Vulnerable Web Application

Sushil

Table of Content

Section 1 (Static code scan)

- Issue: B106
- Issue: B108
- Issue: B303
- Issue: B311
- Issue: B320
- Issue: B603
- Issue: B703

Section 2

- Assess the Web Application

Section 3 (Security Report)

- Broken Authentication
- Broken Access
- Sensitive Data Exposure
- Cryptographic failures
- HTML Injection
- Cross-Site Scripting (XSS)

Section One: Static Code Scan

Vulnerability Risk Matrix

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

- The above chart represents a Vulnerability Risk Matrix - -
- which helps assess the overall risk severity of a vulnerability by combining its impact and likelihood.
- Impact measures how severe the consequences would be if the vulnerability is exploited.
- Likelihood represents how probable it is that the vulnerability will be exploited.

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:	*A07:2021-Identification and Authentication Failure*
Remediation recommendation	
Keep your database login details out of the actual code.Instead, store them safely in environment variables –this is a common and secure practice used in the industry.It helps protect sensitive data and make you app easier to manage across different environments.This way, even if someone sees your code, your credentails stay hidden.It also mitigates risks related to CWE-259(use of hardcoded password) and A07:2021-Identification and authentication failures .	

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:	*A01:2021 - Broken Access control* *A04:2021 - Insecure Design* *A05-2021 - Security Misconfiguration*

Remediation recommendation

Use of a hard coded temporary directory can lead to several issues including Information disclosure, Data tampering, Denial of Service, This may compromise the application's confidentiality, integrity, or availability.

temp_path = "/tmp/tempfile.txt"

This make the temporary directory & file static , which can be predicted or brute forced .Instead of this create function to create temp file with random name without a predefined directory or in restricted directory.

Issue: B108

Remediation recommendation

Developer must attach to use secure coding practices. Temporary file name have to be generated randomly , making it difficult to predict.manage

temporary files in more privileged environment , this could involve creating the files in directory with restricted access.

Enforce strict security policies around file operations.

Temporary file have to deleted automatically after the process exit.

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:	*Medium*
OWASP TOP 10 reference:	*A02:2021- Cryptographic Failures* *A07:2021- Identification and Authentication Failures*
Remediation recommendation	
The code uses the MD5 algorithm to store or process the passwords. MD5 is deprecated cryptographic function that is no longer secure due to known collision and preimage attacks. When used for password hashing, It allows attackers to quickly compete hash collisions or reverse hashes using rainbow tables and brute-force tools. *This flaw also increases the risk of broken authentication, since weak hashes make it trivial for attackers to recover passwords and access user accounts .*	

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:	*Low*
OWASP TOP 10 reference:	*A02:2021-Cryptographic Failures*

Remediation recommendation

The code's using Python's random.choice() which anyone can predict if they know the seed value, so hackers can basically guess what "random" values your system will generate next - not good for security stuff like passwords or tokens. Replace it with the secrets module which taps into your computer's most secure randomness source, making predictions impossible even for attackers with supercomputers. Just swap random.choice(letters) with secrets.choice(letters) in line 40 of init_db.py - literally a one-word change but massively upgrades security. The secrets module is built specifically for creating passwords, OTPs, session tokens, and other security-critical stuff where true randomness matters. Bottom line: if it's security-related (authentication, credentials, tokens), always use secrets; if it's just for games or testing, random is fine write this more simple.

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	
"lxml.etree.fromstring(xmlString)“ *This allow to parse a string containing XML into an XML element tree without any input validation or proper senatization. Improper use "lxml" lead to several XML vulnerabilities such as XXE (XML External Entity injection). Billion Laughs / Entity Expansion attacks .* *To avoid this implement "defusedxml.lxml" instead of "lxml" , and implement proper input senatization. Implement the input size limit.*	

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:	*Low (False Positive)*
OWASP TOP 10 reference:	*N/A*
Remediation recommendation	
In this parameter is set to "shell=false" . So it state that input parsing won't get execute in shell. *if "startupcmd" argument is fully validated by the application not by the user , then it will repot as false positive*	

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
```

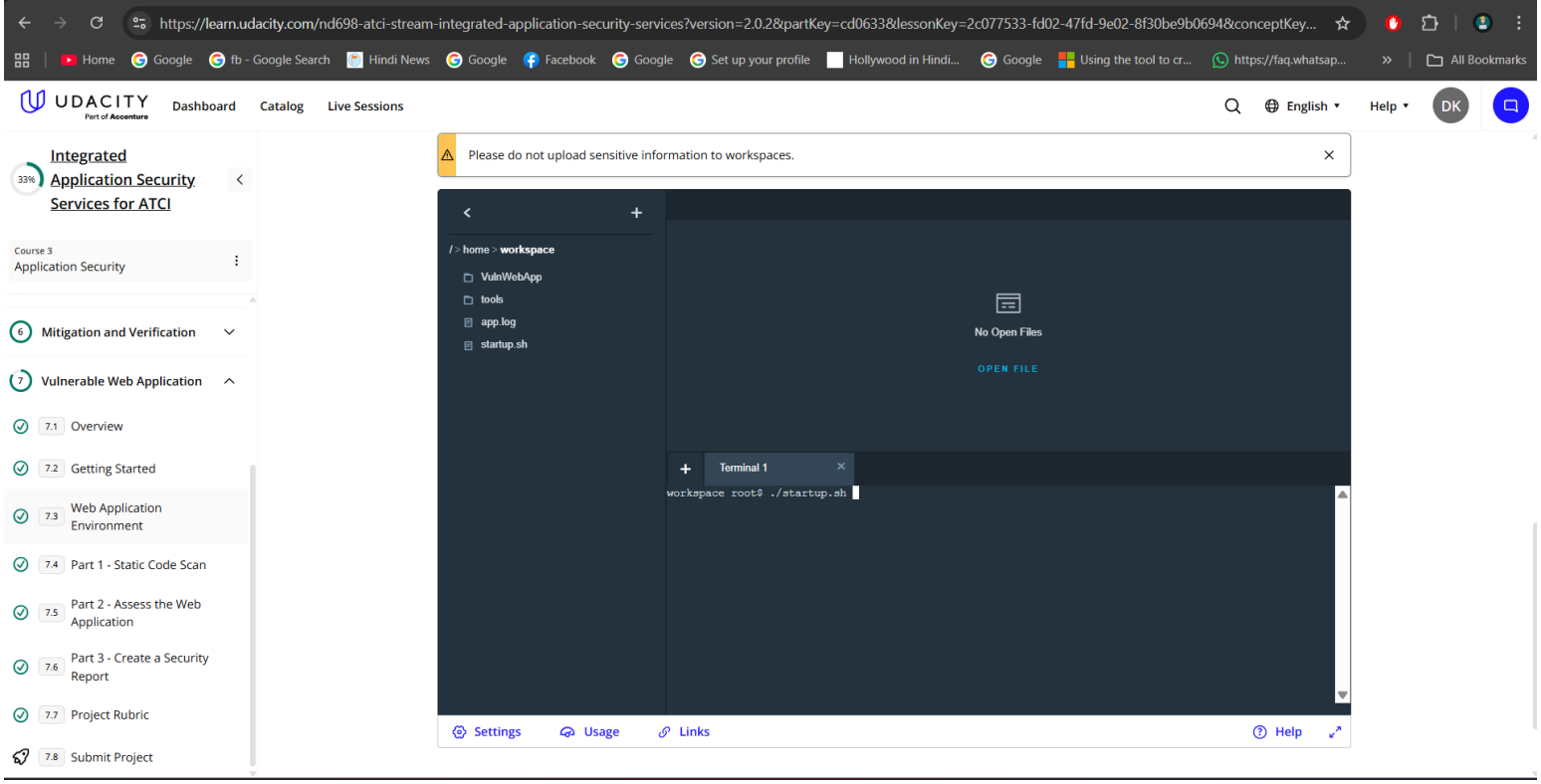
Severity:	*Medium*
OWASP TOP 10 reference:	A03:2021 - Injection (specifically Cross-Site Scripting - XSS

Remediation recommendation

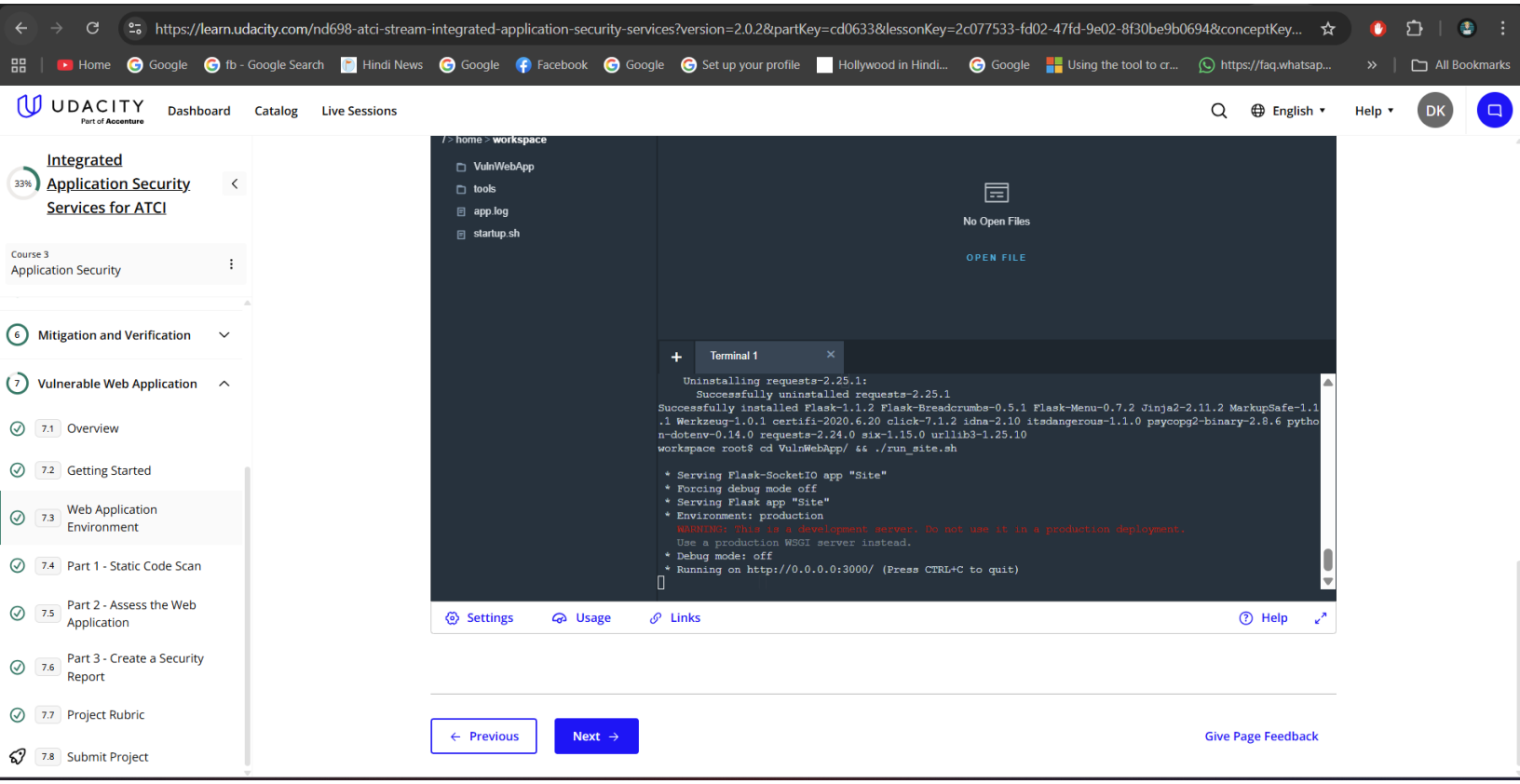
The code uses Django's mark_safe() function on line 23 of create_customer.py to bypass Django's automatic HTML escaping for the password field.
When mark_safe() is applied to user-controlled input, attackers can inject malicious JavaScript that executes in victims' browsers
XSS is one of the most common web application vulnerabilities, allowing attackers to inject client-side scripts into web pages viewed by other users.
Remove mark_safe() *because* MD5 hashes are alphanumeric and don't need

Section Two:

Assess the Web Application



- Start the workspace. Then open the Terminal
- Then run command **./startup.sh**
- Then run **cd VulnWebApp && ./run_site.sh**
- Then access the lab from links



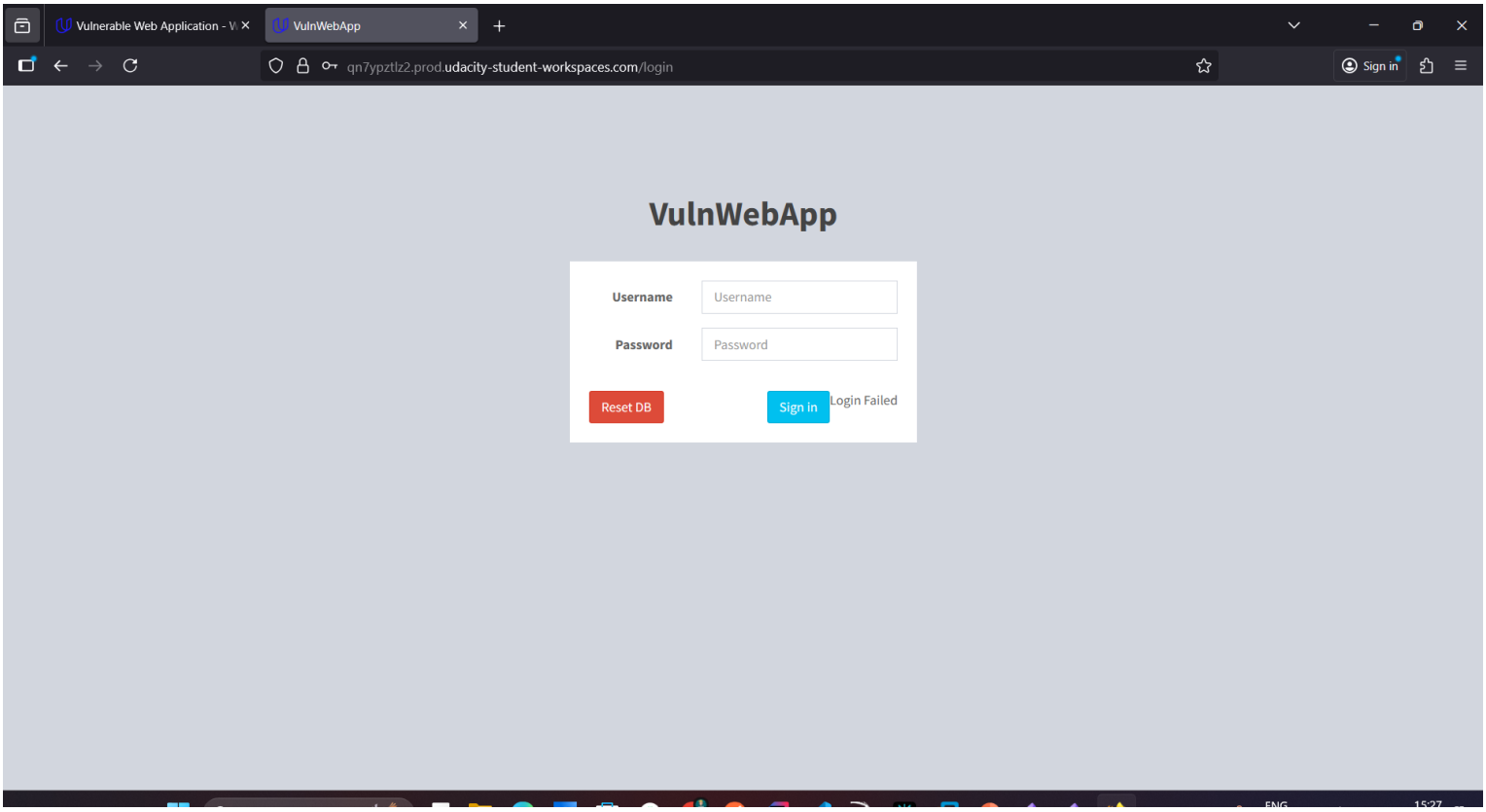
Section Three: Security Report

Broken Authentication

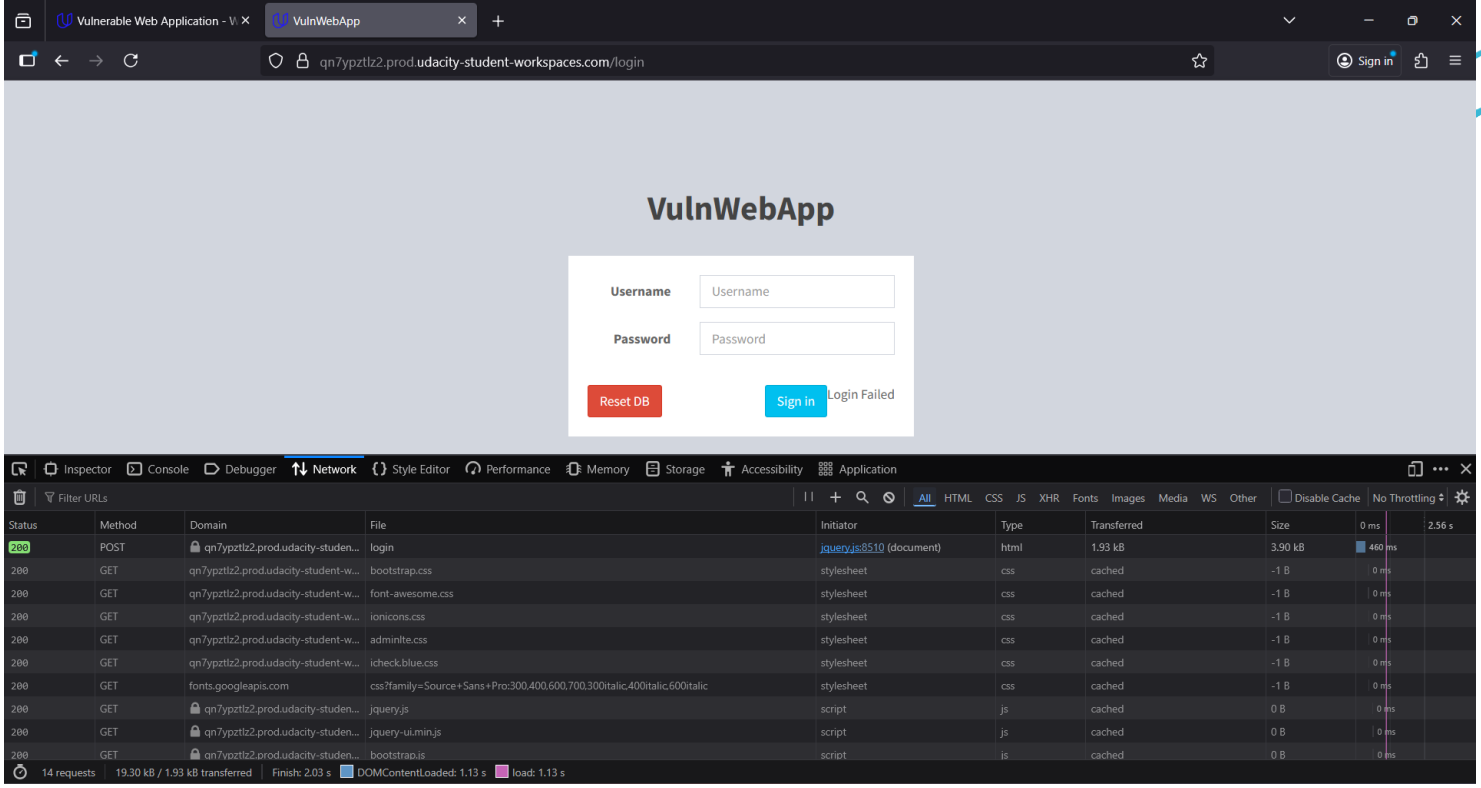
Severity:	*High*
OWASP TOP 10 reference:	A07:2021 – Identification and Authentication Failures
Vulnerability Explanation	
The login functionality is vulnerable to brute-force attacks. An attacker can make unlimited authentication attempts without being blocked or delayed. This allows guessing of valid credentials and may lead to unauthorized access to user accounts or administrative systems.	
Remediation recommendation	
We can Enable MFA (Multi-Factor Authentication): Require OTP for login.We can also Rate Limiting for the restrict the number of login attempts as IP or account(e.g. 5 attempt in 1 hour)We can use Monitor and Alert log all failed login attempts and generate alert for suspicious patterns.We can use password policy for enforce strong password requirements and check against known breached passwords.We also add Implement Account Lock for temporarily lock or delay in login attempts after repeated failures.	

Broken Authentication

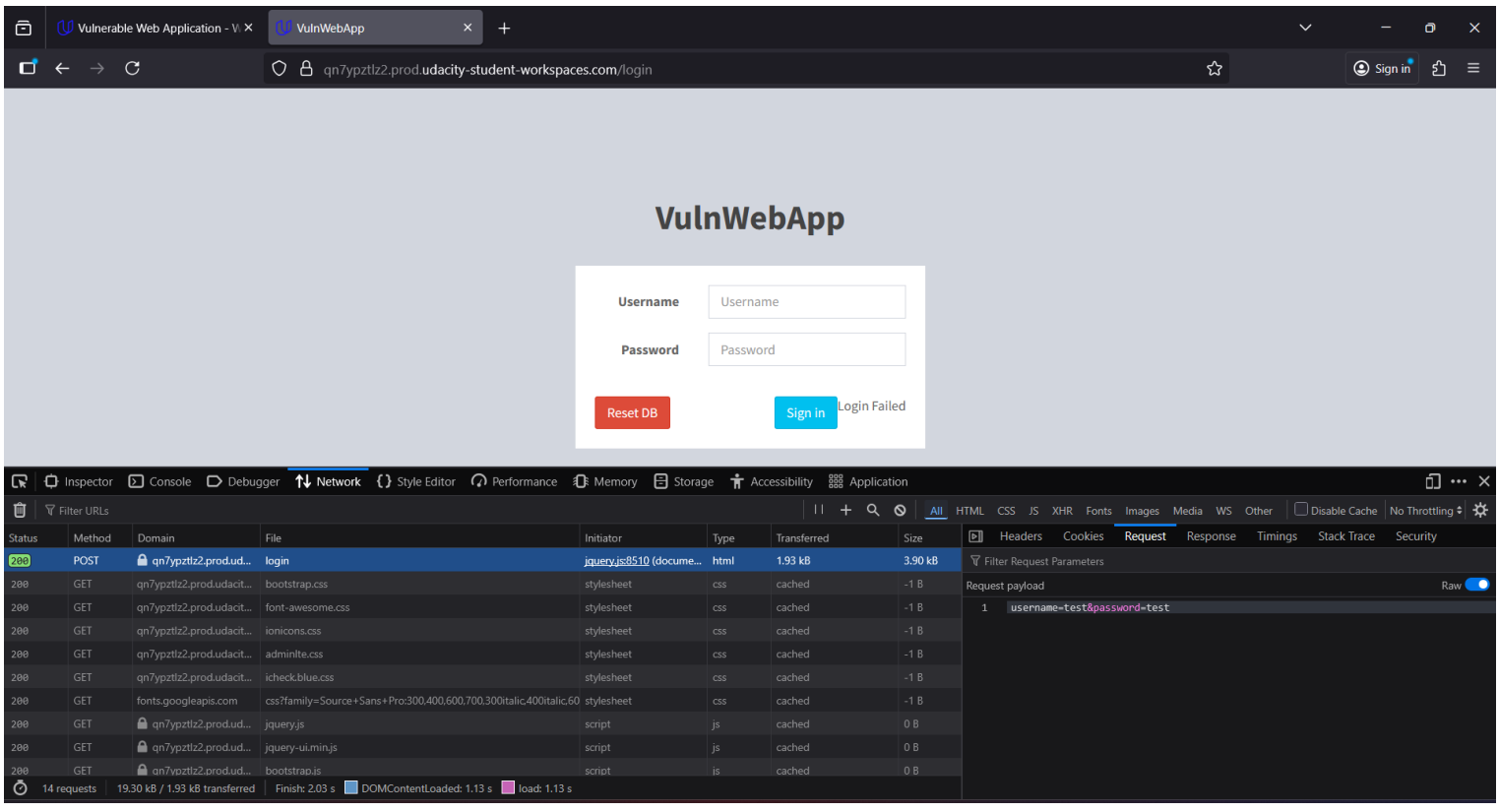
step-by-step walkthrough



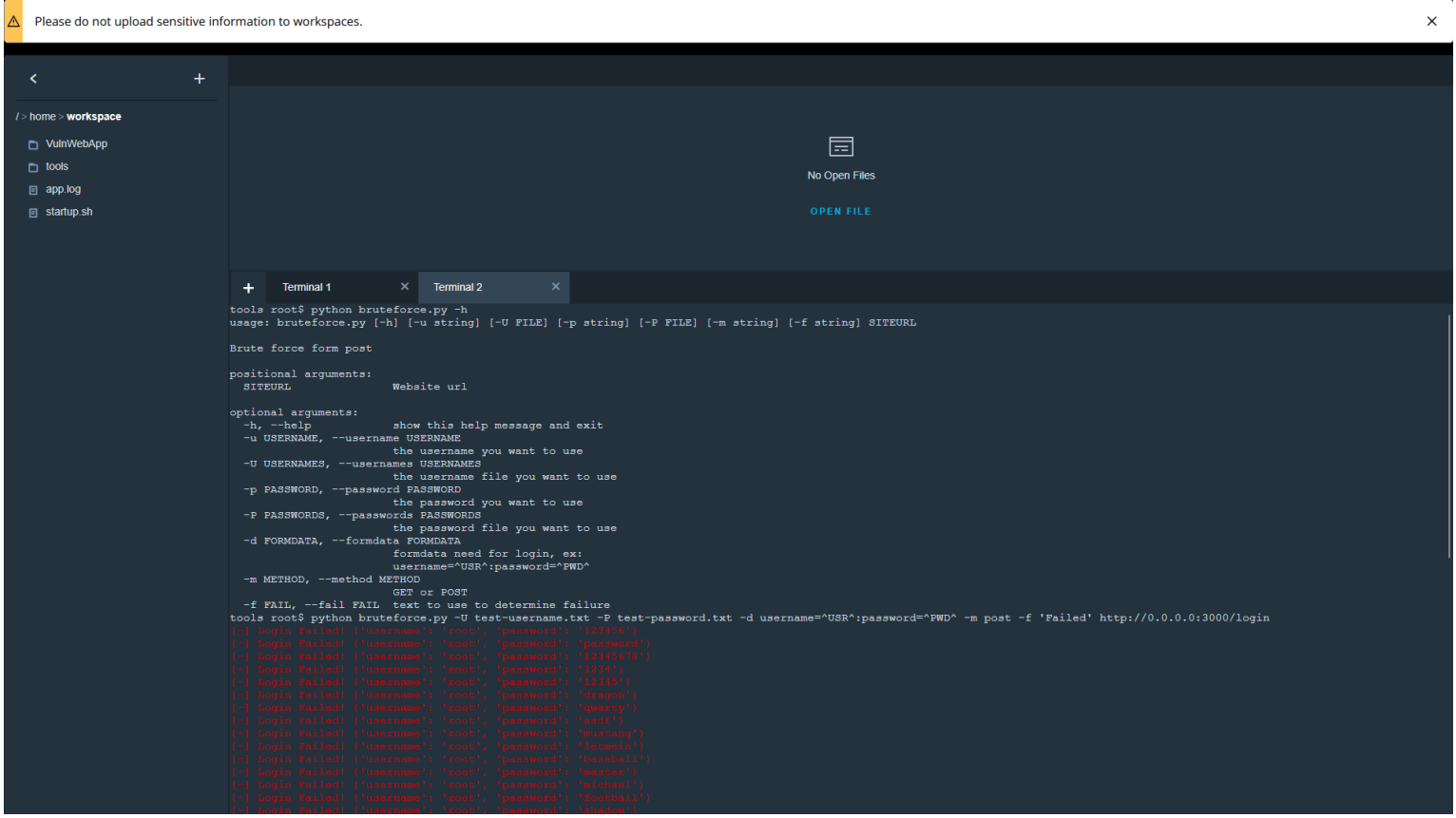
- After visiting the website, it open a **login** page (/login)
- Press "**Reset"** before perform attack.



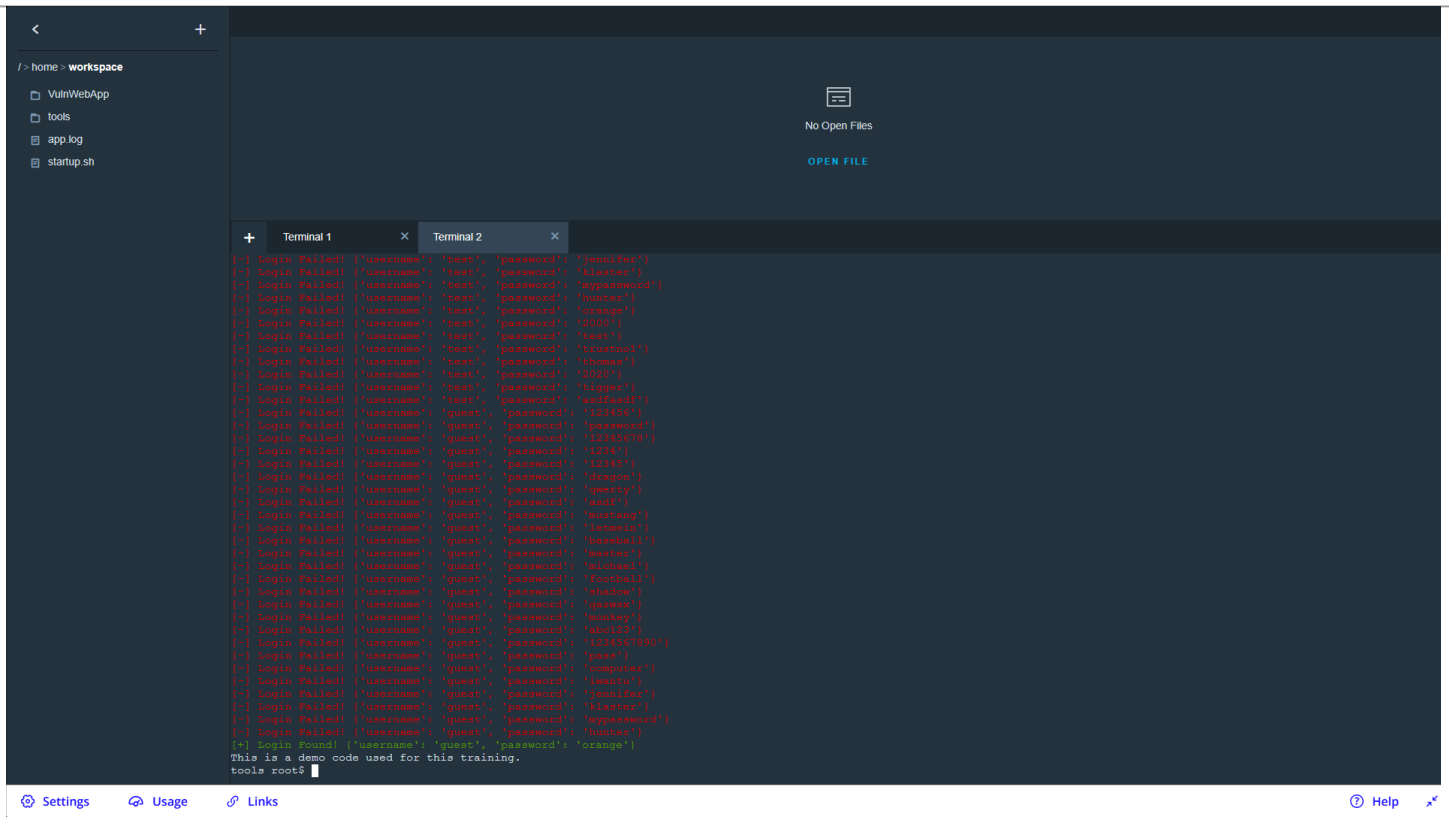
- Open the developer tool & network tab to see the **post request** of the login



- It show the request payload “username=test&password=test” ,
- In these test credentials it gives message **“Login Failed”**



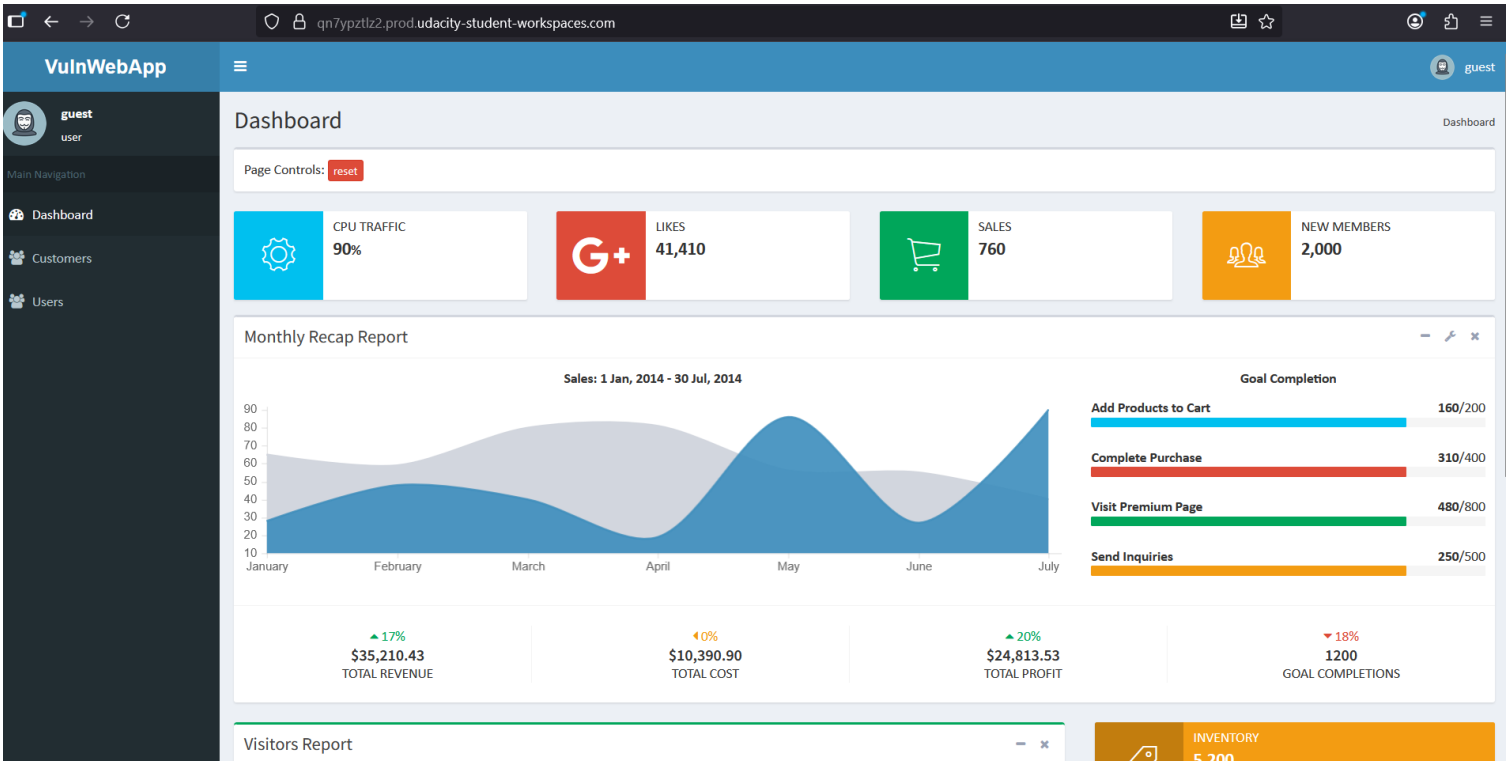
- By using bruteforce tool "bruteforce.py" , craft the command by login payload credentials got from previous screenshot.
- **python bruteforce.py -U test-username.txt -P test-password.txt -d username=^USR^:password=^PWD^ -m post -f 'Failed' http://0.0.0.0:3000/login**
- Use the given username and password file



```
[-] Login Failed! {'username': 'guest', 'password': '1234567890'}
[-] Login Failed! {'username': 'guest', 'password': 'pass'}
[-] Login Failed! {'username': 'guest', 'password': 'computer'}
[-] Login Failed! {'username': 'guest', 'password': 'iwantu'}
[-] Login Failed! {'username': 'guest', 'password': 'jennifer'}
[-] Login Failed! {'username': 'guest', 'password': 'klaster'}
[-] Login Failed! {'username': 'guest', 'password': 'mypassword'}
[-] Login Failed! {'username': 'guest', 'password': 'hunter'}
[+] Login Found! {'username': 'guest', 'password': 'orange'}
This is a demo code used for this training.
tools root$
```

Usage Links

- Brute force attract got successful and get the username and password
- **Username - Guest**
- **Password – orange**
- By these got login in website.



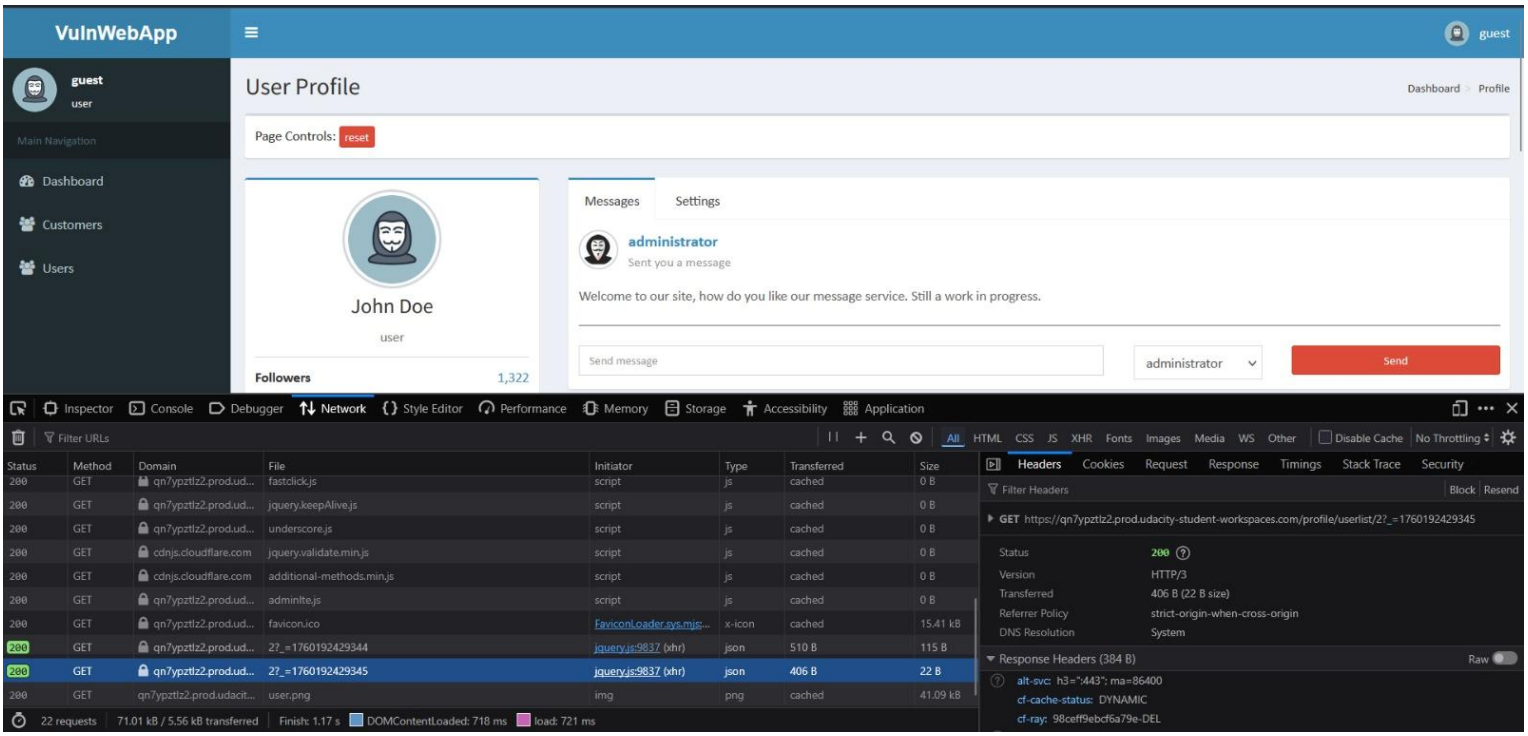
Broken Access

Severity:	*Critical*
OWASP TOP 10 reference:	*A01:2021 – Broken Access Control*
Vulnerability Explanation	
*The application uses client-side cookies **authInfo** to store sensitive user data such as role or authorization level.* *If this value is not validated or signed securely on the server, an attacker can manipulate the cookie (e.g., change "role=user" to "role=admin") and gain unauthorized access to admin functionalities.This is a Broken Access Control issue because the system fails to enforce proper access restrictions on the server side*	
Remediation recommendation	
Do not store roles or permissions in client-side cookies or tokens without proper integrity protection because Always validate roles on the server side.Implement server-side access control checks for every endpoint or action because Each request should verify the user's actual role and privileges from the server's session or database.Apply the Principle of Least Privilege because Grant only the minimum required permissions to each role.Add server-side validation and logging because Detect and block suspicious role changes.	

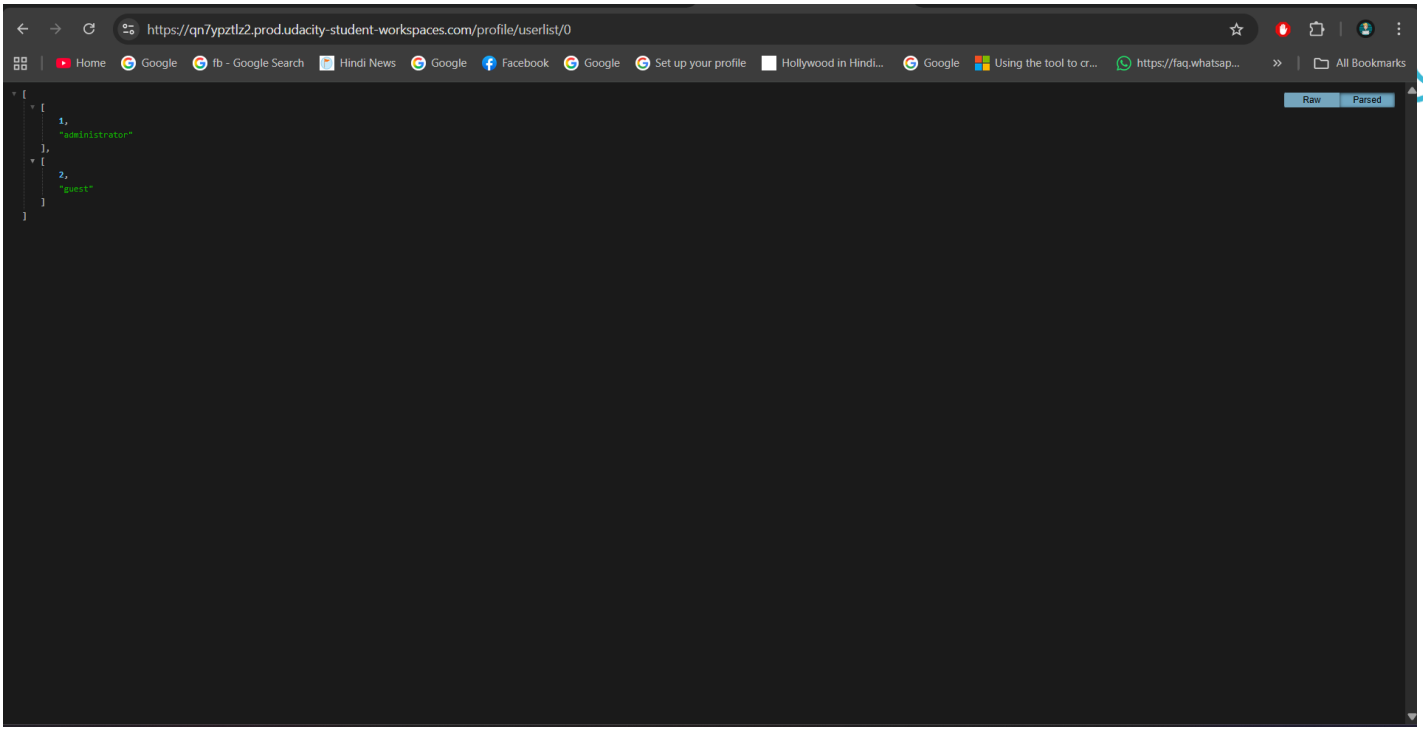
Broken Access

Step-by-step walkthrough

- We are logged in as guest user.
- Go to profile page. One developer tool then network tab
- Refresh page.

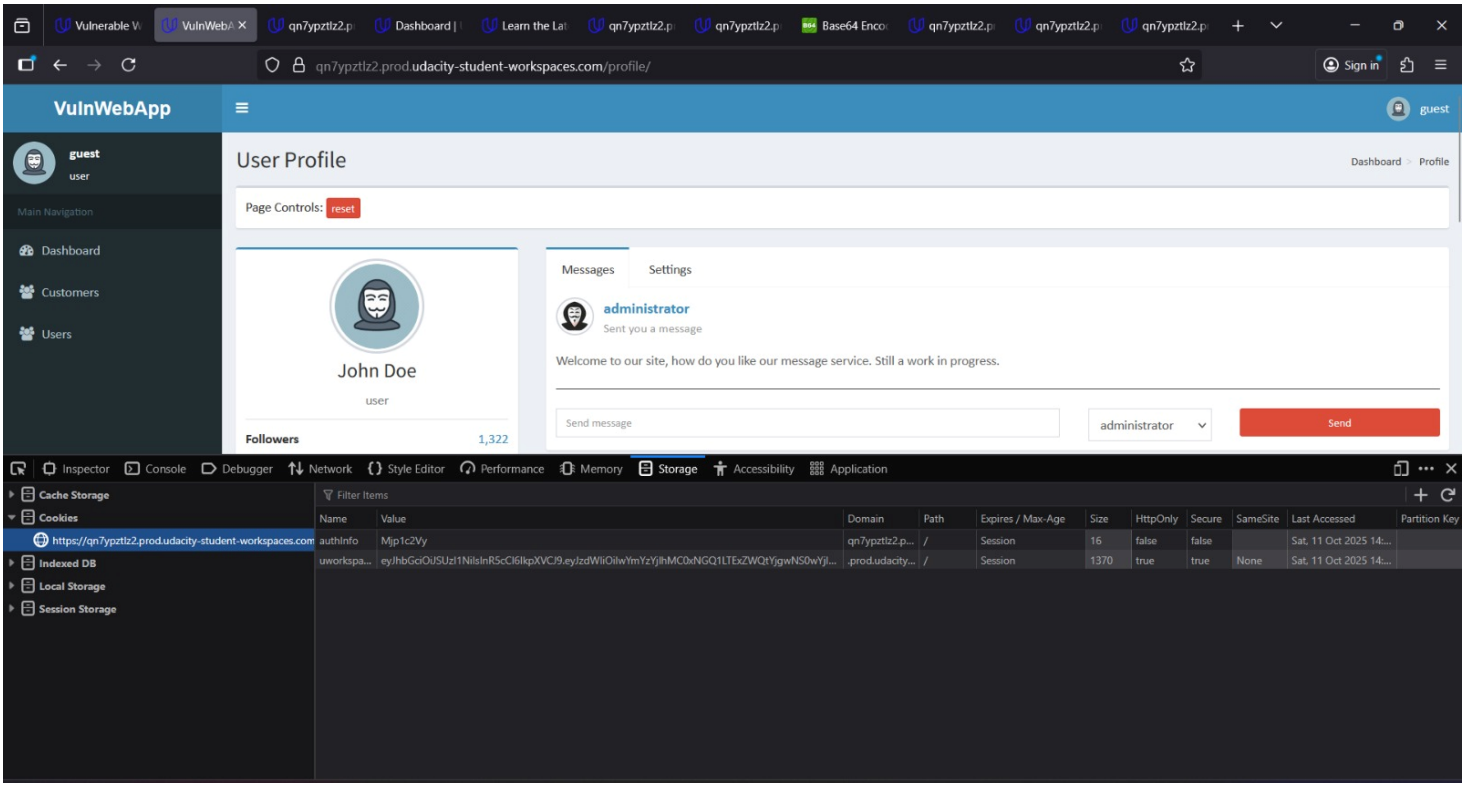


- We can see a request below which get the api **“/profile/userlist”**
- Which show the data of user id

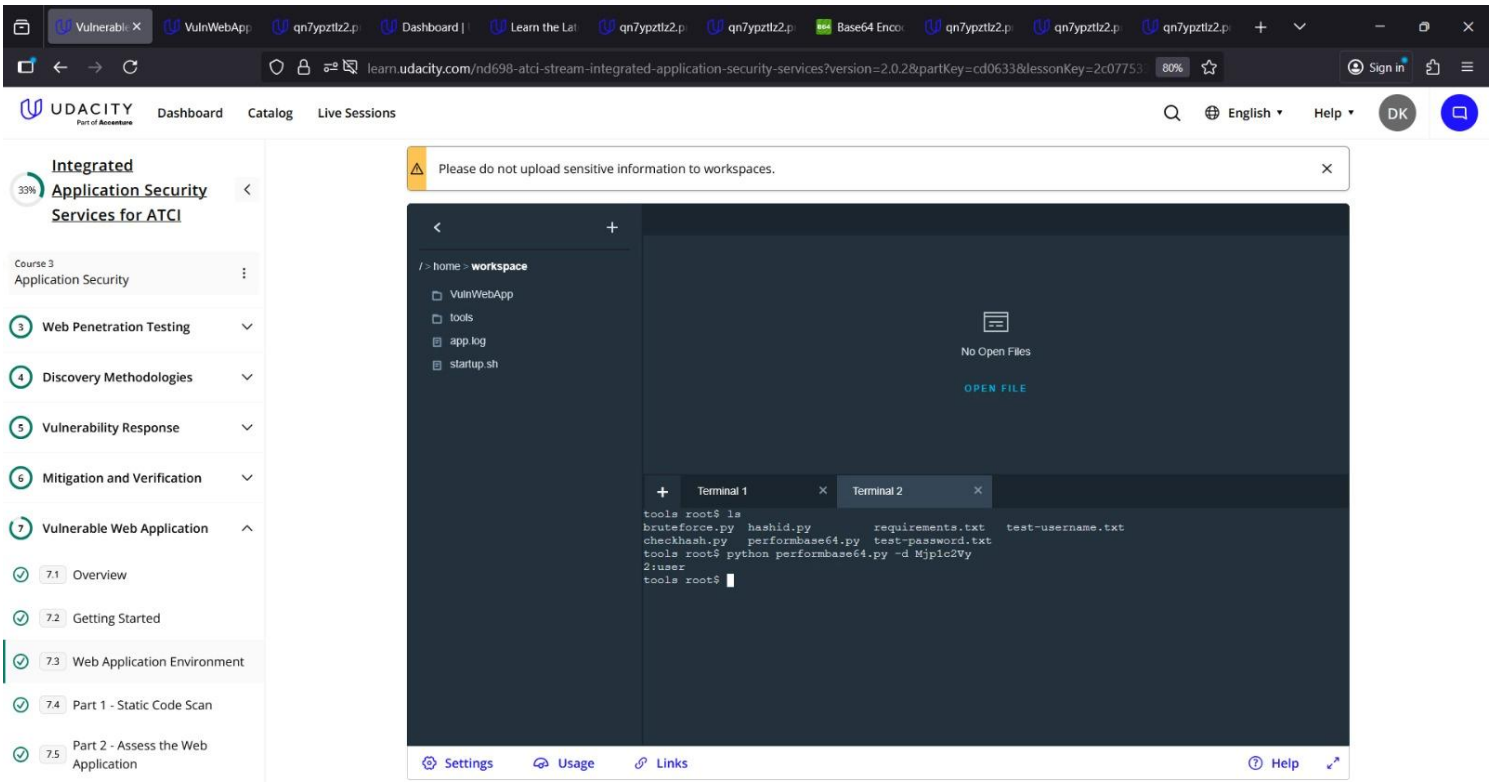


- On hitting this url with **/profile/userlist/0**
- it show the admin id

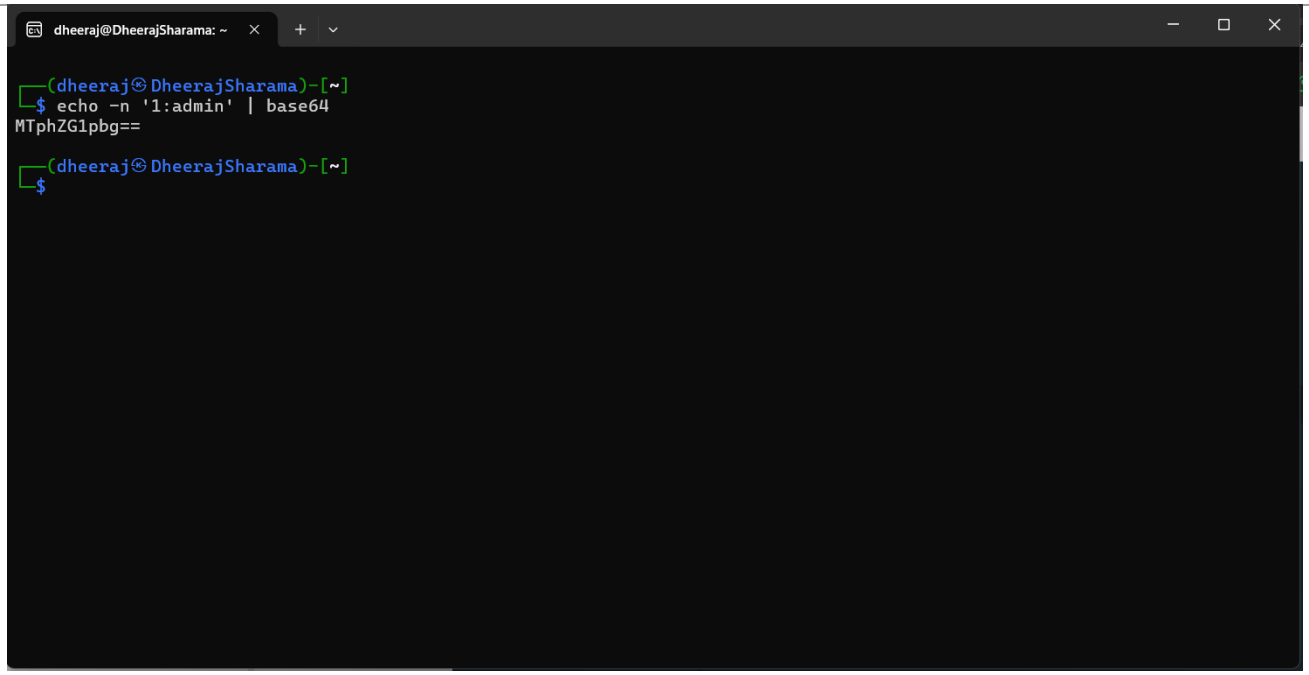
- Next open developer tools and got to storage cookies
- We can see the **authinfo** cookie its base64



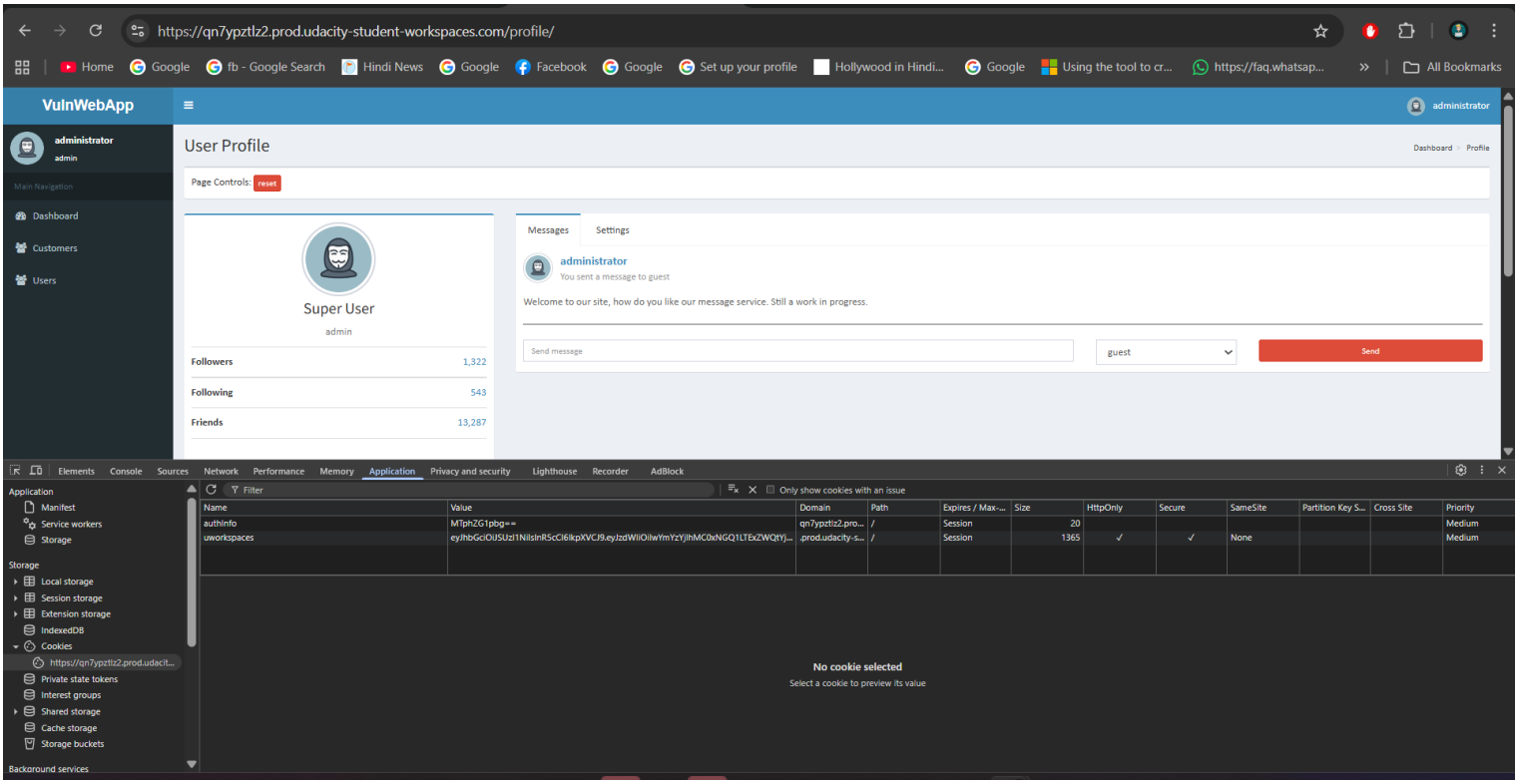
- Use the give tool **performebase64.py** to decrypt base64
- Its value is **2:user**



- Now make **authinfo** cookie for the admin
- Admin id we have founded in **profile/userlist/0**
- Admin cookie will we **1:admin**
- Then encode this to base64



- Not replace the user auth cookie with the admin cookie and refresh the page



- Now we got the **Admin** access

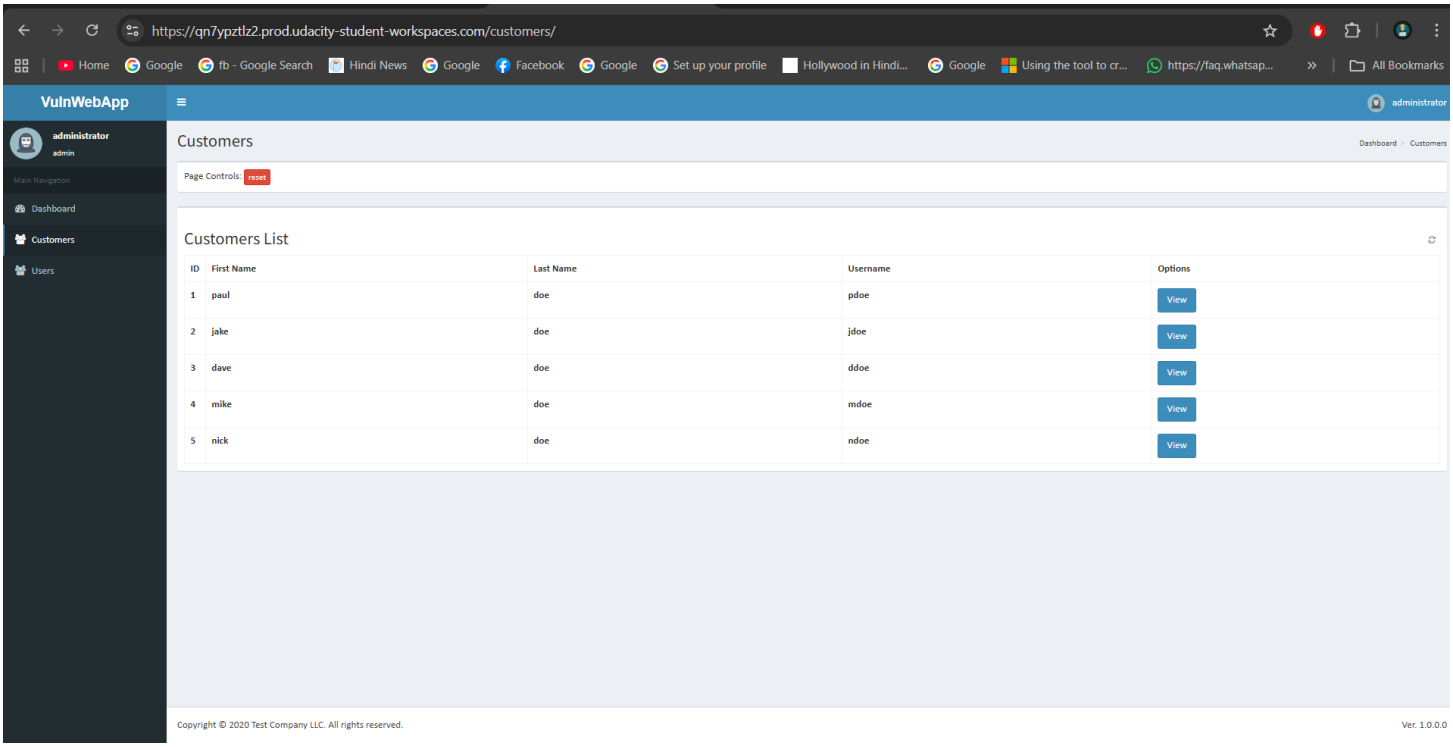
Sensitive Data Exposure

Severity:	*High*
OWASP TOP 10 reference:	*A02:2021-Cryptographic Failures*
Vulnerability Explanation	
The API /customers/id/1 is returning user details without proper authentication or access control. Even though the password seems hashed, exposing it in a public or client-facing API is still dangerous. Attackers can use brute-force or rainbow table attacks to recover the original password. It leaks identifiable customer data (names, usernames).	
Remediation recommendation	
Add authentication & authorization because Require a valid user token or session before anyone can access /customers/id/....Don't expose sensitive data in API responses because Only return the data your frontend actually needs (e.g., not password hashes, tokens, or internal IDs).Hash and salt passwords securely on the server (but never return them).Use proper access control because a user should only see their own data, not anyone else's.securing endpoints, limiting what's returned, and protecting stored data.	

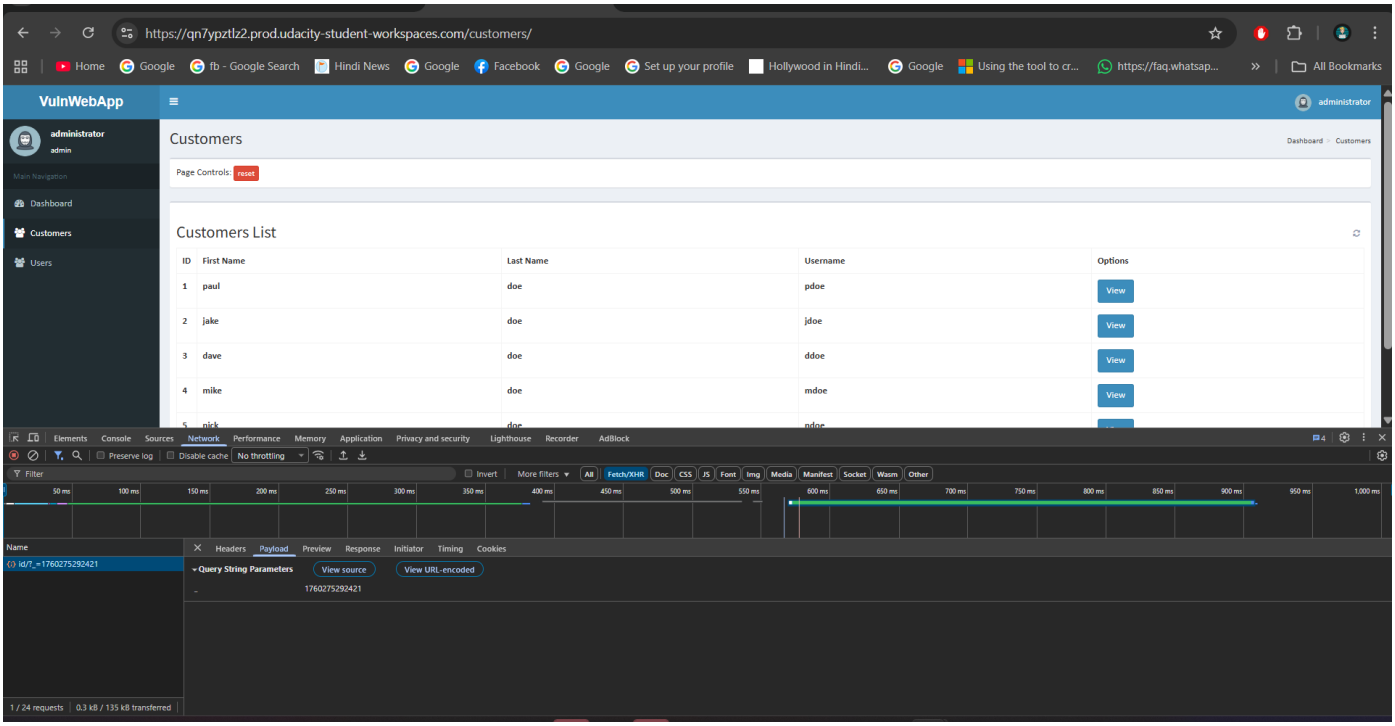
Sensitive Data Exposure

step-by-step walkthrough

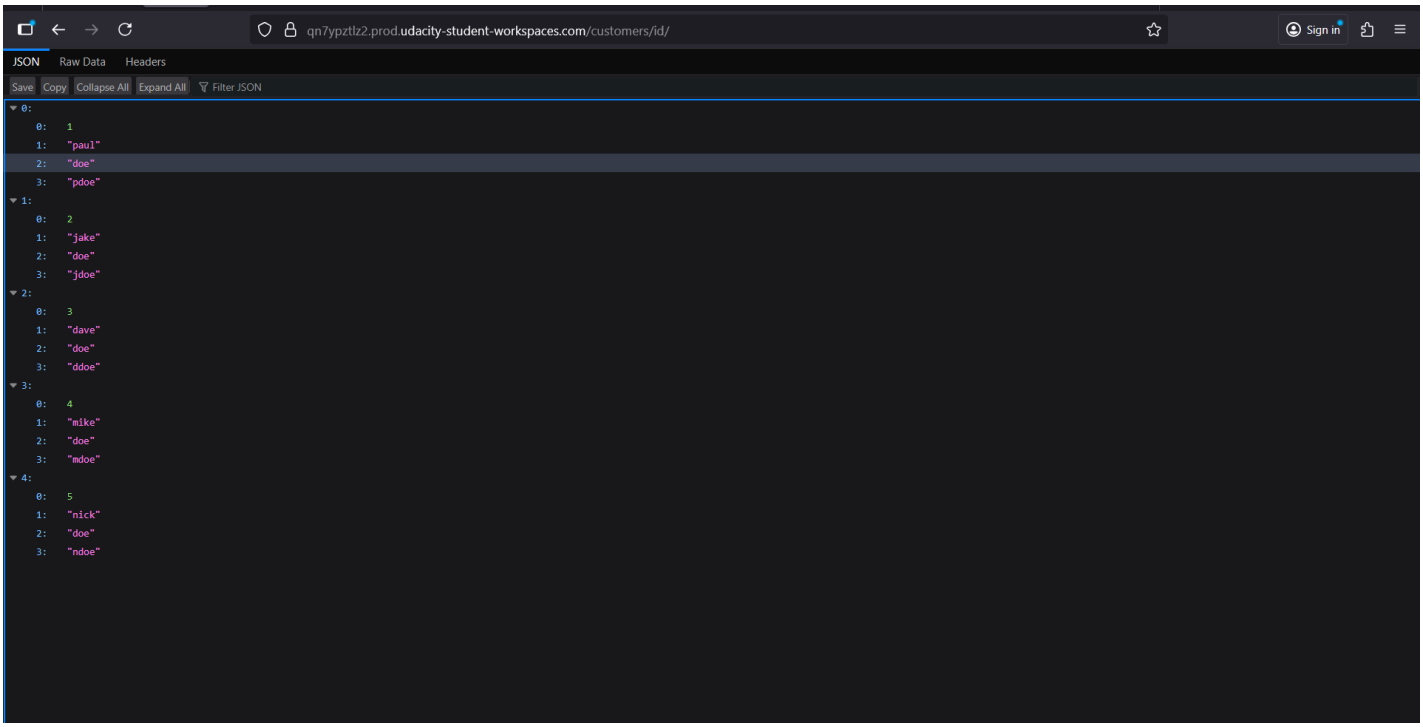
- We got the Admin access earlier
- Now we can see the Customers list



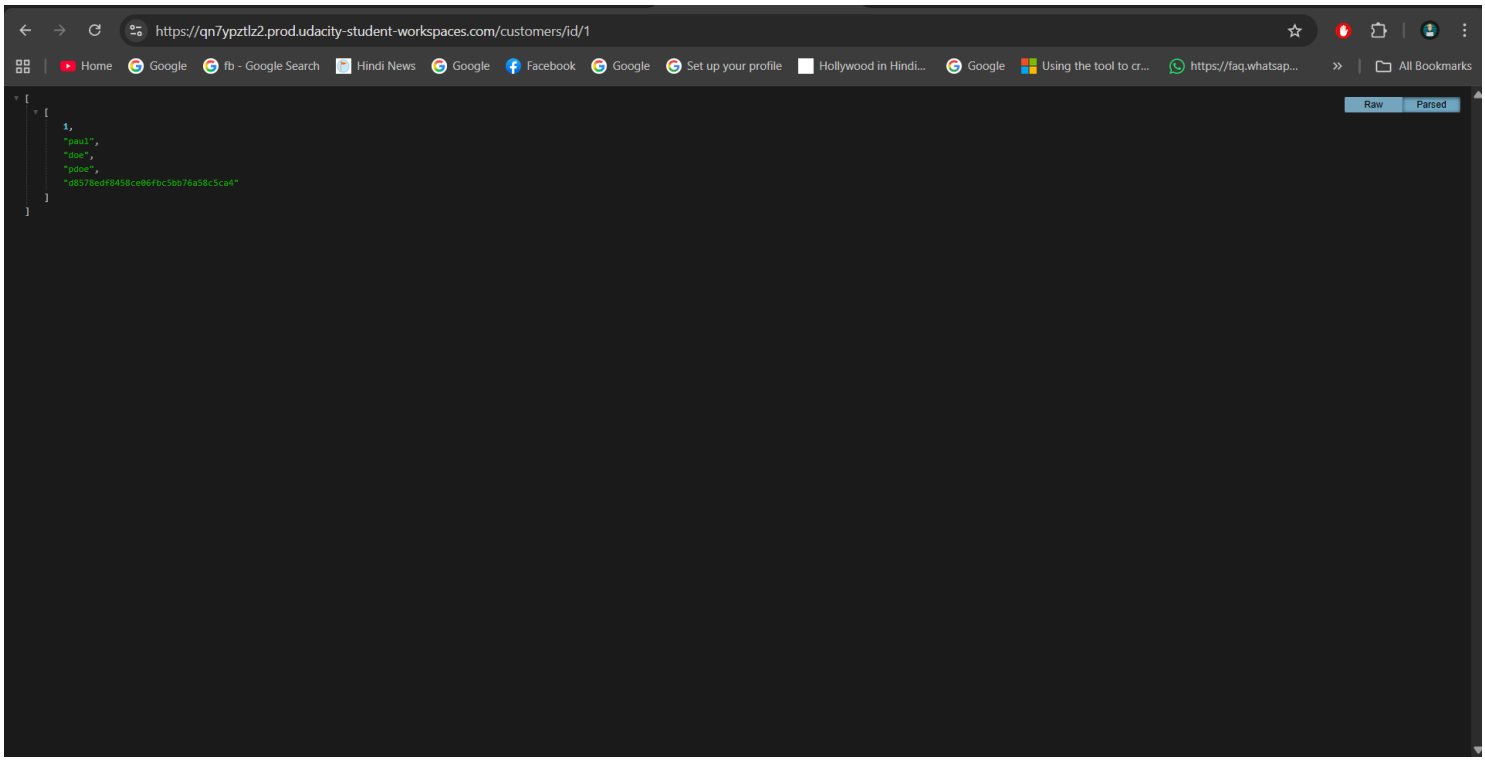
- We look the network request of this page
- It request to API **/customers/id**

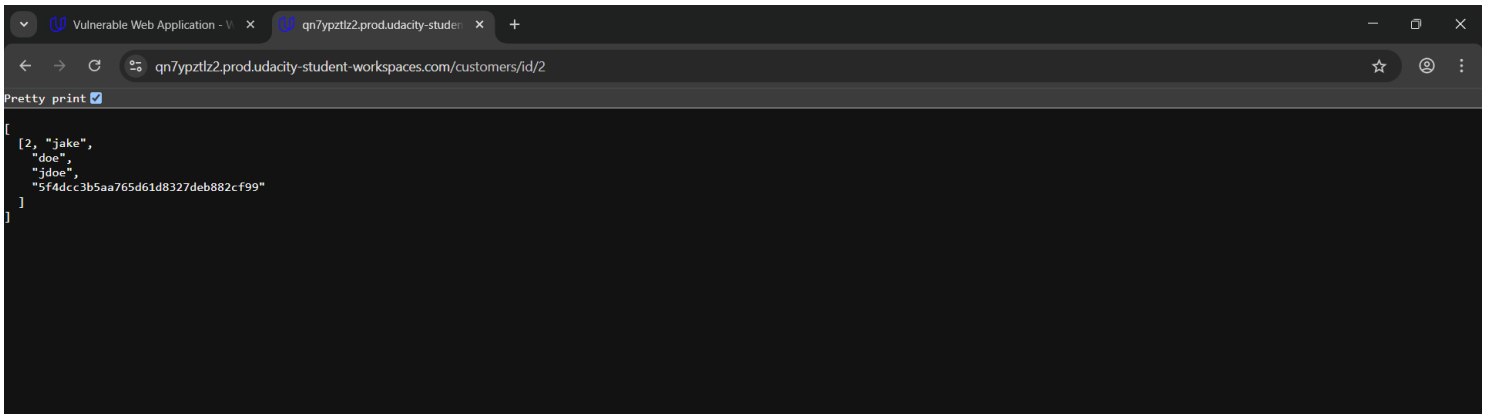


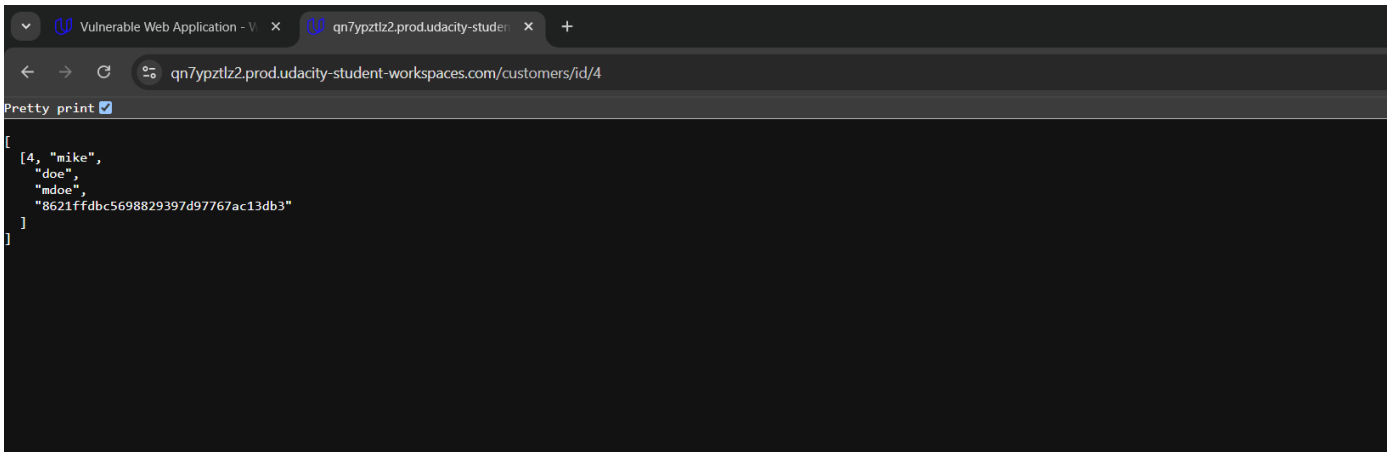
- Opening the request to new tab will hit the Api **/customers/id**
- It will show the Json data of customers



- Now go to specific customer id , which revel the sensitive information
- **Hash of the Password**







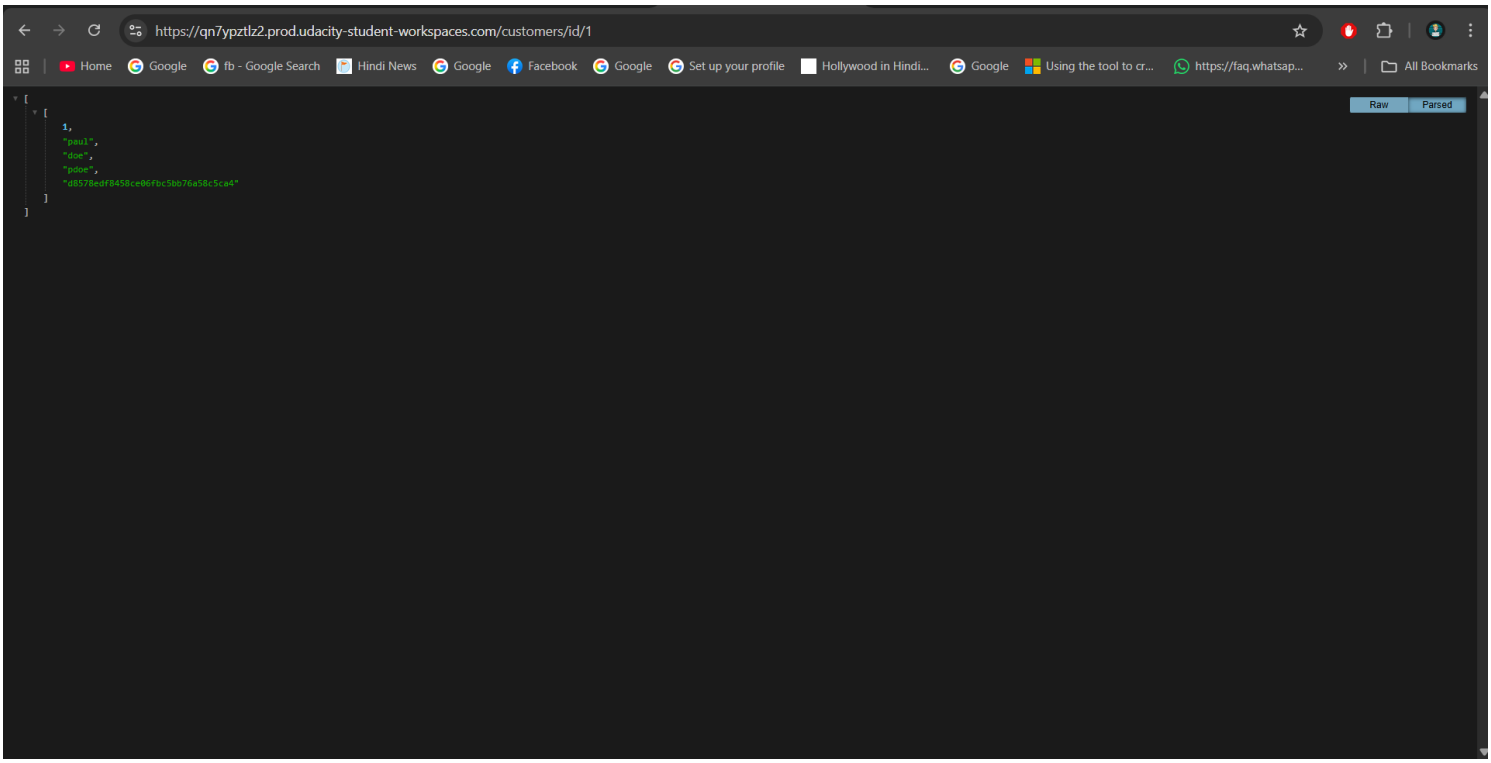
Cryptographic failures

Severity:	*High*
OWASP TOP 10 reference:	A02:2021 – Cryptographic Failures
Vulnerability Explanation	
The system is using weak hash like MD5 to store passwords It might look secure because the password turns into a long string like d8578edf8458ce06fbc5bb76a58c5ca4, but in reality, hackers can crack it in seconds using free online tools or pre-made hash databases.So even though it seems hidden, that string can be quickly turned back into “qwerty” — meaning your users’ passwords are practically lying in plain sight.	
Remediation recommendation	
Stop using MD5 or SHA1 because they are old and insecure. *Enforce strong password rules for users — avoid weak passwords like “qwerty”.* *Add a unique salt to each password before hashing.* *Use strong, modern password hashing algorithms such as: bcrypt.* *Re-hash old passwords because when users log in next time, re-hash their password with a strong algorithm.*	

Cryptographic failures

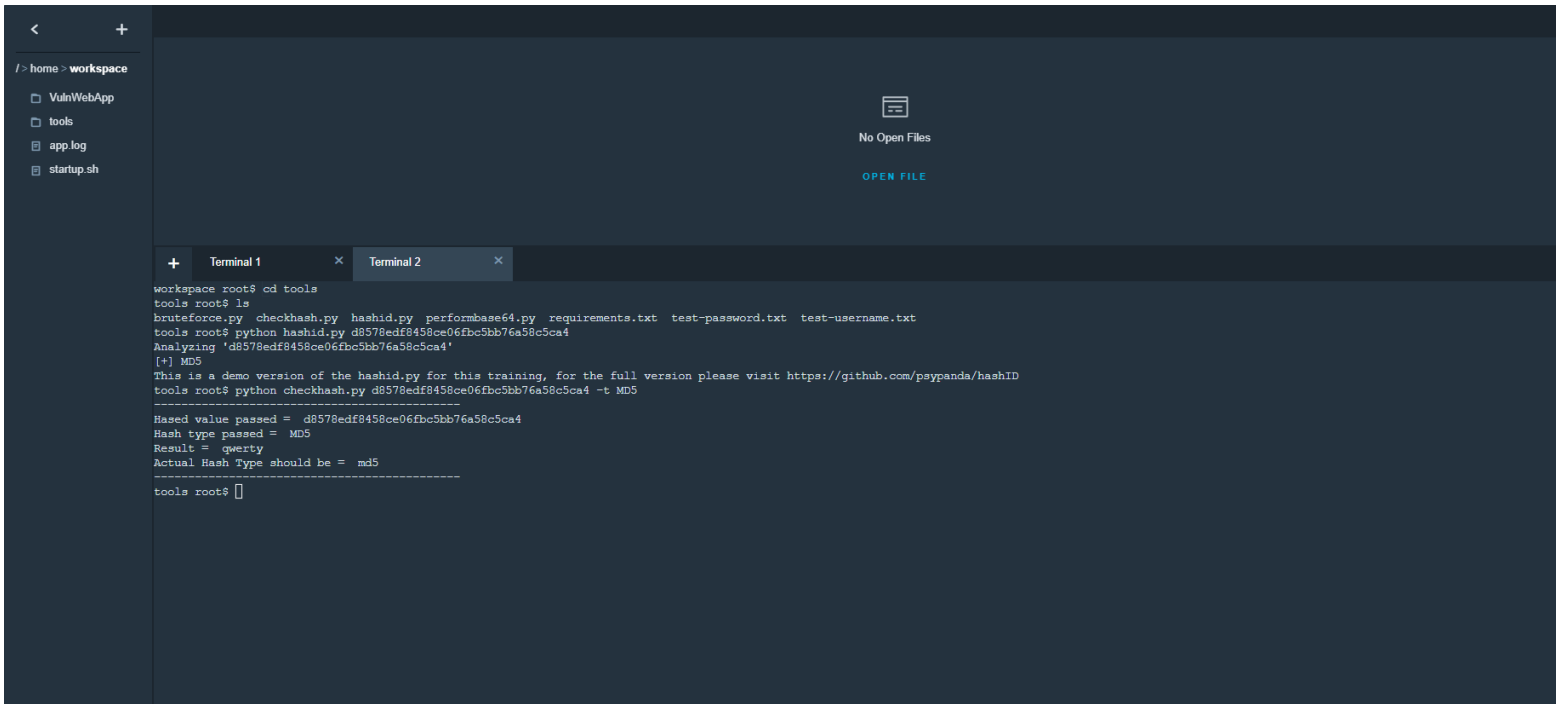
step-by-step walkthrough

- From specific request to customer ID like **/customer/id/1**
- It give us customer data with their Hash Password

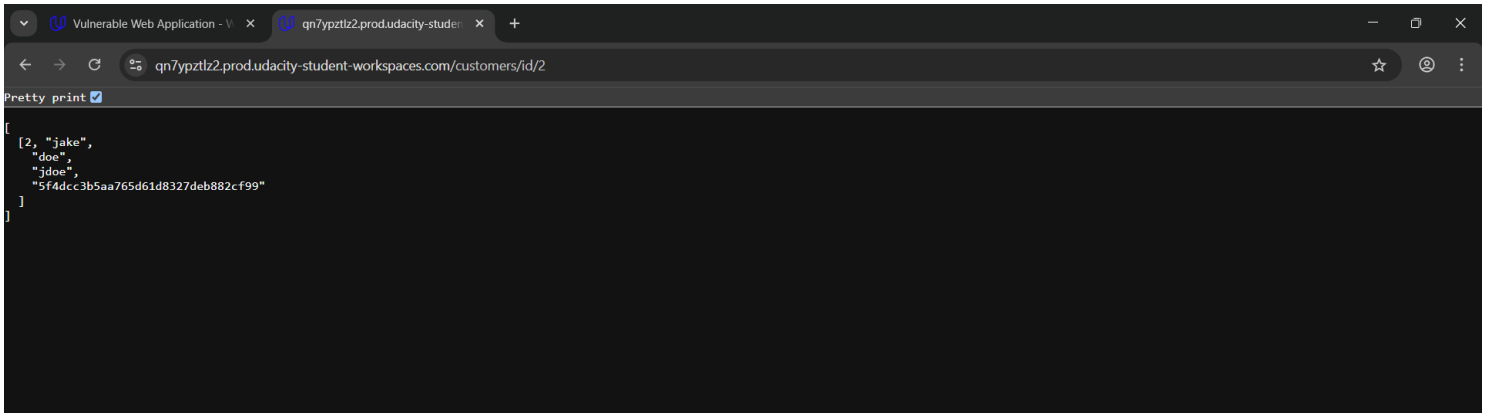


- Hashing used here is very weak & can be cracked by brute force
- Now use the given tool **hashid.py** to check which type of hash is this
- And try to crack the hash with given tool **checkhash.py**

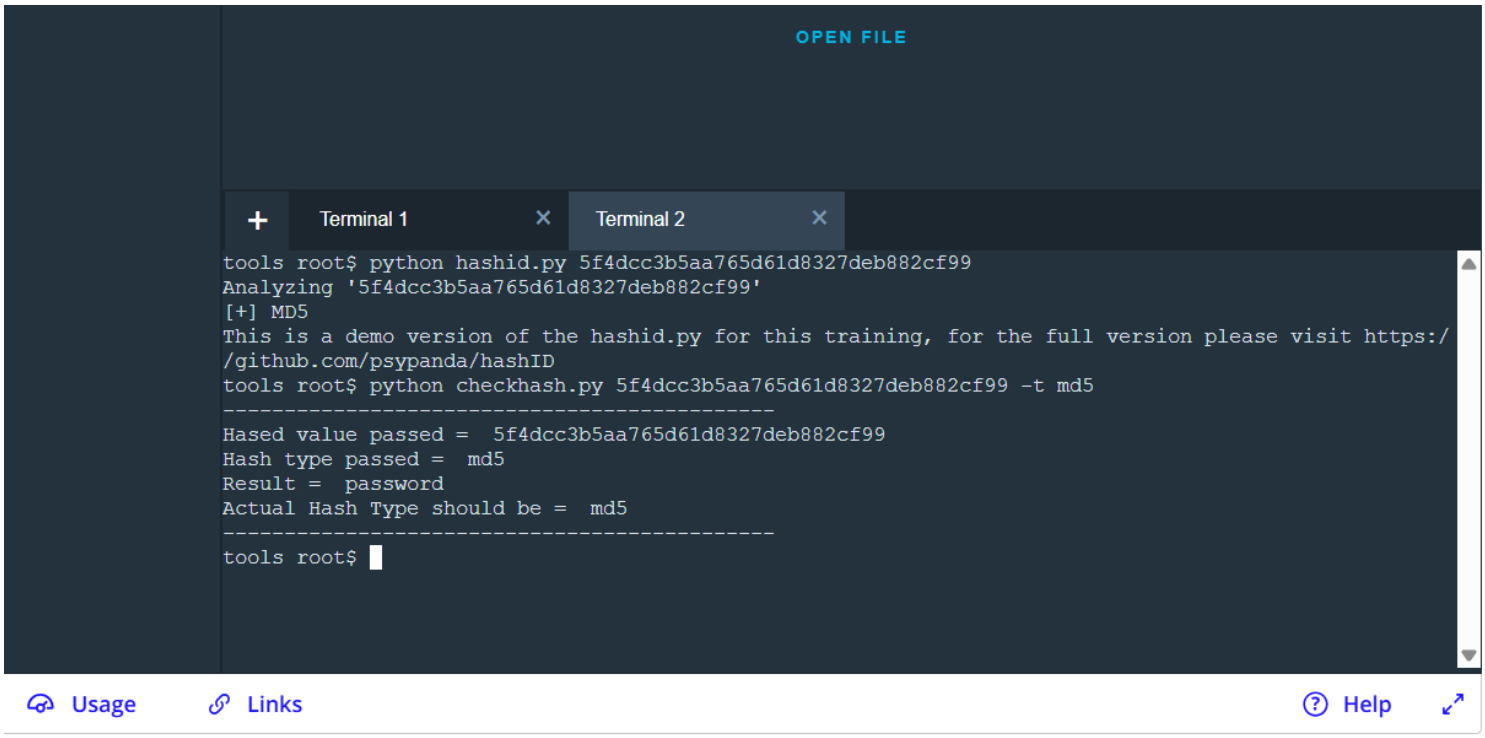
- **python hashid.py “hash here”**
- Use above command to get the type of hash
- We got the hash type is **MD5**
- Now use **python checkhash.py “hash here” –t “hash type”**
- it crack the hash and give the password of customer



- Now let's crack another hash of customer id 2



- By using **hashid.py** we can see the hash type if **md5**
- By using **checkhash.py** and give type as md5 we can crack the hash
- And got the password for customer 2



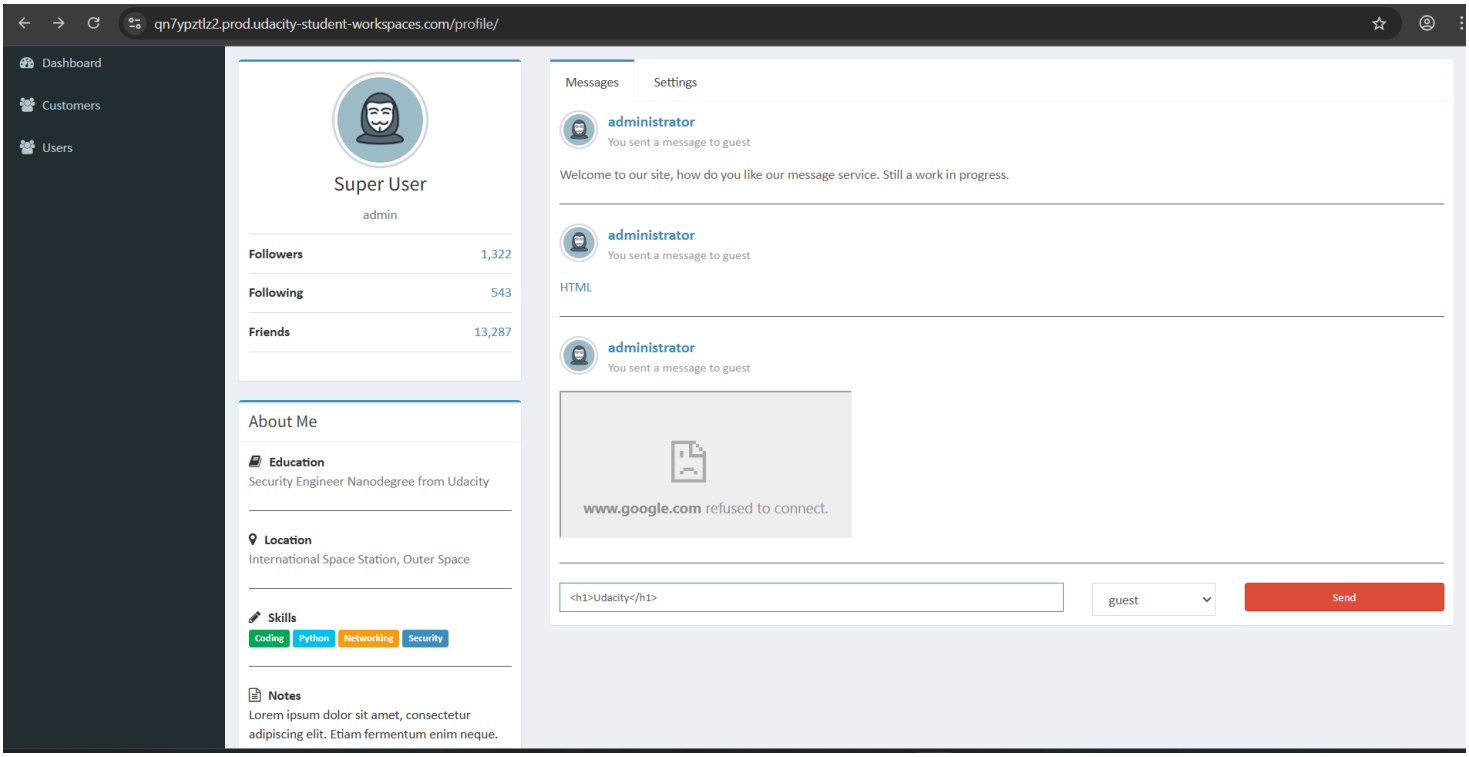
HTML Injection

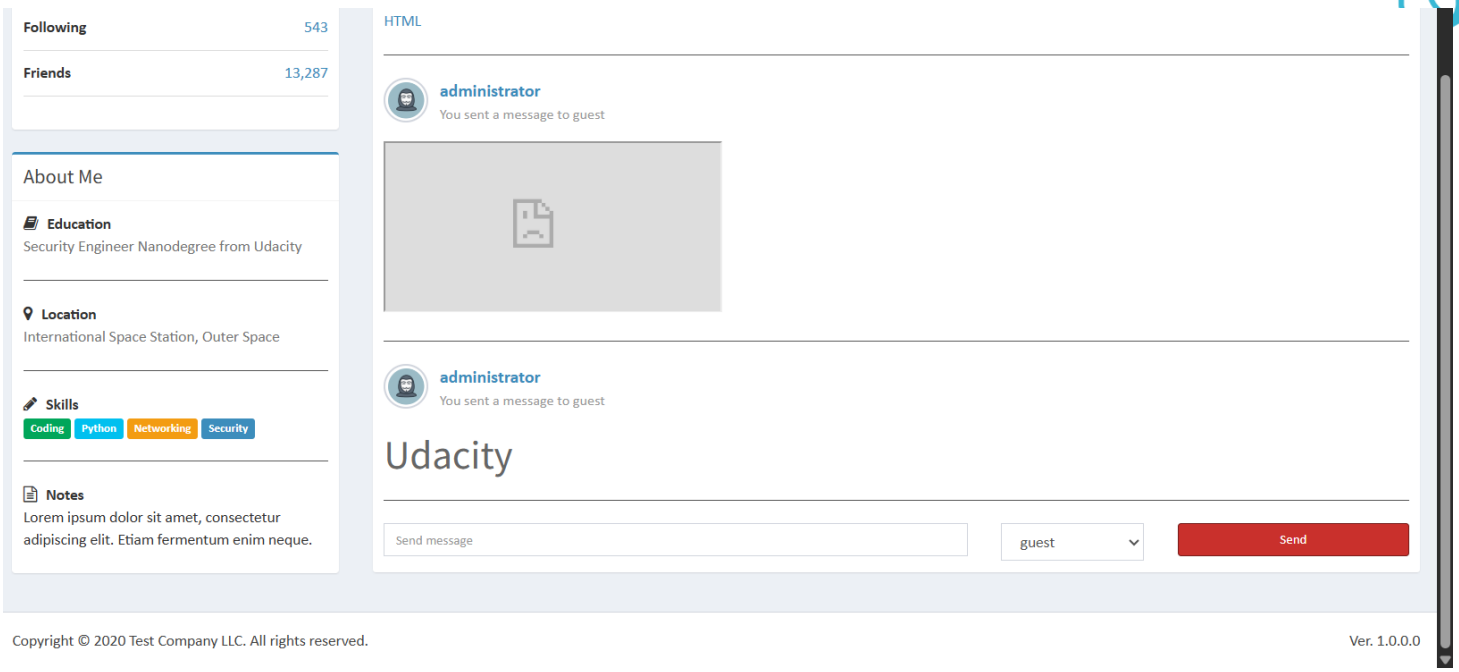
Severity:	*High*
OWASP TOP 10 reference:	A03:2021 – Injection
Vulnerability Explanation	
The website is executing user input directly without cleaning them.If someone puts malicious code like <h1>Udacity<h1> in a message, it will run on user's end as well as to whom message is sent . who opens that page.That means a bad guy can:Steal your login cookie Trick users into doing actions (like changing passwords or sending messages)Or even take over admin accounts	
Remediation recommendation	
Validate and filter inputs because it only allow safe characters or limited HTML if needed. *Sanitize data on both input and output because always treat user input as unsafe until cleaned.* *Escape or sanitize all user input before displaying it because use built-in frameworks or libraries to escape HTML special characters (<, >, ") for example, show Hello as text, not bold.* *Fix it by cleaning and escaping user input, so browser treats it as text — not code.* *Use Content Security Policy (CSP) because it set limits what scripts can run on your page, blocking many XSS attacks.*	

HTML Injection

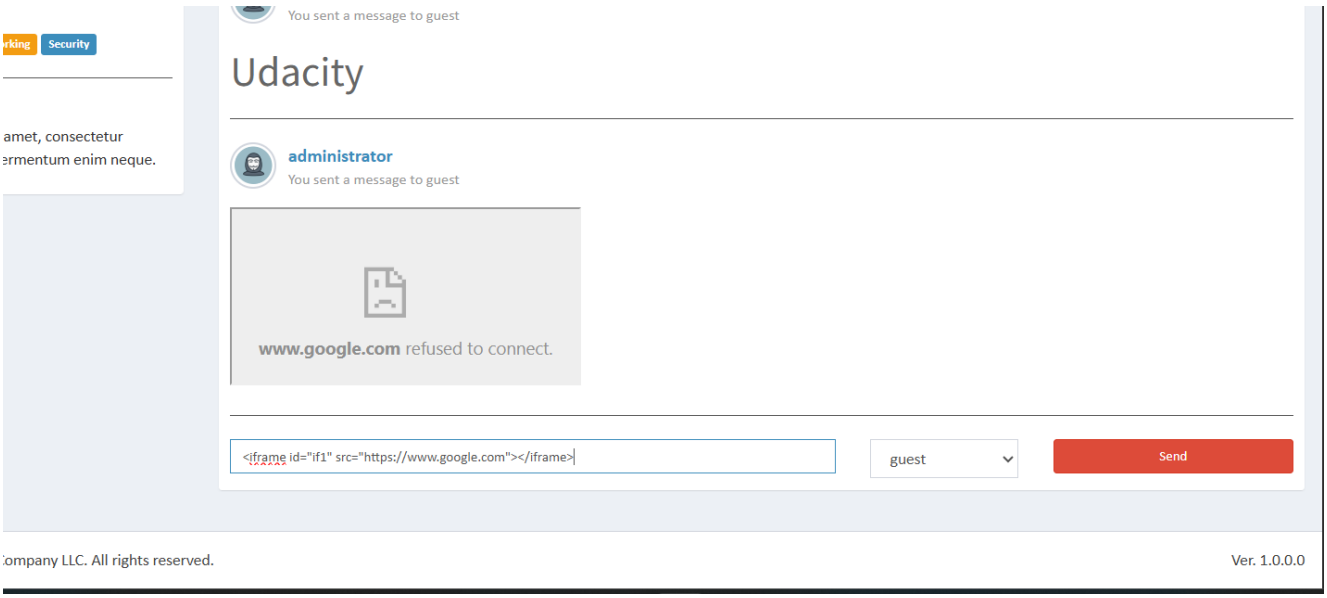
step-by-step walkthrough

- In profile section there is a message box .
- Which is vulnerable to injection vulnerability
- We can inject html codes in it
- And it will execute them
- Here we have executed several codes
- Let's run **<h1>Udacity<h1>**
- The html code will make a header name Udacity





- We can also execute other injection like displaying external website content
- **<iframe id="if1" src="https://www.google.com"></iframe>**
- it will make a frame with the content of external website



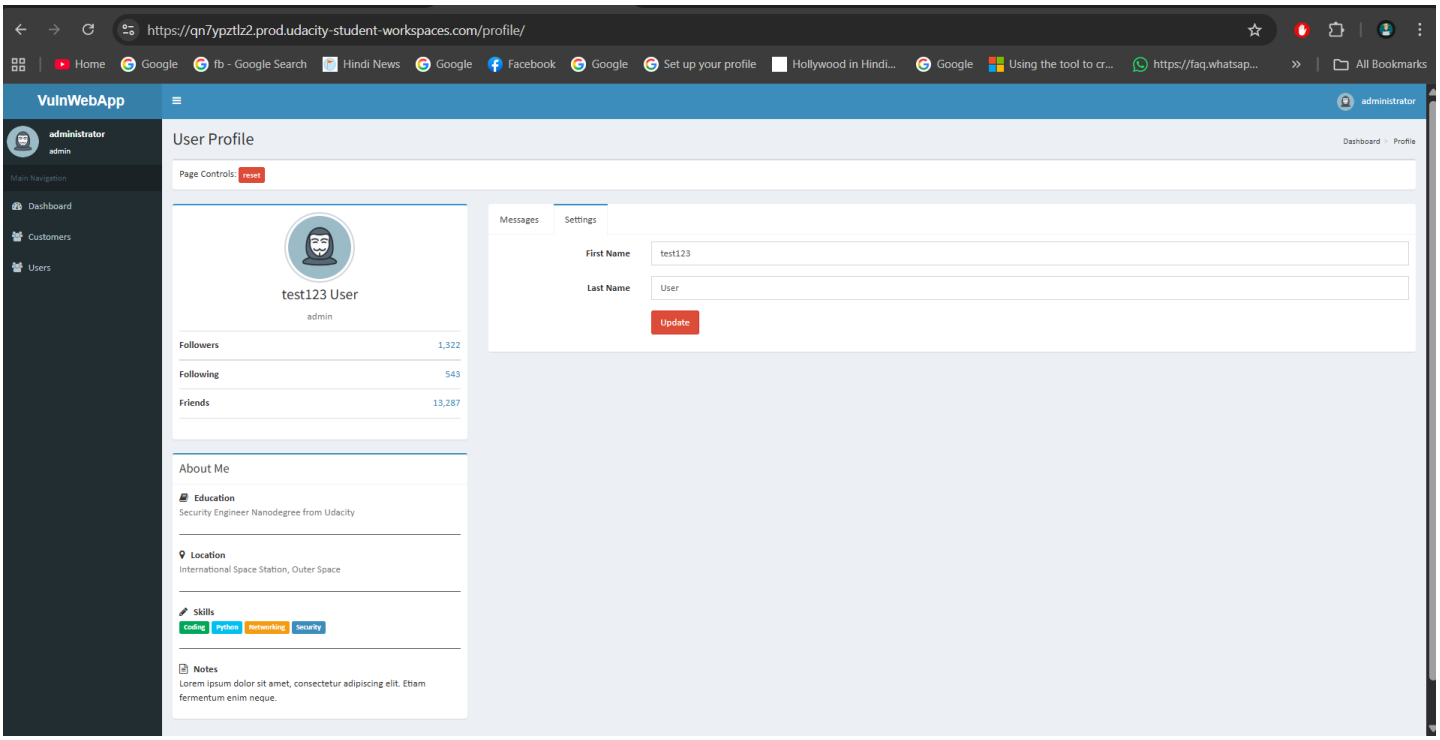
Cross-Site Scripting (Stored XSS)

Severity:	*High*
OWASP TOP 10 reference:	*A03:2021 – Injection (cross site scripting) (Stored)*
Vulnerability Explanation	
The username update filed is vulnerable to stored xss , if the any user update the username to any malicious code the system will update the username to that code on frontend but in backend on admin panel it execute the malicious code, If admin open the users page the code will got executed.	
Remediation recommendation	
Sanitize / escape output — show user input as text, not code and just like convert < to <, > to >. *Validate inputs because it only allow safe characters or whitelisted HTML tags if really needed.* *Add a Content Security Policy (CSP) because it restrict what scripts can run.* *Use a safe templating engine frameworks like React, Angular, or Django auto-escape values.* *First test often try entering <script> in forms to confirm it's blocked.* *Clean the input and escape it before showing it, and boom no more XSS drama.*	

Cross-Site Scripting (XSS)

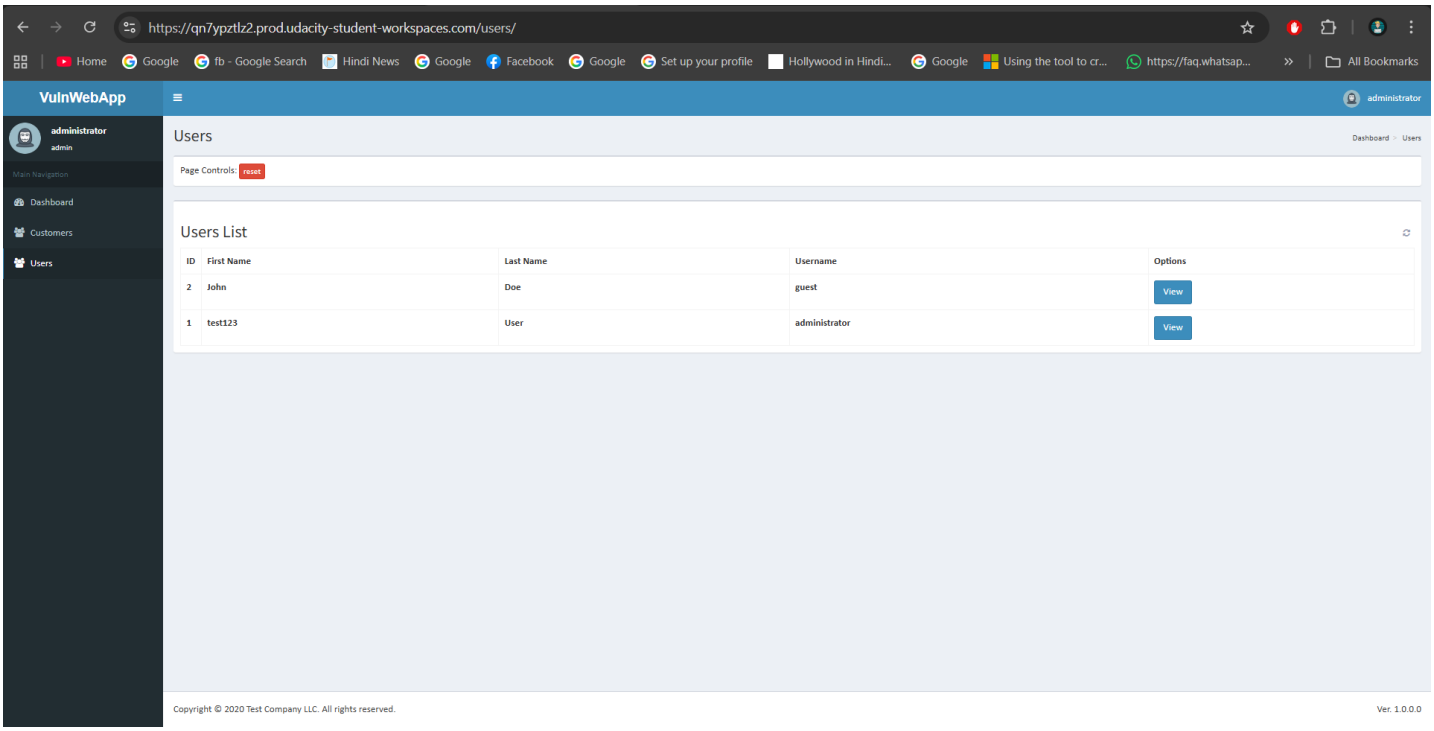
step-by-step walkthrough

- We need admin access here , which we got earlier
- In Profile section there is filed to update's first name and last name

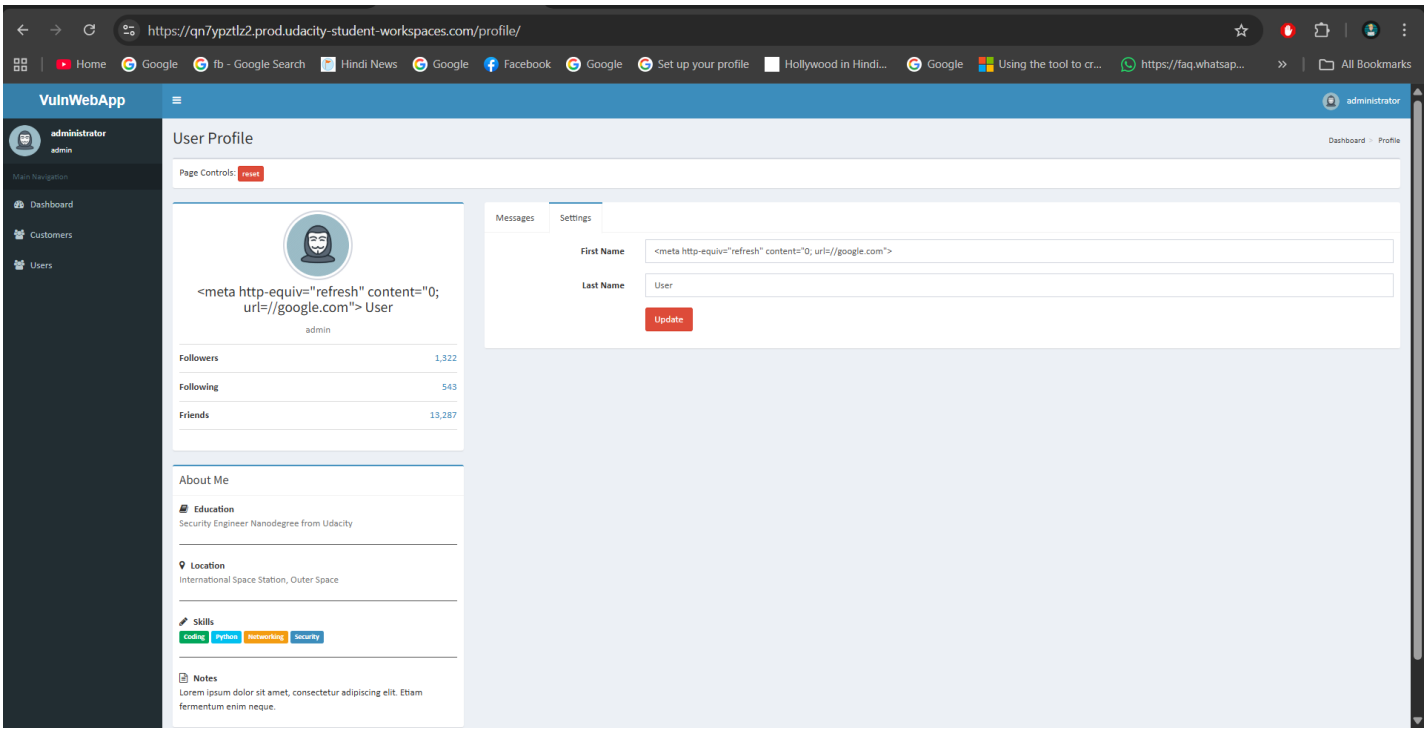


- Try changing username

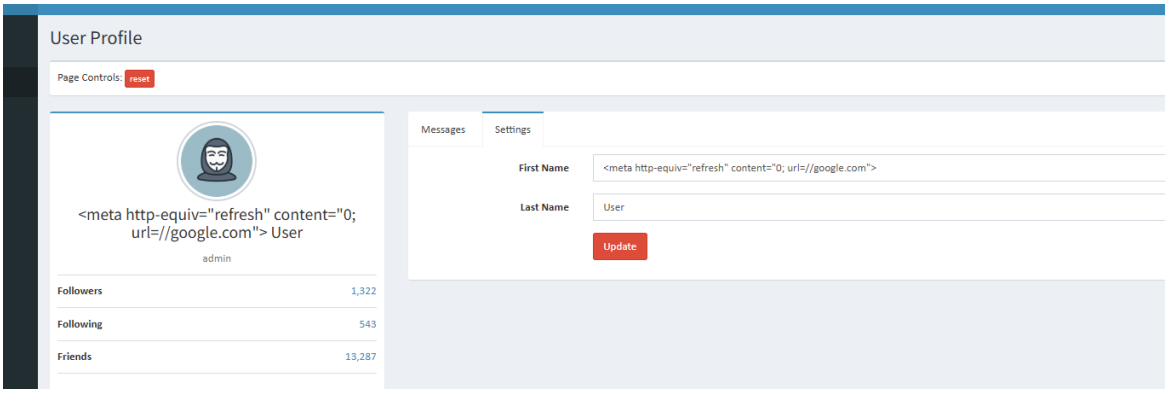
- If we change the username like **test123**
- it will update the username in user list page also

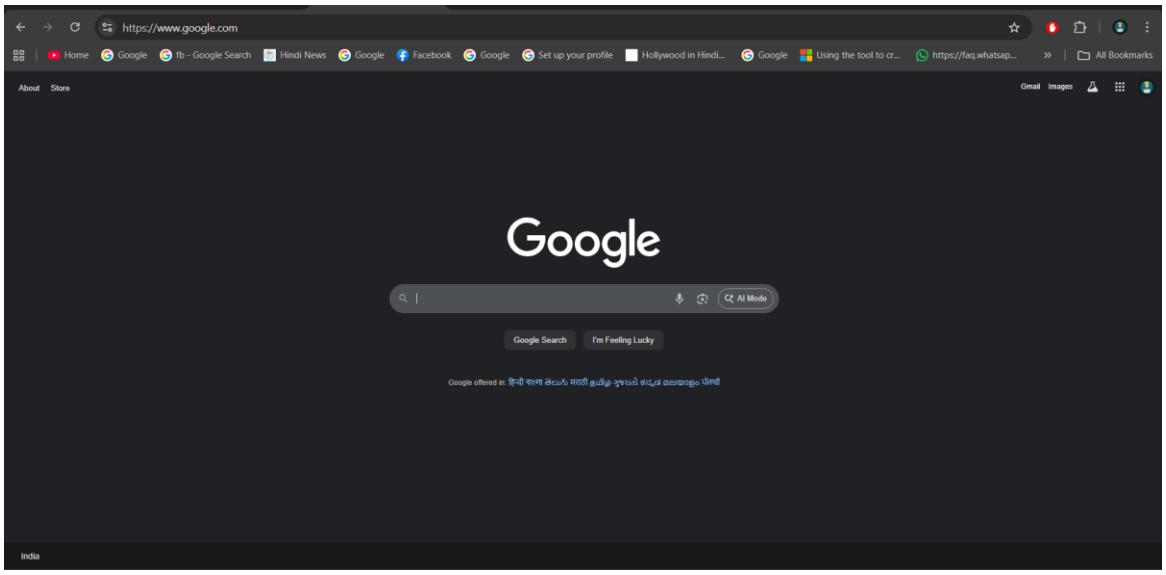


- If we injection any malicious code here it will change and display the username to that
- And upon opening user list it will execute the injected code



- Here we injected the code which will redirect to external page
- **<meta http-equiv="refresh" content="0; url=//google.com">**
- Now username changed to this
- If we go to user list page it will execute the code & redirect us to google.com
- It show the stored XSS vulnerability





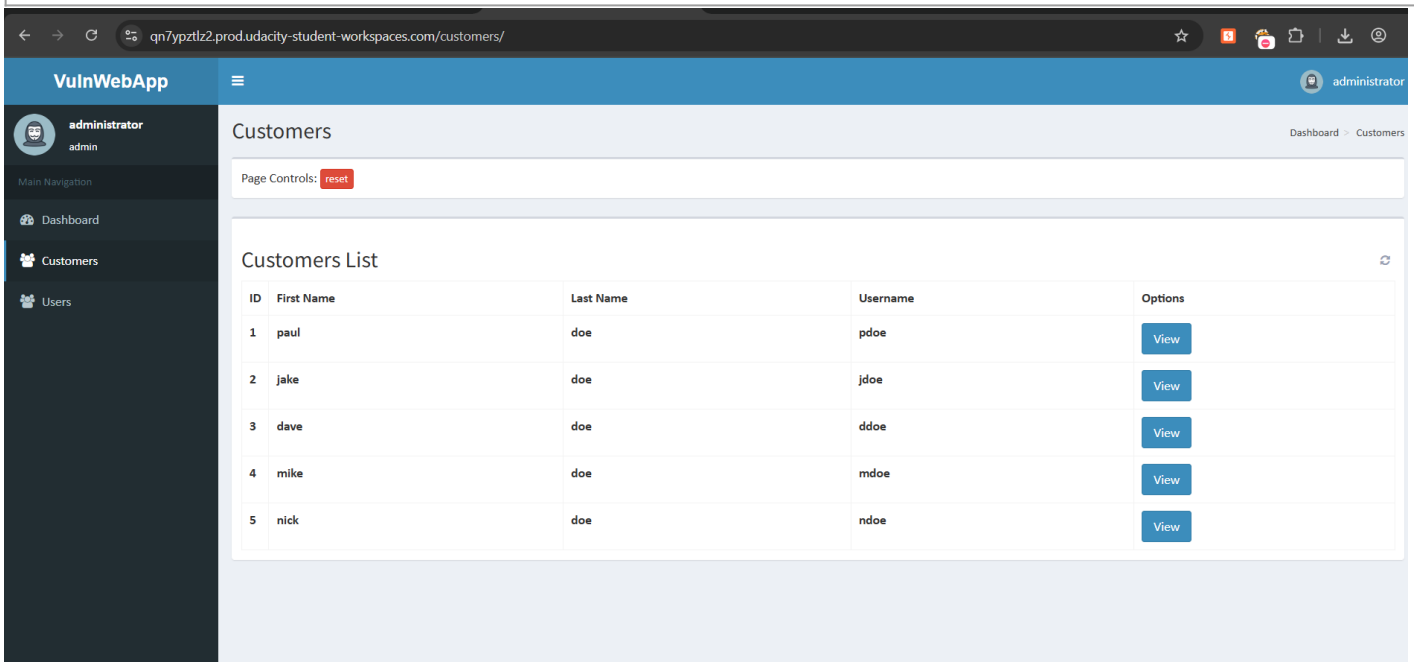
SQLi

Severity:	*Critical*
OWASP TOP 10 reference:	*A03:2021 – Injection*
Vulnerability Explanation	
*In URL **/customers/id/1** , it is vulnerable to SQL injection . Attacker can make a payload which make all condition true to database even if the credential is wrong and it can fetch all the customers data including their personal information their password hashes*	
Remediation recommendation	
To prevent from SQL injection *Use parameterized queries. Validate and filter input from users avoid raw inputs.* *Apply the allowlist, whitelist. Use a strong and secure web application firewall.* *Always limit the database privileges.* *Don't display the errors on the user's side.* *Always monitor logs and deploy automated tools which can create alert when a malicious query is run.*	

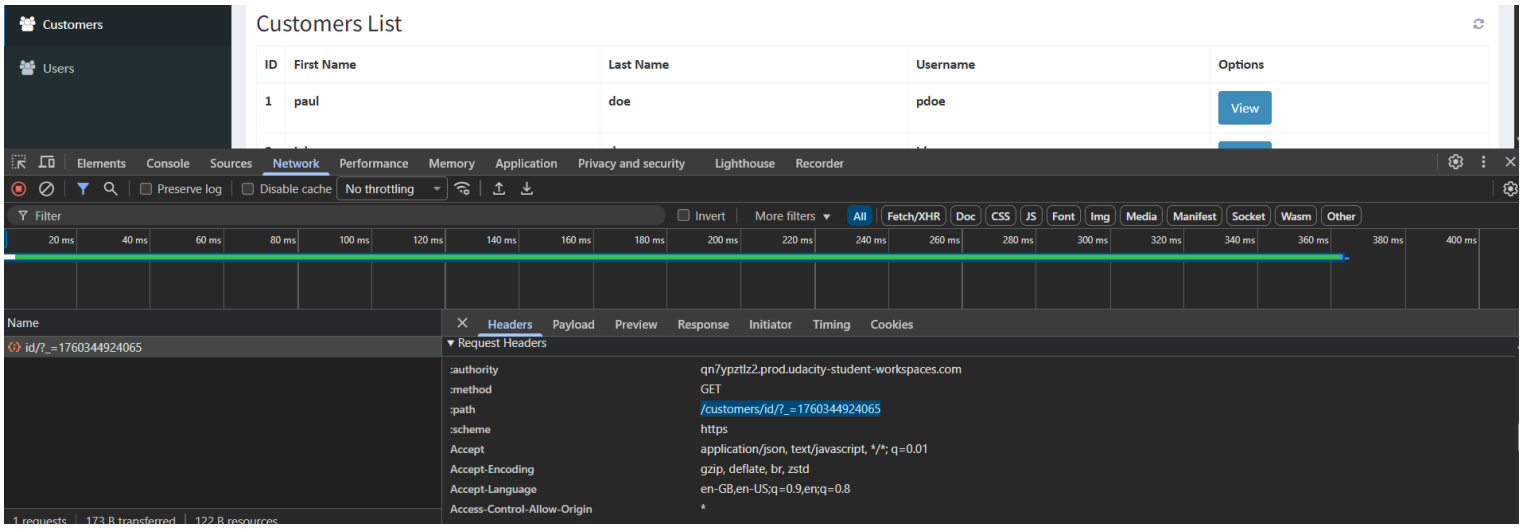
SQLi

step-by-step walkthrough

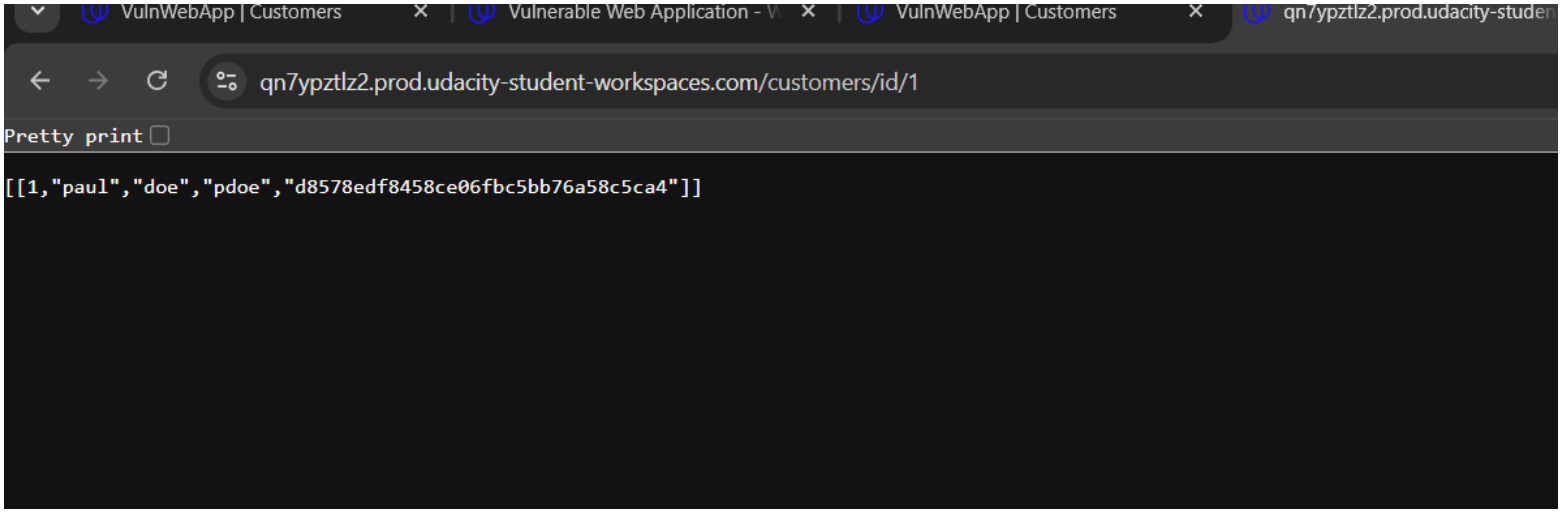
- If we go to page **/customers** , it show list of customers fetched from database



- Now open the developer tool
- Then go to Network tab to see the endpoint request to database



- Now open the api request in new tab
- Which will show the data of customer
- If we hit with the **/id/1** it will show data of customer 1 , But



- If we want to get data of all customers from database
- We have to build a **SQL injection payload**
- Payload worked here is **“1' OR ‘1”**
- By going to URL
- **/customers/id/1’ or ‘1**
- It will trigger the database to a true statement and fetch all customers data