SQL-Identifier Injection Attacks

Cagri Cetin Dmitry Goldgof Jay Ligatti
Department of Computer Science and Engineering, University of South Florida
{cagricetin, goldgof, ligatti}@usf.edu

***Abstract*—This paper defines a class of SQL-injection attacks that are based on injecting identifiers, such as table and column names, into SQL statements. An automated analysis of GitHub shows that 15.7% of 120,412 posted Java source files contain code vulnerable to SQL-Identifier Injection Attacks (SQL-IDIAs). We have manually verified that some of the 18,939 Java files identified during the automated analysis are indeed vulnerable to SQL-IDIAs, including deployed Electronic Medical Record software for which SQL-IDIAs enable discovery of confidential patient information. Although prepared statements are the standard defense against SQL injection attacks, existing prepared-statement APIs do not protect against SQL-IDIAs. This paper therefore proposes and evaluates an extended prepared-statement API to protect against SQL-IDIAs.**

I. Introduction

Injection attacks such as SQL-Injection Attacks (SQLIAs) continue to be considered the most critical web-application vulnerabilities [1].

The following Java code provides a classic example of a SQLIA vulnerability.

```
String sql = "SELECT address FROM Customer
    WHERE password = '" + userInput + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

By entering an input such as `'OR true --`, an attacker can make the executed query be

```
SELECT address FROM Customer WHERE password =
    '' OR true --'
```

where `--` introduces a comment in SQL code and `OR true` bypasses the password check. Under this attack, the executed query returns all addresses in the Customer table.

A particularly problematic subclass of SQLIAs involves the injection of identifiers into SQL statements. We call such attacks SQL-Identifier Injection Attacks, or SQL-IDIAs. As far as we are aware, this paper is the first to specifically define and address SQL-IDIAs.

Identifiers may appear in SQL statements as, for example, names of tables, columns, indexes, databases, views, functions, procedures, or triggers. The following Java code demonstrates a SQL-IDIA vulnerability.

```
String sql = "SELECT * FROM Contact ORDER BY "
     + userInput;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

Here the untrusted user input injects an identifier, referring to a column name in the Contact table, according to which the results will be ordered. By entering an input such as

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic
    WHERE firstName='John' AND lastName='Doe')
     > 0 THEN Contact.lastName ELSE Contact.
    firstName END)
```

an attacker can observe the query results for the Contact table to infer whether John Doe appears in the Demographic table. As described in Section IV, we have successfully mounted such attacks against a deployed Electronic Medical Record web application to leak confidential information about patients.

SQL-IDIAs are of special interest because the existing standard SQLIA-preventing mechanisms—prepared statements—do not protect against SQL-IDIAs.

This paper makes the following contributions. It defines SQL-IDIAs based on concatenating, into SQL statements, identifiers that have propagated from untrusted inputs.

To understand the prevalence of SQL-IDIAs, we analyze 120,412 Java source files from GitHub. The analysis shows that 31.3% of the files are vulnerable to SQLIAs, and 15.7% of the files are vulnerable to SQL-IDIAs.

To demonstrate the impact of SQL-IDIAs, we mount an attack on deployed Electronic Medical Record software. This software uses prepared statements for injecting values like string and integer literals but concatenates identifiers directly into SQL statements. Our attack extracts confidential information, including the reason for a patient's doctor visit.

To prevent SQL-IDIAs, we introduce a new extended prepared-statement API. This new API enables prepared statements to fill placeholders with table and column names. Because the prepared statements restrict these placeholders to be filled only with valid table- and column-name identifiers, attackers are prevented from performing SQL-IDIAs, similarly to the way that standard prepared statements prevent attack injections into string- and integer-literal placeholders. Additional benefits of the new API include extending the safe use of prepared statements by enabling table and column names as parameters, not leaking schema information on invalid inputs, not having drawbacks that existing input-sanitation-based solutions have, and inducing less performance overhead compared to an ad hoc dynamic-whitelisting mechanism.

To the best of our knowledge, this is the first work that (1) defines SQL-IDIAs, (2) analyzes the usage of SQL statements in source files from GitHub, or (3) introduces an extended prepared-statement API for preventing SQL-IDIAs.

II. Prepared Statements

Prepared statements, also known as parameterized queries, are the de facto mechanism to prevent SQLIAs [2]. Figure 1

```
String sql = "SELECT address FROM Customer
    WHERE password = ?";
PreparedStatement stmt = conn.prepareStatement
    (sql);
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

Fig. 1: A Java program using prepared statements.

presents a program that employs prepared statements to perform database operations. At a high-level, preventing SQLIAs with prepared statements involves three main steps. First, an application creates a SQL-statement that has placeholders (i.e., the question mark symbol in the SQL statement) for literals and sends this statement to a database system. Then, when the prepare-statement function is executed, the database parses the statement and creates a statement structure having placeholders. Next, the application fills these placeholders (e.g., by calling the `setString` function) with values. This mechanism enforces that applications fill placeholders with literals, thus preventing SQLIAs.

A major limitation of prepared statements is that only application-level values are allowed to replace placeholders [3]. For example, for applications written in Java, placeholders in prepared statements may only be replaced by Java values such as string literals, integer literals, and Java objects.

This limitation to replace prepared-statement placeholders with only application-level values prevents safe construction of SQL statements having dynamically resolved identifiers [4], [5]. That is, standard libraries do not allow SQL identifiers such as table and column names to replace placeholders in prepared statements. SQL syntax allows identifiers to appear in many statements and clauses, including `create`, `alter`, and `drop` statements and `order by` and `group by` clauses.

Because standard libraries do not allow SQL identifiers to replace placeholders in prepared statements, developers must concatenate dynamically resolved identifiers into SQL statements, though such concatenations create SQL-IDIA vulnerabilities, like the one shown in Figure 2. In fact, prior work showed that every concatenation of untrusted input into a SQL statement can produce a SQLIA [6], [7].

III. SQL-Identifier Injection Attacks

This section defines and presents examples of SQL-IDIAs.

A. Definition of SQL-IDIAs

SQL-IDIA-vulnerable applications are applications that can form a valid SQL statement by concatenating a user-input identifier into the statement.

Definition 1. *An application is vulnerable to a* SQL-IDIA *iff the application constructs a SQL statement S by concatenating an untrusted input i into S and there exists an identifier x such that concatenating x into S in place of i causes S to be a valid SQL statement.*

For example, the application excerpted in Figure 2a is vulnerable to a SQL-IDIA because it can create a valid SQL

```
String sql = "SELECT * FROM Contact ORDER BY "
     + userInput;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

(a) An example Java program vulnerable to SQL-IDIAs.

```
SELECT * FROM Contact ORDER BY firstName
```

(b) The output SQL program when the `userInput` is a valid column name.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic
    WHERE firstName='John' AND lastName='Doe')
     > 0 THEN Contact.lastName ELSE Contact.
    firstName END)
```

(c) A malicious input through the `userInput` to perform a SQL-IDIA.

Fig. 2: An order-by-based SQL-IDIA.

statement by concatenating an identifier into the statement, as shown in Figure 2b.

A SQL-IDIA occurs when a SQL-IDIA-vulnerable application—which would produce a valid SQL statement by concatenating a user-input identifier into the statement—instead concatenates a non-identifer, or an invalid identifier, into the statement in place of a valid identifier.

Definition 2. *A* SQL-IDIA *occurs in a SQL-IDIA-vulnerable application iff the concatenated input i either is not an identifier or is an identifier that, when concatenated into S, makes S an invalid SQL statement.*

For example, a SQL-IDIA occurs when the SQL-IDIA-vulnerable application excerpted in Figure 2a is provided the input shown in Figure 2c. In this case the untrusted input (Figure 2c) is concatenated into the output SQL statement at a position in which an identifier could be valid, yet the untrusted input is not a valid identifier; hence, a SQL-IDIA has occurred.

Definition 2 also considers invalid-identifier injections to be SQL-IDIAs because such injections can leak sensitive database-schema information [8], [9]. For example, an attacker might input a nonexistent column name into the application shown in Figure 2a to cause the DBMS (Database Management System) to raise an exception when executing the generated invalid SQL statement. As with traditional SQLIAs, in cases in which the DBMS raises an exception, the application may output information contained in the exception object to leak database schema such as the database name or the SQL statement being executed.

Although this paper focuses on SQL, identifier-injection attacks are possible in other languages such as XML.

B. Additional Examples

To provide additional familiarity with SQL-IDIAs, we next consider two additional examples, shown in Figures 3 and 4. Both of these examples, as well as the example shown in Figure 2, are abbreviated and simplified versions of actual

```
String sql = "SELECT * FROM Customer WHERE " +
     userInput1 + " BETWEEN ? AND ?";
PreparedStatement stmt = conn.prepareStatement
    (sql);
stmt.setInt(1, userInput2);
stmt.setInt(2, userInput3);
ResultSet rs = stmt.executeQuery();
```

(a) An example Java program vulnerable to SQL-IDIAs.

```
age BETWEEN ? AND ? UNION SELECT * FROM Admin
    --
```

(b) A malicious input through the `userInput1` to perform a SQL-IDIA.

```
SELECT * FROM Customer WHERE age BETWEEN ? AND
     ? UNION SELECT * FROM Admin -- BETWEEN ?
    AND ?
```

(c) The output SQL program with the malicious input (b).

Fig. 3: A column-name-based SQL-IDIA.

Java applications found by our automated GitHub analysis tool, described in Section IV.

Figure 3a shows a program that is vulnerable to a column-name-based SQL-IDIA. In this program, two user-inputs fill placeholders using prepared statements; therefore, SQLIAs are not possible through these inputs. However, a column-name parameter (i.e., `userInput1`) is concatenated into the SQL statement. In normal cases, this program expects the concatenated parameter to be a valid column name. However, attackers can perform SQL-IDIAs by injecting carefully crafted SQL statements. For instance, the program (Figure 3a) outputs the SQL code shown in Figure 3c with the malicious input shown in Figure 3b. This output program can maliciously return all entries from the Admin table (assuming the Customer and Admin tables have the same attributes). The malicious input is not a valid column in the Customer table, so Definition 2 correctly considers this input to be a SQL-IDIA.

Figure 4a shows a program that is vulnerable to a table-name-based SQL-IDIA. This program concatenates a table-name parameter (i.e., `userInput`) into a SQL statement and executes this statement using the standard JDBC (Java Database Connectivity) execute-update function. If an attacker injects the malicious input presented in Figure 4b through the table-name parameter, the program (Figure 4a) outputs the two consecutive SQL statements shown in Figure 4c. These statements cause two different attacks. The first attack adds a new user to the Customer table as an administrator by changing the hardcoded admin value. The second attack—an example of piggy-backing attacks—deletes all entries from the Admin table; the malicious input presented in Figure 4b causes the Java program to execute multiple queries at once.

The existing JDBC API attempts to mitigate piggy-backing attacks by requiring the execute-update function to only execute one SQL statement at a time. The API provides

```
String sql = "INSERT INTO " + userInput + "
    (isAdmin) VALUES ('False')";
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

(a) A SQL-IDIA-vulnerable program through the table name.

```
Customer (name, isAdmin) VALUES ('Mallory',
    'True'); DELETE FROM Admin; --
```

(b) A malicious input through the `userInput` to perform SQL-IDIAs.

```
INSERT INTO Customer (name, isAdmin) VALUES
    ('Mallory', 'True'); DELETE FROM Admin; --
     (isAdmin) VALUES ('False')
```

(c) The output SQL program with the malicious input (b).

Fig. 4: A table-name-based SQL-IDIA.

different functions to execute multiple statements as a batch. However, in practice, some JDBC implementations do not faithfully follow the API specifications. We tested this SQL-IDIA with three different JDBC implementations: H2, SQLite, and MySQL JDBC drivers. Our results showed that these drivers, except the MySQL driver, are vulnerable to this attack. On the other hand, Definition 2 correctly classifies the input shown in Figure 4b as an attack because this input is not a valid table in the database.

IV. GitHub Analysis

This section presents an analysis of source files from GitHub and shows SQL-IDIAs on a deployed web application. This GitHub analysis investigates how SQL statements are constructed in practice and how many of the files are vulnerable to SQLIAs and SQL-IDIAs.

A. Research Questions

To understand the prevalence of SQLIAs and SQL-IDIAs, we ask the following research questions:

- $\mathbf{R_1}$ What percentage of source files use prepared statements, string concatenation, or hardcoded strings for constructing SQL statements?
- $\mathbf{R_2}$ What percentage of source files use both prepared statements and string concatenation in the same file?
- $\mathbf{R_3}$ What percentage of source files use identifier concatenation for constructing SQL statements?
- $\mathbf{R_4}$ What type of identifiers are the most commonly concatenated?

Our GitHub analysis provides empirical answers to these questions.

B. Dataset Collection

We collected a dataset of source files from GitHub, which is the most popular platform to publish open source projects. We used Java source files because Java is one of the most commonly used programming languages [10]. We used GitHub

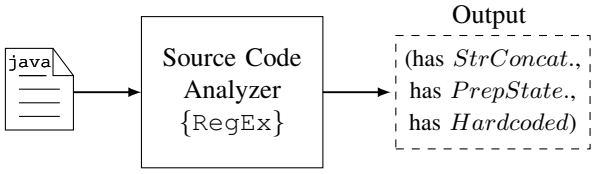


Fig. 5: The operation of the source file analyzer.

Archive [11] because the GitHub website provides limited access to all source files. GitHub Archive is a public database that collects all public GitHub activities (e.g., source files, commits, pull requests) since 2011.

Each file in our dataset contains SQL statements. To determine the files that contain SQL statements, we filtered GitHub Archive with certain keywords (i.e., "executeQuery", "executeUpdate", "createQuery", and "createNativeQuery"). These keywords are used to execute SQL statements in Java. It is noteworthy that these keywords are the functions of the well-known database libraries [12]. To minimize redundancy and false positives, we only considered parent projects (i.e., the projects that are not forked from other projects) and eliminated unit-test files. Our final dataset contains 120,412 Java source files, obtained by filtering 56.7 million Java files.

C. Identifying SQL Usages

We developed an automated tool to determine SQL usages in each source file. Figure 5 depicts the source file analysis operation using this automated tool. As shown in the figure, this tool takes an individual Java source file as an input, matches regular expressions to identify how SQL statements are constructed in the file, and outputs SQL-construction types in three categories. The first category is the dynamic SQL-statement construction using string concatenation (e.g., `"SELECT * FROM table WHERE id=" + userInput`). The second category is the dynamic SQL statement construction with prepared statements (e.g., `"SELECT * FROM table WHERE id=?"`). The last category is the static SQL-statement usage with hardcoded string literals (e.g., `"SELECT * FROM table WHERE id=5"`).

These SQL-construction categories are detected with rules that are encoded in regular expressions. If one of the following rules is matched, a file is classified as belonging to the string concatenation category.

- Concatenating a string literal with a Java identifier (e.g., a variable name) using the plus operator, where the string literal contains SQL keywords. For example, this rule matches `"SELECT * FROM " + userInput`.
- Using at least two append functions, where at least one of the append function takes a Java identifier as an argument and the other takes a string literal containing SQL keywords. For example, this rule matches `append ("SELECT * FROM ").append(userInput)`.
- Using a Java string-format function that contains at least one string literal with SQL keywords and string-format placeholders. For example, this rule matches `String.format ( "SELECT * FROM %s", userInput)`.

To determine prepared-statement usage, we encoded a rule that matches a string literal containing SQL keywords and prepared-statement placeholders (i.e., `?` and `:placeholder_name`). A file is considered as using hardcoded SQL-statements if the file (1) does not contain a string concatenation to form SQL statements, (2) does not contain prepared statements, and (3) only contains string literals with SQL keywords.

These SQL-construction categories determine the files that are vulnerable to SQLIAs and SQL-IDIAs. Assume that the concatenated variables are propagated from untrusted inputs. Then, the source file is considered as vulnerable if it uses at least one string concatenation to construct SQL statements [6], [7]. If a source file does not use string concatenation and employs prepared statements and/or hardcoded SQL-statements, then the file is considered as not vulnerable.

D. Empirical Results

This section discusses the prevalence of SQLIAs and SQL-IDIAs by empirically answering each of the research questions.

$\mathbf{R_1}$ *What percentage of source files use prepared statements, string concatenation, or hardcoded strings for constructing SQL statements?*

Figure 6a summarizes the SQL-usage statistics of 120,412 Java source files. 30.3% of the files construct SQL statements using string concatenation, 30.6% of the files employ prepared statements to form SQL statements, and 22.4% of the files only use hardcoded SQL statements. These results reveal that a significant portion of source files are vulnerable to SQLIAs through string concatenation.

$\mathbf{R_2}$ *What percentage of source files use both prepared statements and string concatenation in the same file?*

Interestingly, 16.7% of the files have both string concatenation and prepared statements. The following answer to R_3 shows that one reason for using both string concatenation and prepared statements in the same file is the identifier limitation of the existing prepared-statement API.

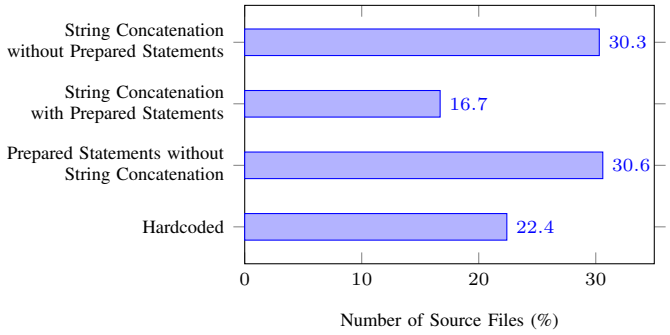
$\mathbf{R_3}$ *What percentage of source files use identifier concatenation for constructing SQL statements?*

Based on the GitHub analysis, 9.6% of the files use only identifier concatenation to form SQL statements. Additionally, 6.1% of the files concatenate identifiers and also employ prepared statements for values in the same file.

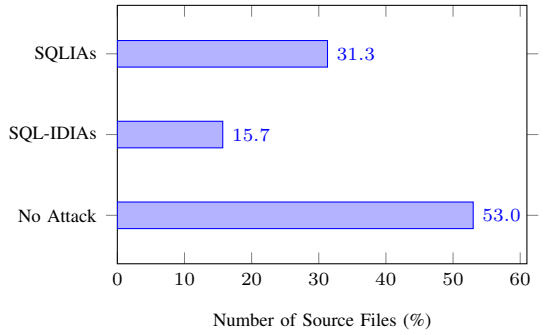
$\mathbf{R_4}$ *What type of identifiers are the most commonly concatenated?*

We analyzed the types of identifiers (e.g., table, column, index, function, procedure names) being concatenated. Based on the analysis, 96% of the identifiers were table and column names.

As a result, the GitHub analysis revealed that identifier concatenation is a real problem. As shown in Figure 6b, 15.7% of the files are vulnerable to SQL-IDIAs. In addition,



(a) SQL-construction statistics of 120,412 Java source files.



(b) Injection-attack vulnerabilities of 120,412 Java source files.

Fig. 6: SQL-construction and injection-attack vulnerability statistics of GitHub Java source files that contain SQL statements.

31.3% of the files are vulnerable to SQLIAs through string concatenation.

E. Attacking a Deployed Application

We present SQL-IDIAs on a large-scale Electronic Medical Record software. As of March 17, 2019, this open-source software is actively maintained and has 21,844 total commits, 61 contributors, and 23 stars at GitHub. Features of this software include managing confidential patient and medication records, scheduling appointments, and managing hospital-related tasks. Our GitHub analysis showed that a Java file in this software concatenates an identifier to form a SQL statement. By manually inspecting the source code, we were able to verify that this software is indeed vulnerable to SQL-IDIAs. We set up an attack environment by running this Electronic Medical Record software on a local computer and creating a test database. Although we have disclosed the identified vulnerability, we do not name the software here because it is widely deployed, and because the vulnerability has not been resolved (as of March 17, 2019).

The software has a web page used to search for employees in a clinic. Users can type a search keyword through this web page, and the software runs the code shown in abbreviated form in Figure 7a. The `keyword` parameter is used with prepared statements; therefore, a SQLIA is not possible through this parameter. However, this program takes the `userInput` parameter for the order-by clause from a hidden form in the web page. Thus, attackers can perform SQL-IDIAs through this order-by parameter by changing the source of the web page in a web browser.

In our first example attack, we injected the SQL expression shown in Figure 7b through the order-by parameter to determine whether a person named John Doe is a patient in the clinic. If John Doe appears in the Demographic table, the search result from the Contact table was sorted by the last name; otherwise, the result was sorted by the first name. This enabled us to determine that John Doe is a patient in the clinic.

To access other confidential information, we need to determine the unique identifier (i.e., `demographicNo`) of John Doe from the Demographic table. This identifier, which manages the relations between tables, can be extracted by injecting the code shown in Figure 7c. This code tests whether the unique identifier of John Doe is less than 5. Performing a binary search allowed us to determine the actual value (i.e., 1) of the unique identifier.

The extracted unique identifier of John can be used to obtain additional confidential information from other tables. For example, the code presented in Figure 7d maliciously detects John's doctor-visit reason. We determined, via binary search, that the reason code for John's doctor visit was 7.

```
String sql = "SELECT * FROM Contact WHERE
    lastName LIKE ? ORDER BY " + userInput;
PreparedStatement stmt = conn.prepareStatement
    (sql);
stmt.setString(1, keyword);
ResultSet rs = stmt.executeQuery();
```

(a) An abbreviated version of actual SQL-IDIA-vulnerable code in Electronic Medical Record software.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic
    WHERE firstName='John' AND lastName='Doe')
     > 0 THEN Contact.lastName ELSE Contact.
    firstName END)
```

(b) A malicious input through the `userInput` parameter to test the patient name in the database.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic
    WHERE firstName='John' AND lastName='Doe'
    AND demographicNo<5) > 0 THEN Contact.
    lastName ELSE Contact.firstName END)
```

(c) A malicious input through the `userInput` parameter to determine the unique identifier of the patient in the database.

```
(CASE WHEN (SELECT COUNT(*) FROM Appointment
    WHERE demographicNo=1 AND reasonCode<5) >
    0 THEN Contact.lastName ELSE Contact.
    firstName END)
```

(d) A malicious input through the `userInput` parameter to determine the patient's doctor-visit reason.

Fig. 7: SQL-IDIAs on Electronic Medical Record software.

With manual inspection of the reason codes in the open source project, we were able to determine that the visit was made for HIV testing.

This technique is not limited to the attacks above. The same technique can be used to extract different confidential information about patients such as medication history, laboratory test results, patient address, or the room that a patient is occupying.

F. Threats to Validity

Our GitHub-analysis methodology is based on pattern matching in single files. This approach relies on the assumption that concatenated variables are propagated from untrusted inputs. This assumption was made because determining the input source for concatenated variables would require techniques such as data flow analysis [13] and compiler optimization [14]. These techniques require source files to be compiled, thus they cannot be applied to our analysis. This limitation exists because our dataset only contains independent source files, not the whole projects and their dependencies.

To estimate the performance of our GitHub-analysis results, we randomly selected and manually inspected 200 source files. The false positive is determined when the tool says there is a string concatenation in the file, but in reality (1) the string concatenation is not used to form a SQL statement, (2) the concatenated variable is inside of the file (e.g., a SQL statement is concatenated with a static field), or (3) the concatenated variable is commented out. The false negative is determined when the tool says there is no string concatenation in the file, but there is actually at least one SQL statement that is created using string concatenation, and the concatenated variable is coming from outside of the source file. To estimate the accuracy, we calculated the proportion of true results (i.e., true positive and negative rates) in all results.

This methodology was used to estimate the performance of our analysis. The false positive rate, false negative rate, and the accuracy of our GitHub analysis are 18%, 7%, and 87%, respectively. The obtained results are promising. However, these results, similarly to existing research relying on GitHub data, may suffer from different threats as discussed in [15].

V. Extended Prepared-Statement API

To prevent SQL-IDIAs, we introduce a new extended prepared-statement API. We also demonstrate the implementability, efficiency, and effectiveness of the extended API by empirically evaluating a prototype implementation.

A. API Functions

To prevent SQL-IDIAs, we introduce the following two functions that can be added to prepared-statement APIs (e.g., Java JDBC and PHP PDO).

- `setColumnName(int parameterIndex, String columnName)`: takes a column name and its index as arguments
- `setTableName(int parameterIndex, String tableName)`: takes a table name and its index as arguments

A possible implementation of these functions consists of three main steps. First, these functions can be added to the prepared-statement API and its corresponding database driver (e.g., the MySQL JDBC Driver). The implementation of these new functions in this step is similar to the existing prepared-statement functions, such as `setString`. Typically, when these new functions are called, their parameters can be stored in an array with parameter indices. These indices indicate the placeholder positions in SQL statements.

Second, the SQL-statement preparation phase for identifiers can be implemented in the DBMS. The standard preparation phase contains two main steps: (1) parsing the SQL statement, and (2) generating an execution plan. The implementation of the parsing step may require changing the SQL syntax of the database in some cases. For example, the syntax does not need to be changed if the databases allow placeholders anywhere in the SQL-statement. The syntax has to be modified to allow placeholders for table and columns if the database syntax only allows certain clauses to have placeholders.

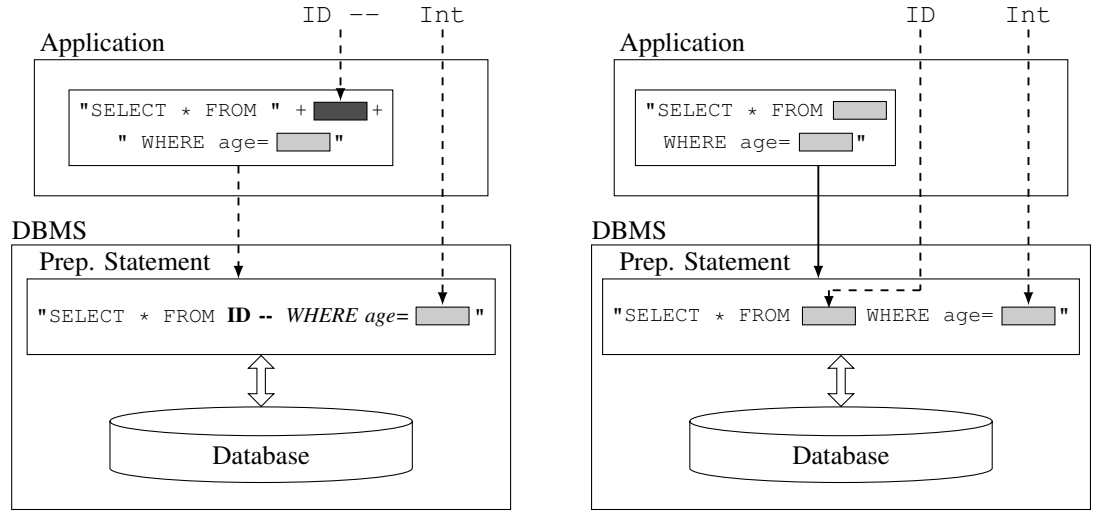
The execution-plan-generation step can include schema verification and statement optimization. In the schema verification, the DBMS checks whether the table and column names in the SQL-statements are valid. For example, given the statement `SELECT id FROM Customer`, the DBMS checks whether the Customer table is in the database and id is an attribute of the Customer table. Although the DBMS can still verify and optimize the non-parameterized table and column names in this step, the verification of parameterized table and column names must be performed while executing the prepared statement.

The last step can involve filling the placeholders with identifiers while executing the prepared statement. This step starts by checking whether dynamic identifiers belong to the schema. The checking operation is straightforward in the case of column names because the DBMS only needs to ensure that a given column belongs to an appropriate table. Dynamically checking table names requires further verification including verifying whether the table belongs to the schema as well as ensuring that the already existing attributes in the SQL statement belong to the given table. Once the verification is complete, the DBMS can create an expression for each parameterized identifier and place these expressions into the prepared statement.

We only considered table and column names in our extended API because our GitHub analysis showed that 96% of the concatenated identifiers were table and column names.

B. Benefits of the Extended API

An illustration of the systems that are vulnerable to SQL-IDIAs due to the usage of existing prepared-statement APIs is shown in Figure 8a. As can be seen, an application (1) takes literals and identifiers as inputs, (2) fills placeholders with literals using prepared statements, and (3) concatenates identifiers to construct SQL statements. The identifier concatenation causes applications to have SQL-IDIA vulnerabilities.



(a) SQL-IDIAs with the existing prepared-statement APIs.

(b) Preventing SQL-IDIAs with the extended prepared-statement API that has identifier capabilities.

Fig. 8: Preventing SQL-IDIAs with the extended prepared-statement API. Data propagated from untrusted inputs are illustrated with dashed arrows. The solid unidirectional arrow indicates the trusted data flow.

As illustrated in Figure 8b, the extended API prevents SQL-IDIAs by filling placeholders with identifiers using prepared statements. Applications can create placeholders for identifiers using the extended prepared-statement API, and the API only allows these placeholders to be filled with valid identifiers. Thus, attackers are not able to perform SQL-IDIAs.

The extended API prevents DBMSs from leaking sensitive schema information by performing a default operation when the input column or table name does not exist in the database. For example, if a parameterized column name is used in an order-by clause and the column name is invalid, the DBMS will order the results by the first column in the table. This operation prevents the information-leakage attack described in Section III.

In addition, these extended API functions do not suffer from the drawbacks of input-sanitation-based approaches. For example, as described in Section VI, incorrect updates to whitelists or blacklists may introduce false positives or false negatives. The extended API functions eliminate such false positives or negatives by dynamically verifying given table and column names in databases before filling placeholders.

C. Empirical Evaluation

A prototype of the extended prepared-statement API was implemented, and the implementation was compared with an existing equivalent prepared-statement function as well as ad hoc whitelisting solutions.

1) Implementation: We implemented a prototype of the `setColumnName` function into the H2 JDBC library [16]. H2 is an open-source relational database management system that is written in Java.

The implementation enables order-by clauses to have column names through the new `setColumnName` function. In our implementation, we have not modified H2's SQL syntax because it allows order-by clauses to have placeholders for values; in fact, order-by clauses can take numerical column indices as parameters with prepared statements.

```
String sql = "SELECT * FROM TestTable WHERE
    col2 < 100 ORDER BY ? ASC";
PreparedStatement stmt = conn.prepareStatement
    (sql);
stmt.setColumnName(1, userInput);
ResultSet rs = stmt.executeQuery();
```

Fig. 9: Usage of the new `setColumnName` function.

Figure 9 shows a program that employs the implemented `setColumnName` function. This program selects entries from a table and orders them by the given column name. At prepare time, when the prepare-statement function is executed, the H2 DBMS parses the SQL statement and creates a query structure having a placeholder for the order-by parameter. When the `setColumnName` is executed, the DBMS stores the column name parameter with its index in an array. Once the execute-query function is executed, the DBMS first validates the column name. If the given column name is invalid, i.e., does not belong to the table, the DBMS sorts the results by the first column in the table to prevent information-leakage attacks through error messages. If the column is valid, the DBMS (1) dynamically creates a column expression, (2) appends this expression to the query structure, and (3) executes the query.

2) Experimental Setup: We compared our `setColumnName` implementation with three different implementations: an existing prepared-statement function and two different ad hoc implementations. Our implementation executes the query shown in Figure 9, and the three other implementations execute equivalent queries. Hence, all of the implementations return the same result-set in the same order

Implementation	Execution Time (ms)		
	Same Input	Random Input	Bad Input
New `setColumnName`	2.11	2.25	2.04
Existing `setInt`	2.13	2.29	1.11
Static Whitelist	2.08	2.18	2.29
Dynamic Whitelist	2.37	4.73	4.07

TABLE I: Average execution times of the implementations over 100 runs.

when the input is the same.

To establish a baseline, we used the existing prepared-statement API's `setInt` function that takes an int-literal as a parameter. By filling the placeholder shown in Figure 9 with a column index using the `setInt` function, we were able to create an equivalent query with `setColumnName`. We could not use other existing prepared-statement functions because an equivalent query cannot be created with any other functions. We also compared our implementation with two different ad hoc solutions. The first ad hoc implementation uses a static-whitelist (i.e., a hash set that contains all column names in the table). The second ad hoc implementation employs a dynamic-whitelist by first querying whether the given column name exists in the database and then executing the actual query.

The `setColumnName` and `setInt` implementations prepare a statement once and execute the statement 100 times. The ad hoc implementations prepare and execute the statement 100 times because column names had to be concatenated into queries. In each execution, we measured the execution time, that is, the real-time. For the first two prepared-statement implementations, the real-time is measured from beginning to setting placeholders and executing the query until finishing obtaining a result-set from the database. For the ad hoc implementations, the real-time is measured from beginning to preparing a statement and executing the query until finishing obtaining a result-set from the database.

We tested all four implementations using a uniform environment. The testing database has a table that contains 100 columns and 1000 rows. Each cell of the table was filled with a random number between 0 and 1000. These random numbers were generated using the standard Java random number library. We used the H2 DBMS to implement the database-relevant operations. All experiments were performed on a MacBook Pro laptop that runs macOS Sierra version 10.12.6 with 16GB of memory and a 2.2GHz Intel quad-core i7 processor.

We conducted three sets of experiments to test the performance of the implementations. In the first experiment, each implementation was given the same column name or column index. In the second experiment, a randomly chosen valid column name or index was given to each implementation in each run, to eliminate caching. In the last experiment, each implementation was given a "bad" input, meaning a randomly chosen column that is not an attribute of the table.

3) Experimental Results: Table I summarizes the performance results of the four implementations. Our implementation has no extra performance overhead over the existing prepared-statement `setInt` function when the input is the same or a random input is provided. For the bad inputs, `setInt` outperformed `setColumnName` because `setInt` does not retrieve a result set from the table and instead throws an exception containing sensitive schema information. In contrast, our implementation returns a result set that is sorted by the first column in the table to prevent information-leakage attacks.

In all experiments, the new `setColumnName` function outperformed the dynamic-whitelist implementation. The static-whitelist implementation slightly outperformed the `setColumnName` function in two experiments. Although this ad hoc approach has a slight performance advantage, whitelisting approaches may introduce nontrivial complexities into application code and may lead to false positives (see Section VI).

To test the effectiveness of our implementation, we mounted the order-by-based SQL-IDIAs described in Figures 2 and 7, and the information-leakage attack described in Section III. The new `setColumnName` function successfully prevented all of these attacks.

To summarize, filling placeholders with column names (1) is practical and efficient as compared to the existing ad hoc approaches, (2) does not introduce extra performance overheads as compared to the existing prepared-statement functions, and (3) is effective against SQL-IDIAs.

VI. Related Work

Due to the popularity of SQLIAs, several dynamic and static analysis methods have been proposed. Dynamic methods (e.g., [6], [7]) and tools (e.g., [17], [18], [19]) aim to mitigate injection attacks at runtime. However, none of them are widely adopted at present due to high performance overheads.

Static analysis tools (e.g., [20], [21], [22]) are also not widely adopted due to high false positives [23]. These false positives result, for example, from imprecision in the information-flow analyses used to determine how untrusted inputs get concatenated into SQL-statement outputs.

Input sanitation is a more common technique for mitigating SQLIAs. Input sanitation may entail whitelisting valid inputs, blacklisting invalid inputs, or escaping special characters [2]. All of these techniques have well-documented drawbacks, including:

- Whitelists and blacklists may introduce nontrivial complexities into application code. For example, creating a new database table may require dynamically changing a whitelist or blacklist of valid table names usable within SQL statements. In addition, incorrect or delayed dynamic updates to whitelists or blacklists may introduce false negatives or positives [24], [3].
- When escaping special characters, some unexpectedly encoded characters may not be properly recognized and escaped, creating false negatives [25]. Examples include SQL smuggling [26], character homoglyph injection [27], and string literal injection without quotes [28].
- Escaping special characters may also introduce SQLIA vulnerabilities. For example, escaping single quotes

in input strings (e.g., converting `\';[code]--` to `\'';[code]--`) may cause an application to output the SQLIA-exhibiting statement `DELETE FROM table WHERE name='\'';[code]--'` [25].
- Applications may be vulnerable to second-order injection attacks when a sanitized input is stored in a database and the stored input is reused without sanitation [28].

Due to these drawbacks, prepared statements are the standard defense against SQLIAs [3].

There have been efforts to build automatic prepared-statement-generation tools [29], [30], [31]. These tools analyze source code and convert concatenated SQL-statements to prepared statements. Although some of the SQLIAs are prevented, none of these tools can prevent SQL-IDIAs due to the fact that prepared statements cannot fill placeholders with identifiers. Utilizing the proposed extended prepared-statement API can enable these tools to prevent SQL-IDIAs.

VII. Conclusions

This paper has defined SQL-IDIAs and demonstrated example SQL-IDIAs on deployed software. To prevent SQL-IDIAs, a new extended prepared-statement API was proposed. This API

- extends the safe use of prepared statements by filling placeholders with table and column names,
- prevents SQL-IDIAs,
- does not leak schema information on invalid inputs,
- does not have drawbacks that existing input-sanitation-based solutions have,
- has been prototyped, and found to perform efficiently and effectively, and
- can be utilized by the existing automatic prepared-statement-generation tools.

The prevalence of SQL-IDIAs was determined by GitHub SQL-construction analysis. The GitHub analysis showed that 15.7% of the SQL-constructing Java source files considered are vulnerable to SQL-IDIAs. These SQL-IDIA vulnerabilities can be prevented with successful adoption of the proposed extended prepared-statement API.

References

[1] OWASP. Owasp top 10 – 2017 The ten most critical web application security risks. 2017. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[2] The open web application security project (OWASP): Sql injection prevention cheat sheet. 2018. [Online]. Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[3] J. Clarke-Salt, *SQL Injection Attacks and Defense*. Elsevier, 2009.

[4] Oracle Community: PreparedStatement and order by. 2003. [Online]. Available: https://community.oracle.com/thread/1340727

[5] Using prepared statements to set table name. 2009. [Online]. Available: https://stackoverflow.com/a/1208477/701325

[6] D. Ray and J. Ligatti, "Defining injection attacks," in *Proceedings of the 17th International Information Security Conference*, 2014, pp. 425–441.

[7] ——, "Defining code-injection attacks," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 2012, pp. 179–190.

[8] B. Smith, L. Williams, and A. Austin, "Idea: Using system level testing for revealing SQL injection-related error message information leaks," in *Proceedings of the International Symposium on Engineering Secure Software and Systems*, 2010, pp. 192–200.

[9] D. Litchfield. Web application disassembly with ODBC error messages. 2001. [Online]. Available: http://www.davidlitchfield.com/WebApplicationDisassemblywithODBCErrorMessages.pdf

[10] T. software BV. Tiobe programming community index. 2018. [Online]. Available: https://www.tiobe.com/tiobe-index/

[11] GitHub Archive. 2018. [Online]. Available: https://www.gharchive.org/

[12] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in Java projects," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 551–555.

[13] A. Sanyal, B. Sathe, and U. Khedker, *Data flow analysis: theory and practice*. CRC Press, 2009.

[14] S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann Publishers, 1997.

[15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the ACM 11th Working Conference on Mining Software Repositories*, 2014, pp. 92–101.

[16] H2 Java SQL Database JDBC Driver. 2018. [Online]. Available: https://github.com/h2database/h2database

[17] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 1181–1192.

[18] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: preventing SQL injection attacks using dynamic candidate evaluations," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 12–24.

[19] W. G. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *Proceedings of the ACM International Conference on Automated Software Engineering*, 2005, pp. 174–183.

[20] C. Nagy and A. Cleve, "A static code smell detector for SQL queries embedded in java code," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, 2017, pp. 147–152.

[21] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," in *IEEE Software*, vol. 25, no. 5, 2008, pp. 22–29.

[22] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in *Proceedings of the USENIX Security Symposium*, vol. 14, 2005.

[23] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the IEEE International Conference on Software Engineering*, 2013, pp. 672–681.

[24] L. K. Shar and H. B. K. Tan, "Defending against cross-site scripting attacks," in *IEEE Computer*, vol. 45, no. 3, 2012, pp. 55–62.

[25] S. Friedl. SQL injection attacks by example. 2017. [Online]. Available: http://unixwiz.net/techtips/sql-injection.html

[26] A. Douglen. Sql smuggling or, the attack that wasn't there, Comsec Consulting Research. 2007. [Online]. Available: http://www.it-docs.net/ddata/4954.pdf

[27] A. Ginsberg and C. Yu, "Rapid homoglyph prediction and detection," in *Proceedings of the IEEE International Conference on Data Intelligence and Security*, 2018, pp. 17–23.

[28] C. Anley. Advanced SQL injection in SQL server applications, an NGSSoftware Insight Security Research Publication. 2002. [Online]. Available: https://crypto.stanford.edu/cs155old/cs155-spring06/sql_injection.pdf

[29] P. Bisht, A. P. Sistla, and V. Venkatakrishnan, "Automatically preparing safe SQL queries," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2010, pp. 272–288.

[30] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities," in *Information and Software Technology*, vol. 51, no. 3, 2009, pp. 589–598.

[31] S. Thomas and L. Williams, "Using automated fix generation to secure SQL statements," in *Proceedings of the Third IEEE International Workshop on Software Engineering for Secure Systems*, 2007, p. 9.