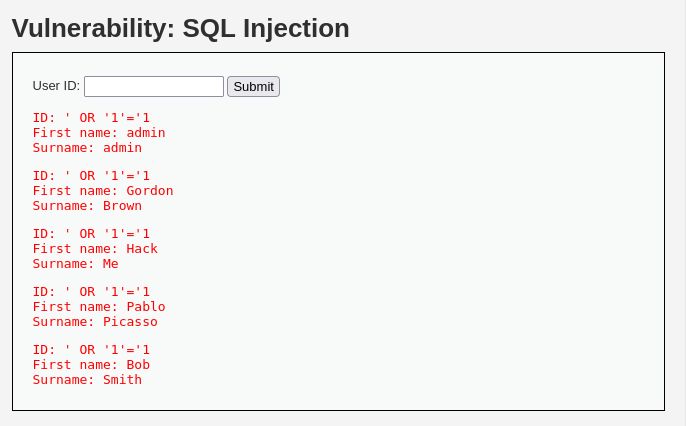
**LAB 7: SQL Injections**

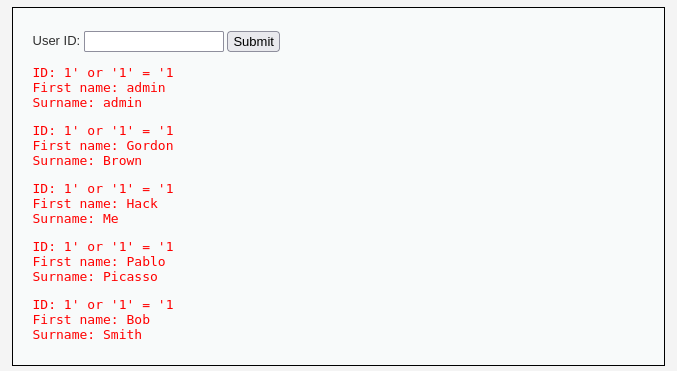
Exercise 7.1: **Identify the various input field on the vulnerable application through which sql queries can be crafted.**

**Input Fields**

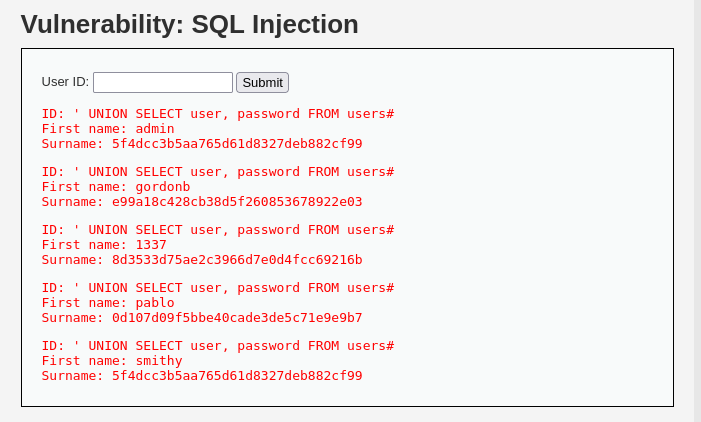
The most common input field for testing SQL injection is the **User ID** field on the SQL Injection page. The vulnerability is often found in the following types of web application inputs:

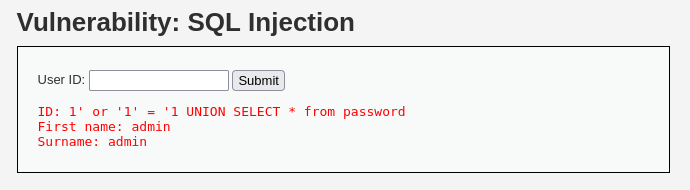
* **Login forms**: This includes fields for username and password. Attackers can use classic payloads like ' OR 1=1 -- to bypass authentication.
* **Search bars**: A search query is a classic example of an input field that might be directly concatenated into a SQL SELECT statement, making it a prime target.
* **User ID or numeric fields**: Even if an input field is meant to accept only numbers, it can still be vulnerable if the application doesn't strictly validate the input type. For example, a user could input ' OR 1=1 into a user ID field.
* **GET parameters in the URL**: In some cases, the input isn't a form field but a parameter in the URL itself (e.g., http://example.com/products.php?id=1). These are also exploitable.



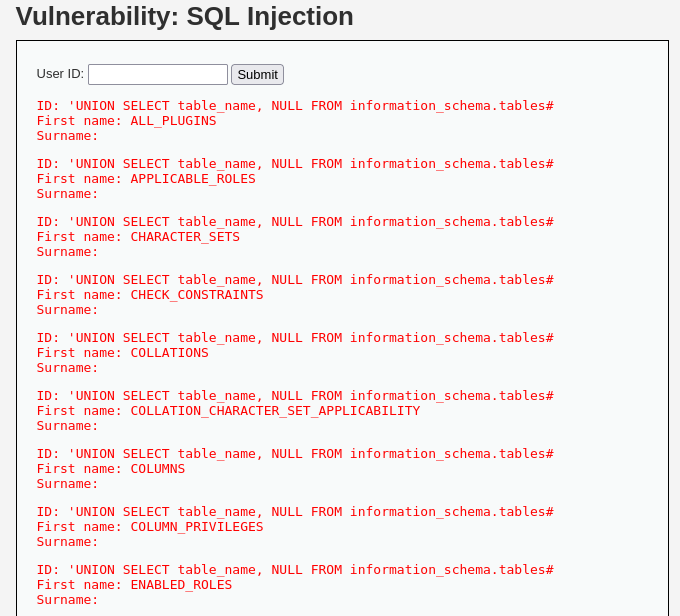


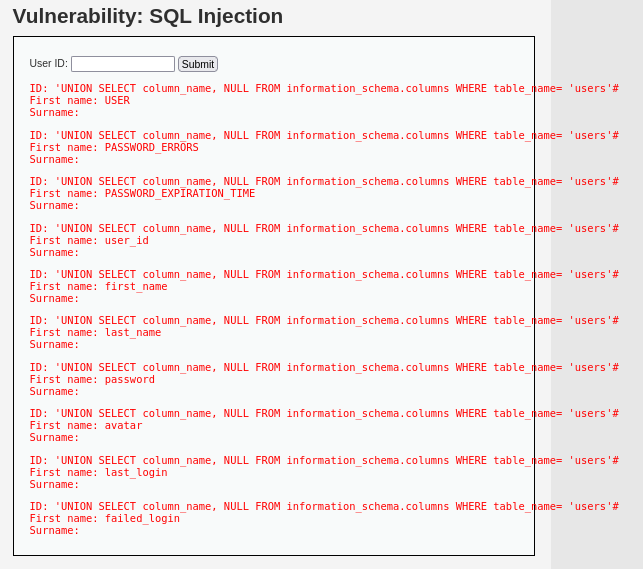
Exercise 7.2: **Get the list of all usernames and password stored in the application using SQL injection**



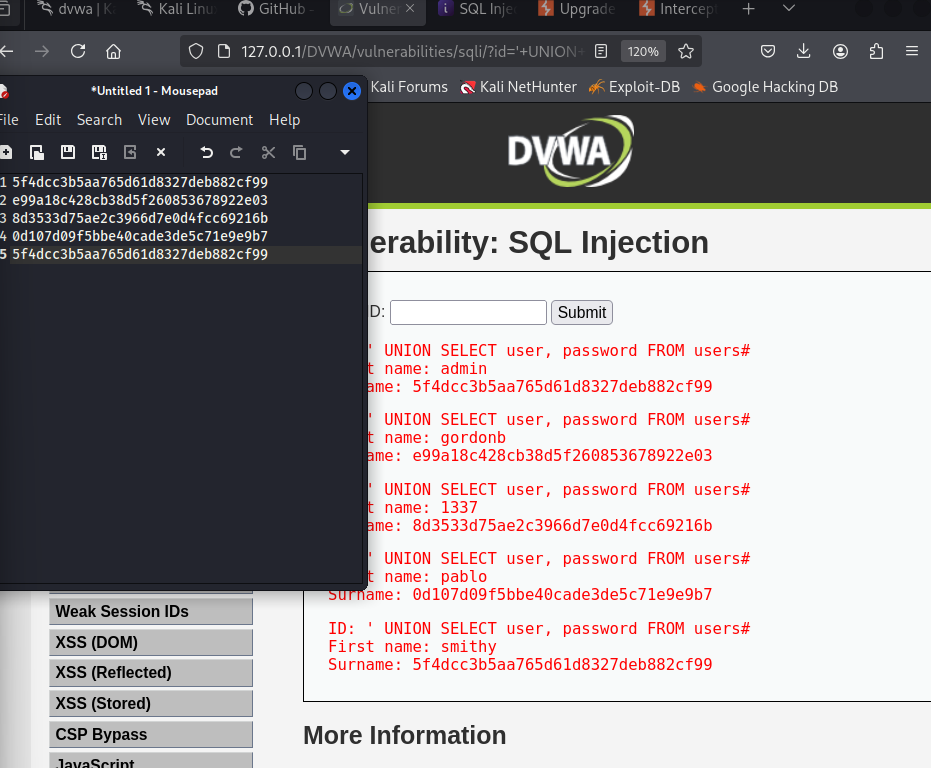


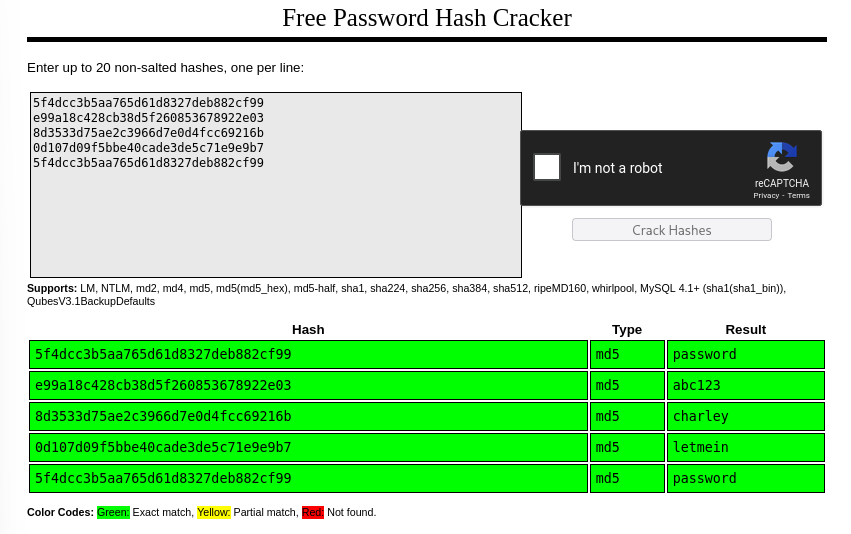
Exercise 7.3: **Get the list of all table names, column names and data stored in it using SQL injection**.

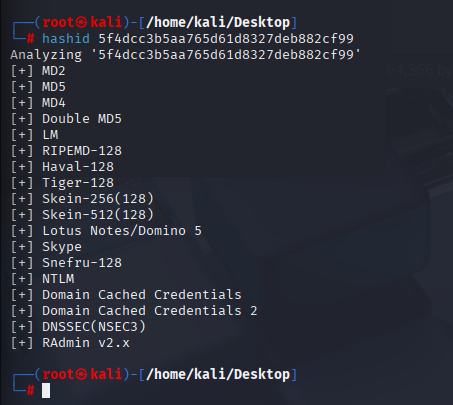


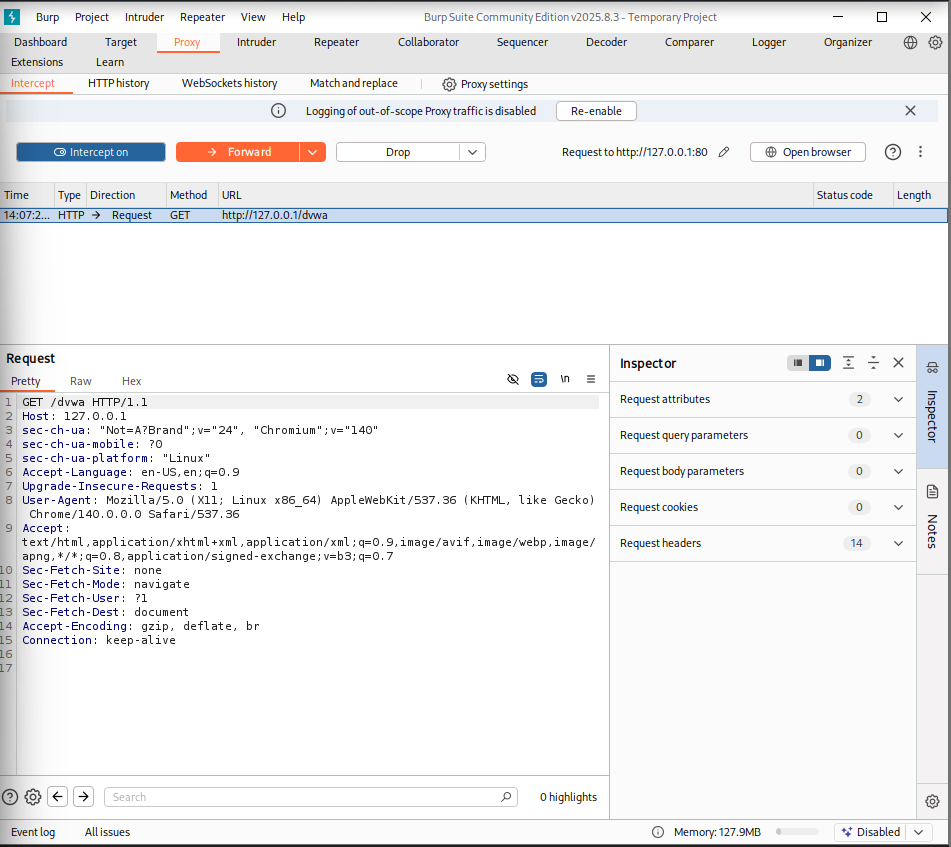


Exercise 7.4: **Store the data thus retrieved in a text file**.

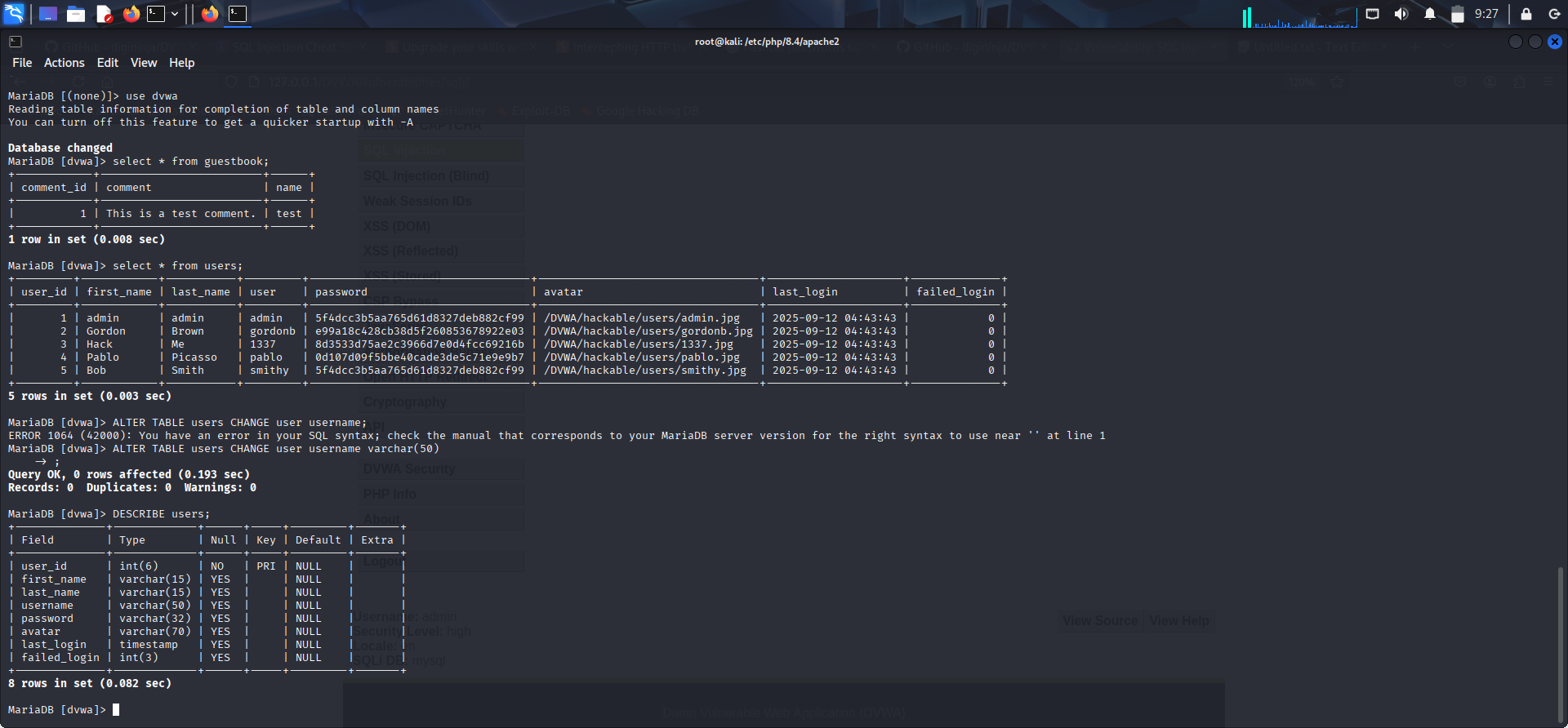




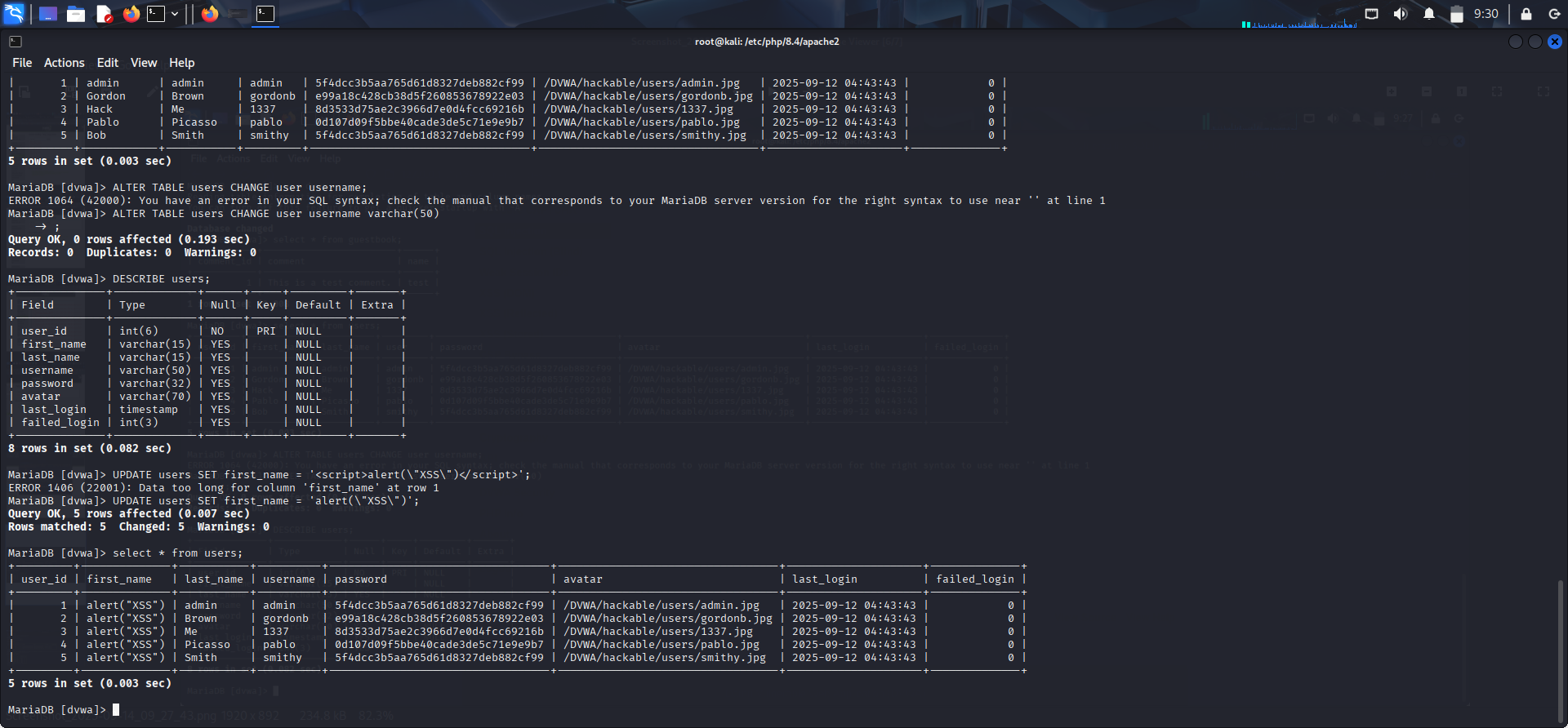




Exercise 7.5: **Modify the tables by renaming a column, deleting a column and adding a column.**



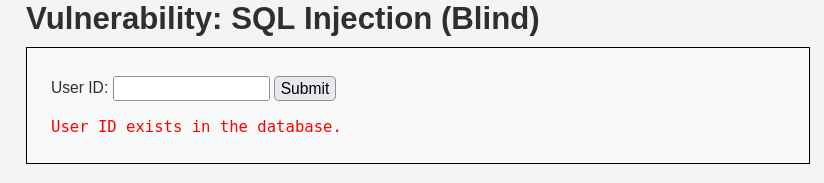
Exercise 7.6: **Tamper the data by updating it with some random garbage or malicious data.**

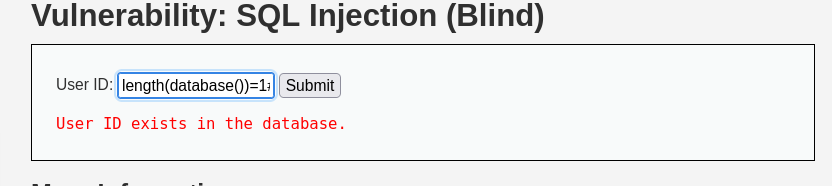


Exercise 7.7: **Delete a table for data loss attack**

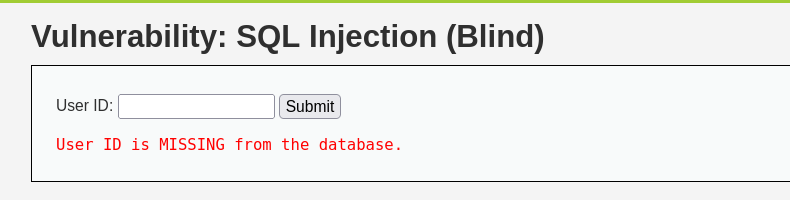
In DVWA, it is not possible to change a column name by entering a value in the userid input because SQL injection here only allows manipulation of data retrieval or update queries, not schema changes.

Exercise 7.8: **Perform Blind SQL injection attack**









Exercise 7.9: **Rewrite the code so as to minimize the SQL injection attacks and validate it through various test cases**.

<?php

if (**isset**($\_SESSION['id'])) {

*// Get input and validate it as an integer (assuming user\_id is numeric)*

$id = $\_SESSION['id'];

if (!**filter\_var**($id, FILTER\_VALIDATE\_INT)) {

die('<pre>Invalid ID format.</pre>');

}

switch ($\_DVWA['SQLI\_DB']) {

case MYSQL:

*// Prepare statement to mitigate SQL Injection*

$stmt = **mysqli\_prepare**($GLOBALS["\_\_\_mysqli\_ston"], "SELECT first\_name, last\_name FROM users WHERE user\_id = ? LIMIT 1;");

if ($stmt === false) {

die('<pre>Database error: Failed to prepare statement.</pre>');

}

**mysqli\_stmt\_bind\_param**($stmt, "i", $id);

**mysqli\_stmt\_execute**($stmt);

**mysqli\_stmt\_bind\_result**($stmt, $first, $last);

if (**mysqli\_stmt\_fetch**($stmt)) {

**echo** "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";

} else {

**echo** "<pre>No user found with ID: {$id}</pre>";

}

**mysqli\_stmt\_close**($stmt);

**mysqli\_close**($GLOBALS["\_\_\_mysqli\_ston"]);

break;

case SQLITE:

global $sqlite\_db\_connection;

*// Prepare statement*

$query = "SELECT first\_name, last\_name FROM users WHERE user\_id = :id LIMIT 1;";

$stmt = $sqlite\_db\_connection->**prepare**($query);

if ($stmt === false) {

die('<pre>Database error: Failed to prepare statement.</pre>');

}

*// Bind parameter*

$stmt->**bindValue**(':id', $id, SQLITE3\_INTEGER);

try {

$results = $stmt->**execute**();

} catch (**Exception** $e) {

**echo** 'Caught exception: ' . $e->**getMessage**();

exit();

}

if ($results) {

$row = $results->**fetchArray**(SQLITE3\_ASSOC);

if ($row) {

$first = $row["first\_name"];

$last  = $row["last\_name"];

**echo** "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";

} else {

**echo** "<pre>No user found with ID: {$id}</pre>";

}

$results->**finalize**();

} else {

**echo** "<pre>Error in fetch: " . $sqlite\_db\_connection->**lastErrorMsg**() . "</pre>";

}

break;

}

}

?>

Enter a valid user ID, like 1.The code should correctly return the first and last name for that user.

Enter a malicious payload like ' OR 1=1 --. The rewritten code will not treat this as a SQL command. Instead, it will look for a user\_id that literally matches the string ' OR 1=1 -- which will return no results, effectively blocking the attack.