SQL Injection

Emanuele Storci

Indice

1 Project Introduction

The project represents a simulation of a web application login page. The objective is to demonstrate possible violations of the fundamental principles of the CIA Triad.

1.1 Attack Types

The goal is to compromise **Confidentiality** by accessing the web application through authentication bypass, even without possessing valid credentials. Subsequently, we demonstrate possible impacts on **Integrity** by acting on accessible data, with the ability to modify and manipulate it. Finally, we compromise **Availability** by deleting the users table from the database, effectively rendering the system unusable. At the end of the attack, the server is no longer operational.

1.2 Operational Context of the Attack

In this project, the web application is implemented using an architecture composed of:

- a **PHP backend** that handles APIs and the web interface, exposed via **Apache**;
- a **MySQL database**, used for storing users and transactions.

The entire environment is managed using **Docker Compose**, in order to ensure ease of execution across multiple machines. In particular, two separate containers are used: one dedicated to the PHP web server, running on port `80`, and one for the MySQL database, configured on port `3306`.

The application contains vulnerabilities related to unsafe server-side SQL query handling, which can be exploited to perform a **SQL Injection** attack. The attack is carried out by forcing malicious interactions with the PHP login endpoint and the user management APIs, bypassing authentication and gaining access to sensitive data.

2 Code Analysis

The project is a web application developed in PHP and managed using Docker, with a simple backend and a MySQL database initialized through SQL scripts. The purpose is to demonstrate a login system and transaction management linked to registered users, with the ability to visualize user roles and performed operations.

2.1 Project Structure

- **index.php**
  This file handles the main application logic:
  - Checks for the existence of the database file:
    - If it exists, a connection is established
    - Otherwise, the database is created and initialized with sample data using the `init.sql` script
  - Handles `POST` requests to the `/login` endpoint, enabling CORS for the `POST` and `OPTIONS` methods
  - Receives `username` and `password` from the request and passes them to the authentication logic
  - Passwords are not hashed; secure password handling is assumed in a real-world scenario
- **ruoli.php**
  Allows visualization of the role associated with the authenticated user (such as `admin` or `user`)
- **utenti.php**
  Manages user-related operations. The `utenti` table is created using the `init.sql` script:

```
CREATE TABLE utenti (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ruolo VARCHAR(255),
    nome VARCHAR(255),
    pass VARCHAR(255)
);

INSERT INTO utenti (ruolo, nome, pass) VALUES
```

```
('admin', 'emanuele', '123'),
('user', 'alessio', '000'),
('user', 'flaminia', 'ciao'),
('user', 'giuseppe', 'roma');
```

- **transazioni.php**
  Displays the transactions associated with users. The `transazioni` table is created using the `transazioni.sql` script:

```
CREATE TABLE transazioni (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_utente INT,
    importo DECIMAL(10,2),
    carta VARCHAR(255),
    data_operazione DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_utente) REFERENCES utenti(id)
);

INSERT INTO transazioni (id_utente, importo, carta, data_operazione) VALUES
(1, 120.50, '4539 1488 0343 6467', '2025-04-10'),
(2, 89.99, '5500 0000 0000 0004', '2025-02-20'),
(3, 250.00, '3400 0000 0000 009', '2025-01-02'),
(1, 50.00, '4539 1488 0343 6467', '2025-05-08'),
(4, 300.75, '6011 0000 0000 0004', '2025-03-14');
```

- **docker-compose.yml**
  Manages the development environment using Docker containers and starts the PHP and MySQL services, ensuring data persistence and accessibility through the local browser

3 Attack Simulation

3.1 Attack Objective

The objective of this attack simulation is to demonstrate how an SQL Injection attack can compromise the three fundamental principles of information security: **Confidentiality**, **Integrity**, and **Availability** (the CIA Triad), using a simple login page as the entry point for the attack.

In this case, the application is written in PHP, and the login logic builds the SQL query by concatenating user-provided input values:

```
$sql = "SELECT * FROM utenti WHERE nome = '$username' AND pass = '$password'";
```

This approach exposes the system to critical security vulnerabilities.

3.2 Expected Behavior

The application successfully performs a login when the provided credentials match a user stored in the `utenti` table. For example, by accessing the system with:

- `username: emanuele`
- `password: 123`

the user is authenticated correctly and can access restricted functionality.

3.3 Attack Using a `Tautology` Payload

It is possible to bypass authentication without knowing valid credentials by exploiting a logical **tautology**, injecting the following payload into the `username` field:

```
' OR 1=1 --
```

The resulting SQL query becomes:

```
SELECT * FROM utenti WHERE nome = '' OR 1=1 --' AND pass = ''
```

The condition `1=1` is always true, and the `--` comment truncates the remainder of the query, effectively bypassing the password check. This results in unauthorized access and compromises the **Confidentiality** of the system.

3.4 Attack Using an `EOL Comment`

If a valid `username` is known (e.g., `flaminia`), it is also possible to bypass the password check using the following payload:

```
flaminia' --
```

The resulting SQL query is:

```
SELECT * FROM utenti WHERE nome = 'flaminia' --' AND pass = ''
```

This allows direct access as an existing user. Both **Confidentiality** and **Integrity** are compromised, as the attacker can access sensitive data and potentially modify it.

3.5 Piggyback Query Attack

Finally, a destructive attack can be performed by concatenating a second SQL statement:

```
'; DROP TABLE utenti --
```

The resulting query becomes:

```
SELECT * FROM utenti WHERE nome = ''; DROP TABLE utenti --' AND pass = ''
```

This attack deletes the entire `utenti` table, making any future login impossible. As a result, the **Availability** of the system is compromised.

4 Final Conclusions

These examples demonstrate how unsafe string concatenation in an SQL query can have devastating effects on a system. It is essential to always use `prepared statements` or `parameterized queries` to prevent this type of SQL Injection attack. This example can be used to clearly illustrate the impact of missing basic security protections.