UDACITY
Part of Accenture

Vulnerable Web Application

[Aditi Chintarevula]

[13/10/25]

Section One: Static Code Scan

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:	*High*
OWASP TOP 10 reference:	*A02:2021 – Cryptographic Failures*
Remediation recommendation	
Do not hardcode passwords in the source code, as it exposes credentials to anyone with code access. *Store secrets securely using environment variables, configuration files with restricted access, or secret management tools*	

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:	*Medium*
OWASP TOP 10 reference:	*A05: Security Misconfiguration*
Remediation recommendation	
Avoid using hardcoded temporary file paths as they can be predictable and exploited by attackers. *Use system-provided secure temporary directories, such as Python's tempfile module (tempfile.NamedTemporaryFile()), to safely create temporary files.*	

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:	*High*
OWASP TOP 10 reference:	*A02:2021 - Cryptographic Failures*
Remediation recommendation	
Do not use MD5 (or other weak hashes like MD2, MD4, SHA1) for storing passwords or sensitive data. *Use secure, salted hashing algorithms such as bcrypt, scrypt, or argon2 to ensure strong resistance against brute-force and collision attacks.*	

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:	*FALSE POSITIVE*
OWASP TOP 10 reference:	*FALSE POSITIVE*
Remediation recommendation	
This is a false positive, as the code only generates a non-security-critical random reset code. *No change is required, but if used for authentication or tokens, secrets should replace random*	

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:	*High*
OWASP TOP 10 reference:	*A05:2021 – Security Misconfiguration*
Remediation recommendation	
Avoid parsing untrusted XML with lxml.etree.fromstring as it can lead to XXE or other XML-based attacks. *Use safe alternatives like defusedxml.fromstring() and validate or sanitize XML input before processing.*	

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:	*Critical*
OWASP TOP 10 reference:	*A03:2021- Injection*
Remediation recommendation	
Validate and sanitize all user-controlled input before using it in subprocess calls to prevent command injection.	

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:	*High*
OWASP TOP 10 reference:	*A03:2021- Injection*
Remediation recommendation	
Avoid using mark_safe() on untrusted or user-provided input, as it can enable cross-site scripting (XSS) attacks. *Sanitize or escape user-provided data before rendering, and only use mark_safe() on trusted, pre-validated HTML content.*	

Section Three: Security Report

VulnWebApp (VWA) Security Report

Code Revision: 1.0.0.0

Company: USociety

Report: VWA251013

Author: Aditi Chintarevula

Date: 2025-10-13

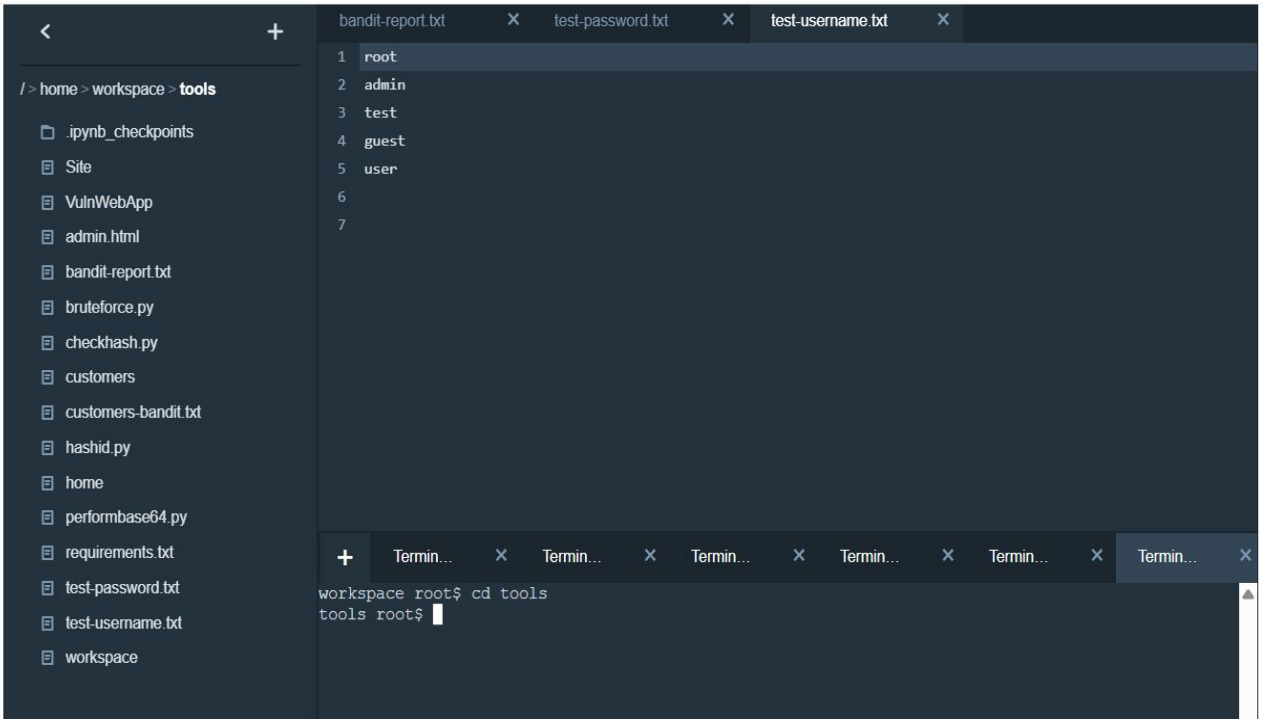
Broken Authentication

Severity:	*High*
OWASP TOP 10 reference:	*A07:2021 - Identification and Authentication Failures*
Vulnerability Explanation	
The application's login form is susceptible to a credential brute-force attack. The system fails to implement effective controls like rate limiting or account lockouts after multiple unsuccessful login attempts. This allows an attacker to continuously submit large lists of potential passwords against a valid username (like 'guest' or 'administrator') until a successful credential pair is found	
Remediation recommendation	
Implement strict rate limiting policies that restrict the number of login attempts per user account and per source IP address over a defined period. Accounts should be temporarily locked out after reaching a configurable threshold (e.g., 5 failures in 5 minutes) to prevent automated guessing.	

Broken Authentication

Steps to Reproduce the Issue:

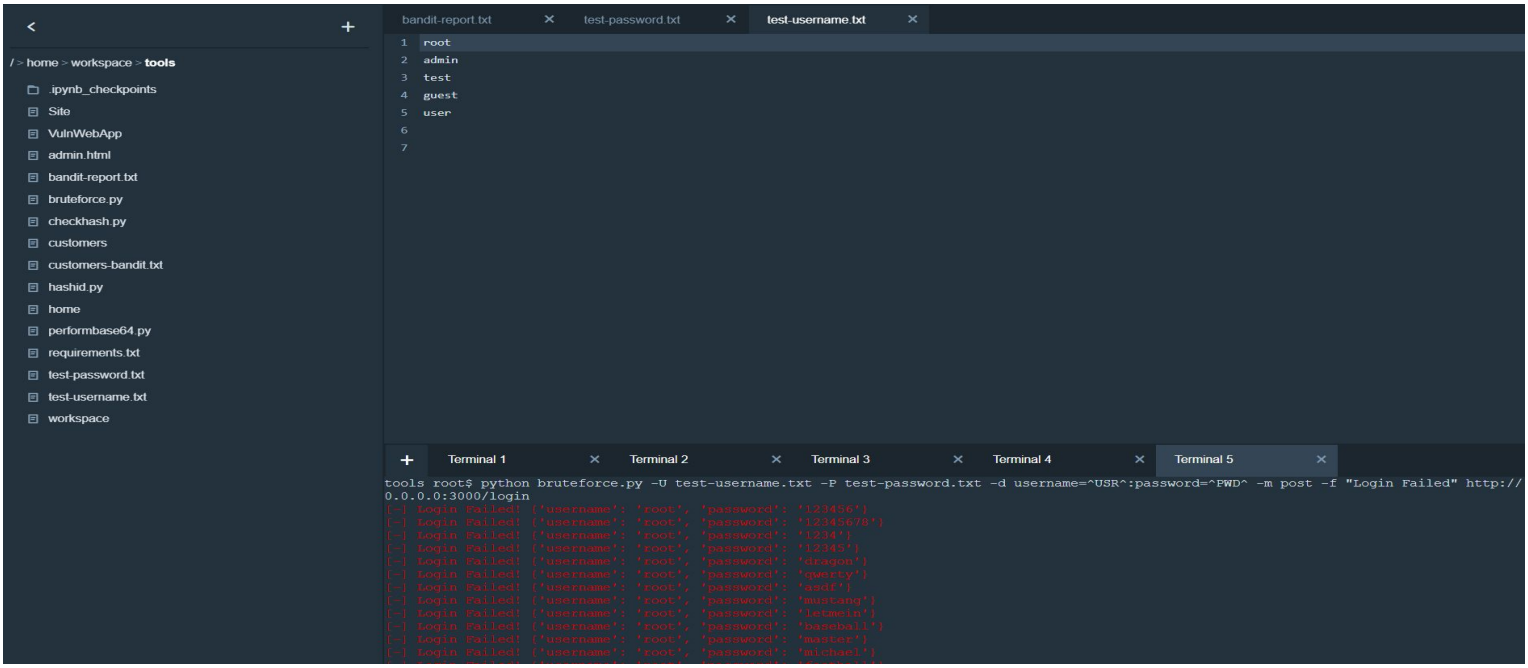
- *Prepare the Attack: Open the workspace Terminal and navigate to the /workspace/tools/ directory.*

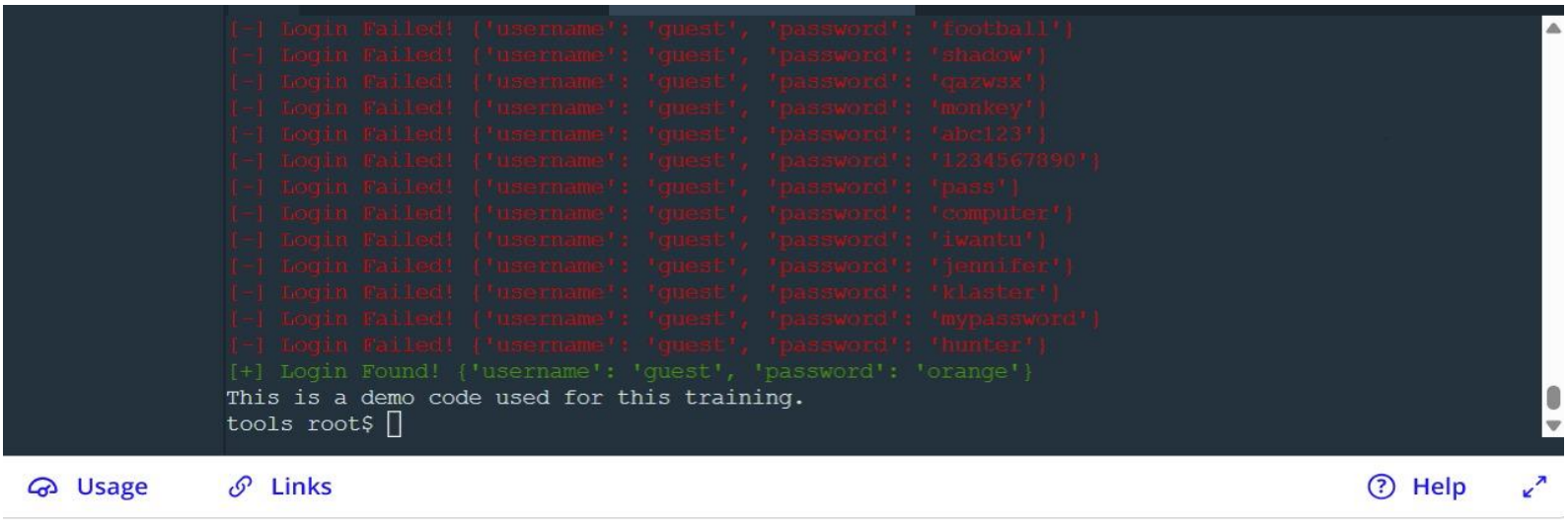


- *Execute Brute Force: Run the bruteforce.py script against the login URL, targeting a known username (e.g., guest) and iterating through the password list.*

Broken Authentication

python bruteforce.py -U test-username.txt -P test-password.txt -d "username={USER}&password={PASS}" -m post -f "Login Failed"http://0.0.0.0:3000/login

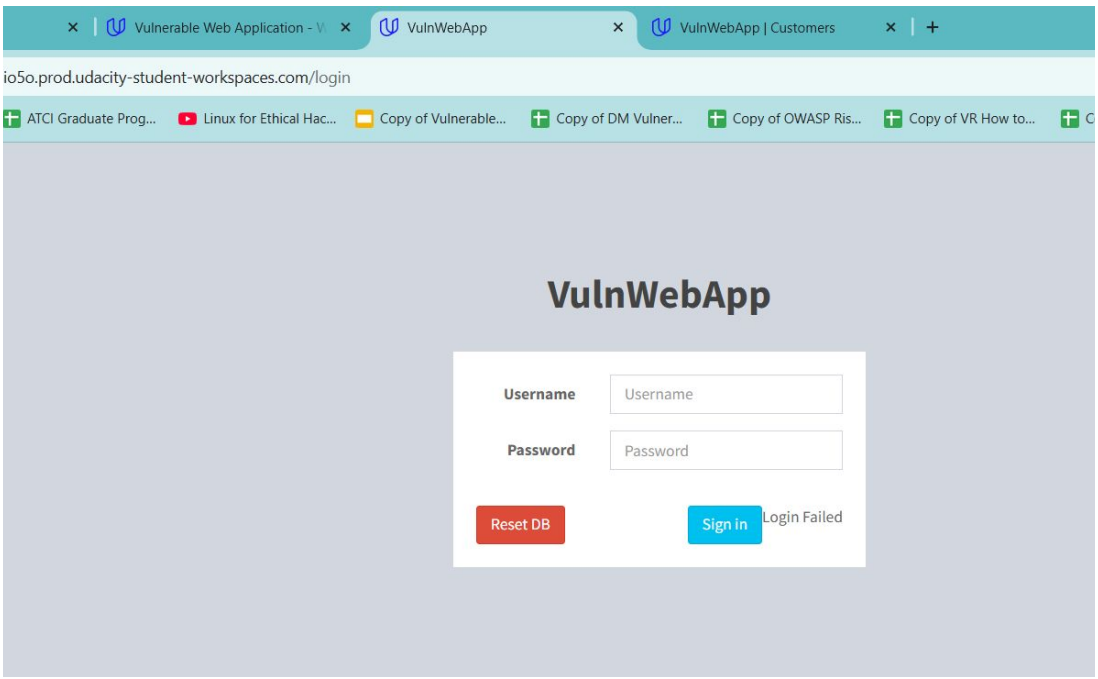




Broken Authentication

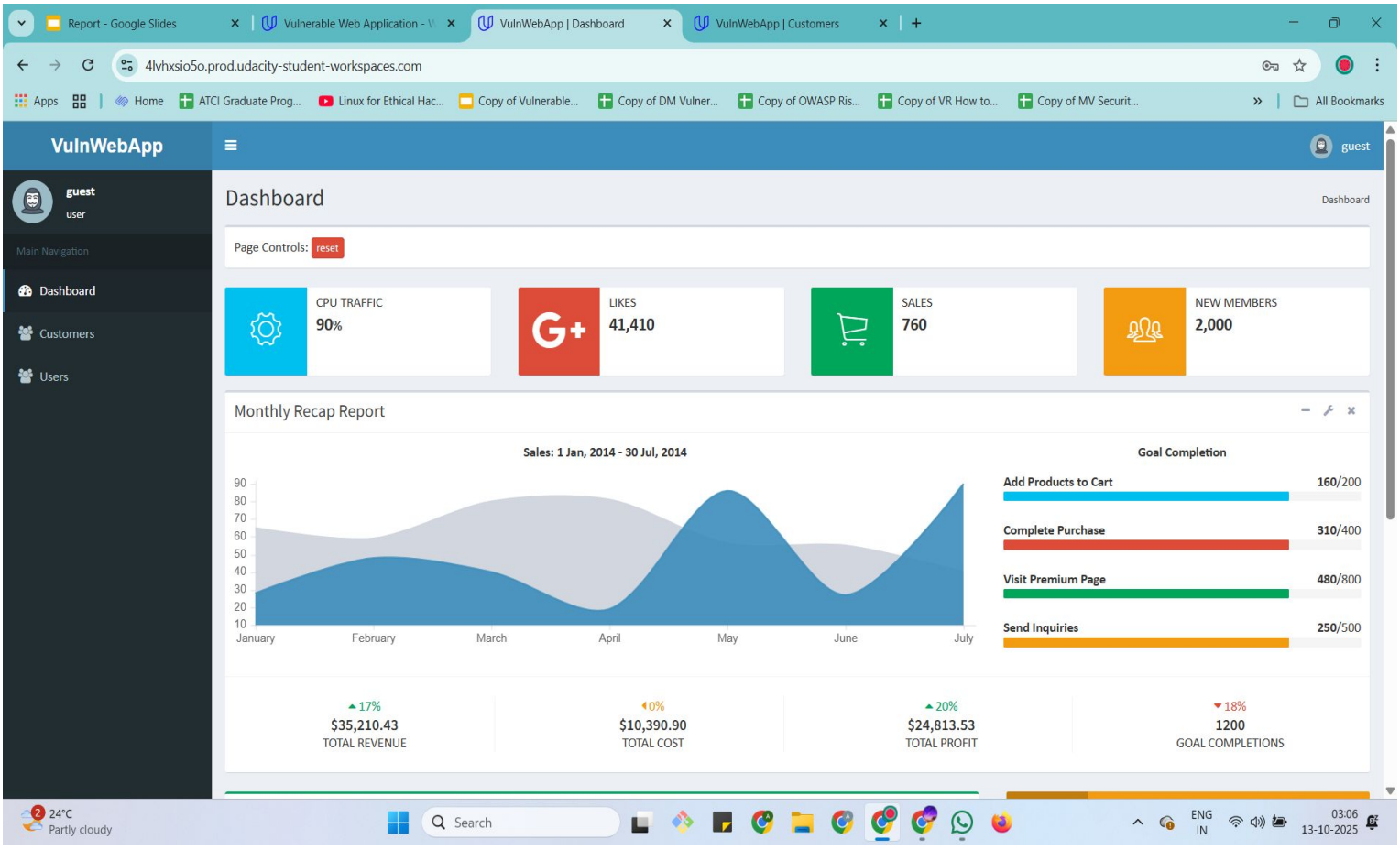
- *Confirm Login: The script will output the successful username and password (guest and orange respectively), proving the lack of protection.*

Before :



Broken Authentication

After :



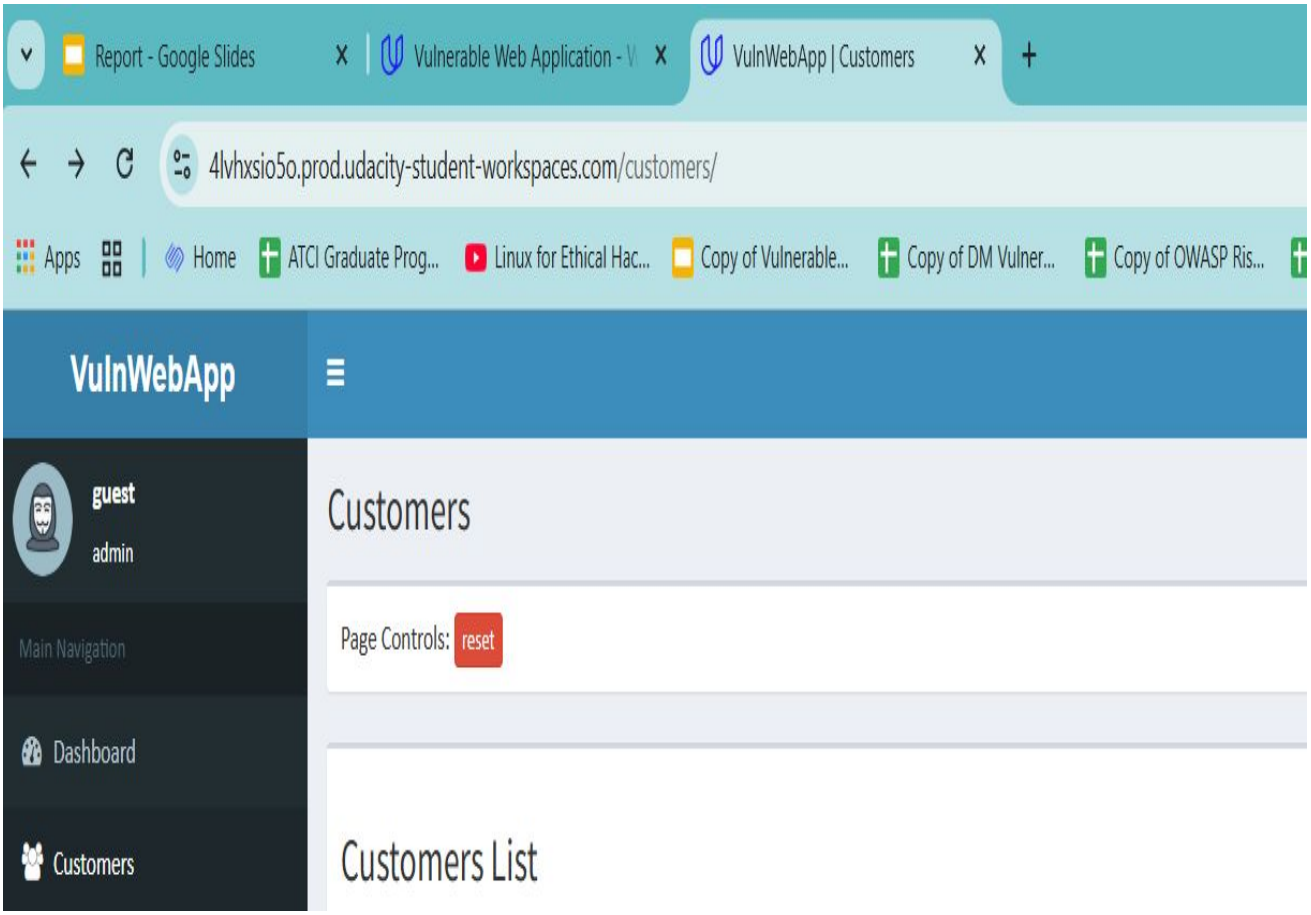
Cross-Site Scripting (XSS)

Severity:	*High*
OWASP TOP 10 reference:	*A03:2021 - Injection*
Vulnerability Explanation	
The application fails to sanitize or validate input provided via URL parameters. An attacker can craft a malicious URL containing a JavaScript payload, which the application attempts to process. While the payload execution was blocked by the Web Application Firewall (WAF) , the attempt confirms that the application code is vulnerable to accepting and processing malicious script data in a Reflective context.	
Remediation recommendation	
Implement Input Validation and Context-Aware Output Encoding: All data read from URL parameters (query strings, path variables) must be rigorously validated against an allow-list of expected characters and formats. Additionally, implement output encoding to convert special characters like <, >, and " to their HTML entity equivalents if the input is ever reflected on the page.	

Cross-Site Scripting (XSS)

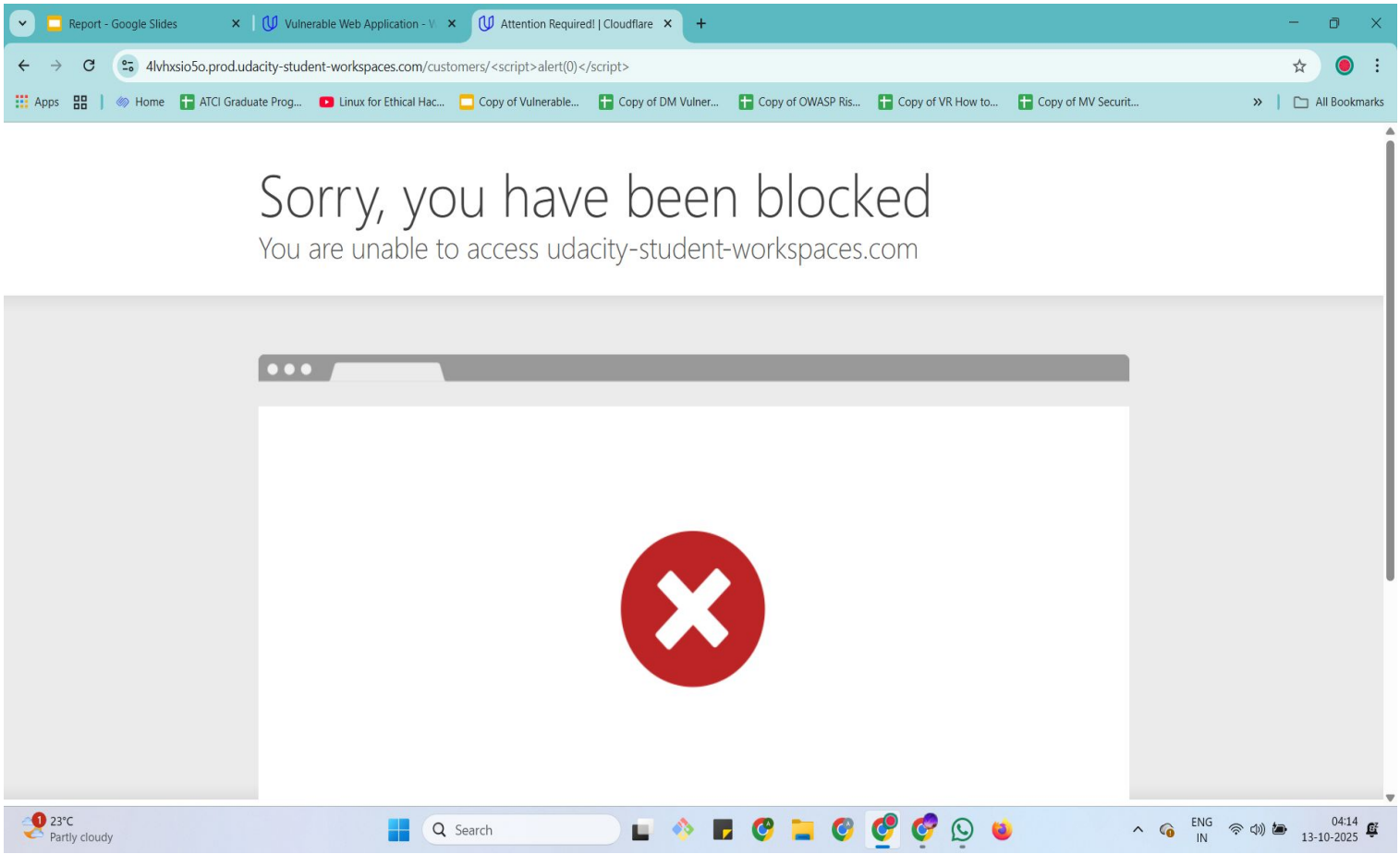
Steps to Reproduce the Issue:

1. *Identify Vulnerable Target: Locate a page URL that reflects input from a query parameter (e.g., a search page).*



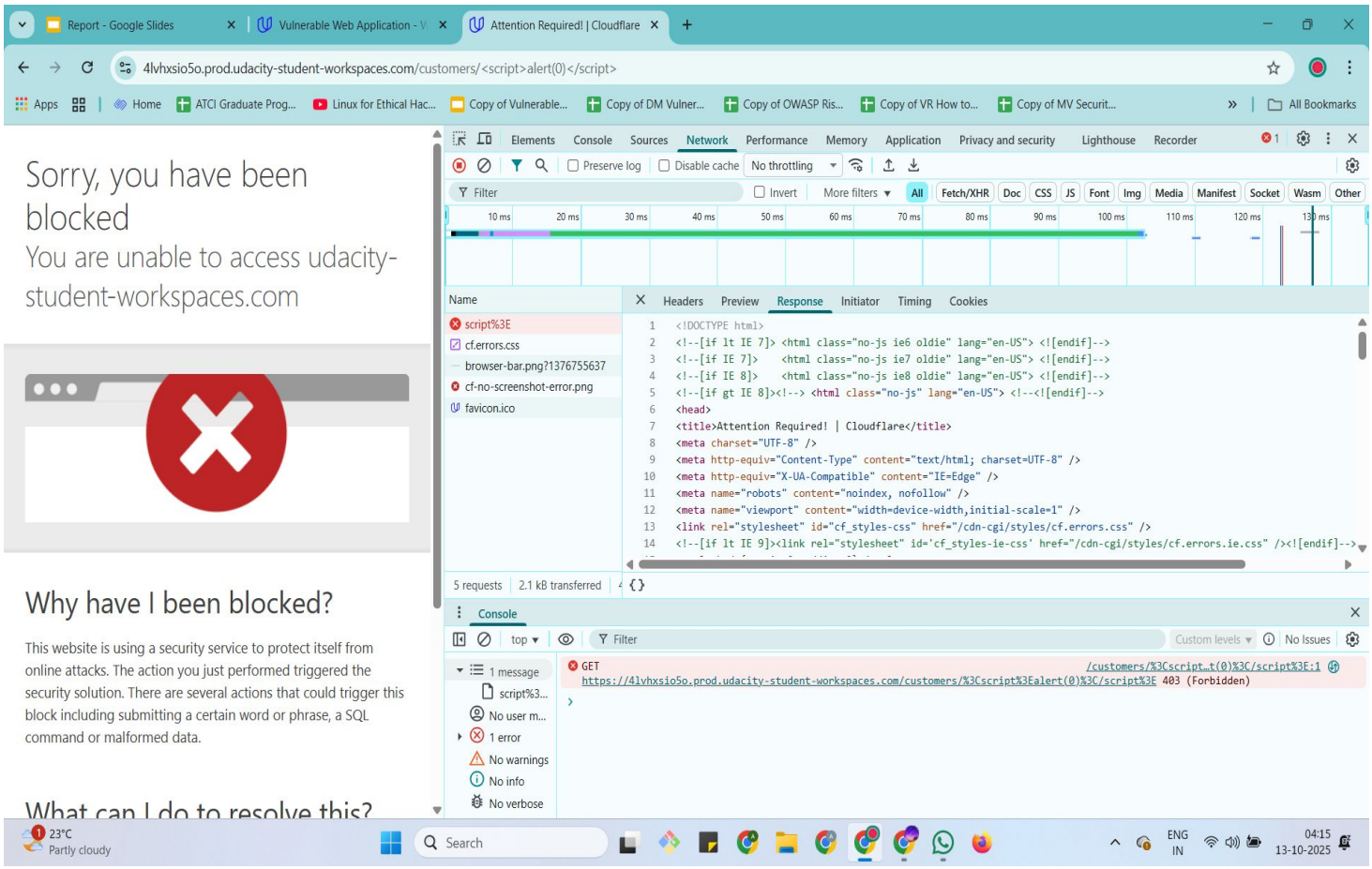
Cross-Site Scripting (XSS)

- *Append Malicious Script: Append a script like <script>alert(0)</script> to the vulnerable page URL.*
- *Execute and Observe: Load the crafted URL. A successful attack would cause the browser to execute the alert(0) function, proving a Reflective XSS vulnerability.*



Cross-Site Scripting (XSS)

- *WAF Confirmation: Observe the response. The WAF's action (e.g., blocking the payload) indicates that while the application code is vulnerable, the WAF is currently acting as a layer of defense.*



Broken Access

Severity:	*Critical*
OWASP TOP 10 reference:	*A01:2021 - Broken Access Control*

Vulnerability Explanation

The application allows vertical privilege escalation by permitting users to modify their role flag via the client-side authinfo cookie. The server validates session integrity but fails to re-check the authorized user's role against the true database record. This permits a regular "guest" user to assume "admin" privileges simply by modifying a cookie value, granting them access to restricted areas like the "Customers" and "Users" dashboards.

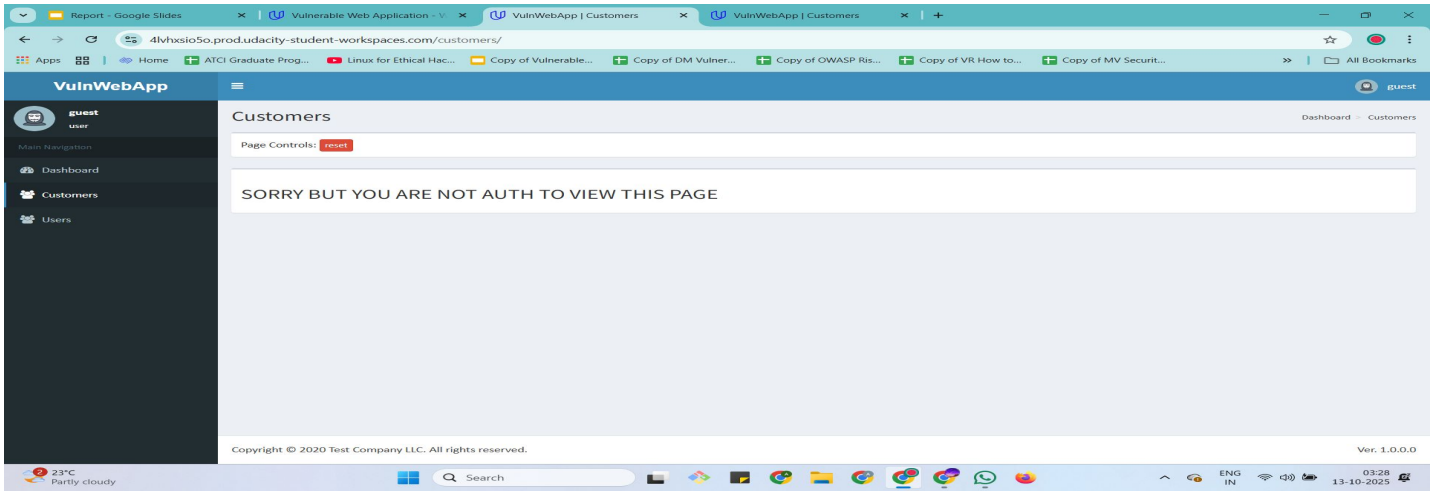
Remediation recommendation

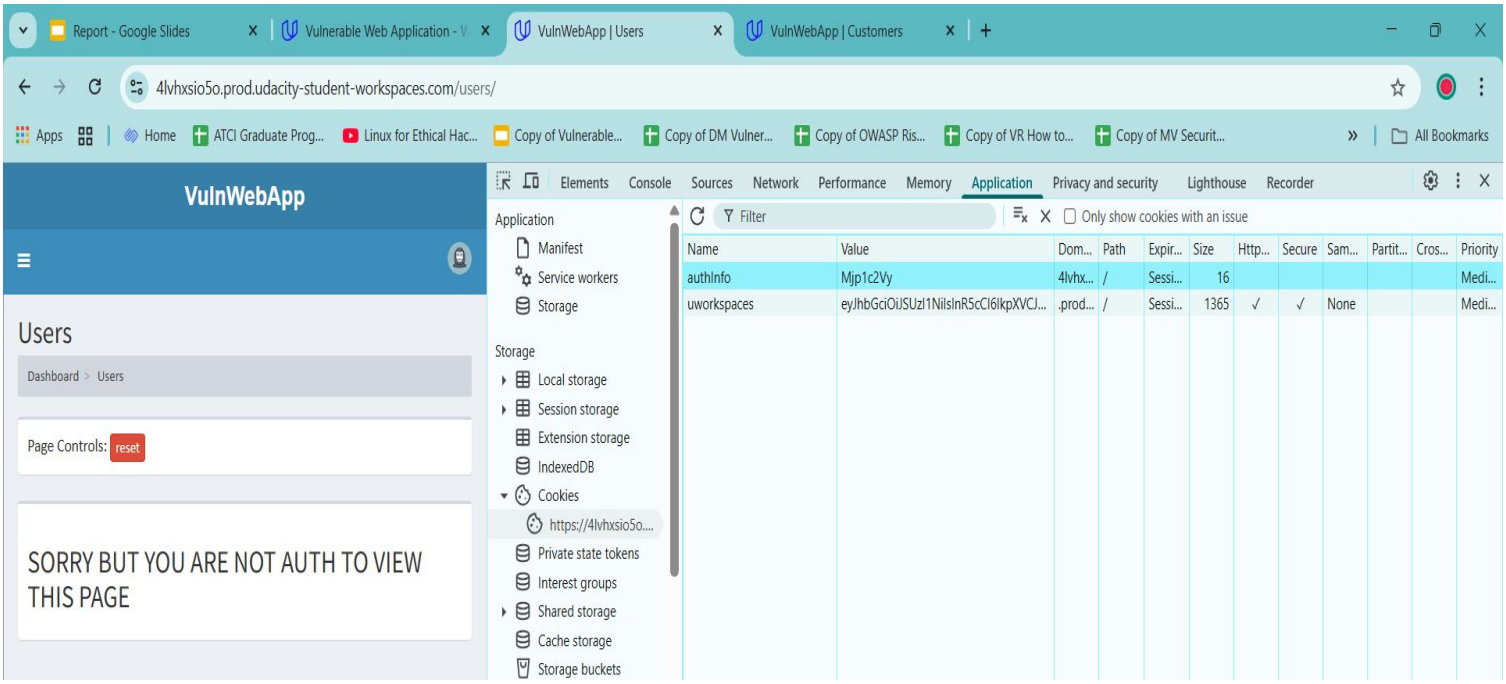
Enforce Server-Side Authorization: Eliminate authorization logic from client-side cookies. All user roles and privileges must be checked against a secure, server-side data store on every request to a restricted endpoint. Implement a strong validation check (e.g., if (request.user.role !== 'admin') { deny access }) before executing any restricted function.
Steps to Reproduce the Issue

Broken Access

Steps to Reproduce the Issue

1. *Capture Cookie: Log in and open Developer Tools (F12). Navigate to the Application tab and copy the value of the authinfo cookie.*
   - *Value : Mjplc2Vy*





Broken Access

1. *Decoding the Cookie: Navigate to the /workspace/tools/ directory in the Terminal and use the provided script to decode the cookie value.*

Decoding

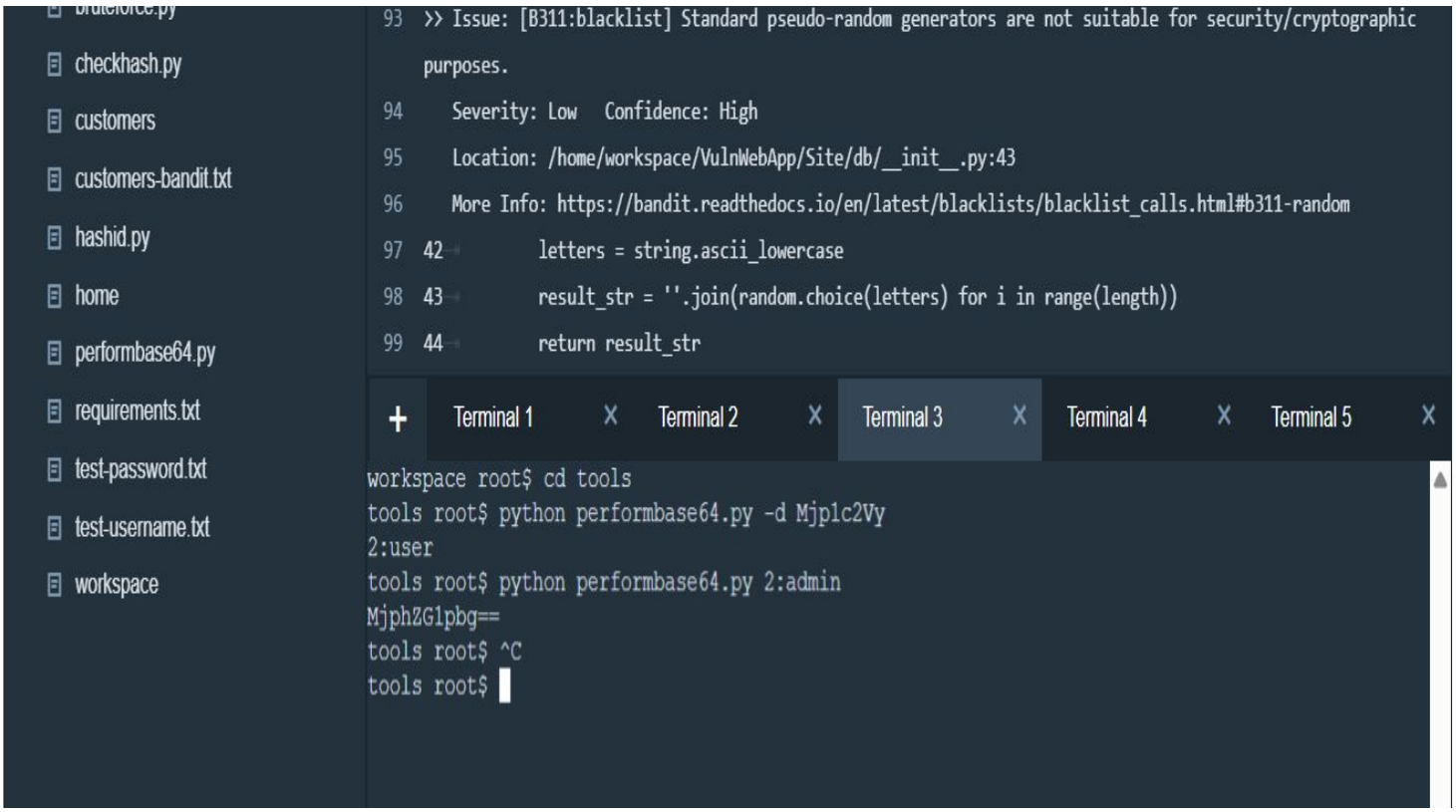
tools root$ python performbase64.py -d Mjp1c2Vy

2:user

Encoding

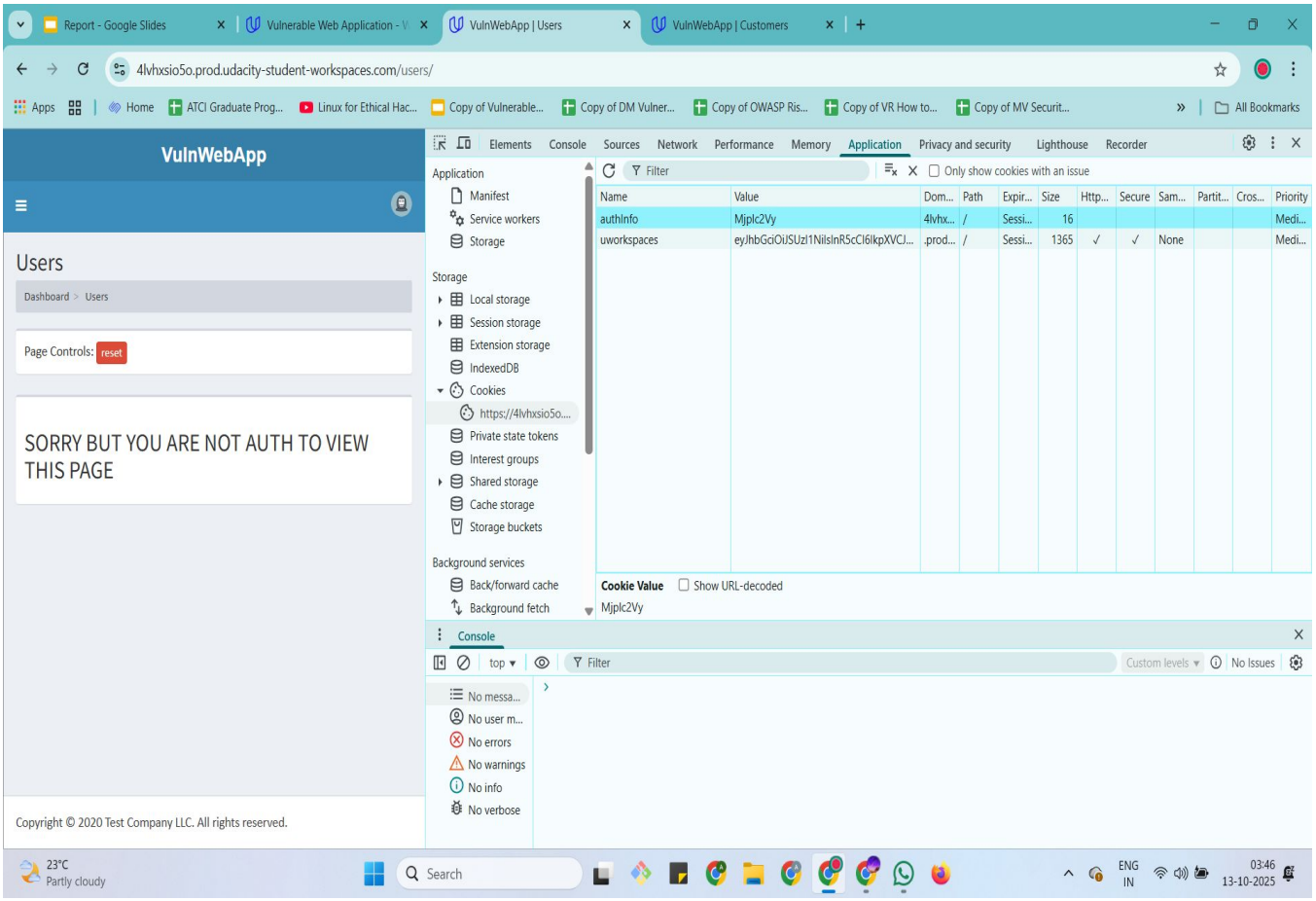
tools root$ python performbase64.py 2:admin

MjphZG1pbg==



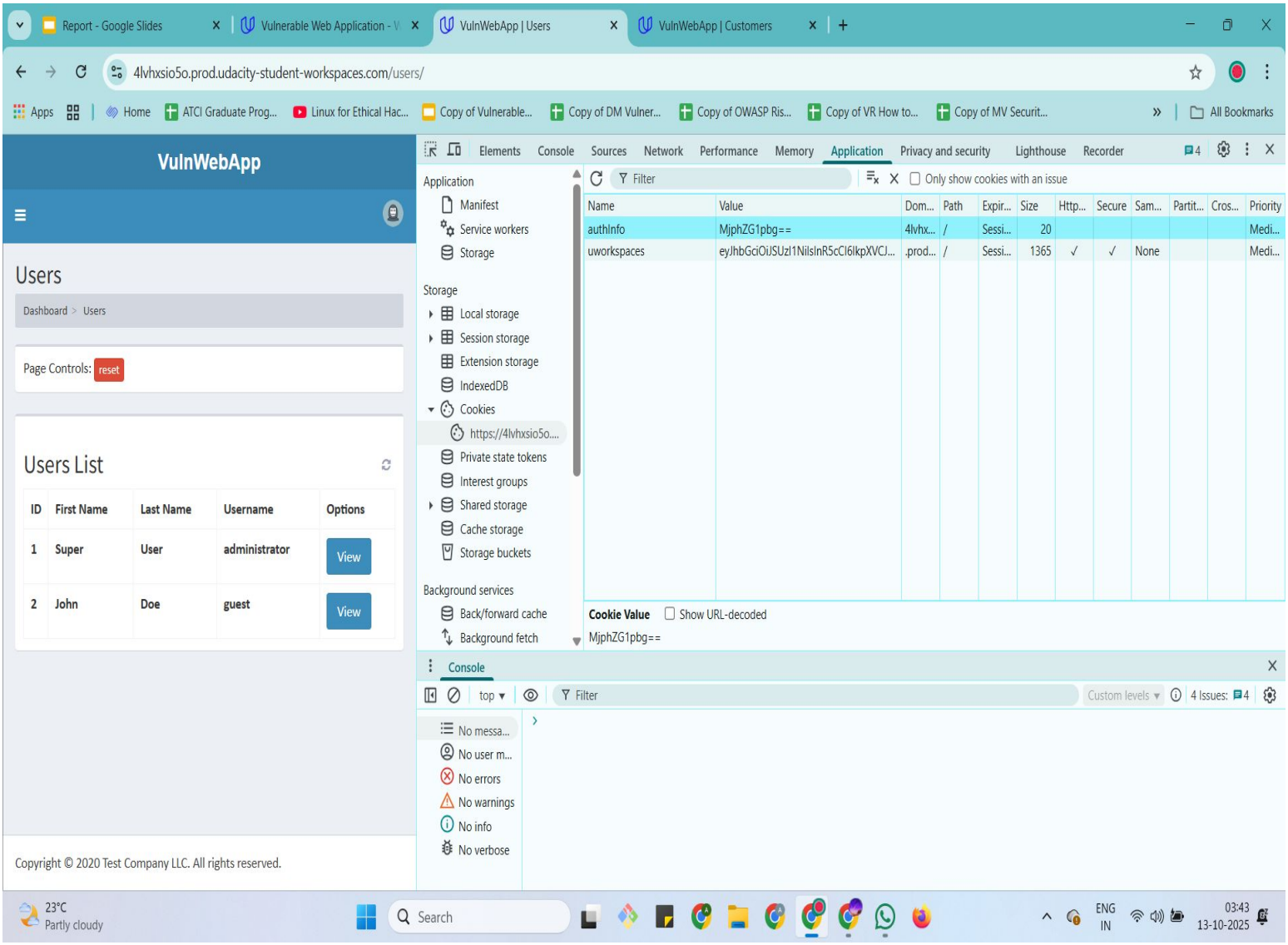
Broken Access

- *By substituting the cookie value with the encoded administrator string (MjphZG1pbg==), an attacker can bypass server-side authorization and immediately assume administrative privileges from a lower-level guest account.*
  Before:



Broken Access

After :



Sensitive Data Exposure

Severity:	*High*
OWASP TOP 10 reference:	*05:2021 - Security Misconfiguration*

Vulnerability Explanation

The application grants administrative privileges based on a client-side flag in the authinfo cookie. This allows an unprivileged "guest" user to perform Vertical Privilege Escalation by simply modifying their session token.The server fails to verify the user's authentic role against the backend database upon request.

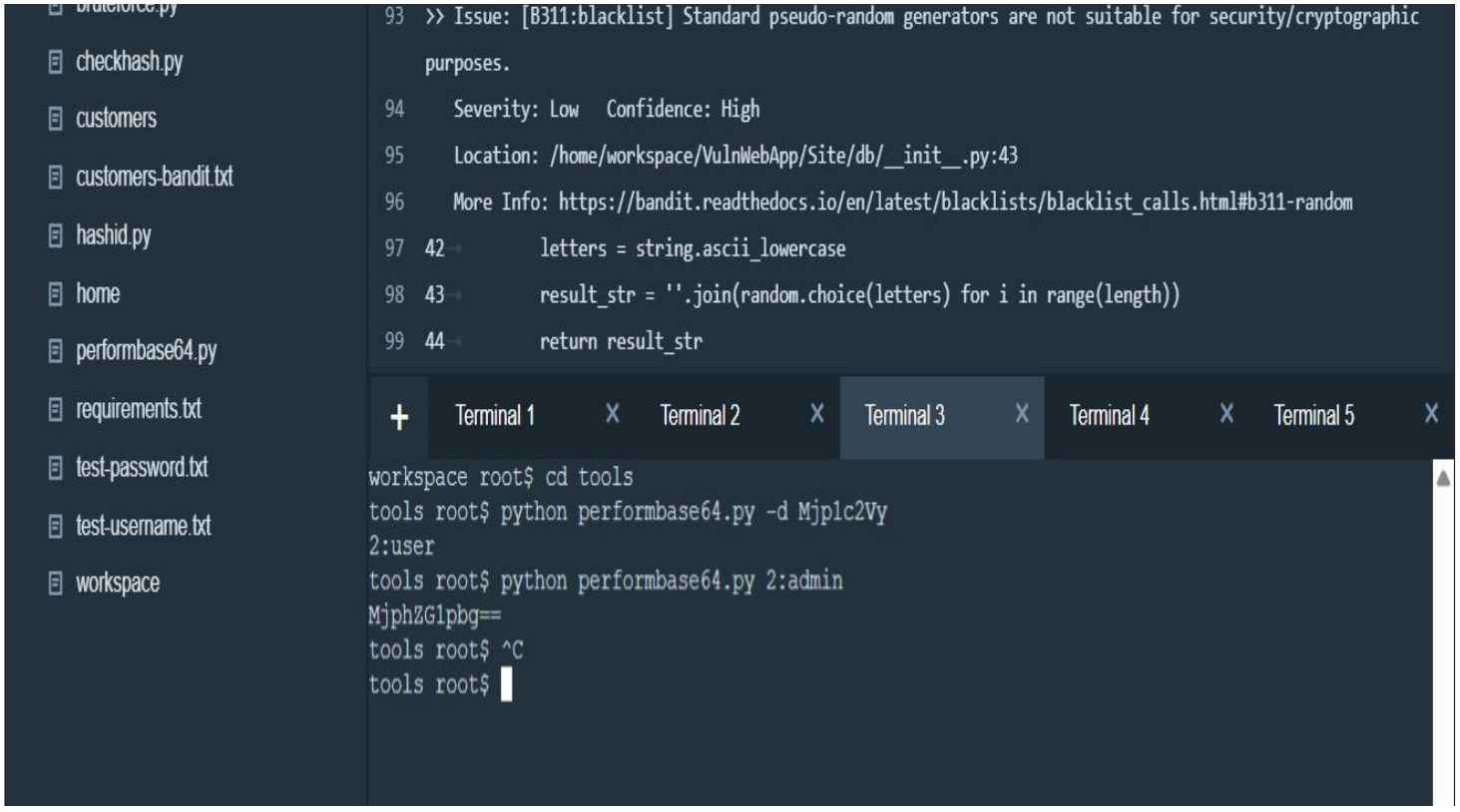
Remediation recommendation

Enforce Server-Side Authorization: *Eliminate the use of client-side data (e.g., cookies) for making critical access control decisions.All user roles and privileges must be stored exclusively in secure server-side sessions, decoupling them entirely from the client.The server must validate the user's authority against this authoritative data store on every single request to a restricted administrative endpoint.*

Sensitive Data Exposure

Steps to Reproduce to Issue:

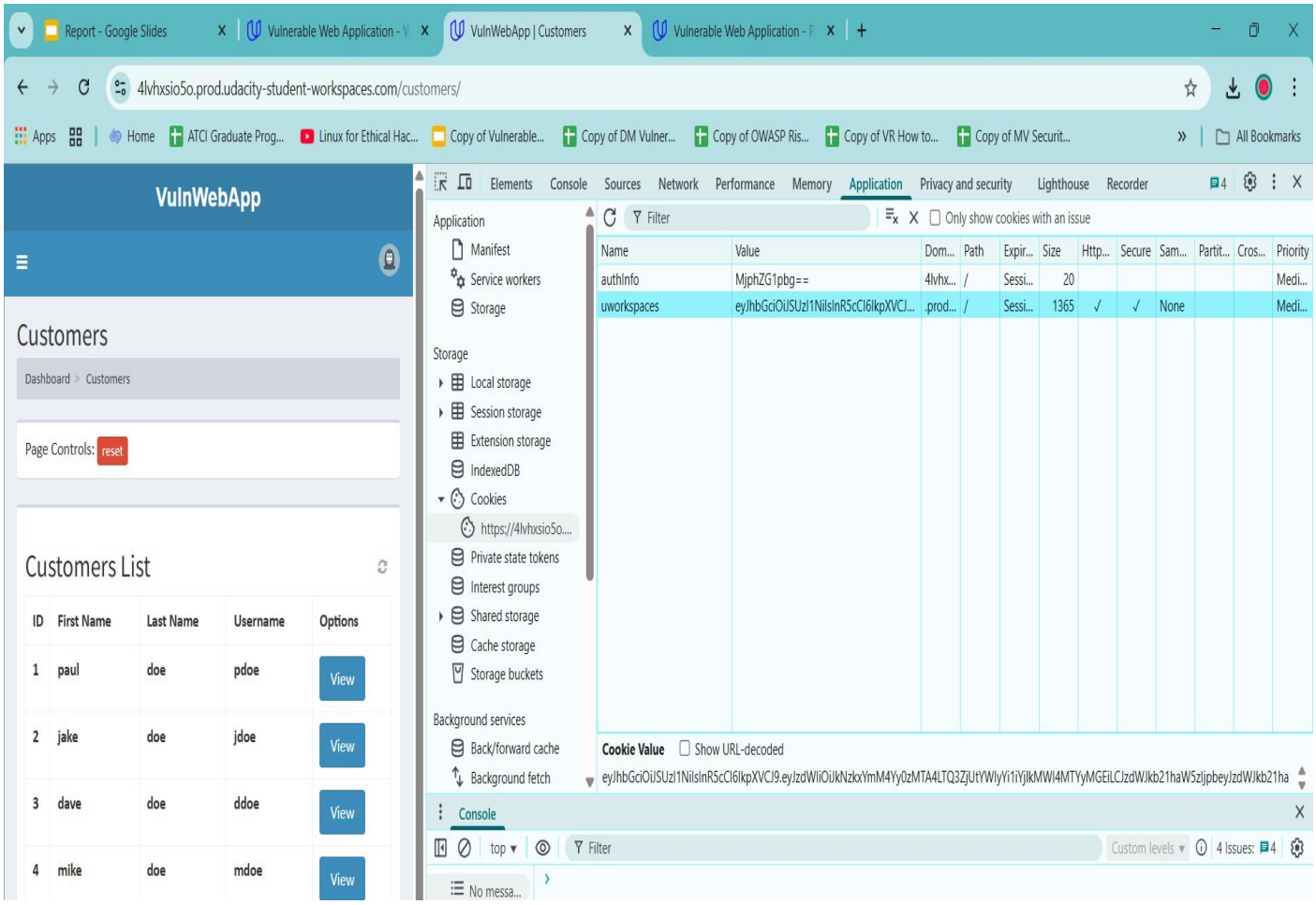
- *Decode the authinfo cookie.*
- *Use the script to re-encode the..*
  python performbase64.py -d Mjplc2Vj



Sensitive Data Exposure

- *Inject New Cookie Value:*

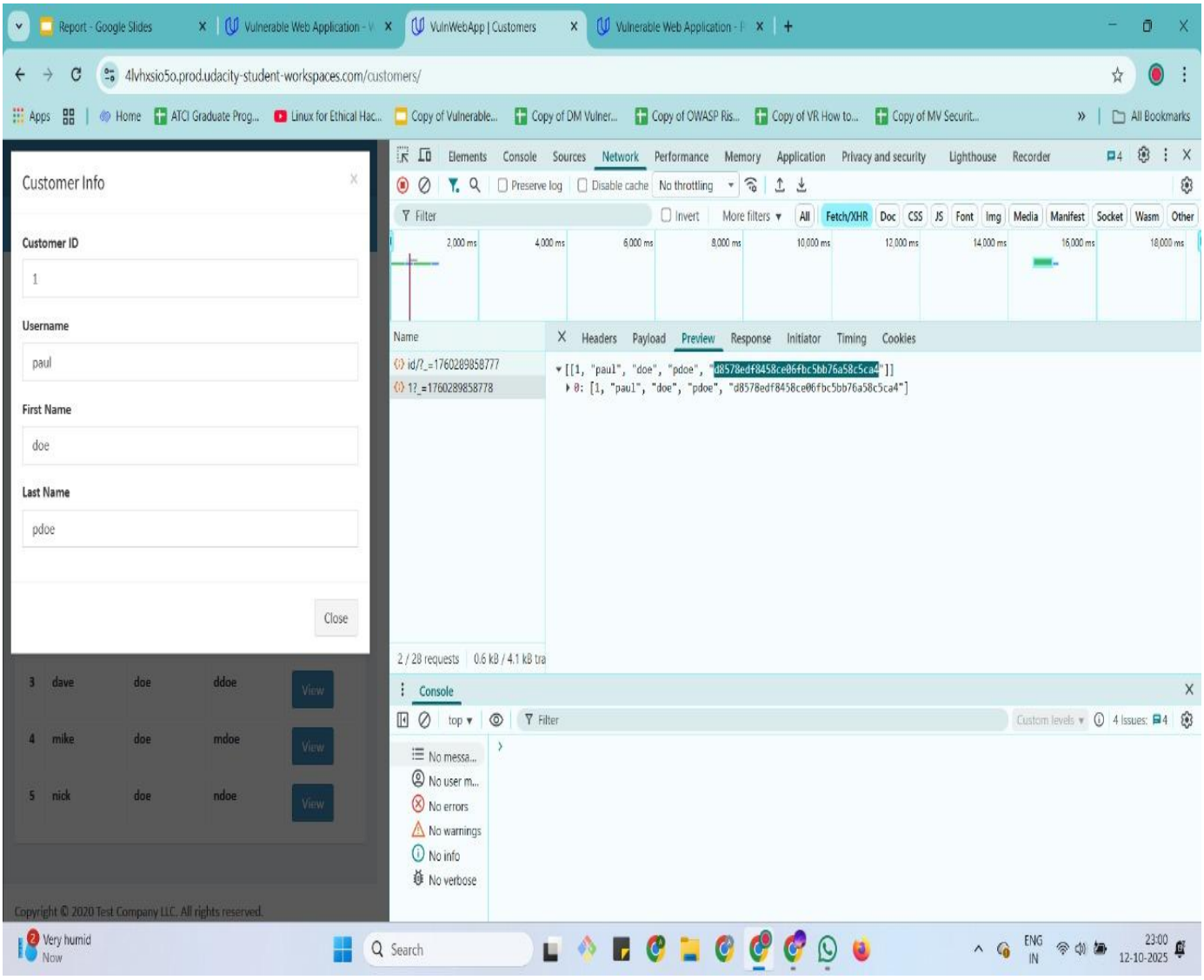
  Use the browser's Developer Tools (Application Tab) to manually replace the existing authinfo cookie value with the new, re-encoded string.



Sensitive Data Exposure

- *Verify Escalation:*

  Refresh the page to send the manipulated cookie to the server.Observe the application interface. New restricted features (like "Customers" and "Users" menus) are now visible, confirming unauthorized administrative access for the 'guest' user.

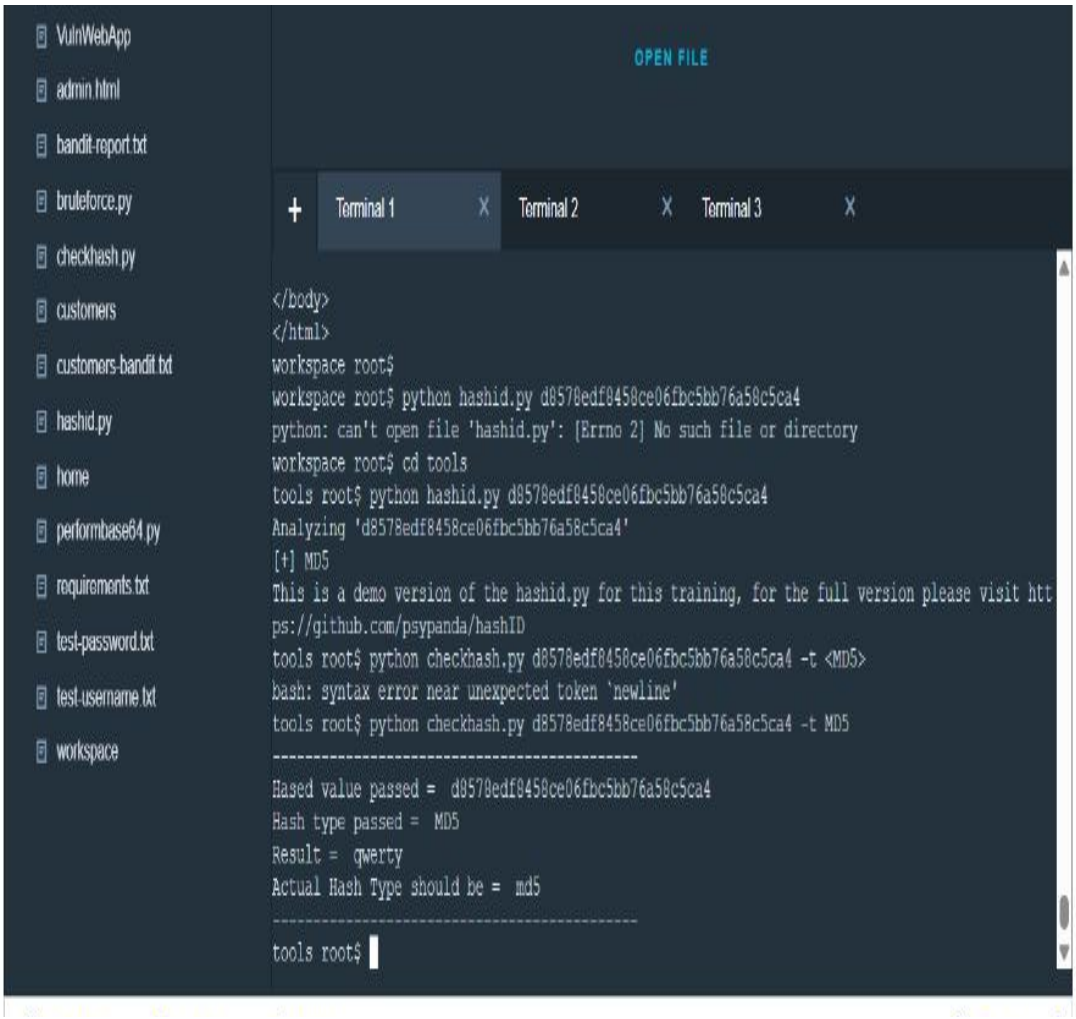


Sensitive Data Exposure

- *The below commands can be used to discover the hashed values Example :*

  python hashid.py d8578edf8458ce06fbc5bb76a58c5ca4

  python checkhash.py d8578edf8458ce06fbc5bb76a58c5ca4 -t MD5



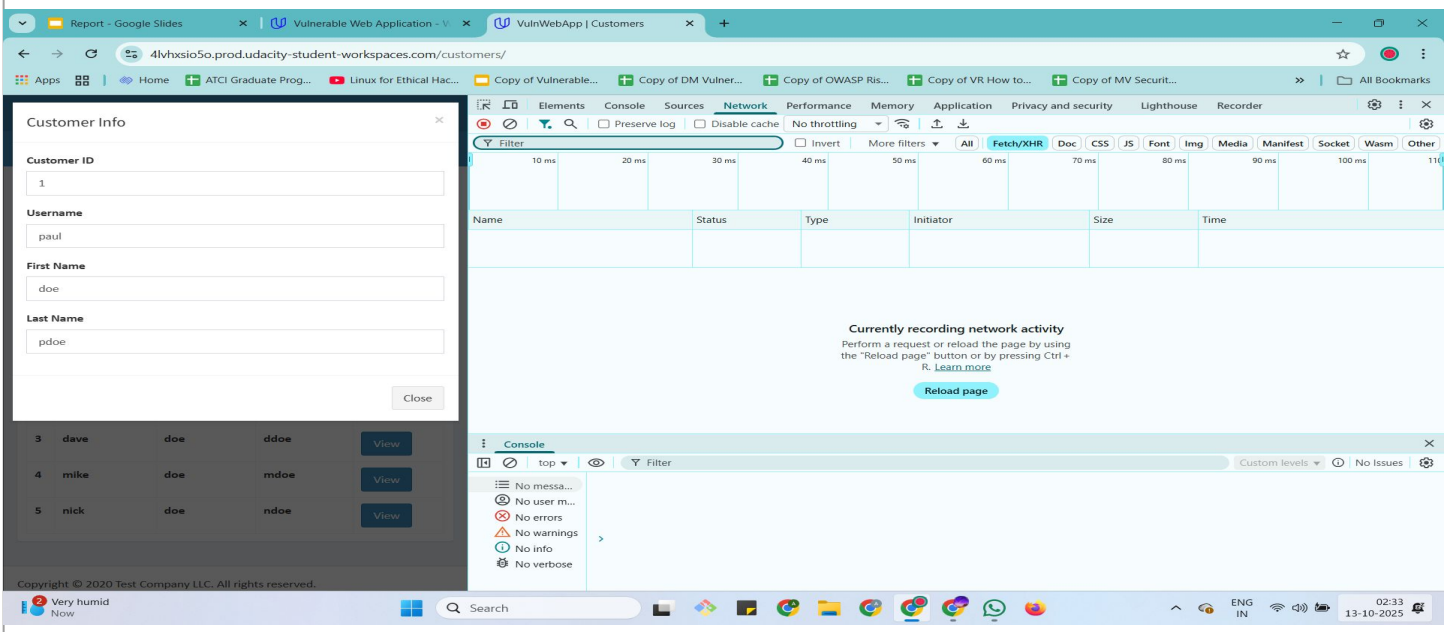
SQLi

Severity:	*Critical*
OWASP TOP 10 reference:	*A03:2021- Injection*
Vulnerability Explanation	
The application's API endpoint used for retrieving customer details via the path parameter (/customers/id/{ID}) fails to properly sanitize or parameterize user-controlled input. This allows an attacker to inject SQL metacharacters and commands into the path, manipulating the database query. Although the Web Application Firewall (WAF) blocked the full exploitation, the injection attempt itself was successfully detected, confirming the underlying code flaw.	
Remediation recommendation	
Adopt a strategic coding standard mandating the use of prepared statements or parameterized queries for all database calls. Do not construct SQL queries by concatenating user-supplied strings. If using a framework, ensure the ORM's built-in parameterization is used correctly for all path or query parameters.	

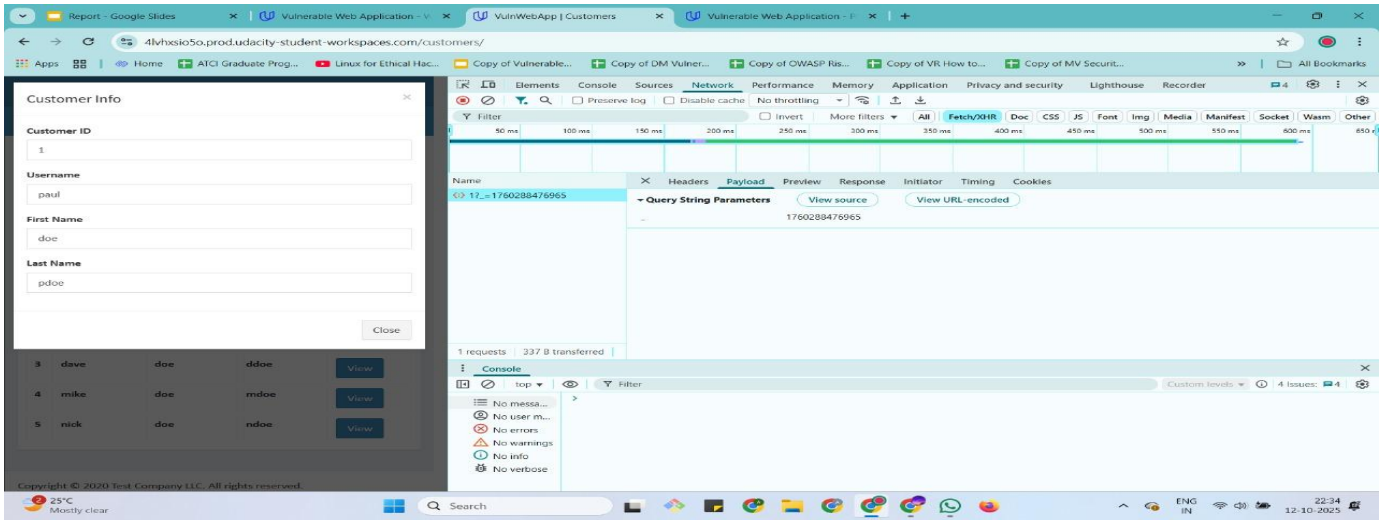
SQLi

Steps to Reproduce the Issue

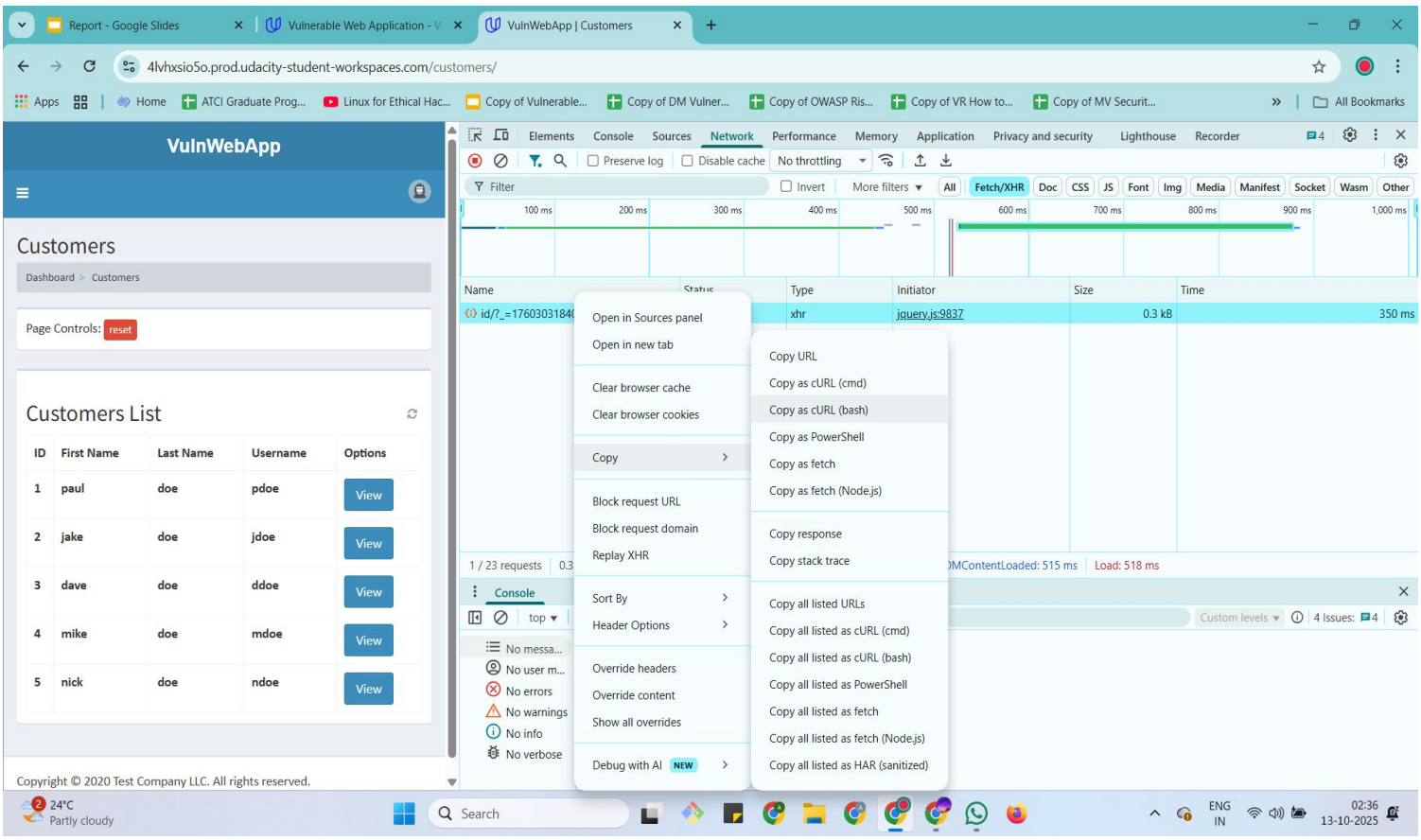
- *Open Network Console: Open Developer Tools (F12), go to the Network tab.*



- *Identify Target: Click "View" on any customer to find the hidden API request (e.g., /customers/id/1?...). Copy the request as a cURL command.*

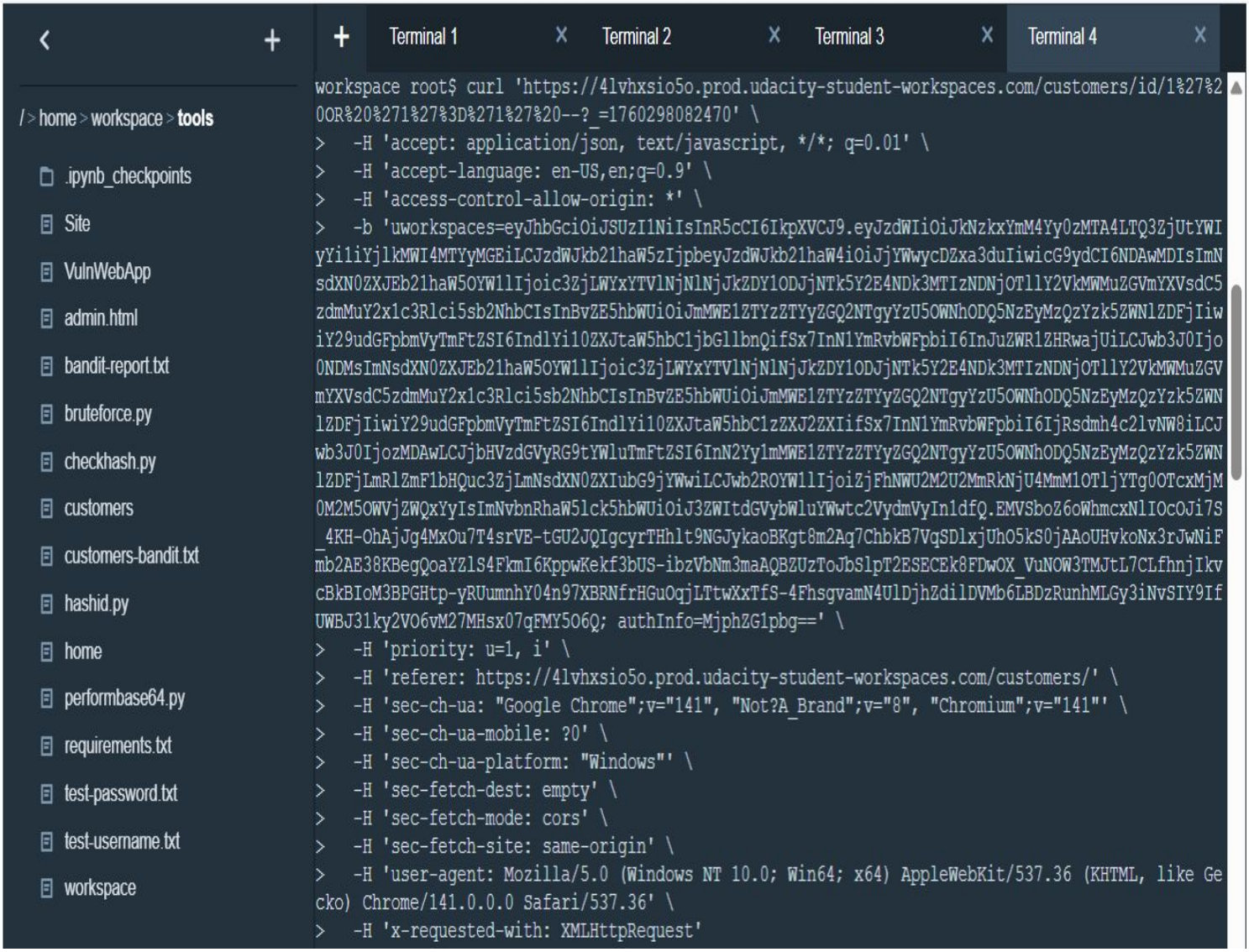


SQLi



- *Inject Payload: In the Terminal, modify the cURL command's URL, replacing the customer ID (/id/1) with the SQL Injection payload, ensuring the payload is URL-encoded to bypass basic URL validation.*
  - *Payload: 1' OR '1'='1' --*
  - *Injected Path Fragment: /id/1%27%20OR%20%271%27%3D%271%27%20--*

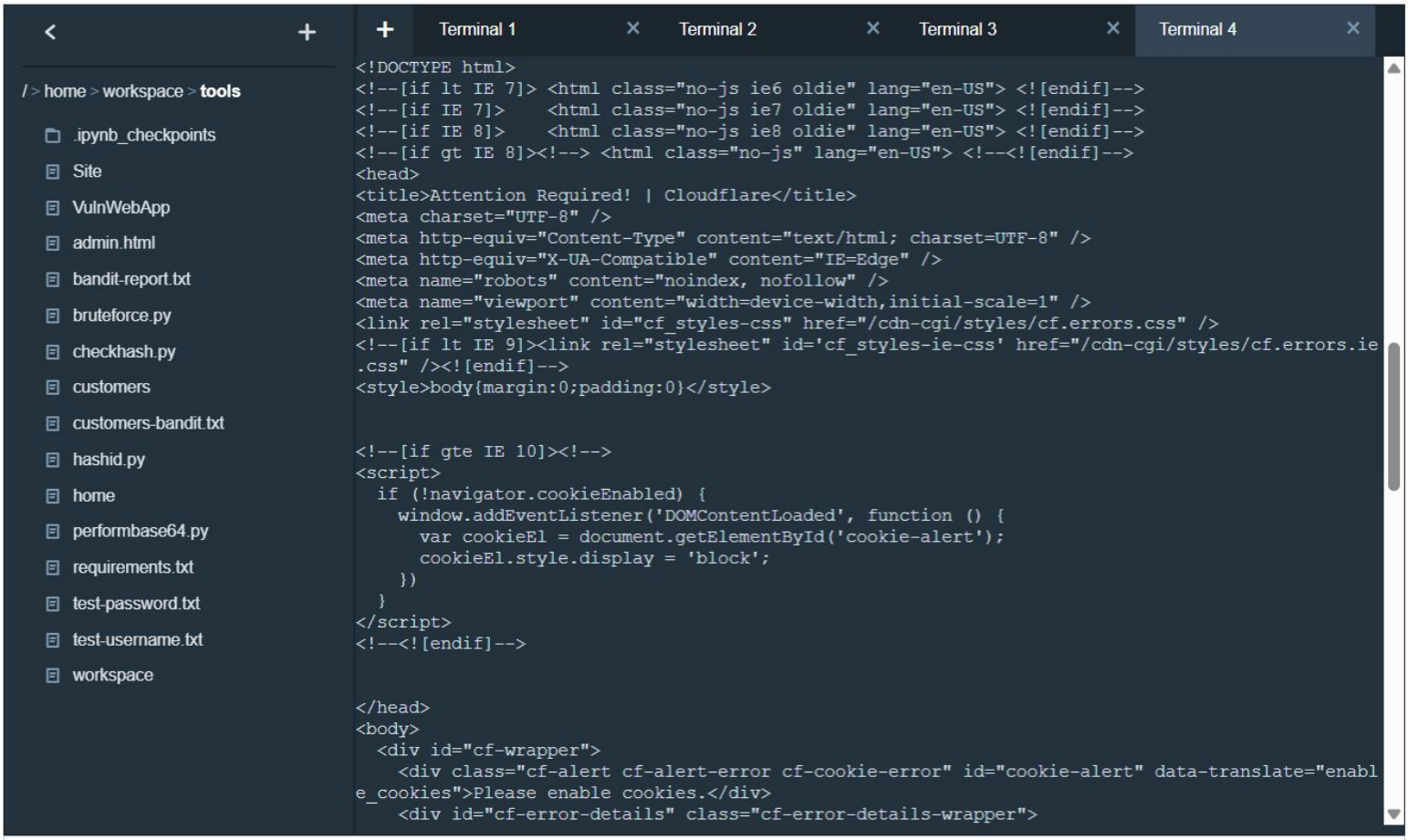
SQLi


/ > home > workspace > tools
.ipynb_checkpoints
Site
VulnWebApp
admin.html
bandit-report.txt
bruteforce.py
checkhash.py
customers
customers-bandit.txt
hashid.py
home
performbase64.py
requirements.txt
test-password.txt
test-username.txt
workspace
Terminal 1
Terminal 2
Terminal 3
Terminal 4
workspace root$ curl 'https://4lvhxsio5o.prod.udacity-student-workspaces.com/customers/id/1%27%20OR%20%271%27%3D%271%27%20--?_=1760298082470' \
> -H 'accept: application/json, text/javascript, */*; q=0.01' \
> -H 'accept-language: en-US,en;q=0.9' \
> -H 'access-control-allow-origin: *' \
> -b 'uworkspaces=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkNzkxYmM4Yy0zMTA4LTQ3ZjUtYWIyYiliYjlkMWI4MTYyMGEiLCJzdWJkb21haW5zIjpbeyJzdWJkb21haW4iOiJjYWwycDZxa3duIiwicG9ydCI6NDAwMDIsImNsdXN0ZXJEb21haW5OYW1lIjoic3ZjLWYxYTVlNjNlNjkzZDY1ODJjNTk5Y2E4NDk3MTIzNDNjOTllY2VkMWMuZGVmYXVsdC5zdmMuY2x1c3Rlci5sb2NhbCIsInBvZE5hbWUiOiJmMWE1ZTYzZTYyZGQ2NTgyYzU5OWNhODQ5NzEyMzQzYzk5ZWNlZDFjIiwiY29udGFpbmVyTmFtZSI6IndlYi10ZXJtaW5hbC1jbGllbnQifSx7InN1YmRvbWFpbiI6InJuZWR1ZHRwajUiLCJwb3J0Ijo0NDMsImNsdXN0ZXJEb21haW5OYW1lIjoic3ZjLWYxYTVlNjNlNjkzZDY1ODJjNTk5Y2E4NDk3MTIzNDNjOTllY2VkMWMuZGVmYXVsdC5zdmMuY2x1c3Rlci5sb2NhbCIsInBvZE5hbWUiOiJmMWE1ZTYzZTYyZGQ2NTgyYzU5OWNhODQ5NzEyMzQzYzk5ZWNlZDFjIiwiY29udGFpbmVyTmFtZSI6IndlYi10ZXJtaW5hbC1zZXJ2ZXIifSx7InN1YmRvbWFpbiI6IjRsdmh4c2lvNW8iLCJwb3J0IjozMDAwLCJjbHVzdGVyRG9tYWluTmFtZSI6InN2Yy1mMWE1ZTYzZTYyZGQ2NTgyYzU5OWNhODQ5NzEyMzQzYzk5ZWNlZDFjLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiLCJwb2ROYW1lIjoiZjFhNWU2M2U2MmRkNjU4MmM1OTljYTg0OTcxMjM0M2M5OWVjZWQxYyIsImNvbnRhaW5lck5hbWUiOiJ3ZWItdGVybWluYWwtc2VydmVyIn1dfQ.EMVSboZ6oWhmcxNlIOcOJi7S_4KH-OhAjJg4MxOu7T4srVE-tGU2JQIgcyrTHhlt9NGJykaoBKgt8m2Aq7ChbkB7VqSDlxjUhO5kS0jAAoUHvkoNx3rJwNiFmb2AE38KBegQoaYZlS4FkmI6KppwKekf3bUS-ibzVbNm3maAQBZUzToJbSlpT2ESECEk8FDwOX_VuNOW3TMJtL7CLfhnjIkvcBkBIoM3BPGHtp-yRUumnhY04n97XBRNfrHGuOqjLTtwXxTfS-4FhsgvamN4UlDjhZdilDVMb6LBDzRunhMLGy3iNvSIY9IfUWBJ31ky2VO6vM27MHsx07qFMY5O6Q; authInfo=MjphZG1pbg==' \
> -H 'priority: u=1, i' \
> -H 'referer: https://4lvhxsio5o.prod.udacity-student-workspaces.com/customers/' \
> -H 'sec-ch-ua: "Google Chrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141"' \
> -H 'sec-ch-ua-mobile: ?0' \
> -H 'sec-ch-ua-platform: "Windows"' \
> -H 'sec-fetch-dest: empty' \
> -H 'sec-fetch-mode: cors' \
> -H 'sec-fetch-site: same-origin' \
> -H 'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36' \
> -H 'x-requested-with: XMLHttpRequest'

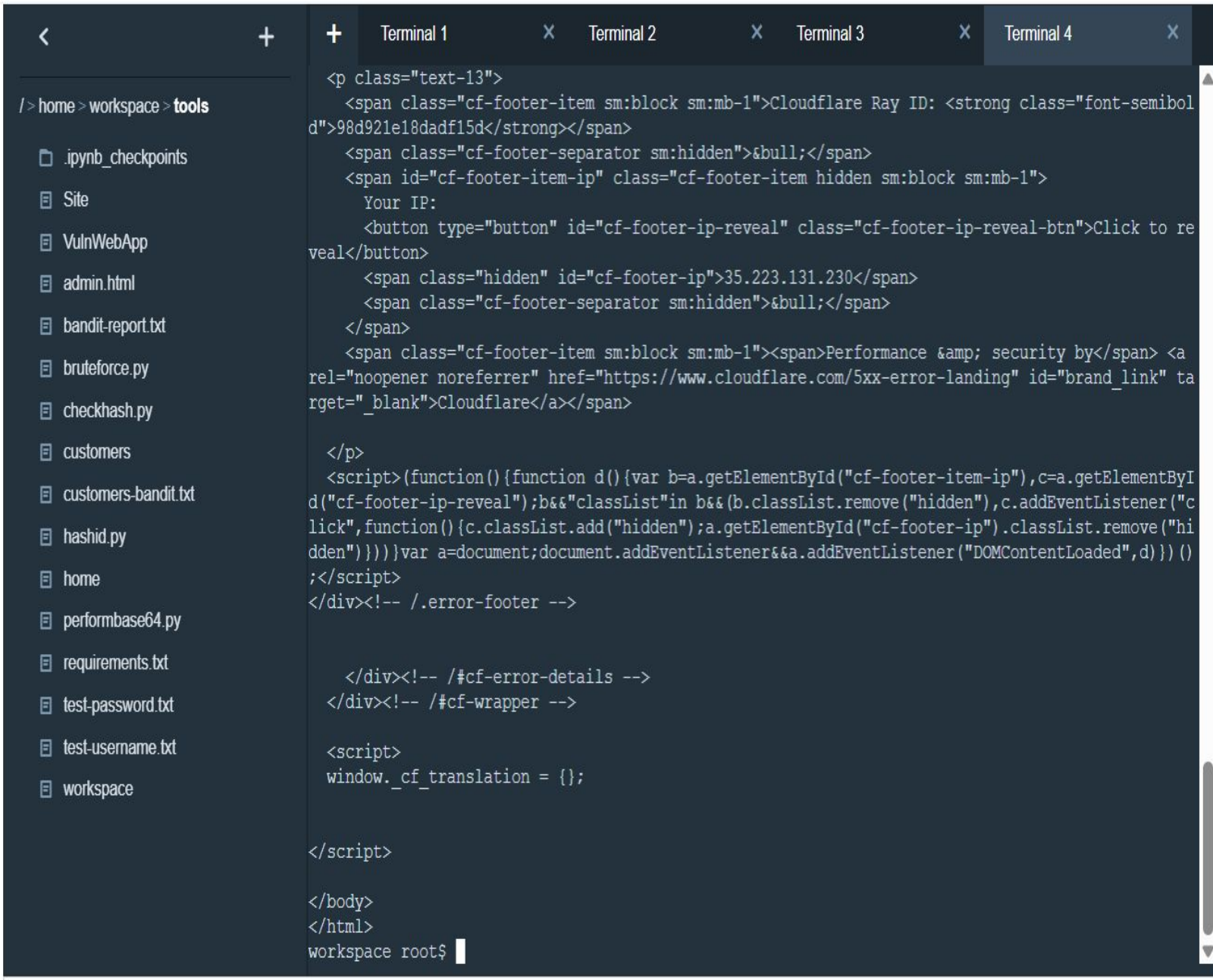
SQLi

RESULT :

The Terminal output will return the HTML content of the WAF block page (e.g., Cloudflare block), confirming that the malicious injection was attempted, but blocked by the external defense. This proves the application code itself is vulnerable.



SQLi


/ > home > workspace > tools
.ipynb_checkpoints
Site
VulnWebApp
admin.html
bandit-report.txt
bruteforce.py
checkhash.py
customers
customers-bandit.txt
hashid.py
home
performbase64.py
requirements.txt
test-password.txt
test-username.txt
workspace
Terminal 1
Terminal 2
Terminal 3
Terminal 4

Cloudflare Ray ID: 98d921e18dadf15d
•

Your IP:
<button type="button" id="cf-footer-ip-reveal" class="cf-footer-ip-reveal-btn">Click to reveal</button>
35.223.131.230
•

Performance & security by Cloudflare

<script>(function(){function d(){var b=a.getElementById("cf-footer-item-ip"),c=a.getElementById("cf-footer-ip-reveal");b&&"classList"in b&&(b.classList.remove("hidden"),c.addEventListener("click",function(){c.classList.add("hidden");a.getElementById("cf-footer-ip").classList.remove("hidden")}))}var a=document;document.addEventListener&&a.addEventListener("DOMContentLoaded",d)})();</script>
<!-- /.error-footer -->
<!-- /#cf-error-details -->
<!-- /#cf-wrapper -->
<script>
window._cf_translation = {};
</script>
</body>
</html>
workspace root$

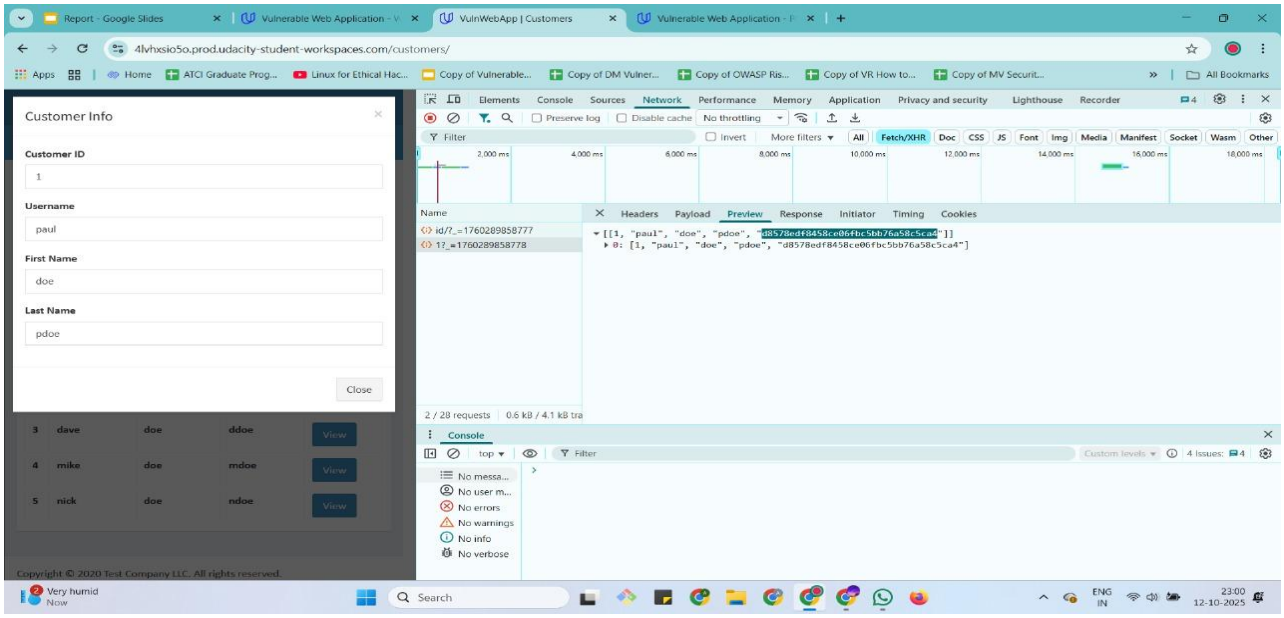
Algorithmic Failure (A05/Misconfiguration)

Severity:	*High*
OWASP TOP 10 reference:	*05:2021 - Security Misconfiguration*
Vulnerability Explanation	
The application uses a deprecated and cryptographically weak hashing algorithm (e.g., MD5, confirmed by analyzing the hash found in the API response) for storing user passwords. Since MD5 is extremely fast and vulnerable to rainbow table and brute-force attacks, the resulting hashes are trivial to reverse back into plaintext passwords using offline tools. The weakness is a failure to properly configure a robust security component.	
Remediation recommendation	
Mandate Strong, Slow Hashing: Immediately migrate all user password storage to a modern, deliberately slow, and salted hashing algorithm (e.g., Argon2 or bcrypt). Implement a migration strategy to re-hash existing user passwords upon their next successful login attempt.	

Algorithmic Failure (A05/Misconfiguration)

Steps to Reproduce the Issue:

- *Identify Hash: Find a user hash (d8578edf8458ce06fbc5bb76a58c5ca4) from the API response (via the Network tab)*

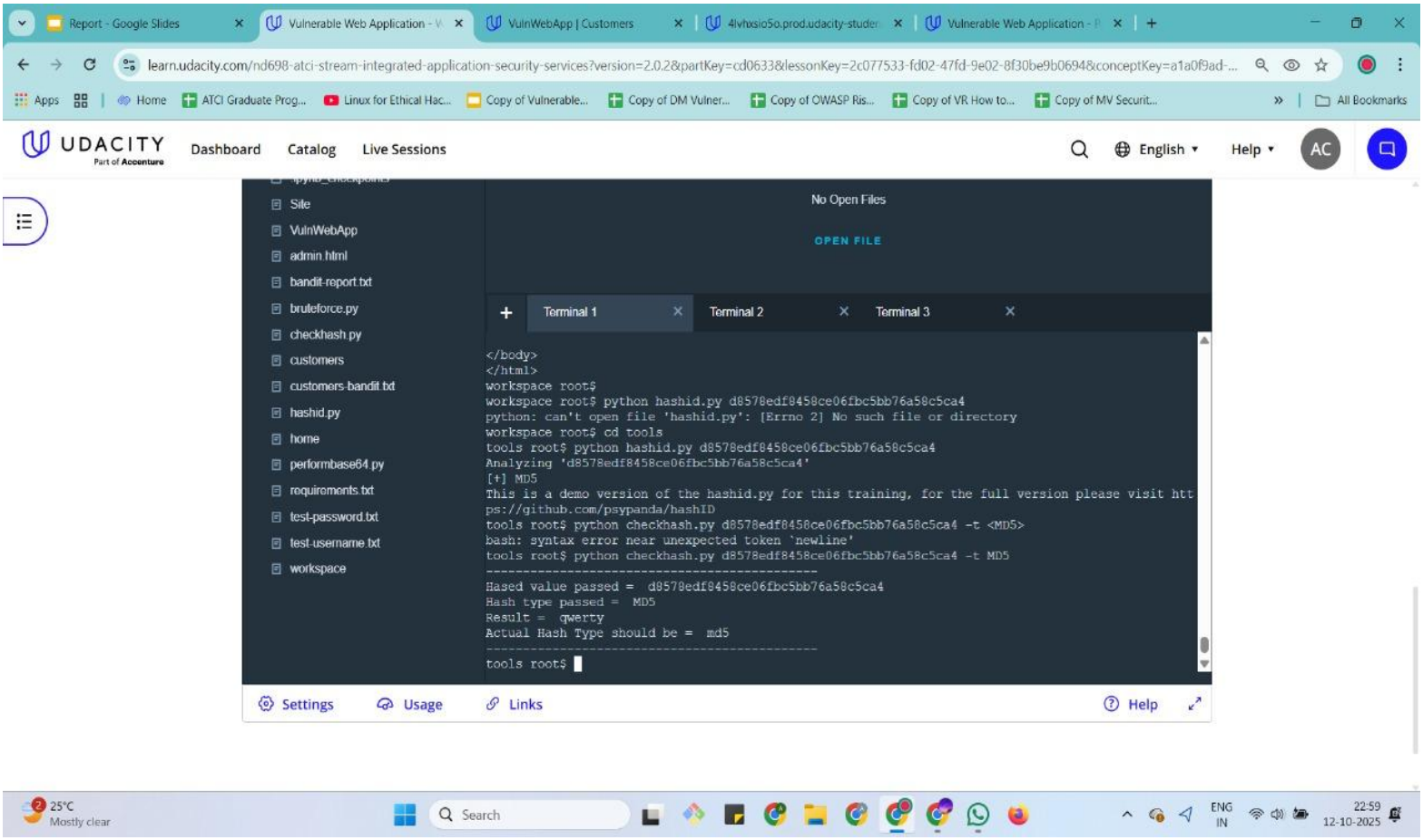


- *To Determine Algorithm: In the Terminal, use hashid.py to identify the hashing algorithm.*

  python hashid.py d8578edf8458ce06fbc5bb76a58c5ca4

-

Algorithmic Failure (A05/Misconfiguration)



- *Crack Hash: Use checkhash.py with the confirmed algorithm (e.g., -t md5) to quickly crack the hash and retrieve the plaintext password (e.g.,qwerty "). The time taken to crack the hash confirms the severity of the weak configuration*