A05:2025 — Injection

January 6, 2026

Document Summary

This document consolidates the provided content for *A05:2025 — Injection* into a structured, print-ready reference, including background context, scoring metrics, vulnerability indicators, prevention guidance (command/data separation, safe APIs, parameterization, validation and escaping), practical attack scenarios (SQL, HQL/ORM, OS command injection), references, and the mapped CWE list.

Contents

1 Background

Injection falls two spots from #3 to #5 in the ranking, maintaining its position relative to A04:2025 — Cryptographic Failures and A06:2025 — Insecure Design. Injection is one of the most tested categories, with 100% of applications tested for some form of injection.

It has the greatest number of CVEs of any category, with 37 CWEs in this category. Injection includes:

- **Cross-site Scripting (XSS)** (high frequency / low impact), with more than 30,000 CVEs, and
- **SQL Injection** (low frequency / high impact), with more than 14,000 CVEs.

The massive number of reported CVEs for CWE-79 (Improper Neutralization of Input During Web Page Generation, *Cross-site Scripting*) lowers the average weighted impact of this category.

2 Score Table

Metric	Value
CWEs Mapped	37
Max Incidence Rate	13.77%
Avg Incidence Rate	3.08%
Max Coverage	100.00%
Avg Coverage	42.93%
Avg Weighted Exploit	7.15
Avg Weighted Impact	4.32
Total Occurrences	1,404,249
Total CVEs	62,445

Table 1: Provided scoring summary for Injection.

3 Description

An injection vulnerability is an application flaw that allows untrusted user input to be sent to an interpreter (e.g., a browser, database, or the command line) and causes the interpreter to execute parts of that input as commands.

3.1 Vulnerability Conditions

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Unsanitized data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.

- Potentially hostile data is directly used or concatenated. The SQL or command contains both structure and malicious data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, ORM, LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters.

Detection is best achieved by a combination of source code review and automated testing (including fuzzing) of all parameters, headers, URL, cookies, JSON, SOAP, and XML inputs. Adding static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline can help identify injection flaws before production deployment.

Related Topic: LLM Prompt Injection

A related class of injection vulnerabilities has become common in LLMs. These are discussed separately in the OWASP LLM Top 10, specifically LLM01:2025 Prompt Injection.

4 How to Prevent

The best means to prevent injection requires keeping data separate from commands and queries.

4.1 Primary Controls

1. **Use safe APIs and parameterization:** Prefer safe APIs that avoid the interpreter entirely, provide a parameterized interface, or migrate to ORM tools.

   Important Note on Stored Procedures

   Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with `EXECUTE IMMEDIATE` or `exec()`.

2. **Apply layered mitigations when separation is not possible:** When you cannot fully separate data from commands, reduce risk using validation and escaping techniques.

4.2 Defense-in-Depth Techniques (Residual Risk)

- **Positive server-side input validation:** Use allow-list validation where feasible. This is not a complete defense, as many applications require special characters (e.g., text areas or mobile APIs).
- **Context-aware escaping:** For residual dynamic queries, escape special characters using the escape syntax appropriate for the interpreter.

  Warning on Escaping and Dynamic Structure

  SQL structures such as table names and column names cannot be escaped. User-supplied *structure names* are therefore dangerous and are a common issue in report-writing software. Techniques that rely on parsing and escaping complex strings are error-prone and not robust in the face of minor system changes.

5 Example Attack Scenarios

5.1 Scenario #1: SQL Injection via String Concatenation

An application uses untrusted data in the construction of the following vulnerable SQL call:

Vulnerable SQL Construction (Java)

```
String query = "SELECT * FROM accounts WHERE custID='"
  + request.getParameter("id") + "'";
```

An attacker modifies the `id` parameter in their browser to send: `' OR '1'='1`. For example:

Illustrative Request

```
http://example.com/app/accountView?id='OR'1'='1
```

This changes the meaning of the query to return all records from the `accounts` table. More dangerous attacks could modify or delete data or even invoke stored procedures.

5.2 Scenario #2: ORM/HQL Injection via Concatenation

An application's blind trust in frameworks may result in queries that are still vulnerable. For example, Hibernate Query Language (HQL):

Vulnerable HQL Construction

```
Query HQLQuery = session.createQuery(
  "FROM accounts WHERE custID='"
  + request.getParameter("id") + "'"
);
```

An attacker supplies: `' OR custID IS NOT NULL OR custID='`. This bypasses the filter and returns all accounts. While HQL has fewer dangerous functions than raw SQL, it still allows unauthorized data access when user input is concatenated into queries.

5.3 Scenario #3: OS Command Injection

An application passes user input directly to an OS command:

Vulnerable Command Construction

```
String cmd = "nslookup " + request.getParameter("domain");
Runtime.getRuntime().exec(cmd);
```

An attacker supplies `example.com; cat /etc/passwd` to execute arbitrary commands on the server.

6 References

- OWASP Proactive Controls: Secure Database Access
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, and ORM Injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications — OAT-014
- PortSwigger: Server-side template injection
- Awesome Fuzzing: a list of fuzzing resources

7 List of Mapped CWEs

CWE	Title
CWE-20	Improper Input Validation
CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component (*Injection*)
CWE-76	Improper Neutralization of Equivalent Special Elements
CWE-77	Improper Neutralization of Special Elements used in a Command (*Command Injection*)
CWE-78	Improper Neutralization of Special Elements used in an OS Command (*OS Command Injection*)
CWE-79	Improper Neutralization of Input During Web Page Generation (*Cross-site Scripting*)
CWE-80	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
CWE-83	Improper Neutralization of Script in Attributes in a Web Page
CWE-86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages
CWE-88	Improper Neutralization of Argument Delimiters in a Command (*Argument Injection*)
CWE-89	Improper Neutralization of Special Elements used in an SQL Command (*SQL Injection*)
CWE-90	Improper Neutralization of Special Elements used in an LDAP Query (*LDAP Injection*)
CWE-91	XML Injection (aka Blind XPath Injection)
CWE-93	Improper Neutralization of CRLF Sequences (*CRLF Injection*)
CWE-94	Improper Control of Generation of Code (*Code Injection*)
CWE-95	Improper Neutralization of Directives in Dynamically Evaluated Code (*Eval Injection*)
CWE-96	Improper Neutralization of Directives in Statically Saved Code (*Static Code Injection*)
CWE-97	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page
CWE-98	Improper Control of Filename for Include/Require Statement in PHP Program (*PHP Remote File Inclusion*)
CWE-99	Improper Control of Resource Identifiers (*Resource Injection*)
CWE-103	Struts: Incomplete `validate()` Method Definition
CWE-104	Struts: Form Bean Does Not Extend Validation Class
CWE-112	Missing XML Validation
CWE-113	Improper Neutralization of CRLF Sequences in HTTP Headers (*HTTP Response Splitting*)
CWE-114	Process Control
CWE-115	Misinterpretation of Output
CWE-116	Improper Encoding or Escaping of Output

CWE	Title
CWE-129	Improper Validation of Array Index
CWE-159	Improper Handling of Invalid Use of Special Elements
CWE-470	Use of Externally-Controlled Input to Select Classes or Code (*Unsafe Reflection*)
CWE-493	Critical Public Variable Without Final Modifier
CWE-500	Public Static Field Not Marked Final
CWE-564	SQL Injection: Hibernate
CWE-610	Externally Controlled Reference to a Resource in Another Sphere
CWE-643	Improper Neutralization of Data within XPath Expressions (*XPath Injection*)
CWE-644	Improper Neutralization of HTTP Headers for Scripting Syntax
CWE-917	Improper Neutralization of Special Elements used in an Expression Language Statement (*Expression Language Injection*)