

SESSION REPORT

SQL Injection: Find and Fix

Applied security lab: an Oracle EMPLOYEES search app built with PyQt5 and python-oracledb

COURSE	Securite des applications
PROGRAM	AEC Infrastructure Informatique et Cybersecurite
DATABASE	Oracle, table EMPLOYEES
DATE	September 9, 2026

1. Scenario

A small internal application searches the EMPLOYEES table of an Oracle database. The user supplies a last name and a department number, and the application builds a SQL query and displays the matching employees. The application works correctly under normal use, but the way it builds the query introduces a SQL injection vulnerability.

2. The vulnerable query

The original code built the WHERE clause by concatenating the user-supplied values directly into the query string:

```
CODE
query = ("SELECT employee_id, first_name, last_name, email, job_id, salary, department_id "
        "FROM employees WHERE last_name = '" + lastname + "'" AND department_id = " + department)
```

The last name is wrapped in quotes; the department number, being numeric, is concatenated without quotes, which turns out to be the exploitable part.

3. Analysis

The last name and department number, typed into the two search fields, come directly from the user and are inserted into the WHERE clause by direct string concatenation.

Concatenation is dangerous here because it mixes data and SQL code in a single string: everything the user types is sent to Oracle as executable SQL, so operators or conditions, such as OR 1=1, can be slipped in and change the logic of the query.

A data value is a simple value the query compares against, such as 90 or 'King'. SQL syntax is the structure of the query itself, such as the keywords SELECT, WHERE, AND, and OR. Concatenation lets user input cross from the first category into the second.

4. Reproducing the problem

Entering 90 OR 1=1 in the department field, with a normal last name, changed the generated clause to:

```
CODE
WHERE last_name = 'd' AND department_id = 90 OR 1=1
```

Because AND binds tighter than OR, the clause is evaluated as (last_name = 'd' AND department_id = 90) OR 1=1, and since 1=1 is always true, the query returned every row in the table (107 rows observed) instead of the handful matching the intended filters.

5. Fix

Rewrote the query to use python-oracledb bind variables instead of concatenation, so the last name and department are always sent to Oracle as data, never as part of the SQL text:

CODE

```
query = ("SELECT employee_id, first_name, last_name, email, job_id, salary, department_id "  
        "FROM employees WHERE last_name = :lastname AND department_id = :department")  
  
cursor.execute(query, {"lastname": lastname, "department": department})
```

6. Validation

Test	Last name	Department	Result
Normal search	King	90	Search continued to work as before.
Another normal search	Kochhar	90	Returned only the matching rows.
Former attack input	King	90 OR 1=1	No longer altered the query; returned no extra rows.

7. Analysis questions

Why is a parameterized query safer than one built by concatenation? Because the database receives the SQL structure and the values separately: the values are always treated as data and never interpreted as code, so a malicious input can no longer alter the query.

Why could the department field be interpreted as a SQL expression in the vulnerable version? Because it was concatenated directly and, being numeric, inserted without quotes, so it landed raw inside the WHERE clause, where OR 1=1 was read as SQL operators rather than a value.

Why is a blacklist of keywords like OR or SELECT not a robust fix? Because it can be bypassed many ways (case, spacing, comments, encoding) and also blocks legitimate data; it treats the symptom, not the cause, which is mixing data and code.

What changed between the query and the results before and after the fix? Before, the query varied with the input and could contain injected SQL (...OR 1=1 returned 107 rows); after, the query text stays fixed and the input is only ever a value, so only the correct rows come back.

Another good practice to limit the impact of a SQL injection, if one were still present: least privilege, granting the application account only the minimum rights it needs (for example SELECT on employees), so that a successful injection could not read or modify other data.

8. Conclusion

A value typed by a user should never be able to change the structure of a SQL query. Bind variables enforce that separation at the driver level, which is why they fix the vulnerability generally instead of just blocking one specific attack string.