TAPS: Automatically Preparing Safe SQL Queries

Prithvi Bisht
University of Illinois at Chicago
Chicago, Illinois, USA
pbisht@cs.uic.edu

A. Prasad Sistla
University of Illinois at Chicago
Chicago, Illinois, USA
sistla@cs.uic.edu

V.N. Venkatakrishnan
University of Illinois at Chicago
Chicago, Illinois, USA
venkat@cs.uic.edu

ABSTRACT

We present the first sound program transformation approach for automatically transforming the code of a legacy web application to employ PREPARE statements in place of unsafe SQL queries. Our approach therefore opens the way for eradicating the SQL injection threat vector from legacy web applications. This extended abstract is based on our paper [4] that appeared in the Financial Cryptography and Data Security (FC'2010) conference.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access; H.2.0 [**General**]: Security, Integrity, and Protection; I.2.2 [**Automatic Programming**]: Program Transformation; D.2.5 [**Testing and Debugging**]: Symbolic Execution; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, Reverse Engineering, and Reengineering

General Terms

Security, Algorithms, Languages

Keywords

Static Program Transformation, Security by Construction, Symbolic Evaluation, SQL Injection

1. INTRODUCTION

In the last decade SQL Injection attacks (SQLIA) have emerged as a serious threat to Web Applications [2]. SQLIA are a prime example of malicious input that change the behavior of a program by sly introduction of query structure into the input strings. An application that does not perform input validation (or employs error-prone validation) is vulnerable to SQL injection attacks. Although useful as a first layer of defense, input validation often is hard to get right and has been cited as the number one cause of vulnerabilities in web applications [3].

There is an emerging consensus in the software industry that using PREPARE statements, a facility provided by many database platforms, to construct SQL queries constitutes a robust defense against SQL injections. PREPARE statements are objects that contain precompiled SQL query structures (without data). This allows a programmer to easily *isolate* and *confine* the "data" portions of the SQL query from its "code", *avoiding* the need for (error-prone) sanitization of user inputs.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.
ACM 978-1-4503-0244-9/10/10.

The existing practice to transform an *existing application* to make use of PREPARE statements requires extensive manual effort. The programmer needs to obtain a detailed understanding of the program that includes identification of all inter-procedural control and data flows that generate vulnerable SQL queries. Furthermore, these flows have to be analyzed to obtain the equivalent code for PREPARE statement generation. Each such control flow needs to be carefully transformed while ensuring that the changes do not alter semantics of the program in any undesirable fashion. Furthermore, additional manual verification may be needed to ensure that the semantics of the transformed program on non-attack inputs is the same as the original program. This process could be tedious, sometimes error-prone, and certainly expensive for large-scale web applications.

The objective of this paper is to develop a *sound* method to *automate* the above transformation to PREPARE statements. This will overcome the deficiencies of manual approach, and would result in considerable savings of program development costs. However, designing a sound method is extremely challenging because a completely automated method needs to replicate the human understanding of the program logic that constructs SQL queries. Quite often, this understanding of program logic is guided by additional documentation such as high-level system designs, flow charts and low level program comments. An automated method that aims to eliminate / minimize human effort cannot depend on the availability or use of any such additional specifications. The web application code is, therefore, the only specification available to our method, from which an understanding of the program logic needs to be automatically extracted to guide the transformation.

The main contribution of this paper is to address this challenge by developing the first automated *sound* program transformation approach that retrofits an existing (legacy) web application to make use of PREPARE statements. We develop a new method that constructs a *high-level* understanding of a program's logic directly from its *low-level* string operations. This method relies on a novel insight that a program's low-level string operations along any particular control path can be viewed as a derivation of a symbolic SQL query that is parameterised by its inputs. Our method directly uses this derivation to identify and isolate any unsafe string operations that may otherwise result in injection attacks. The isolated operations are then rewritten using PREPARE statements, effectively eliminating the SQL injection attack vector from the web application.

Our approach is implemented in a tool called TAPS (Tool for Automatically Preparing SQL queries) which is the first reported sound tool in the literature to perform this transformation. TAPS has been successfully applied to several real world applications, including one with over 22,000 lines of code. In addition, some of these applications were vulnerable to widely publicized SQL injec-

tion attacks present in the CVE database, and our transformation renders them safe *by construction*. This approach will assist developers and system administrators to automatically retrofit their programs with the "textbook defense" for SQL injection.

2. PROBLEM STATEMENT

We use the following code snippet as running example: it applies a (filter) function (`f`) on the input (`$u`) and then combines it with constant strings to generate a query (`$q`). This query is then executed by a *SQL sink* (query execution statement) at `line 6`.

```
1.  $u = input();
2.  $q1 = "select * from X where uid LIKE '%";
3.  $q2 = f($u); // f - filter function
4.  $q3 = "%' order by Y";
5.  $q = $q1.$q2.$q3;
6.  sql.execute($q);
```

The running example is vulnerable to SQL injection if input `$u` can be injected with malicious content and the filter function `f` fails to eliminate it. For example, the user input `' OR 1=1 -` provided as `$u` in the above example can break out of the expected string literal context and add an additional `OR` clause to the query. Typically, user inputs such as `$u` are expected to contribute as literals in the parse structure of any query, specifically, in one of the two literal *data contexts*: (a) a string literal context which is enclosed by program supplied string delimiters (single quotes) (b) in a numeric literal context. SQL injection attacks violate this expectation by introducing input strings that do not remain confined to these literal data contexts and directly influence the structure of the generated queries.

`PREPARE` statements confine all query arguments to the expected data contexts and allow a programmer to declare (and finalize) the structure of every SQL query in the application. Once constructed, the parse structure of a `PREPARE` statement is frozen and cannot be altered by malformed inputs. The following is an equivalent `PREPARE` statement based program for the running example.

```
1.  $q = "select...where uid LIKE ? order by Y";
2.  $stmt = prepare($q);
3.  $stmt.bindParam(0, "s", "%".f($u)."%");
4.  $stmt.execute();
```

The question mark in the query string `$q` is a "place-holder" for the query argument `%f($u)%`. In the above example, providing the malicious input `u = ' or 1=1 -` to the prepared query will not result in a successful attack as the actual query is parsed with these placeholders (`prepare` instruction generates `PREPARE` statement), and the actual binding to placeholders happens *after* the query structure is finalized (`bindParam` instruction). Therefore, the malicious content from `$u` cannot influence the structure of query. In this paper, we aim to replace all queries generated by a web application with equivalent `PREPARE` statements. A web application can be viewed as a SQL query generator, that combines constant strings supplied by the program with computations over user inputs.

3. OUR APPROACH

As mentioned earlier, user inputs are expected to contribute to SQL queries in string and numeric data literal contexts. Our approach aims to isolate these (possibly unsafe) inputs from the query by replacing existing query locations in the source code with `PREPARE` statements, and replacing the unsafe inputs in them with safe placeholder strings. These placeholders will be bound to the unsafe inputs during program execution (at runtime).

In order to do this, we first observe that the original program's instructions already contain the programmatic logic (in terms of string operations) to build the structure of its SQL queries. This leads to the crucial idea behind our approach: *if we can precisely identify the program data variable that contributes a specific argument to a query, then replacing this variable with a safe placeholder string (?) will enable the program to programmatically compute the `PREPARE` statement at runtime.*

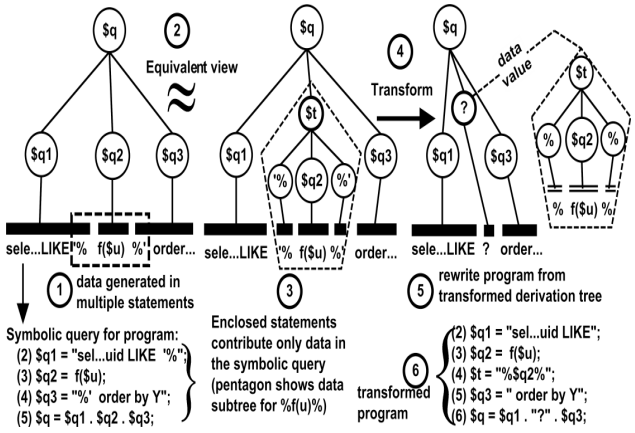

Figure 1: TAPS**: step (1) generates symbolic queries, steps (2-3) separate data from queries, step (4) removes data from symbolic queries, and steps (5-6) produce transformed program.**

The problem therefore reduces to precisely identifying query arguments that are computed through program instructions. In our approach, we solve this problem through symbolic execution, a well-known technique in program verification. Intuitively, during any run, the SQL query generated by a program can be represented as a symbolic expression over a set of program inputs (and functions over those inputs) and program-generated string constants. For instance, by symbolically executing our running example program, we obtain the following symbolic query expression :

```
SELECT...WHERE uid LIKE '%f($u)%' ORDER by Y
```

Notice that the query is expressed completely by constant strings generated by the program, and (functions over) user inputs. Once we obtain the symbolic expression, we analyze its parse structure to identify data arguments for the `PREPARE` statement. In our running example, the only argument obtained from user input is the string `%f($u)%` . Our final step is to traverse the program backwards to the program statements that generate these arguments, and modify them to generate placeholder (?) instead. Now, we have changed a data variable of a program, such that the program can compute the body of the `PREPARE` statement at runtime.

In our running example, after replacing contributions of program statements that generated the query data argument `%f($u)%` with a placeholder (?), `$q` at line 5 contains the following `PREPARE` statement body at runtime:

```
SELECT...WHERE uid LIKE ? ORDER by Y, %$q2%
```

The corresponding query argument is the value `%$q2%`. Note that the query argument includes contributions from program constants (such as `%`) as well as user input (through `$q2`) .

Approach overview. Figure 1 gives an overview of our approach for the running example. For each path in the web application that leads to a query, we generate a derivation tree that represents the structure of the symbolic expression for that query. For our example, `$q` is the variable that holds the query, and step

1 of this figure shows the derivation tree rooted at `$q` that captures the query structure. The structure of this tree is analyzed to identify the contributions of user inputs and program constants to data arguments of the query, as shown in steps 2 and 3. In particular, we want to identify the subtree of this derivation tree that confines the string and numeric literals, which we call the *data subtree*. In step 4, we transform this derivation tree to introduce the placeholder value, and isolate the data arguments. This change corresponds to a change in the original program instructions and data values. In the final step 5, the rewritten program is regenerated. The transformed program programmatically computes the body of the `PREPARE` statement in variable `$q` and the associated argument in variable `$t`.

Conditional Statements. TAPS first individually transforms each control path that could compute a SQL query. To do so it creates program slices from System Dependency Graph. Once each individual path is transformed it then makes changes to the source code of the original program. One issue that arises while doing so is that of a *conflict*: when path P_1 and P_2 of a program share an instruction I that contributes to the data argument, and I may not undergo the same transformation in both paths. We detect such cases before making any changes to the program and avoid transformation of paths that may result in conflicts.

Loops. TAPS summarizes loop contributions using symbolic regular expressions. The goal is essentially to check if placeholder (?) can be introduced in partial SQL queries computed in loop bodies. Given the loop summary, TAPS requires that the loop contribution be present in a "repeatable" clause of the SQL grammar. To do so we require statements in the loop body to satisfy the following rules: (1) the statement is of the form $q \rightarrow x$ where x is a constant or an input OR (2) it is left recursive of the form $q \rightarrow qx$, where x itself is not recursive, i.e., resolves to a variable or a constant in each loop iteration. If these conditions hold, we introduce placeholders in the loop body. This strategy only covers a small well defined family of loops. However, our evaluation suggests that it is quite acceptable in practice.

Limitations. TAPS requires developer intervention if either one of the following conditions hold: (i) program changes query strings containing placeholder (?) (ii) a well-formed SQL query cannot be constructed statically (iii) SQL query is malformed because of infeasible paths (iv) conflicts are detected along various paths (v) query is constructed in a loop that cannot be summarized.

4. EVALUATION

Test suite. Table 1 column 1 lists SQLIA vulnerable applications used in prior research and applications with known SQLIA exploits from Common Vulnerabilities and Exposures (CVE 2009) repository. This table lists their codebase sizes in lines of code and any known CVE vulnerability identifiers (column 2 and 3), number of analyzed SQL sinks and control flows that execute queries at SQL sinks (column 4 and 5), transformed SQL sinks and control flows (column 6 and 7).

Transformation Metrics. For the three largest applications, TAPS transformed 93%, 99% and 81% of the analyzed control flows and achieved a 100% transformation rate for the rest of them. As TAPS depends on symbolic evaluation, it did not transform flows that obtained queries at run time e.g., the `Warp CMS` application used SQL queries from a file to restore the application's database. In two other instances, it executed query specified in a user interface. In both these cases, no meaningful `PREPARE` statement is possible as external input contributes to the query structure. The limitations of the SQL parser implementation were responsible for two of the three failures in the `Utopia news pro` application and the rest are discussed below. A total of 18 control flows used loops that violated restrictions imposed by TAPS and were not transformed (11 - `Warp CMS`, 1 - `Utopia news pro`, 6 - `AlmondSoft`). We also found 23 instances of queries computed in loops, including a summarization of `implode` function, that were transformed. For untransformed flows TAPS precisely identified statements to be analyzed e.g., the `Warp CMS` application required 195 LOC to be manually analyzed instead of complete codebase of 22K LOC. This is approximately two orders of magnitude reduction in LOC to be analyzed.

Application	Size LOC	CVE ID 2009	Analy. SQL Sinks	Analy. Ctrl. Flows	Trans. SQL Sinks	Trans. Ctrl. Flows
WarpCMS	22,773	-	14	200	14	186
UtopiaNP	7,323	-	2	336	2	333
AlmondSoft	6,633	3226	22	33	17	27
PortalXPTE	5,121	3148	122	122	122	122
GravityBoard	2,422	1277	62	62	62	62
MyNews	1,792	0739	1	34	1	34
Auth	284	0738	1	5	1	5
BlueBird	288	0740	1	5	1	5
Yap Blog	264	1038	2	6	2	6

Table 1: Effectiveness suite applications, transformed SQL sinks and control flows: TAPS **transformed over 93% and 99% of the analyzed control flows for the two largest applications.**

TAPS changed a small fraction of original Lines of Code as a result of this transformation (1–7%). TAPS was assessed for performance overhead on a microbench that manipulated varying sized query arguments. Over this stress test, we did not find any noticeable deviations in response times of the transformed and the original application.

Tool demo. TAPS is available for evaluation at `http://sisl.rites.uic.edu/taps`. A test harness with several challenging cases is also available at this site.

5. CONCLUSION

We presented TAPS, a static program transformation tool that modifies web applications to make use of `PREPARE` statements. The tool has been evaluated with several open-source applications to assess its effectiveness. Our approach provides evidence that it is possible to successfully design retrofitting techniques that guarantee security (by construction) in legacy applications, and eliminate well known attacks.

6. REFERENCES

[1] JDBC: Using a prepared statements. http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html.

[2] Symantec Internet Security Threat Report, volume XI. Technical report, Symantec, March 2007.

[3] OWASP. The ten most critical web application security vulnerabilities. http://www.owasp.org.

[4] BISHT, P., SISTLA, A. P., AND VENKATAKRISHNAN, V. Automatically Preparing Safe SQL Queries. In *FC'10: Proceedings of the 14th International Conference on Financial Cryptography and Data Security* (Tenerife, Canary Islands, Spain, 2010).