SQLi or SQL Injection is an attack where one leaks or peaks through back-end database that wasn't meant to be shown by injecting malicious code by exploiting application vulnerabilities.
Attackers locate a vulnerable input into a website or web application and exploit this vulnerability using user input in the form of the SQL query.
A successful SQLi attacks allow attackers to modify database information, access sensitive data, gain unauthorized access on the database, and recover files from the system.

Example : Using SQLi to Authenticate as Administrator

This example shows how an attacker can use SQL injection to circumvent an application's authentication and gain administrator privileges.
Consider a simple authentication system using a database table with usernames and passwords. A user's POST request will provide the variables user and pass, and these are inserted into a SQL statement:

sql = "SELECT id FROM users WHERE username='" + user + "' AND password='" + pass + "'"

The problem here is that the SQL statement uses concatenation to combine data. The attacker can provide a string like this instead of the pass variable:

password' OR 5=5

The resulting SQL query will be run against the database:

SELECT id FROM users WHERE username='user' AND password='pass' OR 5=5'

Because 5=5 is a condition that always evaluates to true, the entire WHERE statement will be true, regardless of the username or password provided.
The WHERE statement will return the first ID from the users table, which is commonly the administrator. This means the attacker can access the application without authentication, and also has administrator privileges.
A more advanced form of this attack is where the attacker adds a code comment symbol at the end of the SQL statement, allowing them to further manipulate the SQL query. The following will work in most databases including MySQL, PostgreSQL, and Oracle:

' OR '5'='5' /*

Source: SQL Injection Attack: Real Life Attacks and Code Examples (brightsec.com)

To prevent SQLi attacks, one can:

- Filter inputs
- Restrict database code
- Restrict database access
- Maintain and monitor the application and database
- Avoid using dynamic SQL
- Avoid placing user-provided input directly into SQL statements
- Prefer prepared statements and parameterized queries, which are much safer
- Use stored procedures instead of dynamic SQL
- Remove potential malicious code elements such as single quotes