IDS00-J. Prevent SQL injection

SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Suppose a system authenticates users by issuing the following query to a SQL database. If the query returns any results, authentication succeeds; otherwise, authentication fails.

```
SELECT * FROM db_user WHERE username='<USERNAME>' AND
                            password='<PASSWORD>'
```

Suppose an attacker can substitute arbitrary strings for `<USERNAME>` and `<PASSWORD>`. In that case, the authentication mechanism can be bypassed by supplying the following `<USERNAME>` with an arbitrary password:

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

The authentication routine dynamically constructs the following query:

```
SELECT * FROM db_user WHERE username='validuser' OR '1'='1' AND password='<PASSWORD>'
```

If `validuser` is a valid user name, this `SELECT` statement yields the `validuser` record in the table. The password is never checked because `username='validuser'` is true; consequently, the items after the `OR` are not tested. As long as the components after the `OR` generate a syntactically correct SQL expression, the attacker is granted the access of `validuser`.

Similarly, an attacker could supply the following string for `<PASSWORD>` with an arbitrary username:

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

producing the following query:

```
SELECT * FROM db_user WHERE username='<USERNAME>' AND password='' OR '1'='1'
```

`'1'='1'` always evaluates to true, causing the query to yield every row in the database. In this scenario, the attacker would be authenticated without needing a valid username or password.

Noncompliant Code Example

This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a `char` array, the database connection is created, and then the passwords are hashed.

Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument `username` into the SQL command, allowing an attacker to inject `validuser' OR '1'='1`. The `password` argument cannot be used to attack this program because it is passed to the `hashPassword()` function, which also sanitizes the input.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new
            com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection =
      PropertyManager.getProperty("db.connection");
    // Can hold some value like
    // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // Create hash of password
  }

  public void doPrivilegedAction(String username, char[] password)
                                 throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // Handle error
    }
    try {
      String pwd = hashPassword(password);

      String sqlString = "SELECT * FROM db_user WHERE username = '"
                         + username +
                         "' AND password = '" + pwd + "'";
      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery(sqlString);

      if (!rs.next()) {
        throw new SecurityException(
          "User name or password incorrect"
        );
      }

      // Authenticated; proceed
    } finally {
      try {
        connection.close();
      } catch (SQLException x) {
        // Forward to handler
      }
    }
  }
}
```

Noncompliant Code Example (`PreparedStatement`)

The JDBC library provides an API for building SQL commands that sanitize untrusted data. The `java.sql.PreparedStatement` class properly escapes input strings, preventing SQL injection when used correctly. This code example modifies the `doPrivilegedAction()` method to use a `PreparedStatement` instead of `java.sql.Statement`. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument `username` into the prepared statement.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new
            com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection =
      PropertyManager.getProperty("db.connection");
    // Can hold some value like
    // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // Create hash of password
  }

  public void doPrivilegedAction(
    String username, char[] password
  ) throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // Handle error
    }
    try {
      String pwd = hashPassword(password);
      String sqlString = "select * from db_user where username=" +
        username + " and password =" + pwd;
      PreparedStatement stmt = connection.prepareStatement(sqlString);

      ResultSet rs = stmt.executeQuery();
      if (!rs.next()) {
        throw new SecurityException("User name or password incorrect");
      }

      // Authenticated; proceed
    } finally {
      try {
        connection.close();
      } catch (SQLException x) {
        // Forward to handler
      }
    }
  }
}
```

Compliant Solution (`PreparedStatement`)

This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the `username` argument, preventing an attacker from submitting an arbitrarily long user name.

```
public void doPrivilegedAction(
  String username, char[] password
) throws SQLException {
  Connection connection = getConnection();
  if (connection == null) {
    // Handle error
  }
  try {
    String pwd = hashPassword(password);

    // Validate username length
    if (username.length() > 8) {
      // Handle error
    }

    String sqlString =
      "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
    ResultSet rs = stmt.executeQuery();
    if (!rs.next()) {
      throw new SecurityException("User name or password incorrect");
    }

    // Authenticated; proceed
  } finally {
    try {
      connection.close();
    } catch (SQLException x) {
      // Forward to handler
    }
  }
}
```

Use the `set*()` methods of the `PreparedStatement` class to enforce strong type checking. This technique mitigates the SQL injection vulnerability because the input is properly escaped by automatic entrapment within double quotes. Note that prepared statements must be used even with queries that insert data into the database.

Risk Assessment

Failure to sanitize user input before processing or storing it can result in injection attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS00-J	High	Probable	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
CodeSonar	5.4p0	**FB.SECURITY.SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING** **FB.SECURITY.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE**	A prepared statement is generated from a nonconstant String Nonconstant string passed to execute method on an SQL statement
Coverity	7.5	**SQLI** **FB.SQL_PREPARED_STATEMENT_GENERATED_** **FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE**	Implemented
Findbugs	1.0	**SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE**	Implemented
Fortify	1.0	**HTTP_Response_Splitting** **SQL_Injection__Persistence** **SQL_Injection**	Implemented

Klocwork		**SV.DATA.BOUND** **SV.DATA.DB** **SV.HTTP_SPLIT** **SV.PATH** **SV.PATH.INJ** **SV.SQL**	Implemented
Parasoft Jtest	2020.2	**BD-SECURITY-TDSQL**	Protect against SQL injection
SonarQube	6.7	**S2077** **S3649**	Executing SQL queries is security-sensitive SQL queries should not be vulnerable to injection attacks

Related Vulnerabilities

CVE-2008-2370 describes a vulnerability in Apache Tomcat 4.1.0 through 4.1.37, 5.5.0 through 5.5.26, and 6.0.0 through 6.0.16. When a `RequestDispatcher` is used, Tomcat performs path normalization before removing the query string from the URI, which allows remote attackers to conduct directory traversal attacks and read arbitrary files via a `..` (dot dot) in a request parameter.

Related Guidelines

SEI CERT C Coding Standard	STR02-C. Sanitize data passed to complex subsystems
SEI CERT C++ Coding Standard	VOID STR02-CPP. Sanitize data passed to complex subsystems
SEI CERT Perl Coding Standard	IDS33-PL. Sanitize untrusted data passed across a trust boundary
ISO/IEC TR 24772:2013	Injection [RST]
MITRE CWE	CWE-116, Improper Encoding or Escaping of Output

Android Implementation Details

This rule uses Microsoft SQL Server as an example to show a database connection. However, on Android, `DatabaseHelper` from SQLite is used for a database connection. Because Android apps may receive untrusted data via network connections, the rule is applicable.

Bibliography

[OWASP 2005]	A Guide to Building Secure Web Applications and Web Services
[OWASP 2007]	OWASP Top 10 for Java EE
[Seacord 2015]	IDS00-J. Prevent SQL Injection LiveLesson
[W3C 2008]	Section 4.4.3, "Included If Validating"

← ↑ →