Finding Injection Vulnerabilities in Server-side SQL Libraries using Symbolic Execution

Kangqi Ni, Xiangyu Li
Georgia Institute of Technology, USA, {kangqi.ni, xiangyu.li}@cc.gatech.edu

***Abstract*—To be written.**

I. Introduction

SQL injection vulnerabilities used to be a prevalent, and serious security threat to web applications. Because of insufficient validation and sanitization of untrusted user input, attackers could trick the server side code to execute in unintended ways by using carefully constructed inputs, causing unauthorized access to sensitive data.

Most SQL injection attacks happen when the user inputs include strings. These string from user inputs are typically intended to be used only as string values, instead of SQL commands. However, if the string provided by the user could potentially break out of its string context, part of the user input would be used as SQL keywords, violating the intended semantic of the SQL query.

Consider the following simple SQL query:

```
SELECT * FROM login
WHERE username = 'name';
```

where 'name' is an input from the user. If the user specified name to be "fake-user' OR 1 = 1;", the query result would return everything in the login table regardless of username. This is because the single quotation mark in the user-specified string broke out of the string context of the SQL query, and thus the "OR 1 = 1;" changed the semantic of the where clause. To prevent this, the server side need to perform appropriate sanitization of user inputs, escaping the single quotation character, which could break out of the string literal context.

Recently, the threat of SQL injection vulnerabilities is greatly mitigated because the widely use of higher-level SQL libraries instead of constructing the raw query statement by the application itself and sending it to the SQL engine directly. The SQL libraries are typically well written and tested in terms of sanitizing user inputs and preventing SQL injections.

However, in case that the SQL library being used did not perform sufficient inputs sanitization and thus has SQL injection vulnerabilities, the consequence is severe. Application developers usually trust SQL library not to contain this kind of issues and will rely on the library code to perform anti-sql-injection processing for them. Such vulnerabilities are unlikely to be discovered by application developers. Also, a SQL library is potentially used by large population of websites. The security threat has more impact than the security vulnerabilities in one particular website. Moreover, if the vulnerable SQL library is open-source, the attackers would have full knowledge of the bug and could device sophisticated ways to exploit it. Thus, during the development of a SQL library, the developers of the library code wanted to make sure that there is absolutely no such vulnerabilities in the library code.

In this work, we present a technique that detects potential SQL injection vulnerabilities in SQL library code using symbolic execution. The technique works on the SQL-library-API level. In particular, it works on SQL-library APIs that produces valid SQL queries and checks whether the resulting SQL query could potentially expose injection vulnerability, for any arbitrary user inputs. Given an API method in the SQL library, the technique executes the API method symbolically, with the user inputs to the API method being marked as symbolic values. It tracks how the user inputs are used to produce the final SQL query and uses contraint solvers to determine whether there could be any user inputs that manifests a SQL injection attack. We implemented the technique in Python and the implementation works on Python programs.

We conducted an empirical study on the proposed techinque. The subjects we picked should 1) be purely in Python; 2) produces the full SQL query statement in an intermediate step or as the final output. According to these criteria, we picked a real-world SQL library, Python-SQL, as our subject. We also wrote a simple, small SQL library as a benchmark to assess the SQL-injection vulnerability detection capability of our technique. As a note, initially, we hoped that this technique could be used to verify more widely-used SQL libraries such as sqlite on Python, and SQLAlchemy. However, after investigating these subjects, we found that these libraries does not produce the full SQL query. Instead, the logic of executing the SQL query is scattered in the code. Thus, in principle, our approach of finding SQL-injection vulnerabilities does not work on these SQL libraries.

The contribution of the work includes:

- The formulation of a symbolic-execution-based approach that detects SQL injection vulnerabilities in server-side SQL libraries.
- An implementation of the approach that works on SQL libraries written in Python.
- An empirical study that shows the effectiveness of the approach in discovering SQL-injection vulnerabilities.

II. Our Approach and Implementation

A. Overview

Our proposed technique works on API method level. Given an API method of the SQL library, it verifies that no user input to the API method could possibly break out of its intended string context and be executed as SQL keywords. First, the method is executed symbolically, with the input parameters marked as symbolic values. The symbolic execution engine explores as many program paths as possible in a breadth-first manner, until the resource limitation is reached. For each program path it explored, it records the symbolic expression of the resulting final SQL query statement, which is expressed in terms of the symbolic inputs, as well as the path condition from the starting of the symbolic execution to the point where the final SQL query is generated. The tool then uses the symbolic expression of the SQL query and the corresponding path condition of that particular program path to construct the condition, that there are some input being able to break out of its string context and executed as SQL keywords, which in turn indicates a potential SQL injection vulnerability. This condition is sent to the constraint solver to determine its satisfiability. If the condition is satisfiable, it indicates that the vulnerability is present. In this case, the tool also gives an example input that trigger this vulnerability.

We implemented this technique in Python. It consists of the symbolic execution engine, the SQL injection condition generator, and an off-the-shelf constraint solver. The current implementation works for SQL libraries that are written purely in Python, but the approach is generalizable to other languages.

In the remaining part of this section, we'll talk about the implementation of each component of our implementation in details.

B. Symbolic Execution

The symbolic execution engine is implemented in Python, without modification to the Python interpreter or runtime. Strictly, it is implemented as a concolic execution engine that evaluates the program both concretely and symbolically at the same time.

The concolic execution engine is implemented by extending primitive types in Python. The overloaded types, also referred to as concolic types, not only represents the concrete value of the original type, but also keeps track of the symbolic representation of its value, in terms of the symbolic inputs. The current implementation supports concolic int type and concolic string type, which are considered relavent to SQL injection vulnerabilities. Floating point numbers are not tracked symbolically because, in contrary to integer values, which could be used in string-indexing-related operations, they are unlikely to trigger injection vulnerabilities. Array index that depends on symbolic inputs are not tracked either because the string input from the user has to flow into the SQL query to trigger any potential injection vulnerability anyway.

The concolic types track the symbolic expression by adding a attribute to the instance of the original primitive Python type, that points to expression tree of the current concolic value. The concolic types override the common operations of the original primitive types. When the overridden operation is performed on the concolic value, it calculates the concrete result as well as the new symbolic expression associated with the result, which is typically also a concolic type value.

The concolic integer type supports all the built-in operations on integers. One special case is the comparison operation. Comparison of integer type values yields bool type values in Python. However, the bool type cannot be extended according to the language specification. We handle this by just adding its value to the constraint whenever we get a concolic bool value.

The concolic string type partially supports string operations in Python. If the operation is not supported by the concolic type specifically, the program will just treat the string value as an ordinary string and keeps executing. However, in this case, the symbolic information associated with the concolic string would be lost. The set of operations supported by the concolic string type is actually determined by the operations that can be solved by the string constraint solver we used. This is because, although we can represent all string operations easily, but the resulting symbolic expression would not be useful if it cannot be understood by the string constraint solver.

The concolic execution engine starts from a default set of value assignments of the symbolic variables. It evaluates the program concolically until the program finishes and collects the path condition. Then it systematically negates each clause in the path condition. For each new path condition generated in this way, it uses the constraint solver to determine whether it is satiefiable and comes up with an value assignments if it is. The concolic execution engine executes the program with the new value assignments and continue this process iteratively, until all program paths are explored or the resource limit is reached.