SQL Injection

Table of contents

The three methods 1
SQL manipulation — bypassing a login 2
Code injection — going further 2
Risks 3
Protection techniques 3
See also 3

SQL injection is one of the most common threats to a database system: an attacker supplies string input through an (often web-facing) application in a way that changes or manipulates the resulting SQL statement to their advantage. Introduced in 2026-05-11-database-security[1].

It sidesteps everything in database-access-control[2] rather than breaking it — the injected statement runs under the *application's* own, legitimately granted privileges, so no `GRANT` was violated and nothing in the access matrix looks wrong afterwards.

The three methods

Method	What the attacker does
SQL manipulation	Changes an *existing* SQL command — typically by adding conditions to a `WHERE` clause. Classic target: the login form
Code injection	Adds *entirely new* SQL statements or commands, exploiting improper handling of untrusted input

[1] courses/infs1200/2026-05-11-database-security.pdf
[2] courses/infs1200/database-access-control.pdf

Method	What the attacker does
Function call injection	Inserts a database or OS function call into a vulnerable statement, to manipulate data or make a privileged system call

All three have the same root cause: user input is **concatenated into an SQL string**, so the database can no longer tell where the developer's query ends and the user's data begins.

SQL manipulation — bypassing a login

A naive login check builds its query by direct concatenation:

```
SELECT * FROM Users WHERE Username = '<input>' AND Password = '<input>';
```

A normal login (`Jack` / `Pass123`) produces `WHERE Username = 'Jack' AND Password = 'Pass123'`, matches the stored row, and access is granted. But an attacker who knows the username enters this as the *password*:

```
' OR 'x'='x
```

The closing quote terminates the password literal early, and everything after it is read as SQL rather than data:

```
SELECT * FROM Users WHERE username = 'Jack' AND (password = '' OR 'x'='x');
```

`'x'='x'` is always true, so the condition collapses to just `username = 'Jack'`. The row comes back regardless of the actual password, and the application grants access.

Code injection — going further

A semicolon ends one statement and begins another. If the application executes whatever SQL string results, and the driver or DBMS permits statement stacking:

```
wrongpass' OR 'x'='x'; drop table Users;
```

This doesn't merely bypass the login — it destroys the entire `Users` table. The step from manipulation to code injection is small; the damage is not.

Risks

Risk	What it gets the attacker
Database fingerprinting	Identifying the DBMS in use, to target version-specific exploits
Denial of service	Denying service to valid users
Bypassing authentication	Access without valid credentials, as above
Identifying injectable parameters	Learning the backend structure from error messages and behaviour
Executing remote commands	Running harmful commands remotely, e.g. via function call injection
Privilege escalation	Upgrading the attacker's own access level

Note that several of these are *reconnaissance*: verbose error messages are themselves a vulnerability, because they tell an attacker which parameters are injectable and what the schema looks like.

Protection techniques

- **Bind variables (parameterised statements)** — the primary defence. Pass user input as *bound parameters* rather than concatenating it into the SQL string; the driver then treats every parameter value purely as data and never as executable syntax, so a quote in the input can't close a literal. As a bonus, parameterised queries improve performance, since the query plan can be cached and reused across executions.
- **Filtering input (input validation)** — strip or escape characters, such as unescaped quotes, that could break out of a string literal. Useful as defence in depth, but it's a blocklist by nature: it depends on anticipating every dangerous character, whereas bind variables remove the ambiguity entirely.
- **Function security** — restrict which standard and custom database functions are callable, to limit the blast radius of function call injection.

The ordering matters: bind variables fix the *structural* problem (data being parsed as code), while filtering and function restrictions only narrow what an attacker can do once that structural problem exists.

See also

- database-access-control[3] — the privileges an injected statement inherits

[3] courses/infs1200/database-access-control.pdf

- sql-ddl[4] · sql-select-statement[5] · sql-views[6]
- 2026-05-11-database-security[7] — Module 5
- week12-tutorial-5-1-database-security[8] — practice

[4] courses/infs1200/sql-ddl.pdf
[5] courses/infs1200/sql-select-statement.pdf
[6] courses/infs1200/sql-views.pdf
[7] courses/infs1200/2026-05-11-database-security.pdf
[8] courses/infs1200/week12-tutorial-5-1-database-security.pdf