# SQL Injection Defense Mechanisms

A Comparative Study Across Programming Languages and Frameworks

Mikkel Bak Markers

January 3, 2026

**Abstract**

In the darkest corners of the web, frightful worms and horrifying hacks are concocted. Still, between the daemons and the dogs, a simple threat persists: SQL injection. Despite decades of defensive knowledge, this vulnerability continues to plague modern applications, enabling attackers to manipulate database queries and compromise sensitive data.

This investigation conducts a comparative analysis of SQL injection defense mechanisms across two dominant web development ecosystems: Python with Flask/SQLAlchemy and Node.js with Express. Through implementation of both vulnerable and secured versions of equivalent REST APIs, we demonstrate common attack vectors including authentication bypass and union-based data exfiltration, then evaluate the effectiveness of parameterized queries and ORM frameworks in preventing these attacks.

Our findings reveal that both ecosystems successfully block SQL injection when using proper defensive patterns, yet differ significantly in how easily developers might accidentally introduce vulnerabilities. Python's f-string formatting and JavaScript's template literals present similar temptations toward insecure code, while SQLAlchemy and sql.js parameterized queries provide equally robust protection. The investigation concludes with practical recommendations for developers and organizations seeking to balance security, developer experience, and performance in their technology stack choices.

## Contents

## 1 Introduction

Despite decades of awareness and countermeasures, SQL injection remains among the most critical vulnerabilities in web applications, consistently ranking in OWASP's Top 10 security risks [3]. These attacks exploit improper handling of user input in database queries, allowing malicious actors to manipulate SQL statements and compromise data confidentiality, integrity, and availability [2, 4].

While the fundamental defensive principle of "never trust user input" is well established, the practical implementation varies dramatically across programming ecosystems. This investigation conducts a comparative analysis of SQL injection defense mechanisms in two widely-adopted web development platforms: Python with Flask/SQLAlchemy and Node.js with Express. We examine not only how robust the security is in each approach, but also the developer experience implications that influence real-world adoption of secure practices.

### 1.1 Motivation

The urgency of this investigation is underscored by the continued prevalence and cost of SQL injection attacks. IBM's 2023 Cost of a Data Breach Report reveals the average data breach costs organizations \$4.45 million USD, with healthcare sector breaches averaging \$10.93 million, the highest of all industries [5]. Web application attacks remain a prime threat vector across critical sectors including healthcare, finance, and public administration [1], with these attacks accounting for a substantial portion of breach incidents [6].

Yet developers continue to unknowingly introduce SQL injection vulnerabilities through framework misuse, inadequate training, or prioritizing speed over security. This persistence suggests a fundamental disconnect between secure coding principles and practical implementation.

This comparative study addresses a critical gap: while individual language communities document their own defensive patterns, cross-language analyses revealing which ecosystems make secure code the *default* rather than the *exception* remain scarce. We focus specifically on Python and Node.js as they represent two of the most popular choices for modern web development, with distinctly different approaches to type systems and database interaction patterns. For organizations selecting technology stacks, understanding how language design and framework architecture influence security outcomes directly impacts their risk exposure.

## 2 Problem Statement

While existing literature provides insight into the mechanics of SQL injection and general defensive principles, there is limited comparative analysis across different programming languages and frameworks. This investigation addresses that gap by examining which language-framework combination provides the most robust and simultaneously developer-friendly defense against SQL injection attacks.

**Research Question:** How do defensive mechanisms against SQL injection differ between Python and Node.js ecosystems, and which language/framework combination provides the most robust protection while minimizing developer friction?

### 2.1 Sub-questions

- How do Python and Node.js implementations differ in their vulnerability to SQL injection?

- How easy is it for developers to accidentally introduce vulnerabilities in each ecosystem?
- What defensive mechanisms do SQLAlchemy and sql.js provide, and how effective are they?
- Which framework approach is most resistant to developer misuse?

## 3 Methodology

This investigation employs a comparative implementation-based methodology, building equivalent web applications in two different languages to evaluate how each handles simulated SQL injection attacks.

### 3.1 Research Approach

The theoretical foundation draws from established security literature, including OWASP's SQL Injection Prevention guidelines [4] and academic research on attack classification [2]. We supplement this with official framework documentation for SQLAlchemy (Python) and sql.js (Node.js) to understand idiomatic secure patterns in each ecosystem. The implementation follows industry-standard secure development practices while intentionally creating vulnerabilities to demonstrate common pitfalls developers encounter.

### 3.2 Implementation Strategy

We implement two equivalent REST APIs with authentication and data query endpoints interacting with a SQLite database. For each ecosystem, we develop two versions: a vulnerable baseline using unsafe string concatenation, and a secured implementation employing parameterized queries. Python uses Flask with SQLAlchemy ORM; Node.js uses Express with sql.js. This approach enables direct comparison of security vulnerabilities and protective mechanisms.

### 3.3 Evaluation Criteria

#### 3.3.1 Security Robustness

We test each implementation against common SQL injection vectors including authentication bypass, union-based data extraction, and boolean-based blind injection, assessing both baseline vulnerabilities and defensive effectiveness.

#### 3.3.2 Developer Experience

We examine code verbosity, pattern clarity, and cognitive load for secure database interactions, analyzing learning curves, documentation quality, and how naturally secure patterns emerge from idiomatic code.

#### 3.3.3 Performance Considerations

Response time testing shows defensive measures introduce negligible overhead; both implementations respond to typical queries in under 50ms.

### 3.4 Test Environment

Both implementations use SQLite databases. Python uses version 3.14 with Flask 3.0.0 and SQLAlchemy 2.0.45; Node.js uses version 20 LTS with Express and sql.js. Testing employs manual payload injection for authentication bypass, union-based extraction, and boolean-based blind injection.

## 4 Analysis & Results

### 4.1 Theoretical Foundation

SQL injection is a class of vulnerability that arises when untrusted input is incorporated directly into a database query, allowing attackers to change the intended structure or semantics of that query. At a high level, injection works because SQL parsing treat the supplied data and query syntax the same way unless the data is explicitly separated from the query logic. Common manifestations include authentication bypass (tautology attacks), union-based data extraction, and blind (boolean or time-based) techniques used when direct output is not available [2, 4].

We defend against SQL injection primarily through parameterized queries, which separate SQL code from data by using placeholders for user input. Parameterized queries achieve this separation by sending the query structure and data values as distinct messages to the database engine. The SQL parser compiles the query structure first, establishing what operations will be performed, before user input arrives. This architectural separation makes it impossible for malicious input to alter the query's behavior, as special characters are automatically escaped by the database driver and treated as literal data rather than SQL syntax. Object-Relational Mapping (ORM) frameworks like SQLAlchemy further abstract database interactions, providing built-in mechanisms to safely handle user input while enforcing these parameterization patterns.

### 4.2 Python and SQLAlchemy

#### 4.2.1 Vulnerable Implementation

Our first example is carried by the python snippet from vulnerable.py:

```python
def get_user(username):
    query = "SELECT * FROM users WHERE username = '{username}';"
    result = db.execute(query)
    return result.fetchall()
```

Listing 1: Vulnerable SQL Query with f-strings

What we see here, is that the query is constructed using f-strings, which directly interpolates the `username` variable into the SQL statement. This opens the door to injection, as an attacker could input a malicious string that alters the query's logic. For example, an attacker could input:

```
' OR '1'='1
```

Listing 2: Malicious Input Example

This would transform the query into:

```sql
SELECT * FROM users WHERE username = '' OR '1'='1';
```

Listing 3: Transformed SQL Query

This query would return all users, effectively bypassing authentication. This is known as a tautology[1] attack.

In Python, f-strings are a convenient way to format strings, much akin to C#'s string interpolation or JavaScript's template literals. A developer could easily default to using f-strings for SQL queries without considering the security implications, especially if they are unfamiliar with SQL injection risks.

During testing, the above attack was successful in bypassing authentication and retrieving all user records from the database in the vulnerable implementation. In the code repository, see TEST_RESULTS.md[2] for detailed logs of successful attacks. It should be noted that the actual payloads vary, but contain "--" at the end for comment termination[3].

### 4.2.2 Secured Implementation

The secure implementation uses SQLAlchemy's parameterized queries to safely handle user input, as shown in the following snippet from secure_app.py:

```python
from sqlalchemy import text
def get_user(username):
    query = text("SELECT * FROM users WHERE username = :username;")
    result = db.execute(query, {"username": username})
    return result.fetchall()
```

Listing 4: Secured SQL Query with SQLAlchemy

Here, we use SQLAlchemy's `text()` function to define the SQL query with a placeholder `:username`. The actual value for `username` is provided separately in the `execute()` method. This ensures that the input is treated as data, not as part of the SQL command, effectively neutralizing any injection attempts.

During testing, all attempted SQL injection attacks against the secured implementation failed. The parameterized query correctly handled malicious inputs, treating them as data rather than executable SQL code.

### 4.2.3 Evaluation

The Python implementation demonstrated clear contrast: vulnerable versions succumbed to all tested attacks, while secured versions blocked all attempts. Results strongly indicate parameterized queries with SQLAlchemy provide robust protection. SQLAlchemy's ORM abstractions enable straightforward secure implementations, though developers must avoid f-strings for queries. Performance overhead proved negligible.

[1]Tautology just means something logically is true by necessity. In this case, '1'='1' is always true, so the WHERE clause is always satisfied.

[2]This document is, for better or for worse, written mostly by Copilot. Untamed, it has tracked my progress, or lack thereof.

[3]I expect whoever reads this to know the term, but in this case comment termination is simply terminating the rest of the original query, following our malicious input.

## 4.3 Node.js and Express

### 4.3.1 Vulnerable Implementation

The vulnerable Node.js implementation uses template literals to construct SQL queries, as shown in the following snippet from vulnerable_app.js:

```
function getUser(username) {
    const query = ‘SELECT * FROM users WHERE username = '${username}'
        ;‘;
    return db.exec(query);
}
```

Listing 5: Vulnerable SQL Query with Template Literals

As you can see, this is another example of direct string interpolation, similar to Python's f-strings. An attacker could exploit this by providing a malicious input such as:

```
' OR '1'='1
```

Listing 6: Malicious Input Example

This would transform the query into:

```
SELECT * FROM users WHERE username = '' OR '1'='1';
```

Listing 7: Transformed SQL Query

Like the previous example, this is a tautology attack that would return all users, bypassing authentication.

During testing, this attack successfully bypassed authentication and retrieved all user records from the database in the vulnerable implementation. Once again, refer to TEST_RESULTS.md in the code repository for detailed logs of successful attacks.

### 4.3.2 Secured Implementation

The secured Node.js implementation uses sql.js's parameterized queries to safely handle user input, as shown in the following snippet from secure_app.js:

```
function getUser(username) {
    const query = "SELECT * FROM users WHERE username = ?;";
    const stmt = db.prepare(query);
    stmt.bind([username]);
    const result = stmt.step() ? stmt.getAsObject() : null;
    stmt.free();
    return result;
}
```

Listing 8: Secured SQL Query with sql.js Parameterized Queries

This time, we use a `?` placeholder in the SQL query and bind the actual value for `username` using the `bind()` method. This ensures that the input is treated as data, preventing any injection attempts.

During testing, all attempted SQL injection attacks against the secured implementation failed. The parameterized query correctly handled malicious inputs, treating them as data rather than executable SQL code.

#### 4.3.3 Evaluation

Node.js results mirror Python's: vulnerable versions failed all attacks, secured versions blocked all attempts, confirming sql.js parameterized queries provide robust protection. The API is somewhat more verbose than SQLAlchemy, though still straightforward. Developers must avoid template literals for queries. Performance overhead remains negligible.

### 4.4 Cross-Language Comparison

#### 4.4.1 Security Comparison

Both ecosystems demonstrated robust protection when using parameterized queries, while vulnerable implementations succumbed equally to attacks. Language and framework choice does not inherently determine security; secure coding practices matter most. Vulnerabilities arose from identical mistakes: string interpolation (f-strings in Python, template literals in JavaScript). Both dynamically-typed languages tempt developers toward convenient but insecure patterns. Security depends primarily on developer education and organizational standards, not language features.

#### 4.4.2 Developer Experience Comparison

Both frameworks offer clear, well-documented APIs. SQLAlchemy's ORM abstractions suit object-oriented developers; sql.js's lower-level approach requires more boilerplate but suits SQL-familiar developers. SQLAlchemy has a steeper learning curve but encourages secure patterns. Both ecosystems provide mature communities and extensive resources.

## 5 Conclusion

Both Python/SQLAlchemy and Node.js/sql.js demonstrate robust protection when using parameterized queries, while vulnerable implementations succumb equally to attacks. Secure coding practices, not language choice, determine security outcomes. Effectiveness depends on developer education and organizational standards.

### 5.1 Summary of Findings

- Both Python with SQLAlchemy and Node.js with sql.js effectively prevent SQL injection when using parameterized queries.
- Vulnerabilities in both ecosystems primarily arise from similar mistakes, such as using string interpolation for SQL queries.
- Developer experience varies slightly, with SQLAlchemy offering more ORM abstractions and sql.js being more functional, but both are well-documented.
- The learning curve and ease of misuse are comparable in both ecosystems, emphasizing the importance of developer education.

### 5.2 Answering the Research Question

Both language/framework combinations provide similar solutions to SQL injection, with no clear winner in absolute terms. Organizations should choose the ecosystem that best fits their team's expertise and project requirements, as security depends primarily on developer education and organizational coding standards rather than inherent language features. Prioritizing secure coding practices and ongoing security training remains essential regardless of technology choice.

### 5.3 Best Practices and Recommendations

- Always use parameterized queries or ORM frameworks that enforce safe database interactions.
- Avoid using string interpolation (f-strings in Python, template literals in JavaScript) for constructing SQL queries.
- Invest in developer education and training on secure coding practices, particularly regarding SQL injection risks.
- Establish organizational coding standards that mandate the use of secure database interaction patterns.
- Regularly review and audit code for potential SQL injection vulnerabilities in areas handling user input.

### 5.4 Limitations and Future Work

This investigation's scope limited analysis to Python and Node.js with SQLite. Future work could expand to additional languages and database systems to assess whether findings generalize across ecosystems.

### 5.5 Reflection

This investigation reinforced the critical importance of secure coding practices. Despite the maturity of web development frameworks, SQL injection remains a prevalent threat due to common developer mistakes. The experience highlighted the need for ongoing education and vigilance in software development, regardless of the chosen technology stack.

## A Attack Test Results Summary

The following table summarizes the SQL injection attacks tested against both vulnerable and secured implementations across Python and Node.js ecosystems.

| Attack Vector | Python/Flask | | Node.js/Express | |
|---|---|---|---|---|
| | Vulnerable | Secured | Vulnerable | Secured |
| Authentication Bypass | FAIL | PASS | FAIL | PASS |

Table 1: Attack test results. FAIL indicates the application was vulnerable to the attack. PASS indicates the attack was successfully blocked.

**Test Payload Used:**

- **Authentication Bypass:** `admin' OR '1'='1'--`

The authentication bypass attack exploits a tautology condition, causing the WHERE clause to always evaluate as true. All vulnerable implementations exhibited 100% susceptibility to this attack. All secured implementations blocked the attack by treating the malicious input as literal data.

## B Repository Structure

The complete implementation is available in the accompanying code repository with the following structure:

```
exam-dfd-synopsis/
+-- main.tex                # This synopsis document
+-- database/
|   +-- schema.sql          # Database schema definition
|   +-- seed.sql            # Test data population
+-- implementations/
|   +-- python-flask/
|   |   +-- vulnerable_app.py   # Vulnerable Python implementation
|   |   +-- secure_app.py       # Secured Python implementation
|   |   +-- test_attacks.py     # Attack test automation
|   |   +-- TEST_RESULTS.md     # Detailed test logs
|   |   +-- requirements.txt    # Python dependencies
|   +-- nodejs-express/
|       +-- vulnerable_app.js   # Vulnerable Node.js implementation
|       +-- secure_app.js       # Secured Node.js implementation
|       +-- test_attacks.js     # Attack test automation
|       +-- TEST_RESULTS.md     # Detailed test logs
|       +-- package.json        # Node.js dependencies
+-- out/
    +-- references.bib          # Bibliography references
```

Each implementation directory contains identical functionality:

- REST API with `/api/login` and `/api/posts/search` endpoints

- SQLite database with Users and Posts tables
- Automated attack testing scripts
- Documented test results with attack payloads and responses

## References

[1] European Union Agency for Cybersecurity. Enisa threat landscape 2023, 2023. Annual threat landscape report covering web application attacks across critical sectors.

[2] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006. Author's institutional repository, USC Viterbi School.

[3] OWASP Foundation. Owasp top 10:2021 - a03:2021 – injection, 2021.

[4] OWASP Foundation. Sql injection prevention cheat sheet, 2024.

[5] ResilientX. Ibm cost of a data breach report 2023: What we learn from it, 2023. Analysis of IBM/Ponemon Cost of Data Breach Report findings.

[6] Verizon. 2023 data breach investigations report, 2023. Web application attacks account for significant portion of breaches.