkhash.py d8578edf8458ce06fbc5bb76a58c5ca4 -t md5
.....
Hashed value passed = d8578edf8458ce06fbc5bb76a58c5ca4
Hash type passed = md5
Result = qwerty
Actual Hash Type should be = md5
.....
tools roots$
```