Titre: Title:	Insider threat resistant SQL-injection prevention in PHP
Auteurs: Authors:	Ettore Merlo, Dominic Letarte et Giuliano Antoniol
Date:	Mai 2006
Type:	Rapport / Report
Référence: Citation:	Merlo, E., Letarte, D. & Antoniol, G. (2006). Insider threat resistant SQL-injection prevention in PHP (Rapport technique n° EPM-RT-2006-04).

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3138/
Version:	Version officielle de l'éditeur / Published version Non révisé par les pairs / Unrefereed
Conditions d'utilisation: Terms of Use:	Autre / Other

Maison d'édition: Publisher:	Les Éditions de l'École Polytechnique
URL officiel: Official URL:	https://publications.polymtl.ca/3138/
Mention légale: Legal notice:	Tous droits réservés / All rights reserved

EPM–RT–2006-04

INSIDER THREAT RESISTANT SQL-INJECTION PREVENTION INI PHP

Ettore Merlo, Dominic Letarte, Giuliano Antoniol
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Mai 2006

EPM-RT-2006-04

Insider threat resistant SQL-injection prevention in PHP

Ettore Merlo, Dominic Letarte, Giuliano Antoniol
Computer Engineering Department
École Polytechnique de Montréal

Mai 2006

Dépôt légal :
Bibliothèque nationale du Québec, 2006
Bibliothèque nationale du Canada, 2006

EPM-RT-2006-04
Insider threat resistant SQL0-injectionprevention in PHP
par : Ettore Merlo, Dominic Letarte, Giuliano Antoniol
Département de génie informatique
École Polytechnique de Montréal

École Polytechnique de Montréal
Bibliothèque – Service de fourniture de documents
Case postale 6079, Succursale «Centre-Ville»
Montréal (Québec)
Canada H3C 3A7

Téléphone :	(514) 340-4846
Télécopie :	(514) 340-4026
Courrier électronique :	biblio.sfd@courriel.polymtl.ca

Insider threat resistant SQL-injection prevention in PHP

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 are either static sites, programs, or databases. Very often they are a mixture of these three aspects integrating relational databases as a back-end. Web sites require configuration and programming attention to assure security, confidentiality, and trustiness of the published information.

SQL-injection attacks rely on some 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.

Furthermore, insiders may introduce malicious code in a Web application, code that, when triggered by some specific input, for example, would violate security policies.

This paper presents an original approach that combines static analysis, dynamic analysis, and code reengineering to automatically protect applications written in PHP from both malicious input (outsider threats) and malicious code (insider threats) that carry SQL-injection attacks.

The paper also reports preliminary results about 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, sofware re-engineering

1 Introduction

Web applications, are used to distribute information from organizations to different users over a network. Often, they 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. These attacks rely on some weak validation of the textual input that is somehow used to build SQL queries to be submitted to a database.

In some specific contexts, maliciously crafted input, that contains SQL instructions or fragment of these, produces queries whose semantics is different from the one meant by the designers and may threaten the security policies of the underlying databases.

SQL-injections attacks have been described in the literature [9, 10]. Also, regardless of input validation, insiders may introduce malicious code in an application that, when triggered by some specific input, for example, would violate designers intention regarding security and accesses. Insider threats references can be found in [2, 4, 20, 24], but little research work is published about source level security analysis to protect against SQL-injection insider threats.

Although the reasons behind successful SQL-injections attacks are quite known, the problems still persist for several reasons.

Classes of defense against SQL-injection attacks include "defensive programming", sophisticated input validation, dynamic checks, and source level static analysis.

Defensive programming and input validation aim at preventing the insertion of "malicious" strings into SQL queries. Unfortunately they are are fairly sensitive to new patterns of attacks against which defenses were not planned. Furthermore, defensive programming may be better suited for new development since it's programming intensive, but is weaker in addressing the issue of protecting legacy systems.

Dynamic checks are sensitive to obsolescence of checks during time - the application evolves and new legitimate interactions are added into an application -

that may make the number of false positives increase.

Finally, static analysis suffers from issues about precision and execution performance, when complex languages features are considered, such as inter-procedural transfers of data and control, pointers, arrays, polymorphism, and so forth.

This paper presents an original approach to automatically protect applications written in PHP from both malicious input (outsider threats) and malicious code (insider threats) that carry SQL-injection attacks.

The presented approach 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.

Main contributions of this paper are:

- consideration of both outsider and insider threats
- dynamic analysis to automatically build "model-based guards" of legitimates SQL queries
- automatic protection of existing legacy applications written in PHP by appropriately inserting "model-based guards" in source code
- preliminary experimental assessment of the approach obtained on legitimate test cases and both outsider and insider 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 rely on 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 ones are available. Since we place our model validation guards in between Web applications and databases, we are transparent to the particularly adopted DB. However, for the sake of example, and experiments, we will assume that a $MySql$ [5] DBMS is used. We also assume that the DBMS engine implements an extended subset of SQL standard [7]. Also, in the following, we will not make an explicit distinction - unless required - between Databases (DB) and Database Management Systems (DBMS).

As proposed in [16] the presented protection model makes all communication between program variables (either user or programmer variables) and DB-API routines pass through "secured model-based guards" that perform appropriate security checks.

This approach transfers the burden of assuring the protection against SQL-injection attacks from application developers to a core team of developers responsible for security issues. Furthermore, the sheer size of the code to be "secured" is dramatically shrunk, 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".

Another reason why injection attacks may succeed is because the communication between an application and the DB-API routines is performed through plain text. 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 [13], the situation would be different if SQL queries would go through a specific DB-API routine (e.g., getInfo(loginStr, passwdStr)) in an API based architecture, variables can strongly be checked against syntax, domain ranges, and somehow semantic content.

The "getInfo" routine should be a secured one, under the "security" developers control, thus programmers are allowed to call such a routine, but are not allowed to directly interrogate the database to extract login names and passwords and verify the match outside the secured functions.

For example, a login name, may be an identifier (an alphabetic character followed by an arbitrary long sequence of alphanumeric characters) and the valid values of identifiers are those that appear in the DB table that contain registered users. A similar argument can be hold for the password: passwords are fixed length strings which may contain almost any character.

An issue that is not discussed in this paper, but that is worth to mention, is whether passwords themselves may contain SQL syntax. In principle, passwords are arbitrary text strings, but if they contain some SQL syntactic content, some program may be subject to SQL-injection when passwords are retrieved from the database and used in some, possibly malicious, computation.

In general the problem is that of second and higher order SQL-injections [22]. 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.

Higher order SQL-injection is not explicitly in the scope of this paper, so for the purposes of the following discussion, the reader should assume that applications mentioned in this paper are not sensitive to higher order SQL-injections either because the application manipulates passwords and similar free text information in a secure way, or because DB-API are "guarded" as proposed in this paper. Indeed, for the purposes of this paper, we assume that DB-API are "guarded", so passwords and other information stored in a database are assumed free of SQL syntactic content.

If all queries to a database were processed through secured "guards", many classes of SQL-injections and injections in general reported in the literature would be detected because of their mismatch of types and domain values of arguments to those functions.

Our approach aims at the construction of applications protected from SQL-injections by building "secured model-based guards" that perform type and domain validation for parameters of SQL statements. While this could be an appropriate approach for new systems, existing applications often use string based dialog to API for DB accesses, such as the very common "mysql(str)" call. In this context, security checks have to be performed on the strings passed as parameters to DB. This perspective to create model-based monitors for DB queries, has been introduced in [16]. In their approach, the source of model, i.e., the source of knowledge for understanding what are legitimate queries to a DB, is the application itself. Their approach, by using static analysis, builds syntactic models for all queries that can be built by computation on input variables which are considered as having one specific type and taking values in a specific domain.

In our perspective of building an insider-threat resistant model, this is not enough, since we are questioning the very source of the computation. In other words, we cannot take the source code and use static analysis to infer programmers' intention, since we assume that their intention may be malicious and may sabotage DB protection strategies.

The source of legitimate queries is the set of interaction specifications for an application. Following some simple and conventional software development process, just for sake of discussion, we may assume that a Web application went through requirements analysis, specification, design, coding, testing, and deployment phases. At requirements analysis time, requirements for some functionalities had been elicited and legitimate interactions had been described. Interactions would also be more formally described in the Web UI specifications of applications.

Intuitively, from a developer's point of view, quite often a Web form corresponds to one SQL-query or to a small number of variations from a stem query, so it's reasonable to think that specifications should describe the legitimate interactions associated to a Web form.

When, ideally, this high level process is followed, models for legitimate queries can be derived from specifications. Also, scenarios developed during design, would contain information about legitimate "uses" of database accesses depending on different roles and users of the application.

In the context of security analysis, we have to go further. We may envisage scenarios that contain information about "secure" and legitimate uses of the application and also scenarios that contain known and representative attacks to the application.

Both legitimate and security threatening scenarios can be used during the test phase to verify and validate an application under the security perspective.

In this "ideal development" context, protecting DB accesses from injections by users or by programmers would be relatively easy: while adopting the same model-based validation perspective of [16], models could be built from the UI specifications and security scenarios rather than from the source code of the application. If models were built independently from source code development, but consistently with specifications and scenarios, it would be much harder or virtually impossible for a user or a programmer to gain illicit access to database queries by working around specification-based models.

Security models must deal with fixed SQL queries, parameterized queries, and queries with free qualifiers, sub-queries, and other SQL elements. At any given call site, a dedicated and specialized security model call is automatically added into an application by source code transformations, as follows:

$$\begin{array}{c} ... \\ getInfo(str) \\ ... \end{array}$$

where, for example, str may be:

$$\begin{array}{l} SELECT\ info \\ \qquad\qquad FROM\ login =' abc'\ AND \\ \qquad\qquad\qquad passwd =' xyz' \end{array}$$

```
getInfo(String str) {
    if  (loginMatch(str))
        return(mysql(str));
    else
        return(1);
}
```

Figure 1. Secure "getInfo"

Using the approach in [16] and building syntactic models of UI specifications and security scenarios, the implementation could be the following:

```
getInfo(String str) {
    ast = sqlParse(str);
    if  (loginMatch(ast))
        return(mysql(str));
    else
        return(1);
}

bool  loginMatch(AST_type ast)
    if (loginModel.contains(ast)) {
        return(true);
    else
        return(false);
}
```

Figure 2. Parsing based "getInfo"

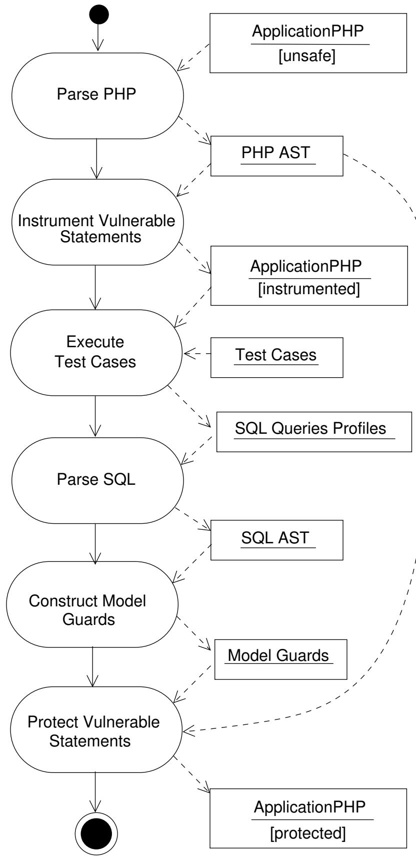

Figure 3. Scheme of instrumentation

Existing systems haven't, in general, been built with the proposed security model in mind. To make adoption of the proposed security model effective, some automatic transition must be put in place for evolving existing systems into secure ones. Again, this requires some shift of effort and responsibility from application developers to the security team, as already mentioned.

We need some automatic help to construct model validation routines for SQL queries. We propose a dynamic analysis that helps in quickly building the model validation routines for legitimate uses of a systems. Fig. 3 depict the overall prevention process.

3 Model-based guards 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 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". This kind of software may or may not be triggered from input data. For example, some malicious code may force some secure table from the database to be printed, in addition to a user's legitimate query. If this SQL code is added to all queries, it may not necessarily be triggered by some input. Of course, this kind of malicious code has very little chances to go undetected during testing, so more dangerous kinds of malicious code are triggered by input data, as, for example, when a particular login name is allowed to obtains secure DB tables when a particular query is presented 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.

The proposed insider threat resistant SQL-injection prevention approach relies on breaking the "appropriate conjunction" of input data, code, and DB access condition that would allow SQL-injection. Intuitively, appropriate "guards" will be inserted between input data, internal code, and DB accesses, so that the appropriate conjunctions of SQL-injection conditions are prevented from occurring. We found 421 occurrences of vulnerable statements in the phpBB version used for the experiments.

As depicted in Fig.3, first we instrument the PHP code to collect samples of queries that should be legitimately use at DB-API call points (often "mysql(str)"). Second we run test cases and implementations of security scenarios to gather the queries that dynamically are generated at different call sites.

Parsing these queries returns the AST's of legitimate queries. At each call site, the set of legitimate queries constitute a small language which is a subset of SQL language. Model validation routines, that we call "guards", can be quickly built by synthesizing SQL sub-grammars that represent the legitimate queries at each call site. The proposed approach is robust in the sense that only explicitly acceptable queries are sent to DB-API. Other strategies, that try to block attack patterns, are, in general, weak in blocking newly conceived attacks. New malicious syntactic variations do not belong to the "allowed" language recognized by model-based guards and are therefore rejected.

The cardinality of legitimate queries used in the experiments is 1590. Legitimate queries are about 18 tokens long on average, with minimum length 4 and maximum one 82. When parsed by the SQL parser, these queries give rise to 417 distinct ASTs for 107 distinct call sites that are used to build model-based guards.

To build model guards from sets of legitimate queries, the process is quite straightforward. Legitimate queries are parsed and the corresponding forest of AST is stored for each call site. Stored AST are a little generalized since constants, strings and other kinds of user data are kept in the AST by their type rather than by their 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.

The proposed approach relies on the assumption that the number of different SQL-query structure that are produce at each call site is quite small, since each call site has a good chance of representing a particular family of queries related to some legitimate composition of user variables.

The effectiveness of automation of the proposed approach relies on the small cardinality of legitimate queries at a call site. 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.

Model guards at call sites are built by parsing an input string that would be sent to a DB-API and by matching the produce AST with the stored references ones for legitimate queries. A match guarantees that the syntactic structure has been allowed for passing down to the DB-API, while it is conservative to reject queries that do not match reference ASTs for a call site. Model-based guards can be automatically synthesized based on a positive sample of legitimate queries.

We can assume that the test sets and security scenario's may not be complete and some legitimate queries may be rejected because they never contributed to the sub-grammars used to generate guards. This is possible, because we try to infer programmers' intention using a subset of legitimate uses. The proposed approach helps in building a transition to secure systems from old systems that haven't necessarily been conceived with security in mind.

Although security specifications would be required, to make sure that the model-based guards be complete at least with respect to specified scenarios, the authors believe that in many application the set of allowed queries is quite small and that the presented approach is advantageous in those cases.

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

Vulnerable statement 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.

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

becomes:

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

from a practical point of view the actual call has been implemented as:

$$\ldots$$
$$call\_site_i : sqlModelGuard(i,\ str,\ db))$$
$$\ldots$$

```
int sqlModelGuard( int i,
                   String str,
                   dbHandlerType db) {
    ast  = sqlParse(str);
    if (astMatch(ast, sqlModel[i]))
        return(mysql(str, db));
    else
        // Rejected string
        return(1)
}
```

Figure 4. Example of model-based guard

The implementation of model-based guards and the source code re-engineering to change calls to *mySql* into calls to the model-based guards are quite straightforward and require little effort. An example of model-based guard is reported in Fig. 4.

A parser for each of these sub-languages constitute 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 using prefix trees based on SQL parser supplied ASTs.

For practical reasons, rather than implementing separate prefix trees, we have implemented the conceptual equivalent of one prefix forest, in which call sites are the entries to individual prefix-trees.

4 Experiments

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

phpBB [6] is a well-known application 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 the reported 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 from the point of view of functionality (so it may still be old with respect to functionalities of current version) but with removed SQL-injection vulnerabilities both in terms of user data triggered injections and/or possibly malicious code injected by an insider.

Unfortunately, security specifications, user interface specifications and specifications in general, for phpBB are not available together with the software distribution. This is likely to be the case of several systems and, to collect patterns of legitimate queries, we have automatically instrumented phpBB by inserting security probes before vulnerable statements. Automatic instrumentation has been achieved by first modifying a Web available PHP grammar [3] by Satyam implemented in JavaCC [1]. Table 1 gives some feature of the parsers built and used for this project.

	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 for the instrumentation.

Simple security test cases and data for the proof-of-concept approach have been built and divided in two classes: that of legitimate cases and that of outsiders' attacks. Execution of legitimate security test cases gives legitimate patterns of SQL usage. Execution of attacks gives SQL-injections that should be blocked in the "secured" version.

Probed strings at vulnerable call points gathered during test execution, constitute the sub-languages of SQL queries that are accepted at different call sites.

To parse SQL queries we have built our SQL parser by considering the *mySql* syntax available in [5]. Features of our SQL parser are reported in Table 1. Simpler SQL grammars are available from the JavaCC grammar repository site [1].

Making phpBB secure consists in automatically replacing calls to *mySql* with calls to the model-based guards, at each call site. Similarly to the problem of security probe instrumentation, we have written a visitor based on our PHP grammar that makes such a replacement automatically on all 421 vulnerable instances of *mySql* calls.

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

4.1 Outsiders SQL-injections experiments

Since the intersection between legitimate cases and attacks is in principle empty, 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 attacks that had been proven successful before software protection.

The purposes of the presented experiments is to assess the number of false positives and false negatives obtained by running the legitimate test cases and outsiders' attacks against the secured application.

	Tests	Outsider attacks	Insider attacks
Total number	1590	176	312
Accepted	1590	0	0
Accepted (%)	100	0	0
Rejected	0	176	312
Rejected (%)	0	100	100

Table 2. Experimental results

Both legitimate accesses and outsider attacks can be simulated by the appropriate sequences of URL and parameters on the URL line. Table 2 presents the results obtained from the execution of 176 outsider attacks.

4.2 Insiders SQL-injections experiments

Experiments to simulate insiders' attacks and to assess the protection against insiders' required some more work. We have assumed that insiders can program anything they want to be fed to the model-based guards.

We have written another visitor based on our PHP grammars that inserts an instance of attack generator before any call to model-based guards, as follows:

$$\begin{array}{rl} & ... \\ & str \ = \ generate\_attack(); \\ call\_site_i: & model\_guard(\ i,\ str,\ db); \\ & ... \end{array}$$

Furthermore, proper input sequence have to be conceived to reach vulnerable statements and activate the malicious code simulated by *generate_attack*().

The purposes of the presented experiments is to assess the number of false positives and false negatives obtained by running insiders' attacks against the secured application. Results obtained from the execution of 312 insider attacks are reported in Table 2.

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 has been detected in the presented experiments). Intuitively, it's somehow expected, since 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.

What's quite remarkable is that the same precision of detection is observed for insider attacks. This is definitely an advantage of our approach with respect to published ones in the literature that didn't address the insider threat perspective.

Nevertheless, false alarms and mis-classified attacks may still happen if unauthorized SQL queries were by mistake 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 because, by design, SQL queries to a database from an network based interactive application, like Web applications, are in a certain way related to the number of forms in the same application. This would suggest that the number of syntactic patterns of SQL queries at the different call points, may be relatively proportional to the number of distinct forms.

Conversely, this approximation may be reduced, when developers from the security team explicitly construct model-based guards hopefully based on security specifications.

Also, the proposed automatic 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.

At one extreme, security developers may even craft by hand specific sub-grammars for protecting fewer call points that receive complex and highly variable queries.

Call points that definitely receive complex queries and potentially all SQL language are full text entries which are reserved for DB administration purpose or system administrator accesses. These kinds of accesses are not covered by the presented approach, but, in general, these call points are accessible only from files and programs that required administrator privileges to be accessed and may not be vulnerable to SQL-injections from non-privileged users.

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

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

Instrumentation of application for SQL profiling and dynamic analysis is linear in the size of the application. Model guards construction is proportional to the cardinality 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, which is often relatively small. Checking the parsed query against a call point model takes, again, a time proportional to the length of the parsed query.

At the current moment, 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 to created that dialog with phpBB, and so on.

On the other hand, advantages of the proposed approach are the portability of the approach to other languages used in network based application like Java. In our laboratory, we have developed, in the past years, 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 somehow 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 [8] and so forth.

6 Related work

AMNESIA [16, 17] 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) [12].

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 work 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.

Significant differences are 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. Their objective is to separate application generated queries from user injected ones. Their approach in intrinsically vulnerable 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 "believe" the application code to build static models of legitimate queries. Malicious code introduced in the application would not be recognized as "trustworthy" by our approach that is intrinsically insider threat resistant.

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) [23] detects and prevents SQL-injection attacks in PHP code.

They modify 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 developers originated parts of SQL strings) and the system may not therefore be protected against insider threats.

JDBC Checker [15] combines automata-theoretic 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. Their tool is used to flag potential errors or verify their absence in dynamically generated SQL queries. JDBC checker can play a role in SQL-injection detection similar to that of JSA in AMNESIA.

SQL DOM [21] 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, their approach shifts the reasoning from the world of dynamically generated queries (that may create runtime problems) to that of compile time declarative checking of compliance with safe classes.

Their approach 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.

Their approach, again, protects an application against malicious user input, while it's oblivious of insider threats.

While some 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, while our approach restricts the static models to the set of queries that are actually and legitimately used by the application. 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 [13] 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.

Other approaches for detecting and protecting applications against SQL-injection can be found in [11, 18, 19, 25]. In previous work [14] the authors detected buffer overflow problems using genetic algorithms.

7 Conclusions

An original approach that combines static analysis, dynamic analysis, and code re-engineering to automatically protect applications written in PHP from both malicious input (outsider threats) and malicious code (insider threats) that carry SQL-injection attacks has been presented, implemented and evaluated.

phpBB Web application has been automatically protected by using the proposed approach. 176 outsider attacks, that had been proven successful before software protection, together with 312 insider attacks have been 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 is needed to evaluate it on larger and more diversified systems.

8 Acknowledgements

This research work has been partially funded by the National 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] National Infrastructure Security Co-ordination Centre. *http://www.uniras.gov.uk/niscc/index-en.html.*

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

[4] U.S. Department of Energy. *http://www.ciac.org/ciac/CIACHome.htm.*

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] C. Del Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1037–1043. IEEE Computer Society press, 2005.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Keeney, D. Cappelli, E. Kowalski, A. Moore, T. Shimeall, and S. Rogers. Insider threat study: Computer system sabotage in critical infrastructure sectors. Technical report, United States Secret Service and CERT Coordination Center/SEI, May 2005.

[21] 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.

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

[23] 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.

[24] M. R. Randazzo, D. Cappelli, M. Keeney, A. Moore, and E. Kowalski. Insider threat study: Illicit cyber activity in the banking and finance sector. Technical report, United States Secret Service and CERT Coordination Center/SEI, August 2004.

[25] 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.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL