

Preventing SQL Injection Attacks in Stored Procedures

Ke Wei, M. Muthuprasanna, Suraj Kothari
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA - 50011
Email: {weike,muthu,kothari}@iastate.edu

Abstract

An SQL injection attack targets interactive web applications that employ database services. These applications accept user inputs and use them to form SQL statements at runtime. During an SQL injection attack, an attacker might provide malicious SQL query segments as user input which could result in a different database request. By using SQL injection attacks, an attacker could thus obtain and/or modify confidential/sensitive information. An attacker could even use a SQL injection vulnerability as a rudimentary IP/Port scanner of the internal corporate network. Several papers in literature have proposed ways to prevent SQL injection attacks in the application layer by examining dynamic SQL query semantics at runtime. However, very little emphasis is laid on securing stored procedures in the database layer which could also suffer from SQL injection attacks. Some papers in literature even refer to stored procedures as a remedy against SQL injection attacks. As stored procedures reside on the database front, the methods proposed by them cannot be applied to secure stored procedures themselves.

In this paper, we propose a novel technique to defend against the attacks targeted at stored procedures. This technique combines static application code analysis with runtime validation to eliminate the occurrence of such attacks. In the static part, we design a stored procedure parser, and for any SQL statement which depends on user inputs, we use this parser to instrument the necessary statements in order to compare the original SQL statement structure to that including user inputs. The deployment of this technique can be automated and used on a need-only basis. We also provide a preliminary evaluation of the results of the technique proposed, as performed on several stored procedures in the SQL Server 2005 database.

1. Introduction

The widespread adoption of the Web as an instant means of information dissemination and various other transactions, including those having financial consequences, has essentially made it a key component of today's Internet infrastructure. These applications and their underlying databases often store confidential or even sensitive data. The potential downtime and damages that could easily amount to millions of dollars have also prohibited many mission critical applications from going online, which could greatly benefit the users. Hence, it is crucial to protect these applications from targeted attacks.

However, the current state of application security leaves much to be desired. The 2002 Computer Security Institute and FBI revealed that, on a yearly basis, over half of all databases experience at least one security breach and an average episode results in close to \$4 million in losses [19]. A recent penetration testing study performed by the Imperva Application Defence Center included more than 250 Web applications from e-commerce, online banking, enterprise collaboration and supply chain management sites and their vulnerability assessment concluded that at least 92% of Web applications are vulnerable to some form of malicious intrusions [18]. Recent U.S. industry regulations such as the Sarbanes-Oxley Act pertaining to information security, try to enforce strict security compliance by application vendors [5] and there is an urgent need to find means of satisfying these security requirements.

SQL-Injection Attack (SQLIA) constitutes an important class of attacks against web applications. By leveraging insufficient input validation, an attacker could obtain direct access to the database underlying an application. Any web application exposed on the Internet or even within a corporate intranet could therefore be vulnerable to SQLIAs. Although the vulnerabilities that lead to SQLIAs are well understood, they persist because of

lack of effective techniques for detecting and preventing them. Defensive programming could, to an extent, protect against certain threats of SQLIAs. But it is impractical to undergo this entire process for protecting legacy systems. Several solutions have been proposed in literature to prevent SQLIAs in the application layer, which combine static analysis of application level programs and runtime validation of dynamically generated SQL-queries with inclusion of user inputs. Although these solutions prevent SQLIAs at the application layer, very little emphasis is laid on securing objects residing in the database layer such as stored procedures which are also potentially vulnerable to SQL Injection Attacks.

Stored procedures are an important part of a modern-day relational database. They add an extra layer of abstraction into the design of a software system, which means that, as long as the interface on the stored procedure stays the same, the underlying table structure could change with no noticeable consequence to the application that is using the database. This extra layer, to some extent, hides some design secrets from the potentially malicious users, such as definitions of tables. By using stored procedures, one could make sure that all the data is always contained in the database and is never exposed. In these databases, the developer is allowed to build dynamic SQL queries ie. SQL statements are built at runtime according to the different user inputs. For example, in SQL Server, EXEC(varchar(n) @SQL) could execute arbitrary SQL statements. This feature offers flexibility to construct SQL statements according to different requirements, but faces a potential threat from SQL Injection Attacks.

In this paper, we propose a novel technique to detect SQLIAs in stored procedures. Our technique builds upon a combination of static and dynamic analyzes techniques. The key intuition behind our technique is that a SQLIA will alter the structure of the original SQL statement and by detecting the difference in the structures, a SQLIA can be identified. Our technique consists of two phases. In the offline phase, the technique uses a parser to pre-process and identify specific SQL statements in the EXEC() call for runtime analysis. In the runtime phase, the technique monitors all dynamically-generated SQL-queries associated with user input and captures the original structure of the SQL statement and checks for compliance after inclusion of the user inputs. When SQLIA is detected, the technique prevents the malicious SQL statements from accessing the database and provides details about the attack.

In addition to describing the structure of our technique, we also present a preliminary evaluation of the proposed technique. We built a prototype SQLIA defense tool for Mi-

crosoft SQL Server 2005, and use it to evaluate our technique on a sample database in it. The results show that our technique is effective and involves negligible runtime overhead, and functions completely transparent to the developer.

2 SQL-Injection in Stored Procedures

In this section, we present a stored procedure that is vulnerable to a SQLIA and explain how an attacker could exploit this vulnerability. We also present various techniques that can be employed to gain illegitimate access to the system as well as the network resources.

A stored procedure is an operation set that is stored. Typically, stored procedures are written in SQL. Since stored procedures are stored on the server side, they are available to all clients. Once the stored procedure is modified, all clients automatically get the new version.

```

1. CREATE PROCEDURE [EMP].[RetrieveProfile] @Name varchar(50),
   @Passwd varchar(50)
2. WITH EXECUTE AS CALLER
3. AS
4. BEGIN
5.     DECLARE @SQL varchar(200);
6.     ...
7.     SET @SQL='select PROFILE from EMPLOYEE where ';
8.     ...
9.     IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
10.    BEGIN
11.        ...
12.        SELECT @SQL=@SQL+'NAME="'+@Name+'" and ';
13.        SELECT @SQL=@SQL+'PASSWD="'+@Passwd+'";
14.        ...
15.    END
16. ELSE
17. BEGIN
18.    ...
19.        SELECT @SQL=@SQL+'NAME="Guest"';
20.    ...
21. END
22. ...
23. EXEC(@SQL)
24. ...
25. END

```

Code 1. Stored Procedure vulnerable to SQL-Injection

A sample stored procedure called with the username and password as user inputs in a variable length string format is shown in Code 1. Notice that, there is an EXEC system function which allow the user to dynamically build a SQL statement in string format and later execute it. This feature

is supported in most other business database products also. Dynamically built SQL statements provide great user flexibility but also face a great threat from SQLIAs. The process of building an SQL statement could be used by the attacker to change the original intended semantics of the SQL statement.

If the stored procedure in Code 1 is called with no values for @Name and @Passwd variables, the following query would get executed:

```
select PROFILE from EMPLOYEE where NAME='Guest'
```

When user inputs are provided for @Name and @Passwd, the following query would get executed:

```
select PROFILE from EMPLOYEE where NAME='name' and  
PASSWORD='passwd'
```

In this scenario, suppose a user gives input for variable @Name as "' OR 1=1 --" and any string, say "null", for the variable @Passwd the query would take the form:

```
select PROFILE from EMPLOYEE where NAME="' or 1=1 --' and  
PASS='null'
```

The characters "--" mark the beginning of a comment in SQL, and everything after that is ignored. The query as interpreted by the database is a tautology and hence will always be satisfied, and the database would return information about all users. Thus an attacker can bypass all authentication modules in place and gain unrestricted access to critical data on the web server.

An SQL-Injection Attack (SQLIA) is a subset of the unverified/unsanitized input vulnerability and occurs when an attacker attempts to change the logic, semantics or syntax of a legitimate SQL statement by inserting new SQL keywords or operators into the statement. This definition includes, but is not limited, to attacks based on tautologies, injected additional statements, exploiting untyped parameters, stored procedures, overly descriptive error messages, alternate encodings, length limits, second-order injections and injection of "UNION SELECT", "ORDER BY" and "HAVING" clauses. A detailed explanation of the different types and forms of SQLIAs and the ways in which they can be exploited are available in the public domain [1] [2] [12] [7].

The widely deployed defense today is to train the programmers and web-developers about the security implications of their code and to teach them corrective measures and good programming practices [17]. However, rewriting or revising the entire lot of existing legacy code is not an easy process and is not a financially viable option for many orga-

nizations. Even this does not guarantee any foolproof defense and hence we need automated processes to detect the vulnerability and eliminate them. Various other techniques like escaping the quotes and limiting the length of user inputs are employed as a quick fix solution. Unfortunately, even these security measures are only inadequate against highly sophisticated attacks [25]. It is of even greater concern that well known database vendor products like Microsoft SQL Server etc. provide attackers direct access to the command line shell and registry using methods like xp_cmdshell, xp_regread etc. Some of the very recent incidents only highlight the magnitude of this problem and hence the urgent need to address it in an appropriate manner [24] [29] [3] [23].

3 Related Work

Various SQLIA detection techniques for the application layer have been proposed in literature, but none of them pay enough attention to SQLIA in stored procedures. Although the mechanism of SQLIA is the same for both stored procedure and application layer program, the same detection technique could not be applied to stored procedures, because of stored procedure's limited programmability and the technique's usability and deployability. Many existing techniques, such as filtering, information-flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQLIAs. Techniques that employ input validation are prone to a large number of false positives and yet there is no guarantee that there are no false negatives. A simple example is to check for single quotes and dashes, and escape them manually. This could be easily beaten by using ASCII representation of these characters such as CHAR(0x27) for single quotes. Safe Query Objects [9] and SQLDOM [22] use encapsulation of database queries to provide a safe and reliable way to access databases but they require developers to learn and use a new programming paradigm. SQLrand [6] provided a radical shift in the way this problem can be approached using query randomization [20]. However, it could be circumvented if the key used for randomization were to be exposed. The use of a machine learning technique trained using a set of typical application queries to detect malicious query models at runtime was proposed by F. Valeur, D. Mutz and G. Vigna [28]. However, like most other learning algorithms, it can generate a large number of false positives in the absence of an optimal query set for training.

Another popular mechanism for application layer program has been static analysis of the code for vulnerabilities [21]. The Java String Analysis library [8] provides a mechanism for generating models for Java strings and can be extended to generate fairly accurate SQL-query models. JDBC-

Checker [13] [14] statically checks for the type correctness of dynamically generated SQL queries. Although these techniques are effective, they cannot capture more general forms of SQLIAs that generate syntactically and type correct queries. Wassermann and Su combine static analysis with automated reasoning in [30] to detect tautologies in the dynamically generated SQL queries, but the other forms of SQLIAs would still succeed rendering the system vulnerable.

Recently, researchers have been exploring the use of static analysis in conjunction with runtime validation [11] to detect instances of SQLIAs. In [10], Buehrer and Weide have proposed the use of parse trees to detect malicious user input, which requires a developer to manually modify new and existing code. In [16] [15], Halfond and Orso have used an automaton construction technique to defend against SQLIAs. As mentioned above, it is hard to predict the exact structure of the intended SQL statement. Also there is an additional runtime analysis overhead in terms of execution time which cannot be avoided due to the sequential nature of the analysis techniques. Additionally, people have also looked at stored procedures as an effective defense against SQLIAs [26]. The use of stored procedures alone does not protect one against SQLIAs as is commonly assumed by most developers, but appropriate use of parameters along with stored procedures is necessary to achieve a minimal defense against such attacks [27] [4].

4 Our Proposed Solution

We propose an SQL-Injection Attack prevention technique here that addresses all types of SQLIAs, as discussed in previous sections. The technique works by combining static analysis with runtime validation. The basis of such a technique is that the control flow graph of the stored procedures can be represented as an SQL-graph which indicates what user inputs the dynamically built SQL statements depend on. By using an SQL-graph, we reduce the set of SQL statements we need to verify, by looking at only a small subset of all the SQL statements in the stored procedure at runtime. During runtime, we retrieve a Finite State Automaton(FSA) from the EXEC(@SQL) procedure call and check the SQL statement with inclusion of user inputs for compliance, flagging them safe or unsafe.

4.1 Static Analysis

To perform static analysis of the stored procedure, we propose a stored procedure parser which extracts the control flow graph from the stored procedure. We label all the EXEC(@SQL) statements in the control flow graph and then backtrack to identify all the statements involved in

the construction of the @SQL statement in the control flow graph. In this process, an SQL-graph as explained below is generated. From the SQL-graph, SQL statements which depend on user inputs are selected and flagged to monitor their structure at runtime. At runtime, we compare the structure of the original intended SQL statement with the dynamically generated SQL statement having user inputs by using a Finite State Automaton. An SQLIA which alters the original structure will be flagged as unsafe and related information would be logged.

4.1.1 SQL-graph Representation

It is possible for a stored procedure to have more than one EXEC(@SQL) statement. Not all the EXEC(@SQL) statements would depend on the user inputs. Only those which need the user inputs to complete the SQL statements are potentially vulnerable to SQLIA. Given that the user input would realistically consist of a few strings only but the number of SQL statements that get executed in a stored procedure could be very large, we now try to optimize the number of queries that need to be further processed at runtime in order to ensure the validity and legitimacy of the dynamically generated statements, using an SQL-graph.

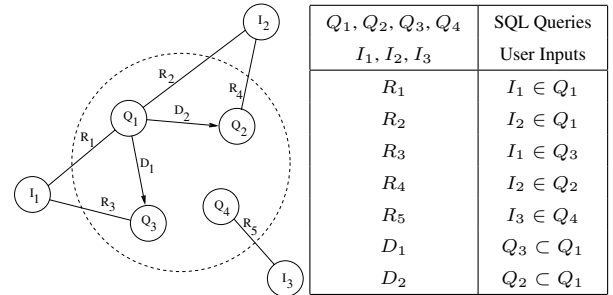


Figure 1. SQL-graph Representation

The SQL-graph in Fig. 1 represents 4 different SQL queries (EXEC statement hotspots) in the stored procedure as nodes within a logical boundary, and 3 different user inputs as being outside the logical boundary. If a particular user input (I) is used in a SQL query (Q), the relationship (R) between the two nodes is indicated by an undirected link between the 2 nodes. We now define dependencies (D) in the SQL-graph as links that point from one SQL query to another SQL query such that the user inputs used by the former is a proper superset of the user inputs used by the latter. For SQL queries that use the same set of user inputs, one of them is chosen as a representative query and is made to point to the others. We see the dependencies represented as directed arrows in the SQL-graph. Drawing equivalence to Code 1, Q_1 represents the SQL statement, while I_1 and

I_2 represent the user inputs *username* and *password*. Q_2 , Q_3 , Q_4 and I_3 could possibly correspond to some other *EXEC statements* in the stored procedure not represented in the code snippet.

The concept of an SQL-graph is used to reduce the runtime scanning overhead by restricting the number of queries that need to be scanned along any execution path that is taken in the stored procedure. SQL queries that do not use user inputs are not included in the SQL-graph. Only the SQL queries that are exposed to the user inputs in some form or the other (string manipulations included) are included in the SQL-graph representation. The choice of such a representation and the resulting benefits in terms of runtime overhead would be explained as part of Runtime Validation.

By using the stored procedure parser, we extract the control flow graph of the stored procedure and label every statement which contains an *EXEC()* call. Then we backtrack in the control flow graph using a breadth first search algorithm from the labeled statement, trying to find all the statements involved in the construction of the queries in the *EXEC()* call. During the process of searching, we keep track of how the query is incrementally built in the stored procedure. SQL queries are differentiated by labeling them according to their sequence in the control flow graph, and user inputs by their position in the arguments list of the stored procedure. After the search reaches the beginning of the stored procedure, an SQL-graph similar to Fig. 1 would be built.

```

1. CREATE PROCEDURE [EMP].[RetrieveProfile] @Name varchar(50),
@Passwd varchar(50)
... 9.     IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
11.     ...
12.     SELECT @SQL=@SQL+'NAME="'+MARK(@Name)+'"
and ';
13.     SELECT @SQL=@SQL+'PASSWD="'+MARK(@Passwd)+'";
14.     ...
16. ELSE
18.     ...
19.     SELECT @SQL=@SQL+'NAME="Guest"';
20.     ...
23. EXEC(SQLIA_CHECKER(@SQL))
24.     ...

```

Code 2. Stored Procedure not vulnerable to SQL Injection

4.1.2 Instrumentation

In order to monitor the structure of the SQL statement at runtime, we need to instrument the original stored procedure. This operation is also implemented by our parser. In

this process, we separate the user inputs from the original SQL statement by pre- and post-pending a mark, which in our implementation is the session id. The mark should be chosen wisely, or it might be guessed by the attacker. In order to avoid collisions with user inputs, each mark (random string of bytes) can be checked for occurrence in any of the user inputs and modified accordingly in the face of a collision or otherwise retained as is. We do not impose any restrictions on the choice of the mark and leave this open as a design issue. Each SQL string is enclosed by the *SQLIA_CHECKER()* function which does the runtime validation and legitimacy check. We implement this function using C#. We use another function called *MARK()* which returns the current session id and pre- and post-pends the argument passed to it with that mark. Instrumentation is done automatically in accordance with the SQL-graph discussed in the previous section. Code 2 shows that instrumented version of Code 1, as illustrated previously.

4.2 Runtime Analysis

During runtime, before the *EXEC()* function is called, the *SQLIA_CHECKER()* function will identify the user input by the current session id and build a finite state automaton as shown in Figure 2. Then, the SQL statement with user inputs filled in (the marks are ignored in this case), is compared against the corresponding finite state automaton to check for validity. If the user inputs cause the dynamically generated SQL queries to not conform to the semantics of the intended SQL queries as in the finite state automata, then they are flagged as SQLIAs, else they are passed through. Figure 2 shows the case where an SQLIA is not caused and the query is passed through ie. it is similar to the automaton construct. Also, it shows the second example where an SQLIA has been caused and hence gets rejected as a potentially malicious query. The literals along both the original structure and the user inputs included structure, as one traverses from the *Start* node to the *End* node, should be identical. The other check that can be enforced is that the length of the finite state automaton chain for a particular instance is exactly the same for the original one and the altered one. Thus SQLIAs employing tautologies and injecting additional statements can be captured by this technique. The case where alternate encoding like *URL Encoding*, *UTF-8* etc. are used by the attackers can also be addressed by requiring the runtime validation to occur only after all the user inputs have been converted to a single encoding as interpreted by the SQL Engine in the database server.

The SQL-graphs for the different programs or applications resident on the application server are computed offline, using static analysis as described above. These SQL-graphs

need to be constantly updated to reflect any changes in the code made by the programmer at any point of time. To facilitate easy modification of the code by the developer in a transparent manner, we have a turn-off option for our parser. The developers simply need to use the turn-off option to remove all the instrumentation and the corresponding SQL-graphs, alter the code as desired and then rebuild these elements again. Every time a client request comes in, the runtime finite state automata of the different SQL queries in the SQL-graph are validated. A Verification Table (VT) is then computed for the different SQL queries indicating whether it can be allowed to pass through or whether it should be dropped before being sent to the database. Now verifying the finite state automata for all the queries in the SQL-graph can be computationally intensive and can be expensive in terms of the runtime processing time for the stored procedure. The concept of the directed dependency in the SQL-graph is used to reduce the total runtime overhead.

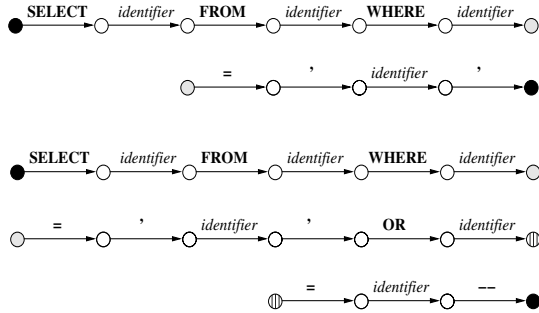


Figure 2. SQLIA Detection: SQL-FSM Violation

If a user input does not cause any SQLIA in one query, it means that it conforms well to the SQL query semantics as defined by the SQL language. Then it is implicitly known that the same input in any other query would also not cause an SQLIA. Hence, we see that if this knowledge is not exploited, we would be redundantly verifying the same user input over and over again in multiple SQL queries in the SQL-graph. The directed dependency in the SQL-graph tells us which SQL queries are supersets of which other SQL queries in the SQL-graph. It would suffice to check only those SQL queries that are supersets of other queries and thus implicitly check the other queries encompassed by it. Thus we filter out all those SQL queries that have no directed dependency edges coming into them and verify only the validity of the finite state automata corresponding to those SQL queries. We thus see that we need to vali-

date only a small fraction of the entire SQL-graph and still achieve SQLIA prevention guarantee. In Fig. 1, it would suffice to check finite state automata corresponding to SQL queries Q_1 and Q_4 . Thus we can achieve optimal SQLIA prevention using runtime validation of the abstraction of the stored procedure source code generated by static analysis.

5 Prototype Evaluation

In order to evaluate our approach, we developed a prototype of our proposed technique and tested it on a sample database in SQL Server 2005. We ran a few trial runs on the database before the new prototype was deployed and repeated the same tests on the new setup. The results of these tests are presented below. The performance metrics we used for comparison were the number of false positives generated, number of false negatives generated and the extra processing overhead during runtime as tested on a few representative stored procedures in the database, having varying number of SQL tokens.

5.1 SQLIA Detection Accuracy

We subjected both the protected and the unprotected database instances to different types of SQLIAs namely use of tautologies, inserting additional SQL statements, second-order SQL injection [1] and various other SQLIAs known in literature. We also tested the impact when a user enters a legitimate (') symbol as part of the user input. The proposed technique detected all types of SQLIAs under all circumstances and in all cases. The details as to how the false positives due to valid user input of (') can be tackled would be provided in the extended version and not here, due to space constraints. The proposed technique is thus a secure and robust solution to defend against SQLIAs.

SQLIA Type	Unprotected Server	Protected Server
Use of Tautologies	Not Detected	Detected
Additional SQL stmts	Not Detected	Detected
Valid User Input of (')	Query allowed	False Positive
Second-Order Injection	Not Detected	Detected
Other SQLIAs	Not Detected	Detected

Table 1. SQLIA Detection Accuracy

5.2 Program Execution Time

The proposed technique introduces two types of overhead. The first overhead is due to the static analysis of the application source code to construct SQL-graph and instrument

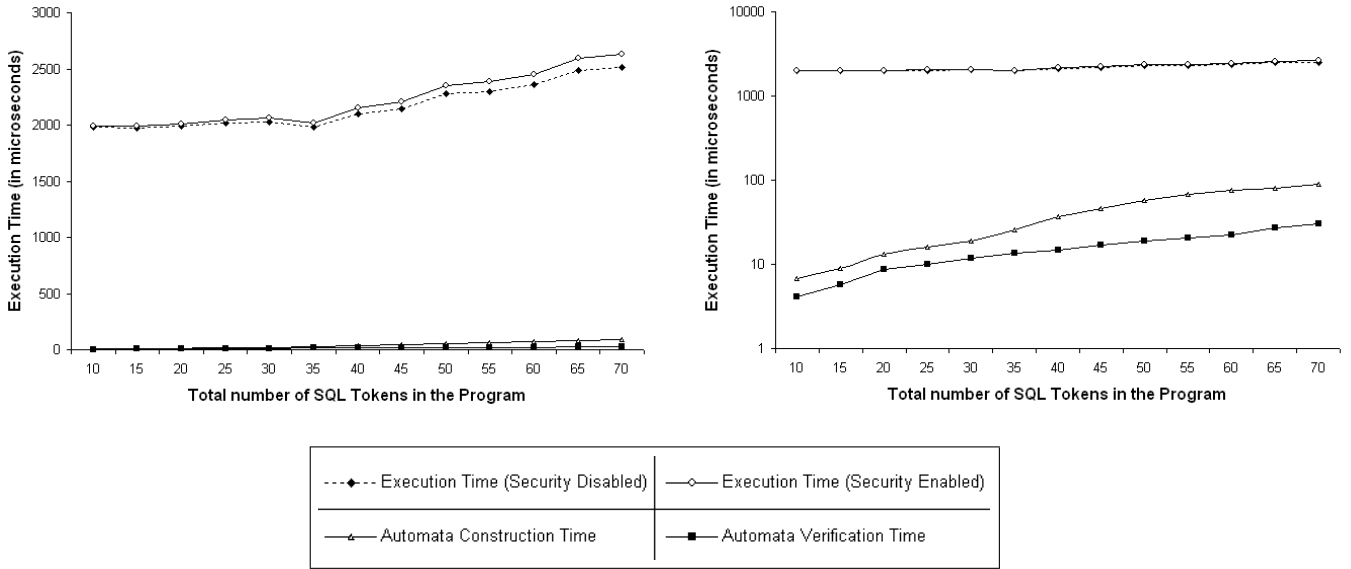


Figure 3. Runtime Analysis II

the code while the second is due to runtime finite state automata construction and validation. As the static analysis is an offline process, the users do not experience the delay induced due to this one-time operation (until next code modification is done).

The runtime validation incurs some overhead in terms of execution time for constructing the finite state automata as well as validating the SQL statement with user inputs. We studied the overhead of running various benchmarking tests on the same database instance under varying program conditions. These experiments were run on a single stored procedure and can easily be extended for multiple stored procedures in the database.

In the first experiment, we varied the number of EXEC() statements in the stored procedure, each having 20 SQL tokens on an average. Fig. 4 shows the runtime execution time for both the instances where security has been enabled and also where it has been disabled. We see that the overhead imposed due to the finite state automata construction and verification is negligible as compared to the execution time for the program.

In the second experiment, we varied the number of SQL tokens in a single EXEC() statement in the stored procedure. Fig. 3 shows the runtime execution time for both the instances where security has been enabled and also where it has been disabled. The graphs have been plotted both on a logarithmic as well as on a linear scale for visual clarity. The graphs also indicate the time required for the finite state

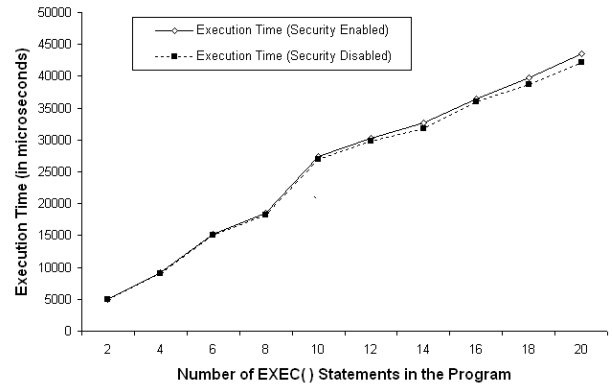


Figure 4. Runtime Analysis I

automata construction as well as for runtime comparison and verification of the dynamically generated SQL query with the finite state automata. The time required for the extra processing overhead is negligible as compared to the normal execution time for the program. Appropriate care was taken to ensure that the SQL queries were not available in the cache at any point of time to obtain the worst case execution times for the program.

6 Conclusions and Future Work

Most web applications employ a middleware technology (scripting engine) designed to request information from a relational database in SQL parlance. SQL injection is a common techniques hackers employ to attack underlying

databases. These attacks reshape the SQL queries, thus altering the behavior of the program for the benefit of the hacker. Several solutions exist to prevent SQLIAs at the application layer, but no concrete solution other than using parameters while coding exist to protect stored procedures in the database layer against SQLIAs. In this paper, we present a fully automated technique for detecting, preventing and reporting SQLIA incidents in stored procedures. The technique abstracts the intended SQL query behavior in an application in the form of an SQL-graph, as a one-time offline procedure using static analysis of the stored procedure source code. This graph is then validated against all the different user inputs at runtime to capture all malicious SQL queries, before they are sent for execution. This graph model helps in capturing all the different types and modes of execution of SQLIAs. We also have provided preliminary evaluation results of the prototype we developed against the various performance metrics affecting database accesses.

As part of future work, we plan to extend our prototype to develop a complete implementation of the proposed architecture. This would then be used as a testbed to evaluate the different web application scripts available in the public domain. We are currently exploring the security implications of incorporating well known randomization algorithms into our model in case the session id which is used to separate the user inputs from the SQL statement might be guessed by the attacker. We are also exploring the possibility of implementing this functionality as a middleware to the database engine, to avoid explicit instrumentation of source code.

References

- [1] C. Anley. Advanced sql injection in sql server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf.
- [2] C. Anley. (more) advanced sql injection. http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf, White Paper.
- [3] B. M. L. Archive. <http://seclists.org/lists/bugtraq/2005/>, 2005.
- [4] P. S. I. Attacks. <http://www.wwwcoder.com/main/parentid/258/site/2966/68/default.aspx>.
- [5] K. Beaver. Achieving sarbanes-oxley compliance for web applications through security testing. http://www.spidynamics.com/support/whitepapers/WI_SOXwhitepaper.pdf, 2003.
- [6] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. *ACNS*, 2004.
- [7] C. Cerrudo. Manipulating microsoft sql server using sql injection. http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf, White Paper.
- [8] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Precise analysis of string expressions. *SAS*, 2003.
- [9] W. R. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. *ICSE*, 2005.
- [10] G. T. B. et. al. Using parse tree validation to prevent sql injection attacks. *SEM*, 2005.
- [11] Y. H. et. al. Securing web application code by static analysis & runtime protection. *WWW*, 2004.
- [12] S. Friedl. Sql injection attacks by example. <http://www.unixwiz.net/techtips/sql-injection.html>.
- [13] C. Gould, Z. Su, and P. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. *ICSE*, 2004.
- [14] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ICSE*, 2004.
- [15] W. G. J. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing sql injection attacks. *ASE*, 2005.
- [16] W. G. J. Halfond and A. Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. *WODA*, 2005.
- [17] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2nd Edition, 2003.
- [18] W. Inc. Only 10% of web applications are secured against common hacking techniques. <http://www.imperva.com/company/news/2004-feb-02.html>, 2004.
- [19] C. S. Institute. Computer crime and security survey. <http://www.gocsi.com/press/20020407.jhtml>, 2002.
- [20] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. *CCS*, 2003.
- [21] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. *USENIX Security Symposium*, 2005.
- [22] R. McClure and I. Kruger. Sql dom: Compile time checking of dynamic sql statements. *ICSE*, 2005.
- [23] O. W. A. S. P. (OWASPD). Top ten most critical web application vulnerabilities. <http://www.owasp.org/documentation/top10.html>, 2005.
- [24] T. R. site exposes Data. <http://www.net-security.org/news.php?id=1593>, 2002.
- [25] K. Spett. Blind sql injection. http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf, White Paper.
- [26] Statements and P. Statements. <http://g.bookpool.com/gp/0507apress/159059407X-2367.pdf>.
- [27] H. to: Protect from SQL Injection in ASP.NET. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000002.asp>.
- [28] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. *LNCS*, 3548:123–140, 2005.
- [29] C. V. N. VU#982109. <http://www.kb.cert.org/vuls/id/982109>, 2005.
- [30] G. Wassermann and Z. Su. An analysis framework for security in web applications. *SAVCBS*, 2004.