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

Kangqi Ni Xiangyu Li Taesoo Kim
Georgia Institute of Technology

Overview

- Workflow
  - Using symbolic execution to explore program paths systematically, with user input marked as symbolic values
  - For each path explored, extract the symbolic expression of the resulting query statement and the path condition.
  - There is a potential vulnerability if the query could potentially contain sensitive characters that are from user input.

Symbolic query representation

Query statement:

```
SELECT * from login where user = '<sym_username>'
```

is represented as:

```
(Concat (Concat "SELECT * from login where user = '"
(Replace sym_username "'" "''") "'")
```

where the variable sym_username is from user input

SQL Injection Condition

Query potentially contains (unescaped) single quote originated from user inputs, allowing breaking out of string contexts.

- SQL injection condition construction
  - remove single quote literals added by the library code from the symbolic expression of the query string
  - remove escaped single quotes from the query string
  - injection condition := (the processed query string could still possibly contain single quotes && pc)

The SQL injection condition is sent to the constraint solver to determine its satisfiability.

Example

```
(Concat (Concat "SELECT * from login where user = '"
(Replace sym_username "'" "''") "'")
```

Injection condition is constructed as:

```
(Contains (Replace (Concat (Concat "SELECT * from login
where user = " (Replace sym_username "'" "''") "") "'''
"") "'")
&&
path_condition
```

Example

```
(Concat (Concat "SELECT * from login where user = '"
(Replace sym_username "'" "''") "'")
```

Injection condition is constructed as:

```
(Contains (Replace (Concat (Concat "SELECT * from login
where user = " (Replace sym_username "'" "''") "") "'"
"") "'")
&&
path_condition
```

Implementation

- Symbolic(concolic) execution engine
  - adapted from the symbolic execution engine in Commuter*
  - performs simultaneous concrete/symbolic execution
  - works for Python programs
  - supports symbolic operations on integers and strings

Commuter: https://github.com/aclements/commuter

Implementation

- Constraint solver
  - Z3 by Microsoft
  - with Z3-str extension from Purdue University to support string operations

Experiment

- **sqlalchemy 0.9.8** ✗
  - http://www.sqlalchemy.org/
  - SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL
- **sqlite3** ✗
  - it provides a SQL interface compliant with the DB-API 2.0 specification

Not able to get complete query due to internal mechanism on prepared statements

Experiment Cont.

- **Python-sql 0.4**
  - https://code.google.com/p/python-sql/
  - python-sql is a library to write SQL queries in a pythonic way

- **Our python sql library**
  - it is a library to do sanity on user inputs by escaping apostrophes

Test Case

- **Python-sql**
  - simple selects
    - 4 test cases
  - select with where condition
    - 3 test cases
  - select with join
    - 1 test case
  - https://pypi.python.org/pypi/python-sql
    - symoblize all the user-defined strings in the query

```
>>> user = Table('user')
>>> select = user.select()
>>> tuple(select)
('SELECT * FROM "user" AS "a"', ())

>>> select = user.select(user.name)
>>> tuple(select)
('SELECT "a"."name" FROM "user" AS "a"', ())
```

Test Case Cont.

- **Our python sql library**
  - simple select
    - 1 test case
    - symbolize table name and column field
  - select with where condition
    - 1 test case
    - symbolize table name, column field and where clause

Example

```
def sanity(self, raw_str):
    sanity_str = raw_str
    if len(raw_str) < 50:
        replace(sanity_str,
                '"', '')
        replace(sanity_str,
                "'", "")
    return sanity_str
```

SQL injection condition:

```
And(Contains(Concat(Concat(Concat
(Concat("SELECT ", Replace
(sym_colname, "'", "")), " FROM "),
sym_tname), ""), "'"), And(And(And
(And(And((Length(sym_tname) < 50) ==
False, (Length(sym_tname) > 50) ==
False), (Length(sym_colname) < 50)
== True), Contains(sym_colname, """)
== False), Contains(sym_colname,
"'") == True), Contains(Replace
(sym_colname, "'", ""), "'") ==
False))
```

Limitation

- **String Solver**
  - Z3-str (FSE’13)
    - **Pro**: support both string and non-string operations
    - **Con**: replace operation is not powerful
      - replace the first occurrence
      - timeout when multiple replace operations are used
  - HAMPI (ISSTA’09)
  - DPRLE (PLDI’09)
  - Rex (ICST’10)

Limitation cont.

```
# Z3-str input file

(declare-variable t String)

(assert (Contains (Replace (Replace t "a" "" ) "a" "" )
"a") )

(check-sat)
(get-model)
```

Always Timeout!

Conclusion

- **Findings**:
  - it is hard to capture complete sql query in some python sql libraries
  - limitations on the state-of-art string solver
- **Designings**:
  - our sql library benchmark
  - sql injection condition encoding
- **Useful when string solver gets improved**