Automated Protection of PHP Applications Against SQL-injection Attacks

Ettore Merlo, Dominic Letarte, Giuliano Antoniol
Department of Computer Science, École Polytechnique de Montréal,
C.P. 6079, succ. Centre-ville Montréal (Québec) H3C 3A7
{ettore.merlo, dominic.letarte}@polymtl.ca
antoniol@ieee.org

Abstract

Web sites may be static sites, programs, or databases, and very often a combination of the three integrating relational databases as a back-end. Web sites require care in configuration and programming to assure security, confidentiality, and trustworthiness of the published information.

SQL-injection attacks exploit weak validation of textual input used to build database queries. Maliciously crafted input may threaten the confidentiality and the security policies of Web sites relying on a database to store and retrieve information.

This paper presents an original approach that combines static analysis, dynamic analysis, and code re-engineering to automatically protect applications written in PHP from SQL-injection attacks.

The paper also reports preliminary results of experiments performed on an old SQL-injection prone version of phpBB (version 2.0.0, 37193 LOC of PHP version 4.2.2 code). Results show that our approach successfully improved phpBB-2.0.0 resistance to SQL-injection attacks.

Keywords: SQL-injection, software security analysis, software re-engineering

1 Introduction

Web applications are used to distribute information to users over a network and often accept interactions from users and perform some accesses to databases. They are based on assumptions about legitimate input that is used to build SQL queries to be submitted to a database. These applications are possibly vulnerable to SQL-injection attacks that exploit weak validation of textual input.

In some contexts, maliciously crafted input containing parts of SQL instructions, produces queries whose semantics are different from those intended by the designers and may threaten the security policies of the underlying databases.

SQL-injections attacks have been described in the literature [7, 8]. Although the reasons behind successful SQL-injections attacks are known and although classes of defense against SQL-injection attacks including "defensive programming", sophisticated input validation, dynamic checks, and source level static analysis exist, problems persist for several reasons. Defensive programming and input validation aim at preventing the insertion of "malicious" strings into SQL queries, but unfortunately they are fairly sensitive to new patterns of unanticipated attacks. Furthermore, defensive programming may be better suited for new development since it is programming intensive, but is weaker in addressing the issue of protecting legacy systems. Dynamic checks are sensitive to obsolescence in time as applications evolve and new legitimate interactions are added, which may increase the number of false positives. Finally, static analysis suffers from issues of precision and execution performance when complex languages features are considered, such as interprocedural transfers of data and control, pointers, arrays, and polymorphism.

This paper presents an original approach to automatically protect applications written in PHP from SQL-injection attacks. It combines static analysis to parse PHP code and SQL queries, dynamic analysis to build syntactic models of legitimate SQL queries, and automatic code re-engineering to protect existing legacy applications from the above mentioned attacks.

The main contributions of this paper are:

- dynamic analysis to automatically build "model-based guards" of legitimate SQL queries

- automatic protection of existing legacy applications written in PHP by appropriately inserting "model-based guards" in source code
- preliminary experimental assessment obtained in legitimate test cases and attacks

Section 2 introduces the proposed approach to prevent SQL-injections; section 3 describes the automatic construction of model-based guards; section 4 presents proof-of-concepts experiments; section 5 discusses the preliminary findings reported as experimental results; section 6 briefly recalls related work, and section 7 concludes this paper.

2 Prevention Approach

Injection attacks in general, and specifically SQL-injection attacks, exploit weak input validation that allows users to somehow manipulate the queries that are passed to an SQL database.

Communication between applications and databases is achieved through proper calls to DB-API routines. A typical example is $mysql(str, db)$ where str is a text string that contains the SQL-query and db is a database "handler".

Several database engines, both public domain and commercial, are available. Since we implement our model validation guards between Web applications and databases, we are not strongly tied to the particularly adopted DB. However, for the sake of example and experiments, we will assume that a $MySql$ [3] DBMS is used. We also assume that the DBMS engine implements an extended subset of SQL standard [5]. Also, in the following, we will not make an explicit distinction unless required between Databases (DB) and Database Management Systems (DBMS).

As proposed in [14] the presented protection model requires all communication between program variables (either user or programmer variables) and DB-API routines to pass through "secured model-based guards" that perform appropriate security checks.

This approach transfers the burden of ensuring the protection against SQL-injection attacks from application developers to a core team of developers responsible for security issues as well as the development and review of "secured model-based guards" software. Furthermore, the sheer size of the code to be "secured" is dramatically reduced, since, while application developers would have to enforce security through the whole application, the proposed approach allows the security against SQL-injection attacks to be enforced only in "secured model-based guards" which are smaller in size than the applications.

Injection attacks exploit the plain text communication between an application and the DB-API routines. When programming variables are transformed and integrated in the text based communication between programs and DB servers, their associated type information is lost.

As reported in [12], the situation would be different if SQL queries went through type based specific DB-API routines. In a typed API based architecture, variables can carefully be checked against syntax, domain ranges, and in some ways semantic content.

While this could be an appropriate approach for new systems, existing applications often use string based plain text dialog to API for DB accesses. In this context, security checks have to be performed on the strings passed as parameters to DB-API routines.

Also, arbitrary text strings that contain some SQL syntactic content may be stored in a DB and later retrieved from the database and used in some possibly malicious computation. This problem is that generally of second and higher order SQL-injections [20]. When data that contains SQL syntax is stored in a DB or on a persistent medium, its retrieval from the store and its subsequent use as part of a dynamic SQL query may cause an SQL-injection in some higher order SQL-injection vulnerable statements.

Another possible source of SQL-injections comes from malicious code that may introduce SQL syntactic content in plain text parameters of DB-API, based on some input conditions. An example may be that of inserting additional queries about confidential DB tables, if the current login is a special login name.

In this paper, we explore automatic construction of model guards based on dynamic analysis of "trusted tests".

3 Model-based guard construction

We use the term "SQL-injection prone software" for software in which some SQL-injection may occur; "vulnerable statements" are DB access statements that may contain part of user supplied input. Input data may cause some of the above defined SQL-injection problems. We use the term "SQL-injection revealing data" for input data or DB stored data that contributes to a successful SQL-injection at a "vulnerable statement". The computation that carries "SQL-injection revealing data" is called an "SQL-injection attack". Similarly, malicious code that causes an SQL-injection is called "SQL-injection revealing software". An example of this is when a special login name is allowed to gain access to confidential DB tables using a specially crafted query to the database. Clearly, to obtain an SQL-injection the appropriate conjunction of "SQL-injection revealing data", possibly "SQL-injection revealing software" and "vulnerable statements" has to occur.

Our approach is based on breaking the "appropriate conjunction" of input data, code, and DB access condition that would allow SQL-injection. Intuitively, appropriate "guards" will be inserted before DB accesses so that the appropriate conjunctions of SQL-injection conditions are prevented from occurring.

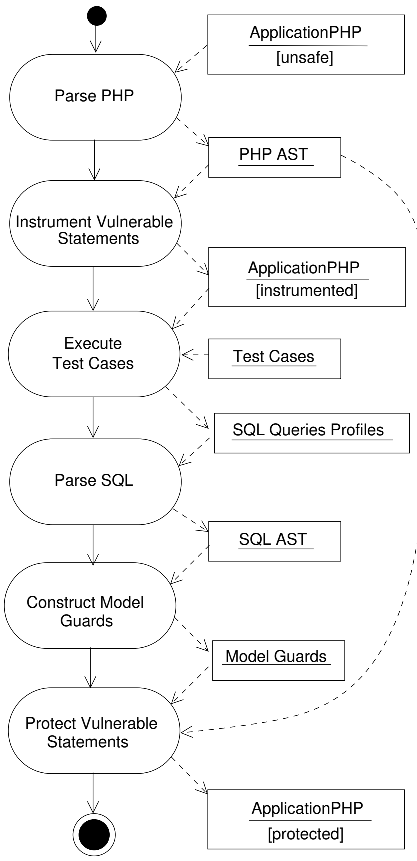

Figure 1. Automated Protection

As shown in Fig.1, we first instrument the PHP code to collect samples of queries that should be legitimately used at DB-API call points. These queries will be considered as a set of "trusted test cases" as further detailed in Section 4. Instrumentation is quite easy to produce since it is simply necessary to add an output instruction before DB-API calls, as follows:

$$\begin{array}{c} ... \\ mysql\_query(\text{ ... } expr \text{ ... }); \\ ... \end{array}$$

becomes:

$$\begin{array}{c} ... \\ \$str = \text{ ... } expr \text{ ... } ; \\ fwrite(\$file\_handle,\ \$str)); \\ \$result\ =\ mysql\_query(\$str); \\ ... \end{array}$$

We then run the trusted test cases to collect the plain text strings corresponding to trusted queries that are generated dynamically at various call sites. The process of building model guards from sets of ASTs corresponding to legitimate queries is quite straightforward. Legitimate queries are parsed and the corresponding forest of ASTs is stored for each call site. ASTs are independently stored so that no generalization between queries is performed. Nevertheless, ASTs are to some extent generalized since constants, strings and other kinds of user data are stored in the ASTs by type rather than by image. Conversely, application dependent identifiers that refer to the database schema and that carry, for example, the names of tables and columns, are considered part of the syntactic structure of SQL queries, to prevent malicious substitution of table or column names in legitimate queries. This allows multiple queries of the same syntactic structure, but with different values of data.

A model guard at one specific call site invokes the SQL parser on the current DB query and obtains the corresponding SQL AST. The produced AST is matched against the stored legitimate reference ASTs for the same call site. A positive match indicates that the current query has a syntactic structure compatible with legitimate queries from the "trusted set". Positively matched queries should be allowed to be presented to the DB-API. Conversely, it is conservative to reject queries that do not match reference ASTs for a call site.

A parser for each of these sub-languages constitutes the model-based guard that filters queries at call sites. In reality, the sub-languages are so simple that we have chosen to implement model guards as hash tables of AST leaves in depth-first order (token types) preceded by the id of call sites.

Reference ASTs are indeed stored as token strings containing token types and not images, but where application table names and field names have become keywords. Local configuration table names have been stored as identifier token types. In the following, these token strings will be called "reference patterns".

An example of trusted query obtained probing the call at line 221 in file search.php is the following:

```
SELECT post_id
FROM phpbb200t_posts
WHERE
poster_id = 2
```

where "phpbb200t_posts" is a configuration dependent local table name, while "post_id" and "poster_id" are field names fixed in the application logic. Indeed the reference pattern corresponding to the previously shown reference query is the following:

```
search.php : 221
SELECT POST_ID
FROM SQL_ID
WHERE
POSTER_ID OP_EQUAL INTEGER_LITERAL
```

We can notice that "phpbb200t_posts" has been recognized as an identifier, while "post_id" and "poster_id" have been considered as keywords of the application. This prevents injection attacks from keeping the same syntactic structure of a legitimate query while replacing field names with confidential ones. In the example "post_id" and "poster_id" can be accessed in any local configuration of phpBB, but they cannot be replaced by other fields. Similarly the poster identifier value 2 can be replaced only by an integer value.

Further details about trusted ASTs are presented in Section 4.

Vulnerable statements identified as reported in [19] can be secured by replacing them with model-based guards that perform the appropriate checks at each call site and call the appropriate DB-API upon successful checks.

$$\begin{gathered}\ldots\\ call\_site_i : mysql(\$str)\\ \ldots\end{gathered}$$

becomes:

$$\begin{gathered}\ldots\\ call\_site_i : sqlModelGuard(\$call\_site_i,\ \$str)\\ \ldots\end{gathered}$$

The implementation of model-based guards and the re-engineering of source code to change calls to *mySql* into calls to the model-based guards are straightforward and require little effort. An example of model-based guard written in Java is reported in Fig. 2.

```
class sqlModelCheckerCl {

    staticHashSetmodel = newHashSet();

    public static void init() {
        // initialize model checker
        // with all reference patterns
        ...
        model.add("search.php : 221 SELECT ...");
        ...
    }

    String parseQuery(String query) {
        // SQL parser implementation
    }

    bool sqlModelGuard(String callSite,
                            String query) {
        if (model.contains(callSite +
             parseQuery(query))) {
            mysql(query);
        } else {
            System.exit(1);
        }
    }
}
```

Figure 2. Conceptual Example of Model Guard

4 Experiments

To validate our approach, we have taken a proof-of-concept perspective about making an application secure from users or insider SQL-injections.

phpBB [4] is an application well known for its abundance of documented opportunities for SQL-injections in its past versions. Indeed, during the past years and versions, an effort has been deployed to remove SQL-injections in phpBB by design.

For our experiments, we have chosen the old version 2.0.0 of phpBB, for which publicly available security bugs have been reported, together with *mySql* version 4.0.26. Our objective is to take an old security flawed version and to automatically transform it into the same application in terms of functionality (so it may still be old with respect to functionalities of current version) but with removed SQL-injection vulnerabilities.

We found 285 occurrences of vulnerable DB-API calls out of 406 ones in the phpBB version used for the experiments, using the vulnerability analysis described in [19]. To collect patterns of legitimate queries, we have automatically instrumented phpBB by inserting security probes before vulnerable statements.

To parse SQL queries we have built our SQL parser by considering the *mySql* syntax available in [3] and by modifying a Web available PHP grammar [2] by Satyam implemented in JavaCC [1]. Features of the SQL parser are reported in Table 1. Other SQL grammars are available from the JavaCC grammar repository site [1].

	PHP	SQL
#Rules	73	2418
Grammar size (LOC)	1221	34133
Parser size (LOC)	11033	118358

Table 1. Parsers Features

An AST visitor that adds instrumentation in the PHP code according to the desired probes and the scoping rules has been implemented in the JavaCC environment.

Legitimate test cases have been obtained by operating the application for a few hours to try to increase the coverage of call sites. There were 1590 obtained legitimate queries. Some of these are duplicate queries occurred during operation. 417 distinct queries were presented at 107 distinct vulnerable call sites.

Some legitimate queries, although with different values and variables, happen to have the same syntactic structure as discussed in Sec. 3. Reference AST patterns from legitimate queries total 155. Fig. 3 shows the length distribution of reference patterns obtained from legitimate queries. Fig. 3 shows the length distribution of reference patterns. Minimum pattern length was 4 tokens, maximum was 236 tokens, and average length was about 35 tokens, with a standard deviation of about 4 tokens.

Fig. 4 shows the distribution of reference patterns over different call sites. Most call sites were associated with only one reference pattern in the legitimate set used for experiments. An average of about 1.4 patterns per call site was observed with a standard deviation of less than one pattern. Maximum number of patterns was 5.

We found that 9 reference patterns begin with "SELECT OP_ASTERISK FROM SQL_ID ... " and are followed by conditions or ordering statements. Although these queries were labelled as legitimate, after reference patterns review, we would recommend further attention, since injecting special characters in SQL_ID and in other tokens may easily allow a user to gain access to any table in phpBB.

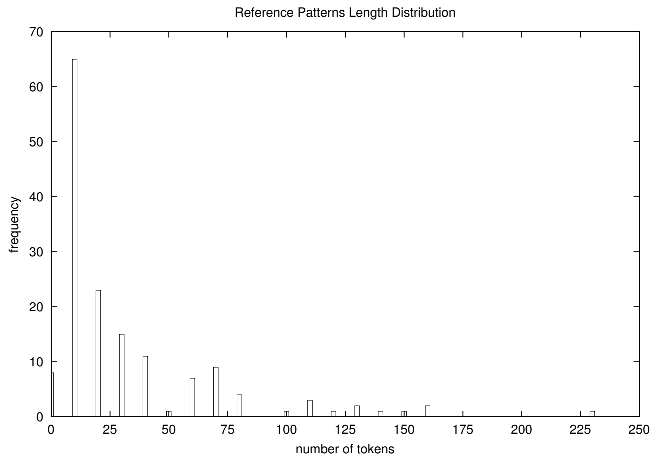

Figure 3. Reference pattern length distribution

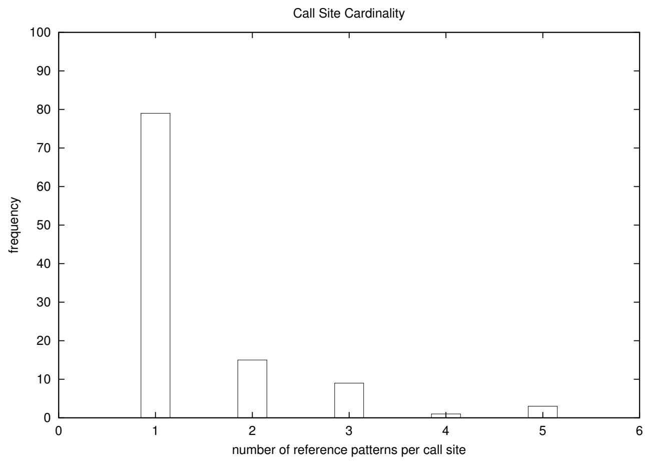

Figure 4. Distribution of patterns over call sites

Making phpBB secure consists of automatically replacing calls to *mySql* with calls to the model-based guards at each call site. Like the problem of security probe instrumentation, we have written a visitor that is based on our PHP grammar and that makes such a replacement automatically on all 285 vulnerable instances of *mySql* calls.

From a phpBB developer point of view, the only changes to the source code have been the replacement of calls to *mySql* with calls to model-based guards. Furthermore, there are 15 lines of new PHP code to transfer calls from phpBB to model guards that have been implemented with 510 lines of Java. phpBB developers' mental model of source code is definitely very little affected by the automatically added protection.

4.1 SQL-injections

Since the intersection between legitimate cases and attacks is in principle empty (unless errors or malicious tests have been inserted and mistakenly accepted by test reviews), the only errors are implementation or re-engineering errors. To assess the strength of the proposed SQL-injection prevention process the protected system has been tested against a set of 176 distinct attacks that had been proven successful before software protection. Injection attacks have been manually constructed from Web available information. Exploits of vulnerabilities are more difficult and tricky to implement than constructing legitimate sets of queries, which is why the number of attacks is smaller than the number of legitimate queries.

The purpose of these experiments is to assess the number of false positives and false negatives obtained by running the legitimate test cases and injection attacks against the secured application. Test cases and data for the proof-of-concept approach have been built and divided into two classes: legitimate cases and injection attacks. Execution of trusted test cases gives legitimate patterns of SQL queries, while execution of injection attacks produces queries that should be successfully blocked in the "secured" version.

	Tests	Injection attacks
Total number	1590	176
Accepted	1590	0
Accepted (%)	100	0
Rejected	0	176
Rejected (%)	0	100

Table 2. Experimental results

Fig. 5 shows the distribution in size of the 176 SQL-injected queries used for experiments. Also, in the reported experiments, only three call sites have been targeted for injection attacks, namely privmsg.php at line 236, search.php at line 601, and viewtopic.php at line 150, because of the time required to manually build customized attacks.

Both legitimate queries and injection attacks have been implemented by inserting an appropriate sequences of URL and parameters on the URL line using a Perl script. Table 2 presents the results obtained from the execution of 176 attacks.

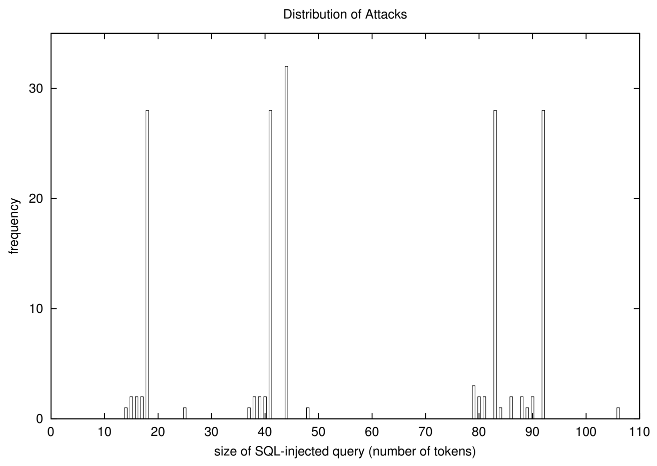

Figure 5. Size distribution of SQL-injected queries

5 Discussion

Experimental results confirm the expectation of a very high success rate and a very low number of false positives and false negatives. Indeed none have been detected in the presented experiments. Intuitively, it's somehow expected, since, because of the limited size experiment set, it's hard to imagine an attack that bears the same SQL language structure of a legitimate query including specific application dependent DB tables and columns names.

Nevertheless, false alarms and mis-classified attacks may still occurr if unauthorized SQL queries were mistakenly allowed in the test sets or in the security scenarios of legacy applications. In general, the precision of the presented automatic protection approach on legacy application is tied to the representativeness of the test cases and scenarios. Model guards are indeed automatically built based on a dynamic approximation of security specifications. Somehow this approach may suffer from the completeness problem of dynamic analysis, but we believe that the impact is much reduced when dealing with fixed forms or with few fixed forms per call site.

Our approach is based on the assumption that very few different SQL-query structures are produced at each call site, since each site has a good chance of representing a particular family of queries related to a legitimate composition of user variables.

Existing systems haven't, in general, been built with the proposed security model in mind. To make adoption of the proposed security model effective, an automated transition must be set up to convert existing systems into secure ones. As already mentioned, this requires some shift of effort and responsibility from application developers to the security team. This approach helps in building a transition to secure systems from old systems that haven't been designed with security in mind. Our automated construction process for model guards is quite simple and there is plenty of room for making these models more complex by leveraging on the power of the underlying general SQL parser that is used to parse queries. The effectiveness of automation is based on the small number of legitimate queries at a call site.

Conversely, developers from the security team may manually construct guards based on security specifications. A manual approach would also be feasible, possibly with better accuracy in terms of completeness and complex query structures, but it would be labor intensive and questionably effective on legacy systems.

Three categories of applications can be identified: those that interact though fixed forms; those that accept a subset of SQL based on some interaction grammar; and those that accept arbitrary plain text SQL queries.

The first category, applications that interact with the users by using fixed forms, should be well-protected by our approach, provided that call points coincide with distinct semantics of fixed forms so that the syntax of a form can be checked at one specific call point. It wouldn't work if all queries are sent to only one call point for API. Further research would be needed to unravel query construction when only one call to DB-API is used. Strategies for SQL-injection protection are often weak in blocking newly conceived attacks. New malicious syntactic variations do not belong to the "reference set" language recognized by model-based guards and are rejected. Our approach is robust in that only explicitly acceptable queries are sent to DB-API. Newly constructed attacks would be rejected in any case. Unfortunately, the price for this robustness is the presence of false negatives, but this becomes negligible when dealing with applications using fixed forms interaction.

Applications belonging to the second category may accept a variable number of forms which should be within a given grammar imposed by the application (think of advanced searches of search engines in which the user can specify arbitrary combinations of identifiers, qualifiers, and relational operators). In these cases the security model should implement the underlying grammar and check that the syntactic structure of queries doesn't extend the intended grammar of the application and the presented approach would not help much for this class of applications.

Finally, applications in the third category make unrestricted use of SQL syntax. Call points that definitely receive complex queries, and potentially all SQL language, and that are full text entries should have access reserved to privileged users or system administrators. Examples can be found in the system administrator parts of an application. In these cases, syntax alone is not enough by definition to filter accesses, and other authorization mechanisms and privilege granting procedures should be used.

These kinds of accesses cannot be protected by the proposed approach, but, in general, these call points should be accessible only from files and programs that required administrator privileges and should not be vulnerable to SQL-injections from non-privileged users. A vulnerability analysis approach that can detect this kind of privilege discrepancy between required security and authorization levels in the context of SQL-injection attack has been described in [19].

Further experiments are required to increase the number of attacks and the number of analyzed systems to assess precision and errors from a more statistically representative perspective.

Dynamic analysis used in our approach suffers from incompleteness. Trusted tests are probably not complete - especially when the number of syntactic patterns of queries at call sites increases and it is far from fixed and small - and some legitimate queries may be rejected because they never contributed to construction of guards. This happens because we try to infer designers' intention using a legitimate set of uses.

Several activities may reduce the impact of the incompleteness of reference patterns. The first is human review of reference patterns, especially when integrated with interaction specifications when available. This may help in cases of fixed forms and a small number of patterns.

Since the average number of patterns per call site is about 1.4 (refer to Sec. 4 for details), it is quite easy to inspect the set of reference patterns for the presented experiments. Further investigation is however needed to assess the average number of patterns in a larger set of more diversified applications.

Furthermore, human reviews may help in identifying call points that are better protected by other mechanism than those proposed in this paper. Namely, call points that are hard to protect may be considered for "interaction grammar" protection, or they may be upgraded to arbitrary SQL queries and protected by authorization granting procedures.

Let's suppose that a malicious programmer introduced in all queries the output request for confidential tables and after the execution of queries filtered out all confidential information and only propagate appropriate information, except when a special login name were eventually presented to the database. At that moment the output filter would be inhibited, so confidential information could leak out. This attack schema would, of course, produce reference patterns that contain the access to confidential information, but would be accepted since the user output is legitimate. Again, human reviews of reference patterns are necessary to limit and prevent this kind of attack. Further research could also be performed by extending the analysis in [19] so that no paths from confidential information to output be available to non-privileged users without executing the proper de-classification procedures.

Rejected SQL strings may also be stored together with some context information such as IP of origin, user name, and time. Some post-processing of rejected strings may periodically be performed and legitimate queries that have been rejected because their structure was not represented in the set used for dynamic analysis may be discovered and considered for addition to the model-based guards.

Further investigation is needed to assess the precision and effectiveness of model guards, created as proposed in this paper, on a wider spectrum of systems.

The implementation model of queries and attacks based on the URL used for the presented experiments is, unfortunately, not general enough, since attacks that require the application being in a vulnerable state may only be achieved by several dialog exchanges with the application and not only one single URL references. This is not addressed in our approach and is left for further research.

Another issue to be investigated in a larger set of attacks is the performance penalty to be paid for model guard checking.

Instrumentation of application for SQL profiling and dynamic analysis is linear in the size of the application. Model guard construction is proportional to the number of test cases and to the average length of SQL queries performed in the tests. Protecting an application by introducing the proper calls to model guards, again, is linear with respect to the application size.

At runtime, parsing an SQL query takes a time proportional to the length of the checked query. Checking the parsed query against a call point model takes, again, a time proportional to the length of the parsed query because of the hash table implementation of reference patterns. Details about average size of reference and attack patterns have been reported in Sec. 4 for the presented experiments.

At present, the proof-of concept prototype implementation is slowed down by the loading of the Java Virtual Machine (JVM) and the parser each time parsing is required for an SQL query and for checking a parsed query against a model guard. Unfortunately, PHP is a memoryless interpreted language that cannot, by itself, keep programs loaded from one execution to the next one. Further research is required to investigate alternative architectures that would avoid repetitive loading of JVM and the guards. It's quite possible to integrate JVM and model guards loading operation with the loading of phpBB. As an alternative, daemons could be created that dialog with phpBB, and so on.

On the other hand, advantages of the proposed approach are its portability to other languages used in network based application like Java. In our laboratory, we have developed several parsers for programming languages and we have access to some research versions of commercial tools. Porting our SQL-injection approach to other languages would simply require to re-write the visitors for instrumenting the application for dynamic SQL profiling and for automatic re-engineering. The SQL parser doesn't need to be changed. From a syntactic point of view, working on languages like Java might even be easier because of their clean syntax and semantics.

Also, the proposed approach may be investigated in the context of other injection problems like those possible in XPath [6] and so forth.

6 Related work

AMNESIA [14, 15] is a tool for SQL-injection protection that uses model-based security by combining static and dynamic analysis. Their approach builds a static model of legitimate SQL queries that could be generated by Java applications using Java String Analysis (JSA) [11].

Models are designed as non-deterministic finite automata (NDFA) whose alphabet is that of SQL keywords, operators, constants, and a special symbol β for user input. At runtime, application generated queries are checked for compliance with the static models. Our research has been inspired by their approach that is similar under the perspective of building static models of legitimate queries, of runtime parsing the application generated queries, and of checking the parsed queries for compliance with the static models.

There are significant differences in the way the static models are built. AMNESIA builds static models based on static analysis of the application and in particular using JSA to infer the set of queries that an application may generate. Its objective is to separate application generated queries from user injected ones. The approach is weaker in protecting from insider attacks, since insider malicious code is recognized by JSA as application generated code and implicitly accepted as trustworthy. We build static models from dynamic analysis of the execution of legitimate test cases and security scenarios. Our approach does not necessarily "believe" the application code to build reference models of legitimate queries.

Another difference is the language in which applications are written. Although building the set of legitimate queries from static analysis of an application is conceptually similar for several imperative languages, in practice semantic differences between languages may prevent reusing much of the string analysis already written for other languages. For example, writing some PHP String Analysis tool may not share much of the code with JSA because of syntactic and semantic differences between the two languages. Conversely, when a parser is available, adapting our approach to another language would simply require writing an instrumentation visitor.

Context-Sensitive String Evaluation (CSSE) [21] detects and prevents SQL-injection attacks in PHP code. It modifies the PHP interpreter to track user provided parts of SQL expressions and to perform the appropriate check to prevent SQL injections. Tracking is performed by automatically marking all user-originated data with meta-data about their "taintedness" and by propagating meta-data through string operations in PHP. No knowledge about the application is required and no new programming discipline or practice is required from the developers. Reported precision is 100%, but developer provided parts of SQL expression are not subject to such SQL injection checks (because SQL content is definitely present in developer provided parts of SQL strings) and the system may not therefore be protected against insider threats or higher order SQL-injections.

JDBC Checker [13] combines automata-theoretical techniques and a variant of the context-free language (CFL) reachability problem to find typing context and scoping information, and to perform SQL type checking on the analyzed programs. This tool flags potential errors or verifies their absence in dynamically generated SQL queries. JDBC checker also plays a role in SQL-injection detection similar to that of JSA in AMNESIA.

SQL DOM [18] automatically extracts an object model of strongly-typed classes from a database schema and uses this model to generate safe queries that access a database from C#. Also, this approach shifts the reasoning from the world of dynamically generated queries, which may create runtime problems, to that of compile time declarative checking of compliance with safe classes. This allows very precise runtime domain value checks on parameters to the SQL DOM object calls. SQL syntactic content in a user supplied parameter can easily be detected and rejected. Again, this protects an application against malicious user input, but it's oblivious of errors, insider threats, or higher order SQL-injections.

While an analogy exists between SQL DOM and our approach, since the structure of their object model is similar to the AST structure of SQL queries when application dependent names of tables and columns are used in the language model (as in our case), their object model corresponds to all queries that can be constructed for a database schema; however, our approach restricts the reference models to the set of queries that are actually and legitimately used by the application at each call point. This set may be significantly smaller than the whole domain of schema dependent SQL queries. Furthermore, developers using SQL DOM would have to learn and adopt a new paradigm of programming. Their approach may better fit new development, while ours is applicable to both new development and evolution or protection of legacy systems.

Safe Query Objects [12] represent queries as statically typed objects, while still supporting remote execution by a database server. Safe query objects use object-relational mapping and reflective meta-programming to translate query classes into traditional database queries. Safe Query Objects can play a role in detecting SQL injections similar to that of SQL DOM.

An approach that uses static and dynamic analysis to protect applications from SQL-injection has been presented in [23]. It computes the Control Flow Graph of stored procedure and at runtime propagates safety or non-safety properties of user inputs to detect changes in the semantic of SQL queries. This approach performs well on a sample database in SQL server 2005, but is sensitive to insider threats and higher order injections.

Other approaches for detecting and protecting applications against SQL-injection can be found in [9, 10, 16, 17, 22].

In [19], an approach based on static analysis to automatically detect statements in PHP applications that may be vulnerable to SQL-injections triggered by either malicious input (outsider threats) or malicious code (insider threats) has been presented. Flow analysis equations that propagate and combine security levels along an inter-procedural Control Flow Graph (CFG) have been defined and presented.

Results of experiments performed on an old SQL-injection prone version of 2.0.0 phpBB show that vulnerability analysis can be used to compute and to identify SQL-injection vulnerabilities both at user and at system level. The computation of security levels presents linear execution time and memory complexity and execution of analysis is fast in practice.

7 Conclusions

An original approach that combines static analysis, dynamic analysis, and code re-engineering to automatically protect applications written in PHP from SQL-injection attacks has been presented, implemented and evaluated.

phpBB Web application has been automatically protected by using the proposed approach. 176 attacks that were successful before software protection were submitted to the newly protected version of the application under study. Experimental results show a very high success rate and a very low number of false positives and false negatives (indeed none has been detected in the presented experiments).

The proposed approach seems very promising on the presented phpBB test case, but further research and assessment are needed to evaluate it on larger and more diversified systems.

8 Acknowledgements

This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors wish to thank José Fernandez, Pierre-Marc Bureau, Antoine Rolland, and Kamel Ayari for their contribution to the discussions about SQL-injection and security.

References

[1] JavaCC. *https://javacc.dev.java.net.*

[2] PHP grammar. *https://javacc.dev.java.net//files/documents/17/14269/php.jj.*

[3] mySql. *http://dev.mysql.com/doc.*

[4] phpBB. *http://www.phpbb.com.*

[5] SQL. *http://www.iso.org.*

[6] XPath. *http://www.w3.org/TR/xpath.*

[7] C. Anley. Advanced SQL injection. In *Technical report*. NGSSoftware Insight Security Research, 2002.

[8] C. Anley. Advanced SQL injection in SQL server applications. In *Technical report*, 2002.

[9] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proc. of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, volume 3089, pages 292–304. Lecture Notes in Computer Science, Springer-Verlag, 2004.

[10] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proc. International Workshop on Software Engineering and Middleware (SEM)*, pages 106 – 113. ACM Press, 2005.

[11] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. of the 10th International Static Analysis Symposium, SAS*, pages 1–18. Springer-Verlag, June 2003.

[12] W. R. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 2005.

[13] C. Gould, Z. Su, and P. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *Proc. of the 26th International Conference on Software Engineering (ICSE) - Formal Demos*, pages 697–698. IEEE Computer Society Press, 2004.

[14] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Automated Software Engineering (ASE)*. Association for Computing Machinery (ACM), Nov 2005.

[15] W. G. J. Halfond and A. Orso. Combining static analysis and runtime monitoring to counter SQL-injection attacks. In *Proc. of the 3rd International ICSE Workshop on Dynamic Analysis (WODA)*. IEEE Computer Society Press, May 2005.

[16] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior. In *Proc. of the 11th International World Wide Web Conference (WWW)*, May 2003.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. of the 12th International World Wide Web Conference (WWW)*, May 2004.

[18] R. McClure and I. Kruger. SQL DOM: Compile time checking of dynamic SQL statements. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 88–96. IEEE Computer Society Press, 2005.

[19] E. Merlo, D. Letarte, and G. Antoniol. Insider and ousider threat-sensitive SQL injection vulnerability analysis in PHP. In *Proceedings of IEEE Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2006 (to appear).

[20] G. Ollmann. Second-order code injection attacks. In *Technical report*. NGSSoftware Insight Security Research, 2004.

[21] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2005.

[22] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of SQL attacks. In *Proc. Detection of Intrusions and Malware Vulnerability Assessment Conference (DIMVA)*, pages 123–140. IEEE Computer Society press, July 2005.

[23] K. Wei, M. Muthuprasanna, and S. Kothari. Preventing SQL injection attacks in stored procedures. In *Proc. Australian Software Engineering Conference (ASWEC)*, pages 191 – 198. IEEE Computer Society Press, 2006.