Using Parse Tree Validation to Prevent SQL Injection Attacks

Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{buehrer,weide,paolo}@cse.ohio-state.edu

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.

1. INTRODUCTION

Web applications employing database-driven content are ubiquitous. Companies whose business model focuses on the Internet, such as Yahoo and Amazon, are obvious examples; but nearly every major company has a web presence, and this presence often uses a relational database. These databases may include sensitive information, such as 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 provide frameworks for web applications, including Sun's J2EE (Java Server Pages, Servlets, Struts, etc.), Microsoft's ASP 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 web application. The bank 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. This attack occurs when user input is parsed as 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 [18, 21].

In this paper, we present a novel runtime technique to eliminate SQL injection. We observe that all SQL injections alter the structure of the query intended by the programmer. By capturing this structure at runtime, we can compare it

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEM 2005 September 2005 Lisbon, Portugal
Copyright 2005 ACM 1-59593-204-4/05/09 ...$5.00.

to the parsed structure after inserting user-supplied input, and evaluate similarity. We assert 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 call to the database, thus lowering runtime costs. Our method aims to satisfy the following three criteria:

- eliminate the possibility of the attack;
- minimize the effort required by the programmer; and
- 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. Our general strategy is not limited to any specific platform since it does not rely on any particular language mechanism or technology. This strategy can be instantiated for any existing web application framework.

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 not to 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 variations exist, but the central 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 application programmer. We present several examples in this section.

2.3 SQL Injection Techniques

2.3.1 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 banking application. Assume that a user has logged into the web site properly. To update their account information, the user navigates to the proper page using a (generated) query string where the user id appears, for example: *details.asp?id=22*. The value of the *id* name-value pair (i.e., the string *22*) is then used by the middle tier to generate the SQL statement *SELECT * FROM users WHERE userid=22*, 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*. Again, the value of *id* is used (i.e., the string *22 OR 1=1*) in constructing the SQL statement, which becomes *SELECT * FROM users WHERE userid=22 OR 1=1*. This query returns all rows of the *users* table.[1]

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.

2.3.2 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[2]. Let us return to our previous example, *SELECT * FROM users WHERE userid=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[14]. For example, if the two queries in the *UNION* clauses 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 users WHERE userid=22 UNION SELECT body,results,1 FROM reports* to try and match the number of columns. The attacker can continue to add dummy columns until an error such as *Syntax error converting the varchar value 'txfrs' 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 can then vary the types accordingly.

2.3.3 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. Given this fact, it seems the attacker does not necessarily benefit from returning more

[1] If a result set contains multiple rows where only a single row was expected, typically the middle tier uses only the first row. In the case of a table of passwords, the first row often corresponds to an administrator, making tautologies a particularly dangerous form of SQL injection.

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

rows than what was intended via the UNION attack. 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 that 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 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, but rather can be placed in any accessible UNC path. For example, SELECT * FROM users WHERE userid=22 UNION sp_makewebtask '\\IP_address\share\test.html','SELECT * FROM users' will create an HTML page of the entire users table at the server IP_address.

2.3.4 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. 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[14]. 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 statement's language. By parsing two statements and comparing their parse trees, 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. A leaf node 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. By restricting our validation to user-supplied portions of the parse tree, we do not hinder the programmer from expressing her intended query.

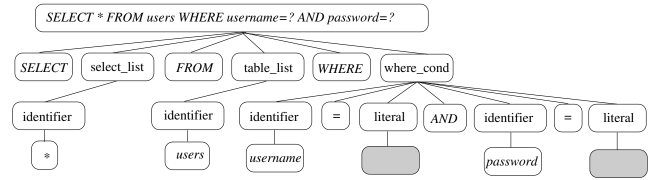

Figure 1: A SELECT query with two user inputs

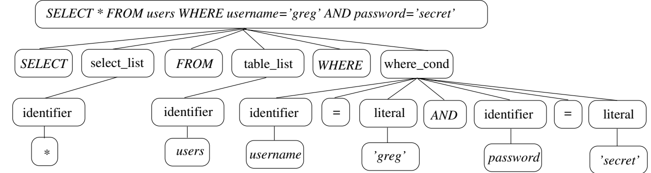

Figure 2: The same SELECT query as in Figure 1, with the user input inserted

An example of her intended query is given in Figure 1. 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.[3] While many programs tend to be several hundred or thousand lines of code, SQL statements are often quite small. This affords the opportunity to parse a query without adding significant overhead.

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. Normally the programmer will append inputs from these fields on the WHERE portion of the query, such as *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 func-

[3]In all the parse tree figures, we have simplified the typical where clause grammar rules due to space constraints.

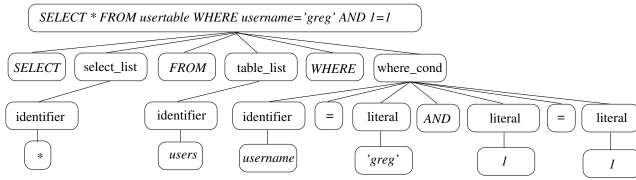

Figure 3: The same SELECT query as in Figure 1, with a tautology inserted

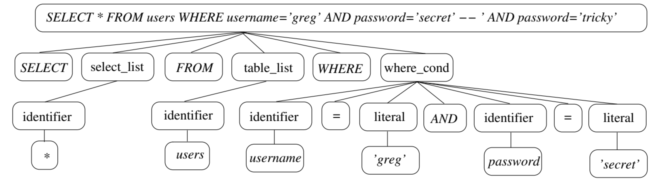

Figure 4: The same SELECT query as in Figure 1, with a subtle shadowing attack

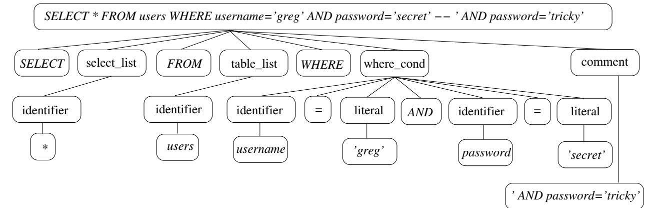

Figure 5: The same user input as in Figure 4, but including the comment as a token

tionality 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 straightforward as the others.

In our last example, we consider an online library application. The programmer has developed a front end application which allows the user to select publications based on his particular criteria. This front end has three user input fields. The first is a list, which supplies the variable to search on, such as author, title, or publisher. The second field is a list to provide the available operators, such as LIKE, greater than, equals, etc. The last field is a free form input to specify text. Two buttons are supplied, which are used to append the clause to the current query as either a conjunction or disjunction. Since the left side of the clause is a column name, and the right side is a user-supplied value, it is not possible for the two to be equal. A string on the right side uses quotes, and a value field uses only number characters. Also, if the programmer had a need to limit the selection, she could easily code her own clause in the middleware, such as ' AND accesslevel< 20'. The list fields which control the variable (author, title, etc) and thus the column of the database table (or view), are not text input by the user. The key in this example is that the user is choosing from a template structure that the programmer has supplied. In this example, although the user impacts the structure of the query, no nodes which are supplied by the user are unknown to the programmer. The user always supplies a clause as a tuple, namely (column,operator,value). Admittedly, this is not due to the parse tree approach, but due to proper front end programming. If the programmer were to allow free text supplied by the user to be parsed as SQL, then security is compromised. By definition, there can be no 'injection' in that case, because the user is not *injecting* SQL. The user is supplying SQL as intended by the programmer.

We would like to emphasize that we are not disallowing the program from using tautologies, or disallowing the programmer from permitting tautologies be supplied by the user. Eliminating tautologies is not the goal. The goal is to eliminate SQL injection, which is to eliminate the user from supplying text that the programmer did not intend to be SQL, but is parsed as such. A common SQL injection technique is to provide a text input such as *' OR 1=1'*, and have this input be parsed as part of the structure of the query.

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 values for all subsequent fields (by commenting out the later portions). An example is illustrated in Figures 1, 2 and 5. The original query is given in Figure 1, 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 2. 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 catches this subtle hazard. The solution is to include the comment as a token in the parse tree. Figure 5 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.

3.3 Implementation

We have implemented 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 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

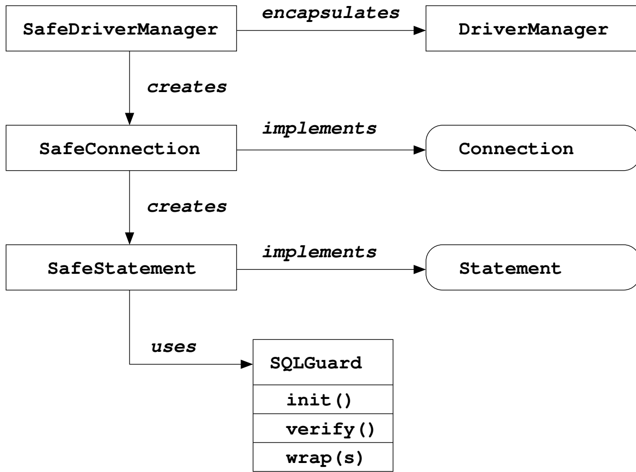

Figure 6: General class structure

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).

The `SQLGuard` class has a private method, `verify()`, which removes the key from the beginning of the query and uses it to identify wrapped user input and 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 sorted vector is used in the case the container application is multithreaded. We parse the query string using ZQL[7], a publicly available parser written in Java.

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 weaken the parse tree similarity check. Instead of verifying that that user leaf nodes are replaced with terminal literals, we would verify that they have 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.

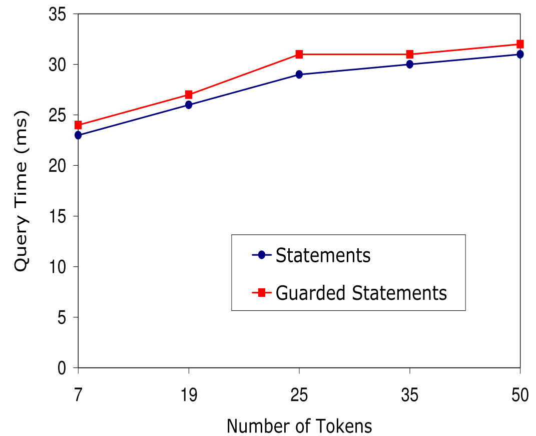

Figure 7: Average database query times for statements and guarded statements vs query size

4. CASE STUDY

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, the *Geological Hazard Inventory Management System*. This application 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 have evaluated our SQL injection elimination technique with respect to its computational overhead, and how easily it incorporates into existing code. In addition, the guarded application handled every injection attempt described in this paper, and all other techniques of which we are aware.

4.1 Execution Overhead

To test the execution time overhead, we baselined our study by timing the application with traditional SQL queries, i.e., without our injection checking. We then modified the queries as outlined in Section 3 and reran the experiment. The web server is a 733MHz Windows 2000 machine with 256MB RAM.

Figure 7 shows the average parse time as a function of the length of the query. During this experiment, the web server was otherwise idle. We recorded the time to parse the query, execute the database call, return the results, and close the connection. This amounts to the code in Figure 10, 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 lines show a small slope, which is due to the increas-

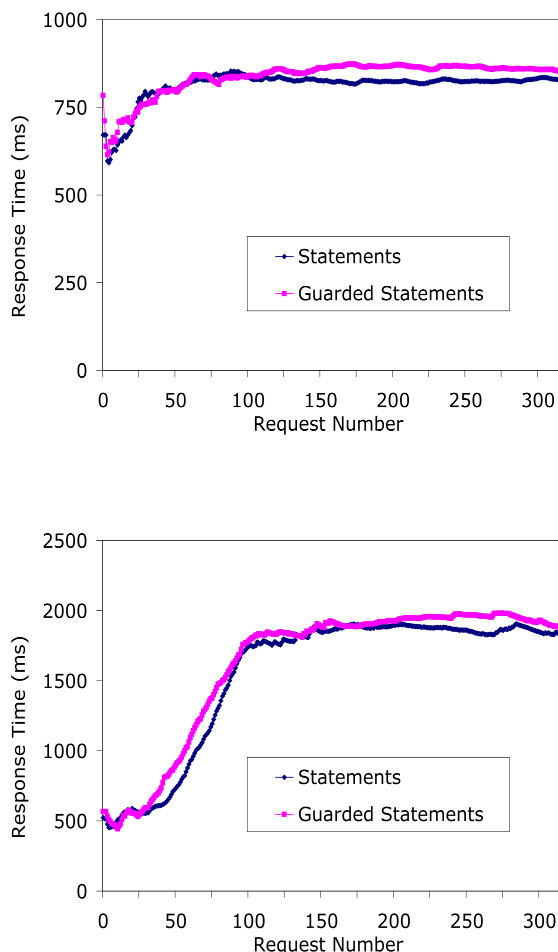

Figure 8: Average page response times for 10 (top) and 25 (bottom) simultaneous users

ing complexity of the query for the database engine.

Our second set of experiments aimed to test our implementation under extreme load. To accomplish this task, we used Apache's JMeter load testing package.[4] 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 was 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; inter-request interval times were taken from a Gaussian distribution with mean 1000 ms and standard deviation 500 ms. Figure 8 shows the response times for load tests with 10 concurrent users and 25 concurrent users. It can be seen

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

Figure 9: Load testing results comparison for traditional Statements and Guarded Statements

Num. Users	Query Method	Response Time (ms)
10	Statements	818
10	Guarded Statements	836
25	Statements	1836
25	Guarded Statements	1882
50	Statements	3552
50	Guarded Statements	3679

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 9 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

One requirement of our solution is for it to be simple to incorporate into both new and existing software. Figure 10 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 to be 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` 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.

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, 9, 8, 19]. The industrial community has also gone to lengths to make programmers aware and provide best practices to minimize the problem [16, 15, 3, 22, 17]. Offutt and Xu [20] made testing for SQL injection attacks a pointed case in their recent web services testing and verification efforts. 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 [21, 10]. 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 [5, 6]. 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

Traditional Method

```
Connection conn = 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 conn = 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();
```

Figure 10: Source code for unmodified SQL query (top) and protected SQL query (bottom)

dashes, and escape them manually. This is easily beaten, as attackers 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 [9, 8, 23].

Wasserman and Su [23] 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. Yet 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, the 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 number of cases for this analysis is 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 [10, 11]. This work is most similar to ours. As with the method above, any technique whose model is built at compile time cannot predict the exact structure of the intended query. Huang *et al.* [12] also employ both static and dynamic techniques, but they require that all sensitive functions have their preconditions precisely defined in annotations (trust policies).

Boyd and Keremytis [1, 13] 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 acknowledged by the authors, many applications export SQL errors [14], and as such, the new tokens would be exposed. External knowledge of the new tokens compromises the usefulness of the technique.

PHP[5] 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 written 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.

A couple commercial platforms have incorporated strong typing to assist programmers with SQL injection. J2EE has prepared statements, and Microsoft's .NET has commands. The effectiveness of prepared statements at eliminating SQL injection attacks is dependent on their implementation, which is contained in database drivers, and thus typically written by third party vendors. 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, Finally, many platforms do not have support for strong-typed technologies.

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 ease-of-use. Our approach blends the strengths of both: it has the convenience of dynamically generated strings and the security of prepared statements.

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

6. CONCLUSION

Most web applications employ a middleware technology designed to request information from a relational database in SQL. SQL 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. That is, effective 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 have made our implementation open to the public to maximize its aid for the larger community.[6] In the near future, we plan to implement our solution on the .NET framework and also in PHP, as well as incorporate automated injection error logging.

7. REFERENCES

[1] S. W. Boyd and A. D. Keromytis. SQLRand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302. Springer-Verlag, June 2004.

[2] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web*, 3(4):205–214, 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*, pages 1–18. Springer-Verlag, August 2003 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*, pages 91–104, August 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*, pages 63–78, January 1998.

[7] P.-Y. Gibello. Zql: A java sql parser. In *http://www.experlog.com/gibello/zql/*, 2002.

[8] 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)*, pages 697–698. IEEE Press, May 2004.

[9] 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)*, pages 645–654. IEEE Press, May 2004.

[10] 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)*, pages 22–28, May 2005. http://www.csd.uwo.ca/woda2005/proceedings.html.

[11] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 11th International World Wide Web Conference (WWW 03)*, pages 148–159, May 2003.

[12] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, and D. Lee. Securing web application code by static and runtime protection. In *Proceedings of the 12th International World Wide Web Conference (WWW 04)*, pages 40–52. ACM Press, May 2004.

[13] 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)*, pages 272–280. ACM Press, October 2003.

[14] D. Litchfield. Web application disassembly with ODBC error messages. In *http://www.nextgenss.com/papers/webappdis.doc*.

[15] P. Litwin. Stop SQL injection attacks before they stop you. In *http://msdn.microsoft.com/msdnmag /issues/04/09/ SQLInjection/default.aspx*, 2004.

[16] O. Maor and A. Shulman. SQL injection signatures evasion. In *http://www.imperva.com/application _defense_center/white_papers/sql_injection _signature_evasion.html*, 2004.

[17] S. McDonald. SQL injection: Modes of attack, defense, and why it matters. In *http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenseandWhyItMatters.php*, 2005.

[18] 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.

[19] A. Nguyen-Tuong, S. Guarnieri, D. Green, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of IFIP Security 2005*. Springer, May 2005.

[20] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. In *Proceedings of the 2004 Workshop on Testing, Analysis and Verification of Web Services (TAV-WEB)*, pages 1–10. ACM Press, July 2004.

[21] W. Security. Challenges of automated web application scanning. In *http://greatguards.com/docs/insightweb.htm*, 2003.

[22] K. Spett. SQL injection: Are your web applications vulnerable? In *SPI Labs White Paper*, 2004.

[23] 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)*, pages 70–78, October 2004.

[6]http://www.cse.ohio-state.edu/ paolo/software/java_sqlguard.htm