SQL Injection Mitigation SOP

# Development Mitigation SOP

SQL injection errors occur when: 1. Data enters a program from an untrusted source. 2. The data is used to dynamically construct a SQL query. This could allow an attacker to modify the statement’s meaning or to execute arbitrary SQL commands.

# Defense Against [DEFECT]

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmers intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but when user-supplied data needs to be included, they create bind parameters, which are placeholders for data that is subsequently inserted.

# Examples

…

String username = ctx.getAuthenticatedUserName();

String itemName = request.getParameter(“itemName”);

String query = “SELECT \* FROM items WHERE owner = ‘ “ + username + “’ AND itemname = ‘ “ + itemName + “’”;

ResultSet rs = stmt.execute(query);

…

## Explanation

The query that this code itends to execute follows:

SELECT \* FROM items

WHERE owner = <username>

AND itemname = <itemName>;

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string “name’ OR ‘a’=’a” for itemName, then the query beomes the following:

SELECT \* FROM items

WHERE owner = ‘wiley’

AND itemname = ‘name’ OR ‘a’ = ‘a’;

The addition of the OR ‘a’ = ‘a’ condition cause the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

SELECT \* FROM items;

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.

## Recommendation

Parameterized SQL statements are constructed using strings of regular SQL, but when user-supplied data needs to be included, they create bind parameters, which are placeholders for data that is subsequently inserted. Bind parameters allow the program to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for the value of each of the bind parameters, without the risk of the data being interpreted as commands.

…

String username = ctx.getAuthenticatedUserName();

String itemName = request.getParameter(“itemName”);

String query = “SELECT \* FROM items WHERE itemname = ? AND owner = ?;

PrepatedStatement smt = conn.prepareStatement(query);

Smt.setString(1, itemName);

Stmt.setString(2, username);

ResultSet results = stmt.execute();

…

## Example

public int syncChanges(ClaimStatusDim entity) {

…

String status – entity.getStatus();

String code = entity.getCode();

Int updated = getJdbcTemplate().update(getUpdateSql(), code, status, code);

…

**Resources**

1. [S. J. Friedl, SQL Injection Attacks by Example](https://www.symantec.com/connect/articles/sql-injection-and-oracle-part-one)
2. [P. Finnigan, SQL Injection and Oracle, Part One, Security Focus, 2002](https://www.symantec.com/connect/articles/sql-injection-and-oracle-part-one)