PROJECT P0

SQL Injection Attack
On Today's Technologies Using
MYSQL Database Queries

Prepared By,

Deepak

Guided By,

Zakir Hussain

TABLE OF CONTENT

1. Introduction

→ **SQL Injection (SQLi)** is one of the most common forms of cyberattacks that exploit vulnerabilities in web applications. An attacker can execute arbitrary SQL code by inserting it into an application's query fields, such as forms, URL parameters, cookies, or HTTP headers. The success of an SQL Injection attack stems from improper validation of user inputs, enabling attackers to interact with the database and potentially access, manipulate, or delete sensitive information.

→ For instance, when a web application doesn't properly escape user input in an SQL query, an attacker can trick the database into running arbitrary SQL commands, such as retrieving information about users or deleting data. This breach can result in severe consequences, including unauthorized access to sensitive information, loss of data, and significant financial and reputational damage for businesses.

→ SQLi attacks have been around for over two decades, but they remain a severe threat to web applications. This attack method exploits the fact that databases are a central component of most web applications, making them an attractive target for malicious actors. An SQLi can lead to the following types of attacks:

- Unauthorized access to protected data.
- Database modification, including adding, updating, or deleting records.
- Execution of administrative operations on the database.
- Denial of service or complete system compromise.

→ According to the Open Web Application Security Project (OWASP), SQL Injection has consistently ranked as one of the top vulnerabilities in web applications. Even though more advanced security techniques have emerged, many applications still fail to implement basic protection mechanisms. Hence, understanding and preventing SQLi attacks is crucial for anyone working in web development or cybersecurity.

2. Types of SQL Injection Attacks

→ There are several distinct forms of SQL Injection attacks, each with varying levels of complexity and risk. Understanding these types helps you protect your database from potential exploitation. Below are the primary types of SQL Injection attacks:

- **Classic or In-Band SQL Injection**: Classic SQL Injection involves injecting SQL code directly into a web application's input fields (e.g., login forms). This is one of the easiest types of attacks to detect since it exploits the same communication channel used to submit the malicious code. Attackers can directly interact with the database, retrieve data, or make modifications.
  - **Example:**

  ```
  SELECT * FROM users
  WHERE username = 'admin' -- ' AND password = 'password';
  ```

- **Blind SQL Injection**: In this type of attack, the attacker sends SQL queries to the database but does not directly receive data in response. Instead, they infer information based on the behavior of the application (e.g., timing, error messages, or Boolean outcomes). Blind SQL Injection is harder to detect since it doesn't return data immediately, but it can still compromise sensitive information over time.
  - **Example:**

  ```
  SELECT * FROM users WHERE id = 1 AND 1=1;
  SELECT * FROM users WHERE id = 1 AND 1=2;
  ```

- **Time-Based SQL Injection**: Time-based SQL Injection relies on the delay in the database's response to specific queries. An attacker sends SQL code that intentionally delays the database's response if a condition is true. This type of attack is often used when other techniques do not work due to limited feedback from the application.
  - **Example:**

  ```
  SELECT * FROM users WHERE IF (username='admin', SLEEP (5), 0);
  ```

- **Out-of-Band SQL Injection**: Out-of-band SQL Injection sends data via an external channel, such as HTTP or DNS. This type of attack is less common but can be very effective when the database has restricted responses or the attacker needs a separate channel to exfiltrate data.
  - **Example:**

  The attacker may use a DNS-based approach to send sensitive data to a remote server.

→ SQL Injection attacks have been a critical risk factor for years due to the nature of web applications relying on databases. As technologies and applications evolve, SQL Injection methods have grown more complex, but so have the prevention measures.

3. Real-Time SQL Injection Attack Example Using MySQL

→ To demonstrate the risk and impact of SQL Injection, let’s go through an example scenario using MySQL. Assume we are working on a web application with a login form that allows users to enter their username and password to access their accounts. The SQL query used in the backend might look something like this:

▪ SELECT * FROM users WHERE username = '$username' AND password = '$password';

→ If the web application doesn’t properly sanitize the input, an attacker can exploit this by inputting the following into the username field:

▪ admin' --

→ The query now becomes:

▪ SELECT * FROM users WHERE username = 'admin' -- ' AND password = '';

→ This modified query will bypass the password check, granting the attacker access as the "admin" user.

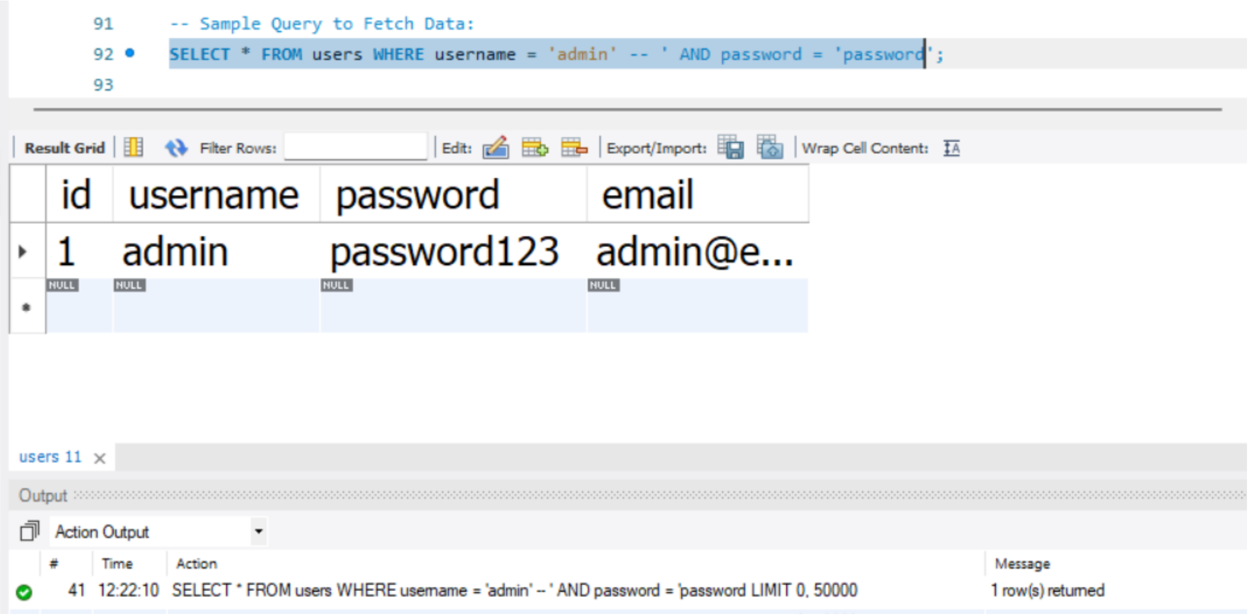
→ Next, we will explore a real-time scenario where we create a database structure and insert data to perform SQL Injection in a controlled environment. Using MySQL Workbench, we will test these queries to demonstrate the effectiveness of the attack and potential solutions for mitigating it.

→ You can create databases, tables, and insert the following data:

→ Database: User_Authentication
→ Tables: users, user_roles, login_attempts, user_sessions, password_resets.

- **Sample Query to Fetch Data**:

- SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password';



```
-- Sample Query to Fetch Data:
SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password';
```

	id	username	password	email
▸	1	admin	password123	admin@e...
*	NULL	NULL	NULL	NULL

	#	Time	Action	Message
✓	41	12:22:10	SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password LIMIT 0, 50000	1 row(s) returned

4. Preventing SQL Injection Attacks

→ Preventing SQL Injection attacks is crucial for safeguarding your web applications and protecting your databases. There are several best practices and techniques you can employ to mitigate SQLi risks:

1. **Use Prepared Statements**: Prepared statements ensure that SQL code and data are separated. The SQL query is defined first, and then user inputs are passed separately, preventing direct manipulation of the SQL query.
   - Example: SELECT * FROM users WHERE username = ? AND password = ?;

2. **Parameterized Queries**: Parameterized queries work similarly to prepared statements and prevent SQL Injection by ensuring user input is treated strictly as data and not executable code.

3. **Input Validation and Sanitization**: Input validation ensures that only valid data is accepted from users. Implement strict validation rules for input fields, such as using regex patterns to filter out potentially dangerous characters like ';--.

4. **Use ORMs (Object-Relational Mappers)**: Using ORM frameworks such as Hibernate or SQLAlchemy can abstract away manual SQL query building, thereby reducing the likelihood of SQL Injection.

5. **Escaping Special Characters**: If you're unable to use prepared statements, you must escape special characters (e.g., quotes and semicolons) in the user input to prevent them from altering the SQL query.

6. **Database Privilege Management**: Ensure that database accounts used by your application have the least privileges necessary. Avoid giving the web application's database user excessive permissions, such as full administrative access.

→ By following these strategies, you can significantly reduce the risk of SQL Injection attacks on your web applications.

5. Database Structure and Setup

→ In this section, we will define the structure of a simple MySQL database, which can be used to demonstrate SQL Injection vulnerabilities. The database setup includes a table named users that holds user credentials, and we will create a few additional tables to simulate a real-world scenario.

→ **Database Creation**

→ First, we will create the database and necessary tables. You can use MySQL Workbench or the MySQL command line to execute these queries.

- CREATE DATABASE User_Authentication;

→ **Table: users**

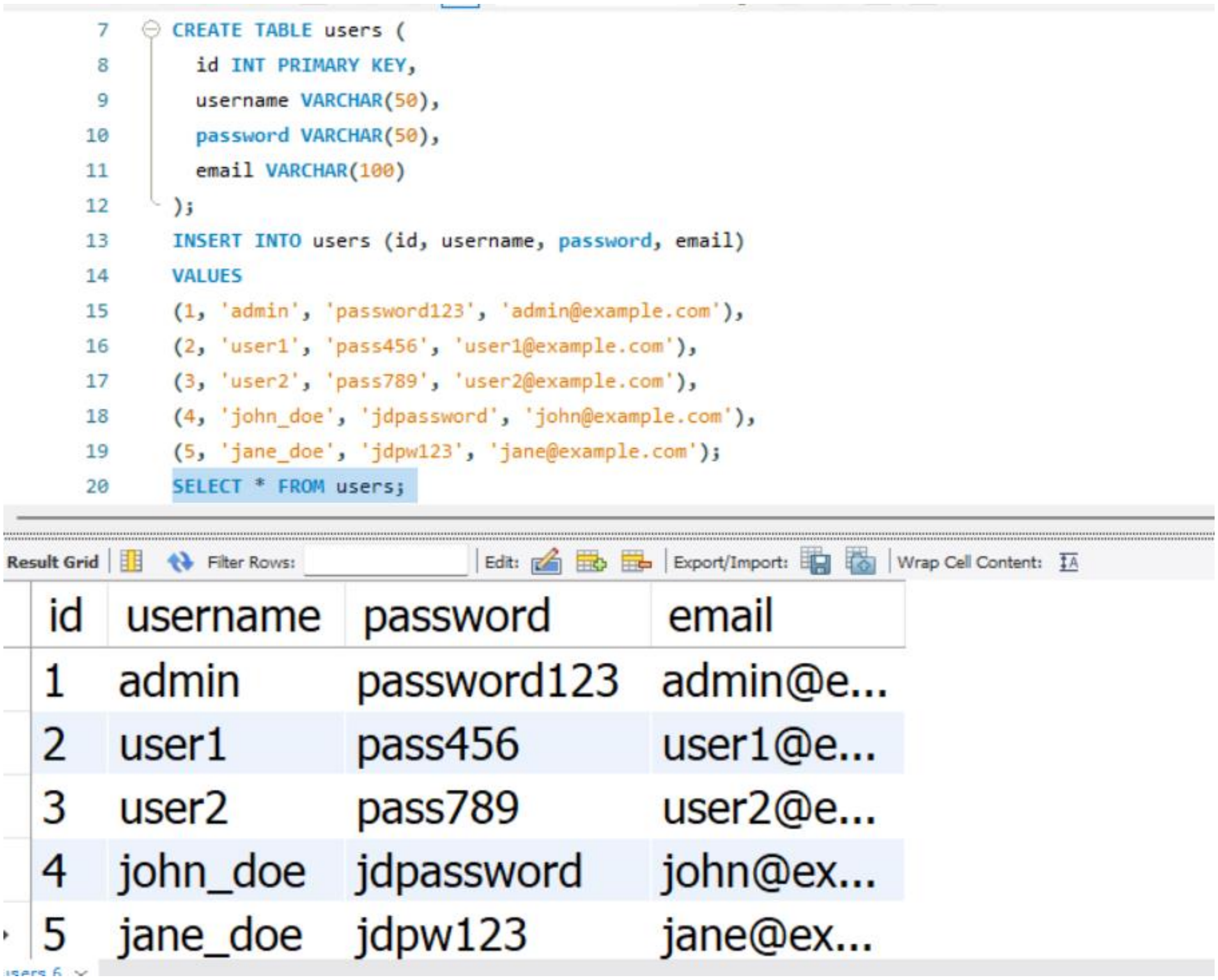
→ This table will store the user information such as id, username, password, and role. It's a simplified example to demonstrate how SQL Injection works with login forms.

-
  ```
  CREATE TABLE users (
      id INT PRIMARY KEY,
      username VARCHAR (50),
      password VARCHAR (50),
      email VARCHAR (100)
      );
  ```

→ **Inserting Sample Data**

→ To populate the users table, we will insert a few records that represent common users and an admin account. This will allow us to test SQL Injection techniques.

-
  ```
  INSERT INTO users (id, username, password, email)
  VALUES
  (1, 'admin', 'password123', 'admin@example.com'),
  (2, 'user1', 'pass456', 'user1@example.com'),
  (3, 'user2', 'pass789', 'user2@example.com'),
  (4, 'john_doe', 'jdpassword', 'john@example.com'),
  (5, 'jane_doe', 'jdpw123', 'jane@example.com');
  ```



```
CREATE TABLE users (
  id INT PRIMARY KEY,
  username VARCHAR(50),
  password VARCHAR(50),
  email VARCHAR(100)
);
INSERT INTO users (id, username, password, email)
VALUES
(1, 'admin', 'password123', 'admin@example.com'),
(2, 'user1', 'pass456', 'user1@example.com'),
(3, 'user2', 'pass789', 'user2@example.com'),
(4, 'john_doe', 'jdpassword', 'john@example.com'),
(5, 'jane_doe', 'jdpw123', 'jane@example.com');
SELECT * FROM users;
```

id	username	password	email
1	admin	password123	admin@e...
2	user1	pass456	user1@e...
3	user2	pass789	user2@e...
4	john_doe	jdpassword	john@ex...
5	jane_doe	jdpw123	jane@ex...

→ This simple setup allows us to create a vulnerable login form for SQL Injection testing. We will later simulate different SQL Injection techniques on this data to extract information.

→ **Additional Tables (Optional)**

→ To simulate a more complex application, we can add extra tables such as user_roles, login_attempts, and user_sessions

→ **Table: user_roles**

→ This table maps users to specific roles, adding another layer of complexity that attackers could exploit in a real application.

-
  ```
  CREATE TABLE user_roles (
    id INT PRIMARY KEY,
    role_name  VARCHAR(50),
    description VARCHAR(100)
  );
  ```

-
  ```
  INSERT INTO user_roles (id, role_name, description)
  VALUES
  (1, 'Admin', 'Administrator role with full access'),
  (2, 'User', 'Regular user with limited access'),
  (3, 'Moderator', 'Can moderate content'),
  (4, 'Editor', 'Can edit content'),
  (5, 'Guest', 'Guest user with minimal access');
  ```

→ **Table: login_attempts**

→ Tracks the number of logins attempts for each user, which can also be manipulated using SQL Injection.

-
  ```
  CREATE TABLE login_attempts (
    id INT PRIMARY KEY,
    username VARCHAR(50),
    attempt_date DATETIME,
    success BOOLEAN
  );
  ```

-
  ```
  INSERT INTO login_attempts (id, username, attempt_date, success)
  VALUES
  (1, 'admin', '2024-09-14 08:30:00', TRUE),
  (2, 'user1', '2024-09-14 09:15:00', FALSE),
  (3, 'john_doe', '2024-09-14 10:00:00', TRUE),
  (4, 'user2', '2024-09-14 11:45:00', FALSE);
  ```

6. Example Queries for SQL Injection Testing

→ Now that we have a functional database, we can move on to testing SQL Injection. Below are some example queries that demonstrate how attackers might exploit vulnerabilities in a poorly secured application.

1. Simple SQL Injection:

→ In a vulnerable application, attackers could manipulate input fields to alter the SQL query. For example, consider the following login query:

- SELECT * FROM users WHERE username = '$username' AND password = '$password';

→ If a user enters the following input into the username field:

- admin' --

→ The query becomes:

- SELECT * FROM users WHERE username = 'admin' -- ' AND password = '';

→ This query effectively bypasses the password check by commenting out the rest of the SQL code. As a result, the attacker is logged in as the admin user without knowing the actual password.

2. Union-based SQL Injection:

→ Union-based SQL Injection is another attack method where an attacker can retrieve data from other tables by combining multiple SQL queries using the UNION operator.

- Example:
  SELECT username, password FROM users WHERE username = 'admin' UNION SELECT database(), version();

→ This query retrieves not only the username and password of the admin user but also information about the current database and MySQL version.

3. **Error-Based SQL Injection:**

→ Attackers can exploit SQL errors to gain information about the database structure. For instance, they might intentionally input incorrect queries to trigger an error message that reveals useful data.

- Example:
  SELECT * FROM users WHERE id = 1' AND 1=2;

→ By submitting this malformed query, the attacker can observe the database's response, which may leak internal information such as table names or field structures.

4. **Blind SQL Injection:**

→ In cases where the database does not return visible errors or data, attackers may rely on Blind SQL Injection techniques. This involves sending queries that change the application's behavior without displaying results.

- Example:
  SELECT * FROM users WHERE username = 'admin' AND IF(1=1, SLEEP(5), 0);

→ If the query causes a delay in the response, the attacker knows that the condition was true, allowing them to extract data incrementally.

7. Execution and Results

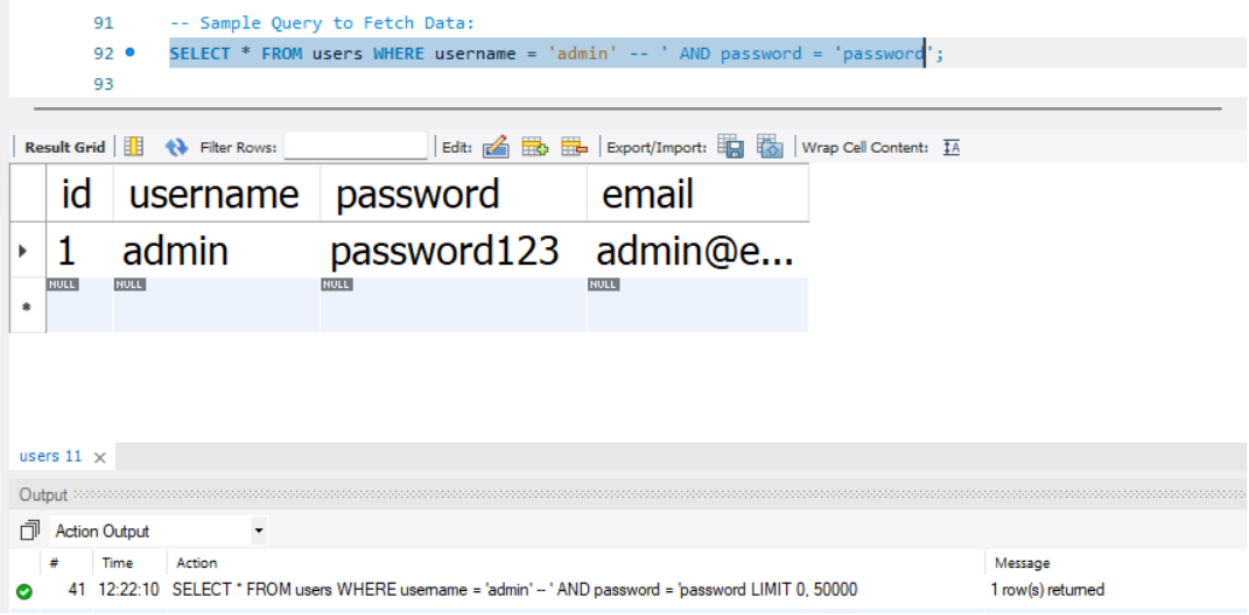
→ In this section, we will execute the SQL Injection queries described above and analyze the results. Using MySQL Workbench, we will run each query and observe how the database responds to different forms of SQL Injection.

1. Simple SQL Injection Results:

→ After executing the simple SQL Injection query:

- SELECT * FROM users WHERE username = 'admin' -- ' AND password = '';

→ You will notice that the query logs in as admin without needing the actual password. This confirms that the application is vulnerable to SQL Injection.



```
-- Sample Query to Fetch Data:
SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password';
```

id	username	password	email
1	admin	password123	admin@e...
NULL	NULL	NULL	NULL

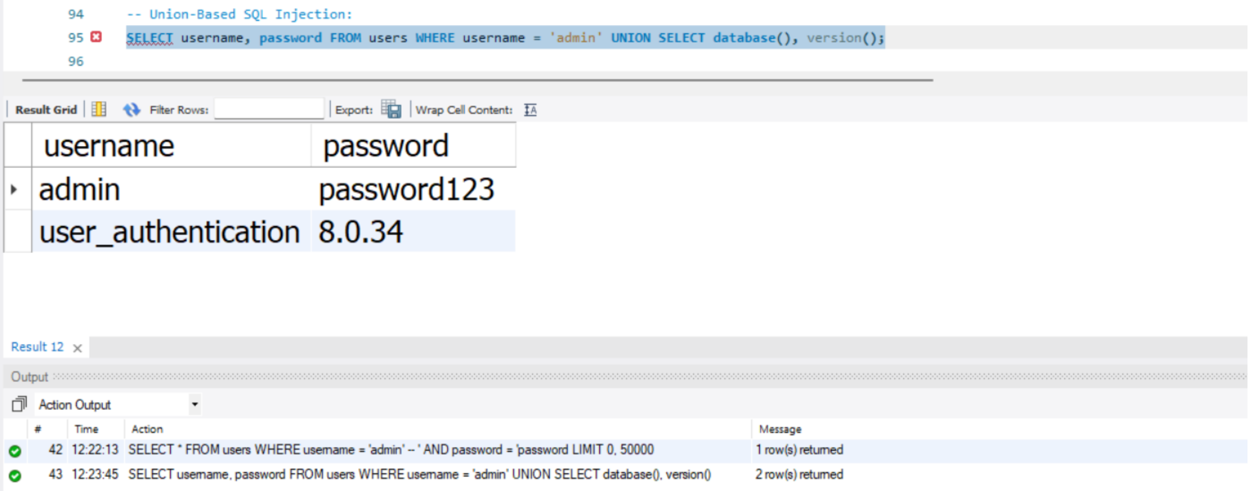
#	Time	Action	Message
41	12:22:10	SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password LIMIT 0, 50000	1 row(s) returned

2. Union-Based Injection Results:

→ The union-based SQL Injection query:

- SELECT username, password FROM users WHERE username = 'admin' UNION SELECT database(), version();

→ should return the username and password of the admin user, as well as information about the database and MySQL version.



```
-- Union-Based SQL Injection:
SELECT username, password FROM users WHERE username = 'admin' UNION SELECT database(), version();
```

username	password
admin	password123
user_authentication	8.0.34

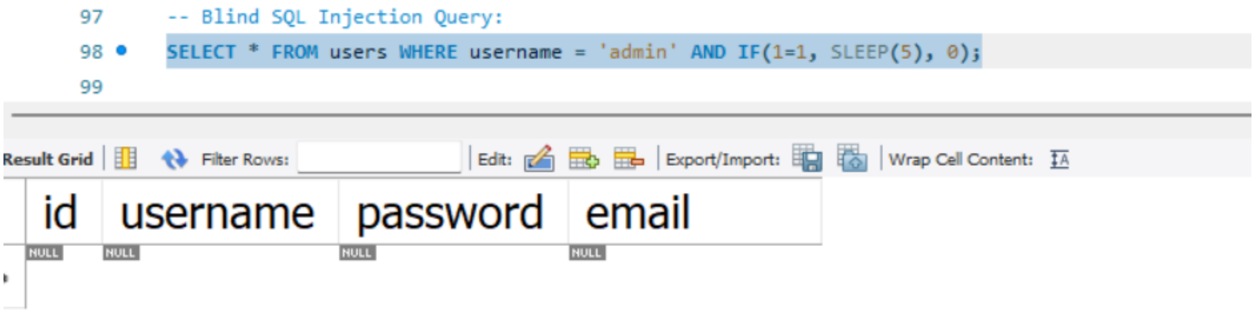
#	Time	Action	Message
42	12:22:13	SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password LIMIT 0, 50000	1 row(s) returned
43	12:23:45	SELECT username, password FROM users WHERE username = 'admin' UNION SELECT database(), version()	2 row(s) returned

3. Blind SQL Injection Results:

→ When executing the blind SQL Injection query:

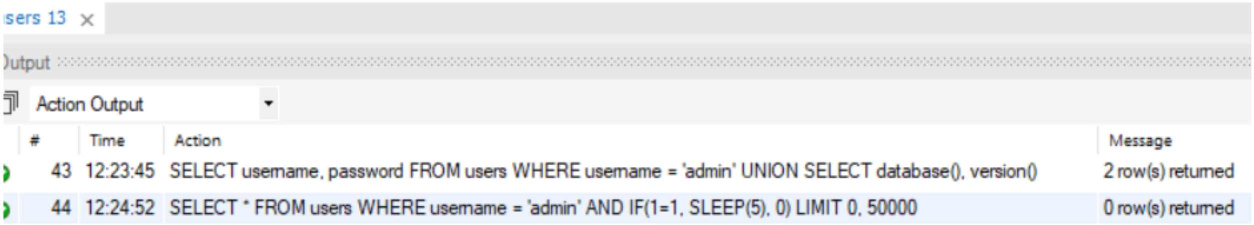
- SELECT * FROM users WHERE username = 'admin' AND IF(1=1, SLEEP(5), 0);

→ you will observe that the query causes a delay in the application's response. This proves that the attacker can infer information from the database without directly seeing query results.



```
-- Blind SQL Injection Query:
SELECT * FROM users WHERE username = 'admin' AND IF(1=1, SLEEP(5), 0);
```

id	username	password	email
NULL	NULL	NULL	NULL



```
#   Time      Action                                                                        Message
43  12:23:45  SELECT username, password FROM users WHERE username = 'admin' UNION SELECT database(), version()  2 row(s) returned
44  12:24:52  SELECT * FROM users WHERE username = 'admin' AND IF(1=1, SLEEP(5), 0) LIMIT 0, 50000           0 row(s) returned
```

4. Error-Based SQL Injection Results:

→ When running the error-based query:

```
SELECT * FROM users WHERE id = 1' AND 1=2;
```

→ you might receive an error message revealing the internal structure of the database.

8. Conclusion

SQL Injection continues to be one of the most dangerous and prevalent vulnerabilities in web applications today. As demonstrated in this project, SQL Injection exploits weaknesses in user input validation, allowing attackers to execute unauthorized SQL queries. The consequences of a successful SQL Injection attack can be devastating, ranging from data theft to complete system compromise.

→ From the examples and techniques outlined in this project, it's clear that even a simple application can be vulnerable if proper security measures are not taken. We showed how common SQL Injection methods, such as union-based and error based attacks, can easily compromise a database.

→ The project also highlighted the importance of preventive measures, such as:

- Using prepared statements and parameterized queries to ensure that user inputs are treated as data rather than executable SQL code.

- Implementing input validation and escaping special characters to reduce the risk of SQL Injection.

- Limiting database permissions to minimize the impact of an attack if it occurs.

→ As web applications continue to evolve, it's essential for developers and database administrators to stay vigilant in applying security best practices. SQL Injection vulnerabilities can often be detected and mitigated with tools like automated vulnerability scanners and code reviews. Moving forward, organizations must prioritize security in their development workflows to protect their applications and data from these increasingly sophisticated threats.