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

Kangqi Ni, Xiangyu Li
Georgia Institute of Technology, USA, {kni3, 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

III. Evaluation

IV. Discussion

Constraint solving plays an important role in web program analysis for the purpose of test generation for coverage, bug finding and vulnerability detection. The reason is that solver-based analysis tools enable more precise analysis with the ability to generate intersting bug-revealing inputs. Furthermore, solver-based analysis tools are often more robust and easier to build than otherwise. However, many string solvers such as HAMPI [2], DPRLE [1], and REX [3] support only string operations. Such logics are not sufficient expressive for many program analysis since non-string operations are also widely used in SQL query checking. More importantly, the string and non-string operations interact in subtle ways leading to program errors that are hard for humans to find without automation. Finally, a string-only analysis will likely miss pure integer or string-to-integer constraints (e.g., length of string) thus resulting in path constraints that are not precise enough, leading to false positive. Therefore, we adopt Z3-str, a satisfiability solver that supports a rich combined logic over string and non-string operations aimed at symbolic, static and dynamic analysis, in our work.

V. Conclusion

In summary, we have made following contributions:

- We discovered certain internal mechanism widely adopted by some python SQL libraries, which prevents from exposing complete SQL query at the library level.
- We revealed the current limtation on state-of-art string solvers.
- We designed our own SQL library benchmark to capture common cases in practice.
- We proposed a practical SQL-injection condition encoding scheme.

References

[1] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 188–198, New York, NY, USA, 2009. ACM.

[2] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[3] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.