Using Parse Tree Validation to Prevent SQL Injection Attacks

Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti
Email: {buehrer,weide,paolo}@cse.ohio-state.edu

Computer Science and Engineering
The Ohio State University
Columbus, OH 43210

Abstract. An SQL injection attack targets interactive web applications that employ database services. Such applications accept user input, such as form fields, and then include this input in database requests, typically SQL statements. In SQL injection, the attacker provides user input that results in a different database request than was intended by the application programmer. That is, the interpretation of the user input as part of a larger SQL statement, results in an SQL statement of a different form than originally intended. We describe a technique to prevent this kind of manipulation and hence eliminate SQL injection vulnerabilities. The technique is based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. Our solution is efficient, adding about 3 ms overhead to database query costs. In addition, it is easily adopted by application programmers, having the same syntactic structure as current popular record set retrieval methods. For empirical analysis, we provide a case study of our solution in J2EE. We implement our solution in a simple static Java class, and show its effectiveness and scalability.
Keywords: SQL Injection, Distributed Systems, J2EE, SQL Parsing

1 Introduction

Web applications employing database-driven content are ubiquitous. Companies whose business model focusses on the Internet, such as Yahoo! and Amazon, are obvious examples but it is important to note that most every major company has a web presence, and this presence often uses a relational database. These databases may include sensitive information, such as personal customer data. Typically, the web user supplies information, such as a username and password, and in return receives customized content. It is also common to have public content available through an external site, and then co-host an intranet on the same web server.

A variety of technologies exist that provide a framework for web applications, including Sun's J2EE (Java Server Pages, Servlets, Struts, *etc.*), Microsoft's Application Server Pages and ASP.NET, PHP, and the Common Gateway Interface (CGI). All of these models make use of a tiered environment. Typically, three

tiers are employed: the presentation tier, middle tier, and data tier. The presentation tier uses a web browser to capture user input and present information output. This is typically HTML, but occasionally intranet applications make use of a custom thin client. The middle tier, also known as the business tier, encapsulates the business logic that drives the application. This software layer is responsible for constructing information requests to the database layer. The database tier is typically a relational database and provides storage services.

As a motivating example, consider an online banking application. The credit union allows clients to log in and view their accounts, make payments, *etc.* Clients provide credentials by typing their username and password into a web page (the presentation tier). The web page posts this information as name:value string pairs (input field:value) to the middle tier. The middle tier uses this input to create a query, or request, to the database layer. This is almost always SQL (Structured Query Language), such as *SELECT * FROM users WHERE username='greg' and password='secret'*. The data layer processes the request and returns a record set back to the middle tier. The middle tier manipulates the data, creates a session, and then passes a subset to the presentation tier. The presentation tier renders the information as HTML for the browser, displaying a menu of account options and personalized information about account balances and transaction history. Notice that, in this example, the middle layer generates an SQL request based on input supplied by the user.

Because of the popularity of these types of applications, techniques to exploit their security vulnerabilities are potentially quite dangerous. One such technique is called SQL injection. It occurs when user input is parsed as additional SQL tokens, thus changing the semantics of the underlying query. Two well-known companies whose public web sites were vulnerable to such attacks are Guess and Petco. Both vulnerabilities were discovered by a curious 20 year old programmer in 2003. As a result, PetCo. exposed 500,000 credit cards, and required a settlement with the Federal Trade Commission[16, 19].

In this paper, we present a novel runtime technique to eliminate SQL query injection. We make the observation that all SQL injections alter the structure of the query intended by the programmer. By capturing this structure at runtime, we can compare it to the parsed structure after inserting user-supplied input, and evaluate similarity. We make the assertion that it is more effective to measure the results of the input than to attempt to validate the input prior to inserting it into the proposed query. By incorporating a simple SQL parser, we can evaluate all user input without requiring a trip to the database, thus lowering runtime costs. Our method aims to satisfy the following three dimensions:

- eliminate the possibility of the attack.
- minimize the effort required by the programmer.
- minimize the runtime overhead.

In addition to describing the structure of the technique, we present an implementation of our solution in a J2EE case study and evaluate it based on these three criteria. We conclude that our solution provides a practical and useful security tool for middleware developers, and make the code publicly available. We

note that our technique can be instrumented into all known web technologies. It requires no particular language or technology, and as such is not limited to a specific platform.

2 Background: SQL Injection

2.1 Web Server Technology

Web applications accept user input via forms in web pages. This input is posted to the server as name-value pairs, both of which are strings. An alternate mechanism to pass information to the server is the query string. The query string is information appended to the end of the URL. On most web servers, a question mark separates the resource from the query string variables. Each name value pair in the query string is separated by an ampersand, and the user is free to edit this input as easily as form inputs.

Because it is common for web servers to not differentiate between variables passed in the query string and those posted in the form, we will consider both as user input.

Web forms can also have hidden fields. These fields are a tool for the programmer to maintain state across web pages. These fields are hidden for aesthetic purposes only. It is relatively easy for an attacker to examine the source code of the web page, and either place a hidden field in the query string or save the form to local storage and modify it. Therefore, we will treat all hidden fields as user-supplied input as well.

2.2 SQL Injection Defined

SQL injection is a type of security attack whereby a malicious user's input is parsed as part of the SQL statement sent to the underlying database. Many techniques exist, but the overriding theme is that the user manipulates knowledge of the query language to alter the database request. The goal of the attack is to query the database in a manner that was not the intent of the programmer. We present several examples in this section.

2.3 SQL Injection Techniques

Tautologies One method to gain unauthorized access to data is to insert a tautology into the query. In SQL, if the where clause of a SELECT or UPDATE statement is disjuncted with a tautology, then every row in the database table is included in the result set.

To illustrate, consider an online message board application. Assume that a user has logged into the web site properly. To update their account information, the user navigates to the proper page. In the query string, the member id is maintained. The web page submits, in the query string, *details.asp?id=22*. In middleware, the query *SELECT * FROM members WHERE memberid=22* is

generated, and the appropriate information is returned back to the user. However, an attacker could easily manipulate this interaction by editing the query string to contain a tautology, such as *details.asp?id=22 OR 1=1*. The resulting query is then *SELECT * FROM members WHERE memberid=22 OR 1=1*, which will select the top user in the users table (user id 1 may not be a valid id). Often, this is an admin, because they are the first to use the application. The attacker is now free to read and modify the admin's credentials.

Checking for tautologies can be difficult, because SQL allows a wide range of function calls and values. Simply checking user input for patterns such as *n=n* or even for an equals sign is not sufficient. For example, other SQL tautologies are *'greg' LIKE '%gr%'* and *'greg'=N'greg'* which are based on operators, such as *LIKE* and *=N* that are not typical math operators.

UNION Queries Another SQL injection technique involves the *UNION* keyword. SQL allows two queries to be joined and returned as one result set. For example, *SELECT col1,col2,col3 FROM table1 UNION SELECT col4,col5,col6 FROM table2* will return one result set consisting of the results of both queries[1]. Let us return to our previous example, *SELECT * FROM members WHERE memberid=22*. If the attacker knew the number and types of the columns in the first query, an additional query such as *SELECT body,results FROM reports* can be appended. For some applications, surmising this information is not difficult. If the programmer is not consuming all exceptions, incorrect SQL queries will generate error messages that expose the needed information[12]. For example, if the two queries in the *UNION* clause have a disparate number of columns, an error such as *All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists* will be returned. The attacker merely changes the query to *SELECT * FROM reports WHERE reportid=22 UNION SELECT username,password,1 FROM members* to try additional queries. The attacker would then continue to add dummy columns until an error such as *Syntax error converting the varchar value 'feet' to a column of data type int* occurs. This signals that the column count is correct, but at least one column type is not. The attacker then alters the types accordingly.

Additional Statements An interesting point when using a *UNION* to select multiple requests is that often only the first row of the result set is used by the web page. In this scenario, the attacker does not necessarily benefit from returning more rows than what was intended. However, a similar but alternative approach is to *UNION* the first query with a new query that does not return addition results. These new queries often perform specific actions on the database which can have disastrous consequences. For example, Microsoft's SQL Server provides two system level stored procedures designed to aid administrators. These tools inadvertently provide powerful features to an attacker. The first is the *xp_cmdshell(string)* command. This function can be *UNION*'d with any

[1] Subject to some constraints, such as matching types.

other query. The effect is essentially that the command passed to the function will be executed on the server as a shell command. Another interesting system procedure is *sp_execwebtask(sql,location)*. This command executes the SQL query and saves the results as a web page at the specified location. The location need not be on the database server, it can be placed in any accessible UNC path. For example, *SELECT * FROM reports WHERE reportid=22 UNION sp_makewebtask '\\IP_address\ share\test.html','SELECT * FROM users'* will create an HTML page of the entire users table onto the server IP_address.

Using Comments SQL supports comments in queries. Most SQL implementations, such as T-SQL and PL/SQL use '- -' to indicate the start of a comment (although occasionally '#' is used). By injecting comment symbols, attackers can truncate SQL queries with little effort. For example, *SELECT * FROM users WHERE username='greg' AND password='secret'* can be altered to *SELECT * FROM users WHERE username='admin' - - AND password="*. By merely supplying *admin' - -* as the username, the query is truncated, eliminating the password clause of the WHERE condition. Also, because the attacker can truncate the query, the tautology attacks presented earlier can be used without the supplied value being the last part of the query, or an int. Thus attackers can create queries such as *SELECT * FROM users WHERE username='anything' OR 1=1 - - and password='irrelevant'*. This is guaranteed to log the attacker in as the first record in the users table, often an administrator.

2.4 Mass SQL Injection Discovery Techniques

It is not required to visit a web page with a browser to determine if SQL injection is possible on the site. Typically, a malicious user will program a web crawler to insert illegal characters into the query string of a URL (or an HTML form), and check for errors in the result. If an error is returned, it is a strong indication that the illegal character, such as ', was passed as part of the SQL query, and thus the site is open to manipulation. For example, Microsoft Internet Information Server by default will display an ODBC error if an unescaped single quote is passed to SQL Server[12]. The crawler simply searches the response text for ODBC messages.

3 SQL Parse Tree Validation

A parse tree is a data structure for the parsed representation of a statement. Parsing a statement requires the grammar of the language that the statement was written in. By parsing two statements and comparing their tree structures, we can determine if the two queries are equal.

When a malicious user successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query do not match. By intended SQL query, we mean that when a programmer writes code to query the database, she has a formulation of the structure of the query. The

programmer-supplied portion is the hard-coded portion of the parse tree, and the user-supplied portion is represented as empty leaf nodes in the parse tree. These nodes represent empty literals. What she intends is for the user to assign values to these leaf nodes. These leaf nodes can only represent one node in the resulting query, it must be the value of a literal, and it must be in the position where the holder was located.

An example of her intended query is given in Figure 3. This parse tree corresponds to the example we presented in Section 1, *SELECT * FROM users WHERE username=? AND password=?*. The question marks are place holders for the leaf nodes she requires the user to provide. While many programs tend to be several hundred or thousand lines of code, SQL (structured query language) statements are often quite small. This affords the opportunity to parse a query without adding significant overhead.

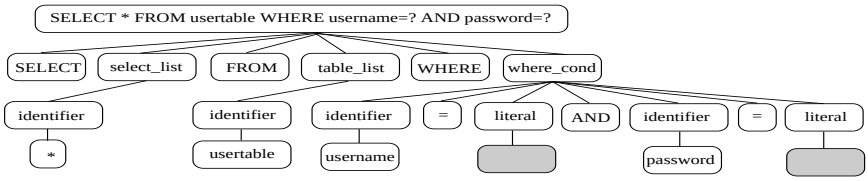

Fig. 1. A SELECT query with two user inputs.

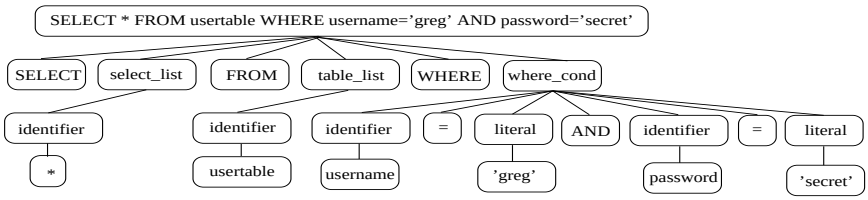

Fig. 2. The same SELECT query as in Figure 3, with the user input inserted.

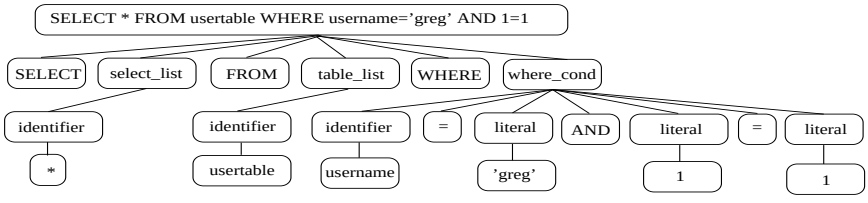

Fig. 3. The same SELECT query as in Figure 3, with a tautology inserted.

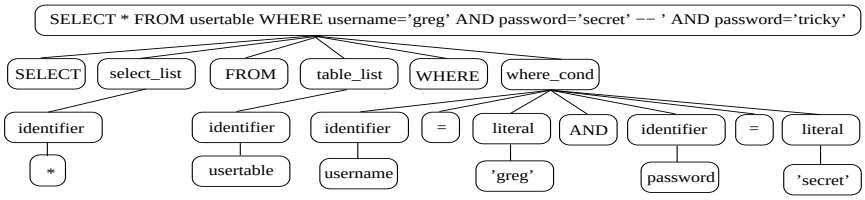

Fig. 4. The same SELECT query as in Figure 3, with a subtle shadowing attack.

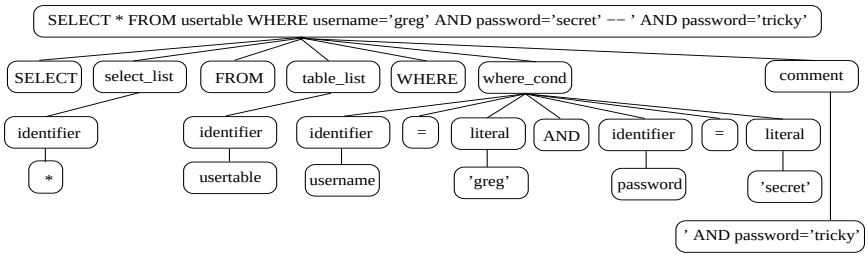

Fig. 5. The same SELECT query as in Figure 3, including the comment.

3.1 Dynamic Queries

One significant advantage over static methods is that we can evaluate the exact structure of the intended SQL query. Many times this structure itself is a function of user input, which does not permit static methods to determine its structure. An example is a search page. One popular free web-based email tool allows users to search through their messages for particular content. This search may or may not include searching through the email body, subject, from field, *etc.* Typically in these types of searches, if the user leaves one or more of these fields empty, the code does not incorporate them into the SQL query Typically, the programmer will append inputs from these fields on the WHERE portion of the query, such as *FROM table1 WHERE subject LIKE '%input%'*.

Another example is result set sorting. Many web applications which allow the user to sift through tabular results permit sorting the table by any of the columns. This functionality is easily accommodated by appending an *ORDER BY column1,column2* clause on the end of the query. Because this clause may or may not be present, and can be any number of columns, static analysis cannot determine the exact query at compile time. As our method establishes the query at runtime, verifying these queries is as straight forward as the others.

3.2 Comment Token Inclusion

One subtle case of attacks deserves special discussion. Our solution compares the parse tree before the user-supplied fields have been inserted and after the user-supplied fields have been inserted. If the user knew the table names and the structure of the targeted SQL query, it is possible he could customize his attack to mimic the query. He could inject the exact query in the first field, and supply and value for all subsequent fields (by commenting out the later portions. An example is illustrated in Figures 3, 3 and 3. The original query is given in Figure 3, with two user-supplied fields (shaded gray). If the user were to supply *greg' and password='secret' - -* for the username field and *tricky* for the password field, the resulting parse tree would be as in Figure 3. The potential vulnerability is that the programmer may assume that since the query parsed properly, the value stored in the request object's password field is the same value which was used to build the query. While the data returned by the query is proper, the state of the program is most likely not the state assumed by the programmer. Fortunately, our parse tree mechanism will catch this subtle hazard. The solution is to incorporate the comment of the query as a token in the parse tree. Figure 3 is the result from the same input, with this comment token included. Because the original query does not have a comment token, the resulting parse tree is no longer a match. This does not restrict the programmer from annotating queries. Had a comment existed in the original query, the parse would still have failed because the value (string literal value) of the two tokens would not be equal. By including the comment as a token, every SQL injection is detected, regardless of whether the query will return the results intended by the programmer.

3.3 Implementation

We implement our solution in Java. At the core of this solution is a single static class, `SQLGuard`, which provides parsing and string building capabilities. The programmer uses this class to dynamically generate, through concatenation, a string representing an SQL statement and incorporating user input.

Each SQL statement string is prepended with `SQLGuard.init()`, which generates and returns a fresh key. By generating a new key for each database query, we allow for inadvertent discovery of a key, because successive page loads will always have a new key. When a user-provided string, `s`, is included within an SQL statement, it is first pre- and post-pended with the current key, by using `SQLGuard.wrap(s)`. It is imperative that the key not be guessable by an attacker. We use a sequence of 8 randomly generated ASCII characters (seeded by clock value, thread id, and an application guid).

Finally, the `SQLGuard` class has a private method, `verify()`, which removes the key from the beginning of the query and uses it in conjunction with the remaining portion of the query string to build two parse trees. The first tree has unpopulated user tokens for user input. The second tree is the result of parsing the string with these nodes filled in with user input The two trees are then compared for matching structure.

We use a static class, and maintain thread identities for each key. A simple sorted vector is used in the case the container application is multithreaded.

3.4 Correctness

The correctness of our approach is predicated on two assumptions. The first is that user input values are intended to be used *only* as leaves in the parse tree. That is, we assume that it is *not* the intent of the application to allow user input to include SQL statements or substatements. One could certainly imagine an application where this is not the case, for example a web tutor for learning SQL! For such situations, we could weakening the parse tree similarity check. Instead of verifying that the only change after user input was that user leaf nodes were replaced with terminal literals, we would verify that they had been replaced by subtrees. In practice, however, we are not aware of any real web applications designed to accept SQL directly from the user and so we have chosen to not implement this weaker similarity check.

The second assumption is that the key value is outside the space of possible user input (*i.e.*, it is unguessable by an attacker) and of possible programmer input (*i.e.*, it is not, itself, a valid SQL token or sequence of tokens). This assumption means that the *only* occurrences of the key in an SQL statement are a result of calls to `SQLGuard.init()` `SQLGuard.wrap()`. If used correctly, then, these key values bracket exactly the instances of string values in the statement arising from user input. Hence, replacing the substrings bracketed by the key values by a single token correctly generates the parse tree of the original, programmed, SQL statement template.

4 Case Study: ODOT Geo Web Application

Our case study is an application built for the Ohio Department of Transportation. The Ohio State University was contracted to engineer and develop a web application named the *Geological Hazard Inventory Management System*, hereafter referred to as Geo. Geo is designed to inventory hazards which adversely affect roads and highways throughout Ohio. It also provides cost estimations for various remediation techniques, allowing state engineers to make decisions and establish priorities for their fiscal resources. The application was built on J2EE using JSP, Java Classes, and Sybase, with Apache Tomcat as the application server. SQL queries are generally built from JSP-supplied user input, using Java's `SQL.Statement` class. We evaluate our SQL injection elimination technique with respect to its computational overhead, and how easily it incorporates into existing code.

4.1 Execution Overhead

To test the execution time overhead, we baseline our study by timing the application with traditional SQL queries, ie without our injection checking. We

then modify the queries as outlined in Section 3 and rerun the experiment. The web server is a 733MHz windows 2000 machine with 256MB RAM. Figure 6 shows the average parse time as a function of the length of the query. During this experiment, the web server was otherwise idle. We record the time to parse the query, execute the database call, return the results, and close connection. This amounts to the code in Figure 9, in the course of an HTTP response. We used several different queries and averaged their times over 20 trials each. Our guarded solution required approximately 1-3 ms more time to execute than traditional Statements, and was not a function of the length of the query. We feel this is because SQL queries are relatively short, typically less than 50 tokens. Both line shows a small slope, which is due to the increasing complexity of the query for the database engine.

Our second set of experiments aim to test our implementation under extreme load. To accomplish this task, we use Apache's JMeter load testing package.[2] It is freely available and tailored for testing the performance of Tomcat applications. JMeter allows one to schedule walks through a web application, including detailed web form submission and thus database querying. This script can exhaustively traverse the application. JMeter can then execute this script with a configurable number of concurrent users, either locally or remotely. It also allows the tester to configure the delay between requests for the simulated users. Our configuration is the same web server as in Experiment 1. We hosted the simulated clients in Santa Clara California to provide a significant response time challenge on the system. We set the users to request web pages aggressively; they use a $(500 + x)$ ms interval, where x was a Gaussian distribution from 0 to 500 ms. Figure 7 shows the response times for load tests with 10 concurrent users and 25 concurrent users. It can be seen that response times increase as the load is applied, eventually leveling out. Under extreme loads, the web server's response time increases significantly for both traditional Statements and Guarded Statements. This is primarily because the server simply does not have the resources to handle the requests. In addition, the overhead for Guarded Statements increases as the load increases. This is due to the additional class instantiations requested. The table in Figure 8 shows summary results for load tests of both query mechanisms for 10 users, 25 users, and 50 users. It can be seen that the overhead, under the worst load, is only 3%.

4.2 Ease of Use

A requirement of our solution is for it to be simple to incorporate into both new and existing software. Figure 9 compares the original source with the modified code for a sample query from the GEO application. As illustrated, the changes are quite minor (in bold face), requiring only two lines modified. The first change is to receive the connection object from the `SafeDriverManager` class, which we implemented. The `getConnection` method is used to obtain a `SafeStatement`

[2] See `http://jakarta.apache.org/jmeter/`.

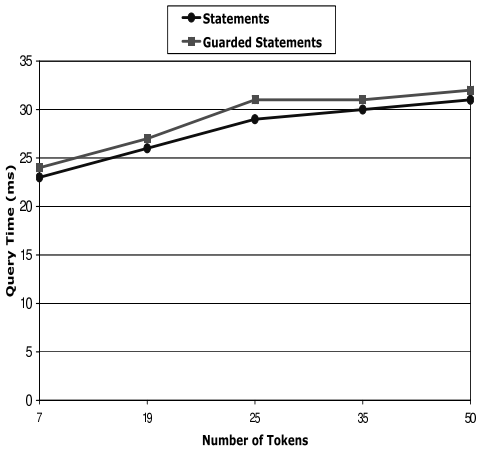

Fig. 6. Average query times for statements and guarded statements vs query size.

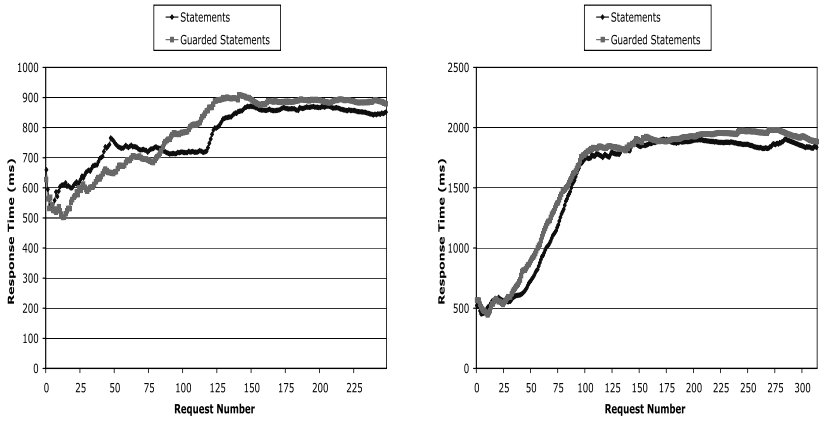

Fig. 7. Execution time for 10 and 25 simultaneous users.

Number of Users	Query Method	Avg Response Time (ms)
10	Statements	861
10	Guarded Statements	875
25	Statements	1857
25	Guarded Statements	1875
50	Statements	3552
50	Guarded Statements	3679

Fig. 8. Load testing results comparison for traditional Statements and Guarded Statements.

object. Because `SafeStatement` implements the standard JDBC `Statement` interface, existing code written for `Statement` also works with our class. The second change is to insert the `SQLGuard.init()` and `SQLGuard.wrap()` methods.

Traditional Method

```
Connection c = DriverManager.getConnection(strDBconn);
Statement s = conn.createStatement();
String q = ''SELECT * FROM reports WHERE id='' + id;
ResultSet RS = s.execute(q);
...
RS.close();
conn.close();
```

Guarded Method

```
Connection c = SafeDriverManager.getConnection(strDBconn);
Statement s = conn.createStatement();
String q = SQLGuard.init() + ''SELECT * FROM reports WHERE id='' + SQLGuard.wrap(id);
ResultSet RS = s.execute(q);
...
RS.close();
conn.close();
```

Fig. 9. Source code for unmodified SQL query and protected SQL query

5 Related Work

SQL injection has been the focus of a flurry of activity over the past two years. Although the security vulnerability has persisted for some time, recent efforts by hackers to automate the discovery of susceptible sites have given rise to increased invention by the research community [1, 8, 7, 17]. The industrial community has also gone to lengths to make programmers aware and provide best practices to minimize the problem [14, 13, 3, 20, 15, 18].

However, a recent study showed that over 75% of web attacks are at the application level and a test of 300 web sites showed 97% were vulnerable to web application attacks[19, 9]. Still, we believe the problem of SQL injection is readily solvable.

It has similarities to buffer overflow security challenges, because the user input extends passed its position in a query [6, 5]. At the heart of the issue is the challenge of verifying that the user has not altered the syntax of the query. As such, much of the work casts the problem as one of user input validation, and focuses on analyzing string inputs [2]. Several technologies exist to aid programmers with validating input. A simple technique is to check for single quotes and dashes, and escape them manually. This is easily beaten, as uses can simply adjust their input for the escaped characters. Programmers must then check for

pre-escaped text, *etc.* In [4], Moeller and Schwartzbach use a two-step process to assess what strings are possible from operations in Java. They first employ flow graphs, and then use these graphs to generate finite automata representing the set of strings possible. Several techniques use this static analysis approach to create models for the potential SQL queries [8, 7, 21].

Wasserman and Su [21] use the static analysis technique from [4] to generate finite state automata for modeling the set of valid SQL commands for each data access. This method offers guarantees of system behavior, and is quite helpful. It has several drawbacks. It cannot handle many queries, such as those with *LIKE*. This limitation is an implementation issue, and it is reasonable to assume that support for these as yet unsupported queries will be available in the future. However, as is a problem with any static design, it cannot model dynamically generated queries well. An example is a search where the user specifies the fields to search on, such as searching through emails. The user may want to search by topic, body, from, *etc.* Because any combination of these fields is possible, static analysis cannot determine the final structure of the query. Finally, their technique to prevent tautology attacks is explicit, and thus unnecessarily complicated. It attempts to ascertain (via SQL tokens) whether a boolean FSA is true. The cases for this analysis are large; it appears they miss simple boolean cases which do not use mathematical operators.

Recently, Halfond and Orso combined these static analysis techniques with dynamic verification [9, 10]. This nice work is most similar to our efforts. As with the method above, any technique whose model is built at compile time cannot predict the exact structure of the intended query, and overestimating will be required. Huang *et al.* [10] also employ both static and dynamic techniques, but they require detailed annotation by the programmer, and all sensitive functions must have preconditioned modeled.

Boyd and Keremytis [1, 11] developed SQLRand, a technique that modifies the tokens of the SQL language: Each token type includes a prepended integer. Any additional SQL supplied by the user, such as *OR 1=1*, would not match the augmented SQL tokens, and would throw an error. This approach is a useful strategy, and can effectively eliminate SQL injection. From a practicality standpoint, the programmer must generate an interface between the database tier and the middle tier which can generate and accommodate the new tokens. This is not a trivial endeavor. Secondly, and more importantly, these new tokens are static. As pointed out by the authors, many applications export SQL errors [12], and as such, the new tokens would be exposed. External knowledge of the new tokens compromises the usefulness of the technique.

PHP[3] has a technology called *Magic Quotes* which escapes input from the Request Object automatically. This feature has caused programmers difficulty. Because it is a server setting, applications are often assuming it is either on or off, and incorrect assumptions create either invalidated input or double escaped input. Attempts to alleviate programmer frustration, such as providing an API to check the setting at runtime, have had mixed results.

[3] http://www.php.net

A couple commercial platforms have incorporated strong typed technologies which assist programmers with SQL injection. J2EE has prepared statements, and Microsoft's .NET has commands. These two examples are fairly effective at minimizing SQL injection. Because they were originally created to allow for multiple queries on the same statement, however, they are not user friendly. Each parameter must be cast as the proper type by the user a priori, and must be set by individual lines of code. In addition, two database trips are required to formulate a model, and finally, most platforms do not support them.

In addition to eliminating SQL injection, we believe that a viable solution must not inconvenience the programmer. The main reason for writing queries as dynamically generated strings (as opposed to prepared statements) appears to be one of ease-of-use. Our approach blends the strengths of both: it has the convenience of dynamically generated strings and the security of prepared statements.

6 Conclusion

Most all web applications employ a middleware technology designed to request information from a relational database in SQL. SQL query injection is a common technique hackers employ to attack these web-based applications. These attacks reshape SQL queries, thus altering the behavior of the program for the benefit of the hacker. It is evident that all injection techniques modify the parse tree of the intended SQL. We have illustrated that by simply juxtaposing the intended query structure with the instantiated query, we can detect and eliminate these attacks. We have provided an implementation of our technique in a common web application platform, J2EE, and demonstrated its efficacy and effectiveness. This implementation minimizes the effort required by the programmer, as it captures both the intended query and actual query with minimal changes required by the programmer, throwing an exception when appropriate. We propose to make our implementation open to the public, to maximize its aid for the larger community. In the near future, we plan to implement our solution on the .NET framework and in PHP as well, as well as incorporate automated injection error logging.

References

1. S. W. Boyd and A. D. Keromytis. SQLRand: Preventing SQL injection attacks. In *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.
2. C. Brabrand, A. Moeller, M. Ricky, and M. Schwartzbach. Powerforms: Declarative client-side form field validation. In *World Wide Web*, 2000.
3. C.Anley. Advanced SQL injection in SQL server applications. In *http:/www.nextgenss.com/papers/advanced_sql_injection.pdf*, 2002.
4. A. Christensen, A. Moeller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium*, 2003.

5. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
6. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
7. C. Gould, Z. Su, and P. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.
8. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.
9. W. G. Halfond and A. Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. In *Online Proceeding of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, St. Louis, MO, USA, May 2005. http://www.csd.uwo.ca/woda2005/proceedings.html.
10. Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *In Proceedings of the 11th International World Wide Web Conference (WWW 03)*, 2003.
11. G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 03)*, 2003.
12. D. Litchfield. Web application disassembly with ODBC error messages. In *http://www.nextgenss.com/papers/webappdis.doc*.
13. P. Litwin. Stop SQL injection attacks before they stop you. In *http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/default.aspx*, 2004.
14. O. Maor and A. Shulman. SQL injection signatures evasion. In *http://www.imperva.com/application_defense_center/white_papers/ sql_injection_signatures_evasion.html*, 2004.
15. S. McDonald. SQL injection: Modes of attack, defense, and why it matters. In *http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenseandWhyItMatters.php*, 2005.
16. R. McMillan. Web security flaw settlement: FTC charges that Petco web site left customer data exposed. In *http://www.pcworld.com/news/article/0,aid,118638,00.asp*, 2004.
17. A. Nguyen-Tuong, S. Guarnieri, D. Green, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In Proceedings of IFIP Security 2005*, 2005.
18. SecuriTeam. SQL injection walkthrough. In *http://www.securiteam.com/securityreviews/5DP0N1P76E.html*, 2002.
19. W. Security. Challenges of automated web application scanning. In *http://greatguards.com/docs/insightweb.htm*, 2003.
20. K. Spett. SQL injection: Are your web applications vulnerable? In *SPI Labs White Paper*, 2004.
21. G. Wasserman and Z. Su. An analysis framework for security in web applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, 2004.