What is SQL Injection attack? Explain with an example

What is an SQL Injection attack?

An **SQL Injection (SQLi)** attack is a web-application security vulnerability that occurs when an attacker is able to insert (or "inject") malicious SQL code into an application's database query. The application treats that injected input as part of the SQL command, which can let the attacker read, modify, or delete data; bypass authentication; or even execute administrative operations on the database.

SQLi happens when user input is concatenated directly into SQL queries without proper handling.

> **High-level:** untrusted input → combined into SQL string → database runs it → attacker controls the query's logic.

Simple example (conceptual / defensive)

Imagine a classic login form where the application checks credentials with code that *naively* builds a SQL string from the user inputs.

Vulnerable pseudo-code (illustrative only):

```sql
-- server-side pseudo-code
username = get_input("username")
password = get_input("password")

query = "SELECT id FROM users WHERE username = '" + username + "' AND password = '" + password +
"';"
result = database.run(query)
if result.has_row():
    login_success()
else:
    login_fail()
```

If the application simply concatenates `username` and `password` into the SQL string, an attacker can craft input that changes the meaning of the query.

Illustrative malicious input idea: an input that makes the WHERE clause always true.
(Shown here for learning/defense—use only in test/lab environments and never against systems you don't own.)

What can go wrong?

Depending on the database permissions and the attacker's skill, consequences include:

- Bypassing authentication (logging in without valid credentials).
- Reading sensitive data (other users' records).

- Modifying or deleting data (DROP/DELETE/UPDATE).
- Running administrative commands or escalating access.
- Extracting database schema and credentials.

How to prevent SQL Injection — best practices

1. **Use parameterized queries / prepared statements**
   Always use the database API's parameter binding instead of string concatenation. This separates code from data so user input can't change SQL structure.

   Example (safe pseudo-code):

   ```sql
   stmt = db.prepare("SELECT id FROM users WHERE username = ? AND password = ?")
   stmt.execute(username, password)
   ```

2. **Use stored procedures carefully**
   Stored procedures can help if they themselves use parameters correctly. Don't build SQL inside the stored procedure from concatenated inputs.
3. **Use an ORM**
   Modern ORMs (when used correctly) avoid raw SQL and reduce risk. Still treat ORM query-building APIs carefully — don't inject raw SQL fragments with untrusted input.
4. **Principle of least privilege**
   The DB account used by the application should have only the needed permissions (SELECT/INSERT/UPDATE on specific tables). Don't use admin/root DB credentials in the app.
5. **Input validation and sanitization**
   Validate input for expected format and length (e.g., username characters, max lengths). Validation is defense-in-depth — it's not a substitute for parameterized queries.
6. **Output encoding**
   If displaying DB results in HTML, encode output to avoid cross-site scripting (XSS) and related issues.
7. **Use Web Application Firewalls (WAFs)**
   A WAF can detect and block common injection patterns as an additional layer of protection.
8. **Logging, monitoring, and regular testing**
   Log suspicious behavior, monitor for unusual queries, and run security testing (static analysis, dynamic scans, and authorized penetration tests) in controlled environments.

How to detect SQLi (quick signs)

- Unexpected, long, or malformed query strings in logs.
- Application errors revealing SQL or schema information.
- Authentication bypasses, sudden spikes of data reads.
- Automated scanner alerts during security tests.

Ethical note

Demonstrations of SQLi are for **learning, testing, and fixing** vulnerabilities only. Never test or exploit systems you do not own or do not have explicit permission to test. Unauthorized attacks are illegal.

If you want, I can:

- Show a concrete safe example in a specific language/framework (e.g., PHP PDO, Python `psycopg2`, Java JDBC) illustrating the vulnerable vs. parameterized code.
- Provide a short checklist you can use to audit a web app for SQLi vulnerabilities. Which would you prefer?

List the types of SQL injection attacks

Types of SQL injection attacks

Here's a compact list of the common types of SQL injection, with a one-line explanation and the typical goal/impact for each:

1. **In-band SQLi**
   - **Union-based SQLi** — attacker uses `UNION` to append a malicious `SELECT` so the DB returns attacker-controlled rows (easy data exfiltration).
   - **Error-based SQLi** — cause the database to produce error messages that leak schema/data (uses DB error output to learn structure).
2. **Inferential (Blind) SQLi**
   - **Boolean-based (content-based) blind** — send payloads that produce true/false responses; infer data bit-by-bit from differences in page content.
   - **Time-based blind** — cause the DB to delay (e.g., `WAITFOR`, `SLEEP`) when a condition is true; infer data from response timing.
3. **Out-of-band (OOB) SQLi**
   - Use features that make the DB send data to an attacker-controlled service (DNS/HTTP callbacks). Useful when in-band channels are blocked or responses aren't returned.
4. **Second-order SQLi**
   - Malicious input is stored benignly (e.g., in a profile) then later used in a different SQL context where it executes. Harder to detect because the injection happens in a later flow.
5. **Blind vs. Visible**
   - **Visible (discoverable)**: the application returns database results or errors directly (makes exploitation easier).
   - **Blind**: no error/data returned — inferential techniques are required (Boolean/time-based).

6. **Stored (Persistent) SQLi**
   - Payloads are stored on the server (e.g., user profile, comments) and executed later when that stored data is used in a query. Overlaps with second-order attacks.
7. **Reflected / Non-persistent SQLi**
   - Attack payload is sent and immediately used in a query and response (similar conceptually to reflected XSS — input leads to immediate effect).
8. **Logical/Boolean-based manipulation**
   - Changing query logic (e.g., `OR 1=1`) to bypass authentication or fetch more rows.
9. **Privilege/escalation & command injection via SQL**
   - Using SQL features to run administrative commands, write files, or execute OS-level commands (depends on DB privileges).
10. **Time/Resource exhaustion (Denial) via SQL**

- Crafting queries that cause heavy load or long-running operations to cause DoS.

Quick notes

- Many of these overlap (e.g., union-based is an in-band visible technique; a payload can be both stored and second-order).
- The attack method chosen depends on the app behavior (errors shown or not), DB features, and the attacker's goals.
- Defenses (parameterized queries, least privilege, input validation, prepared statements) protect against all types.

Want short examples for any specific type (e.g., union-based, boolean blind, or out-of-band DNS exfiltration)? Which language/framework should I use for the demo?

What is Cross-Site Scripting attack ? name and describe its types

What is Cross-Site Scripting (XSS)?

Cross-Site Scripting (XSS) is a web security vulnerability where an attacker injects malicious client-side script (usually JavaScript) into web pages viewed by other users. Because the browser trusts content from the site, the injected script runs in the victim's browser with that site's privileges — letting the attacker steal cookies/session tokens, perform actions as the user, deface content, or load additional malicious code.

Main types of XSS (and short descriptions)

1. Stored (Persistent) XSS

What it is: Malicious script is saved on the server (e.g., in a database, comment, user profile) and later served to any user who views the affected page.
Typical flow: Attacker submits payload → stored by application → page renders stored payload to other users → payload executes in victims' browsers.
Impact: High — wide reach, persistent until removed.
Simple payload example: `<script>fetch('/steal?c=' + document.cookie)</script>` (illustrative).
Prevention: Output-encode/escape when rendering, use a safe templating engine, validate and sanitize inputs, apply Content Security Policy (CSP), least privilege.

2. Reflected (Non-persistent) XSS

What it is: Malicious script is not stored; it is embedded in a link or form and reflected back in the immediate server response (often via a URL parameter). The victim must click a crafted link or submit a request.
Typical flow: Attacker crafts URL with payload → convincing victim to click → server reflects payload into page → payload runs.
Impact: Lower reach than stored XSS but useful for targeted phishing/social engineering.
Simple payload example: `https://example.com/search?q=<script>alert(1)</script>` where the `q` parameter is rendered without escaping.
Prevention: Properly encode user input before rendering, use parameterized APIs, avoid directly inserting user input into HTML, CSP, validate/refuse dangerous inputs.

3. DOM-based (Client-side) XSS

What it is: The vulnerability exists in client-side code (JavaScript). The application's JavaScript reads data from an untrusted source (URL fragment, query string, `document.location`, `document.referrer`, or `innerHTML`) and unsafely writes it into the DOM, causing script execution — without the server ever reflecting or storing the payload.
Typical flow: Attacker crafts link with payload in location/hash → victim visits → client JS processes the payload insecurely → payload executes.
Impact: Can be as severe as other types; detection requires code review of client scripts.
Simple example: `location.hash` parsed and injected into page with `element.innerHTML = location.hash` (unsafe).
Prevention: Avoid using `innerHTML` with untrusted data, use `textContent/innerText`, sanitize/validate client-side inputs, use safe DOM APIs.

Other variations & related terms

- **Self-XSS:** A social-engineering trick where a user is tricked into pasting malicious code into their own browser console. Not a vulnerability in the site itself but a common attack vector via user deception.
- **Mutated XSS (mXSS):** Browser parses/"mutates" input in ways that create XSS even when the developer thought escaping was sufficient; uncommon and tricky.
- **Second-order XSS:** Input is stored harmlessly (e.g., as data) but later used in a different context (or with different escaping) where it becomes executable — overlap with stored XSS logic.

Typical impacts

- Session hijacking (steal cookies/tokens)
- Account takeover (actions as the user)
- Data exfiltration (read page data or make requests)
- Phishing / UI redress (fake dialogs/forms)
- Browser fingerprinting or malware delivery

High-level prevention checklist (quick)

1. Escape/encode output according to context (HTML, attribute, JS, URL).
2. Use safe templating libraries that auto-escape.
3. Use `textContent` instead of `innerHTML` and safe DOM APIs.
4. Validate and sanitize inputs server- and client-side.
5. Apply a strong Content Security Policy (CSP) to block inline scripts and untrusted sources.
6. Use HttpOnly cookies for session tokens (prevents access via JS).
7. Conduct code reviews and automated security testing (SAST/DAST).

Want short concrete examples of vulnerable vs. fixed code in a particular language/framework (e.g., plain HTML/JS, React, PHP, or Node/Express)?