# CWE Detail – CWE-89

## Description

The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2023-32530: SQL injection in security product dashboard using crafted certificate fields

**•** CVE-2021-42258: SQL injection in time and billing software, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-27101: SQL injection in file-transfer system via a crafted Host header, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-12271: SQL injection in firewall product's admin interface or user portal, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2019-3792: An automation system written in Go contains an API that is vulnerable to SQL injection allowing the attacker to read privileged data.

**•** CVE-2004-0366: chain: SQL injection in library intended for database authentication allows SQL injection and authentication bypass.

**•** CVE-2008-2790: SQL injection through an ID that was supposed to be numeric.

**•** CVE-2008-2223: SQL injection through an ID that was supposed to be numeric.

**•** CVE-2007-6602: SQL injection via user name.

**•** CVE-2008-5817: SQL injection via user name or password fields.

**•** CVE-2003-0377: SQL injection in security product, using a crafted group name.

**•** CVE-2008-2380: SQL injection in authentication library.

**•** CVE-2017-11508: SQL injection in vulnerability management and reporting tool, using a crafted password.

## Related Attack Patterns (CAPEC)

* CAPEC-108
* CAPEC-109
* CAPEC-110
* CAPEC-470
* CAPEC-66
* CAPEC-7

## Modes of Introduction

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic.

**•** Implementation: This weakness typically appears in data-rich applications that save user inputs in a database.

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands — Notes: Adversaries could execute system commands, typically by changing the SQL statement to redirect output to a file that can then be executed.

**•** Impact: Read Application Data — Notes: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.

**•** Impact: Gain Privileges or Assume Identity, Bypass Protection Mechanism — Notes: If poor SQL commands are used to check user names and passwords or perform other kinds of authentication, it may be possible to connect to the product as another user with no previous knowledge of the password.

**•** Impact: Bypass Protection Mechanism — Notes: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.

**•** Impact: Modify Application Data — Notes: Just as it may be possible to read sensitive information, it is also possible to modify or even delete this information with a SQL injection attack.

## Potential Mitigations

**•** Architecture and Design: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using persistence layers such as Hibernate or Enterprise Java Beans, which can provide significant protection against SQL injection if used properly. (Effectiveness: N/A)

**•** Architecture and Design: If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Process SQL queries using prepared statements, parameterized queries, or stored procedures. These features should accept parameters or variables and support strong typing. Do not dynamically construct and execute query strings within these features using "exec" or similar functionality, since this may re-introduce the possibility of SQL injection. [REF-867] (Effectiveness: N/A)

**•** Architecture and Design: Run your code using the lowest privileges that are required to accomplish the necessary tasks [REF-76]. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. Specifically, follow the principle of least privilege when creating user accounts to a SQL database. The database users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data. Use the strictest permissions possible on all database objects, such as execute-only for stored procedures. (Effectiveness: N/A)

**•** Architecture and Design: For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. (Effectiveness: N/A)

**•** Implementation: While it is risky to use dynamically-generated query strings, code, or commands that mix control and data together, sometimes it may be unavoidable. Properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict allowlist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88). Instead of building a new implementation, such features may be available in the database or programming language. For example, the Oracle DBMS\_ASSERT package can check or enforce that parameters have certain properties that make them less vulnerable to SQL injection. For MySQL, the mysql\_real\_escape\_string() API function is available in both C and PHP. (Effectiveness: N/A)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When constructing SQL query strings, use stringent allowlists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing SQL injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent SQL injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, the name "O'Reilly" would likely pass the validation step, since it is a common last name in the English language. However, it cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise handled. In this case, stripping the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded. When feasible, it may be safest to disallow meta-characters entirely, instead of escaping them. This will provide some defense in depth. After the data is entered into the database, later processes may neglect to escape meta-characters before use, and you may not have control over those processes. (Effectiveness: N/A)

**•** Architecture and Design: When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. (Effectiveness: N/A)

**•** Implementation: Ensure that error messages only contain minimal details that are useful to the intended audience and no one else. The messages need to strike the balance between being too cryptic (which can confuse users) or being too detailed (which may reveal more than intended). The messages should not reveal the methods that were used to determine the error. Attackers can use detailed information to refine or optimize their original attack, thereby increasing their chances of success. If errors must be captured in some detail, record them in log messages, but consider what could occur if the log messages can be viewed by attackers. Highly sensitive information such as passwords should never be saved to log files. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a user account exists or not. In the context of SQL Injection, error messages revealing the structure of a SQL query can help attackers tailor successful attack strings. (Effectiveness: N/A)

**•** Operation: Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth. (Effectiveness: Moderate)

**•** Operation: When using PHP, configure the application so that it does not use register\_globals. During implementation, develop the application so that it does not rely on this feature, but be wary of implementing a register\_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

**•** SQL (Class: None, Prevalence: Often)

## Demonstrative Examples

**•** N/A

**•** The query that this code intends to execute follows:

**•** If an attacker with the user name wiley enters the string:

**•** Where $user\_input is taken from an untrusted source.

**•** The programmer may have skipped any input validation on $id under the assumption that attackers cannot modify the cookie. However, this is easy to do with custom client code or even in the web browser.

**•** While the programmer applies an allowlist to the user input, it has shortcomings. First of all, the user is still allowed to provide hyphens, which are used as comment structures in SQL. If a user specifies "--" then the remainder of the statement will be treated as a comment, which may bypass security logic. Furthermore, the allowlist permits the apostrophe, which is also a data / command separator in SQL. If a user supplies a name with an apostrophe, they may be able to alter the structure of the whole statement and even change control flow of the program, possibly accessing or modifying confidential information. In this situation, both the hyphen and apostrophe are legitimate characters for a last name and permitting them is required. Instead, a programmer may want to use a prepared statement or apply an encoding routine to the input to prevent any data / directive misinterpretations.

## Notes

**•** Relationship: SQL injection can be resultant from special character mismanagement, MAID, or denylist/allowlist problems. It can be primary to authentication errors.