ISSN: 2250–3676

[IJESAT] INTERNATIONAL JOURNAL OF ENGINEERING SCIENCE & ADVANCED TECHNOLOGY Volume-2, Issue-3, 657 – 663

SQL-INJECTION SECURITY EVOLUTION ANALYSIS IN ASP.NET

Peeyush Mathur[1], Kajal Mathur[2], Pulkit Mathur[3], Puneet Mathur[4]

1Associate Professor, Dept. of Computer Science, SEC, Sikar, Rajasthan, India, ***peeyush14_mathur@yahoo.com***
2Research Scholar, Dept. of Computer Science, SEC, Sikar, Rajasthan, India, ***kajal_mathur2002@yahoo.com***
3Research Scholar, Dept. of Elecronics and Comm. , SBCET, Jaipur, Rajasthan, India, ***pulkitpmathur@yahoo.co.in***
4Research Scholar, Dept. of Elecronics and Comm. ,SBCET, Jaipur, Rajasthan, India, ***puneetpmathur@yahoo.co.in***

Abstract

All the interactive web applications that provide work for databases are target of an SQL injection attack. Such applications gives the permission to the user for input, after that this input added in database request, that's SQL Statement. In SQL injection, the attacker provides user input that outcome in a different database request than was intended by the application programmer. SQL injection is a code injection technique that exploits security vulnerability in a website's software. The vulnerability happens when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and unexpectedly executed. SQL commands are thus injected from the web form into the database of an application (like queries) to change the database content or dump the database information like credit card or passwords to the attacker. SQL injection is mostly known as an attack vector for websites but can be used to attack any type of SQL database. In our project work we describe a technique to prevent this kind of manipulation and hence eliminate SQL injection vulnerabilities. For empirical analysis, we provide a case study of our solution in ASP page. We implement our solution in a simple .NET framework, and show its effectiveness and scalability.

Keywords: *SQL injection, parse tree, statement, commands*

---------- *** ----------

1. INTRODUCTION

The World Wide Web has experienced remarkable growth in recent years. Businesses, individuals and governments have found that web applications can offer efficient and reliable solutions to challenges of communicating and conducing commerce in the 21th century. Various corporate bodies whose business model completely focuses on the Web like Google, Yahoo, Amazon etc. have taken web interactions to newer heights. As many enterprise applications dealing with sensitive financial and medical data turn online, the security of such web applications has come under close scrutiny. Compromise of these applications represents a serious threat to organizations that have deployed them, and also to users that trust these systems to store confidential data. The potential downtime and damages that could easily amount to millions of dollars have also prohibited many mission critical applications, which could greatly benefit users, from going online. Hence, it is crucial to protect these applications from targeted attacks.

SQL Injection Attacks (SQLIAs) constitute an important class of attacks against web applications. SQLIAs can give attackers direct access to the database underlying an application and allow them to leak/alter confidential information [1] or to even execute any malicious code [2]. Every week hundreds of vulnerabilities are being reported in these web applications, and are being actively exploited. The number of attempted attacks every day for some of the large web hosting forms range from hundreds of thousands to even millions. The most common web application attacks are:

- SQL Injection
- Code Injection
- Remote code-Indclusion
- Cross-Site Scripting(CSS)

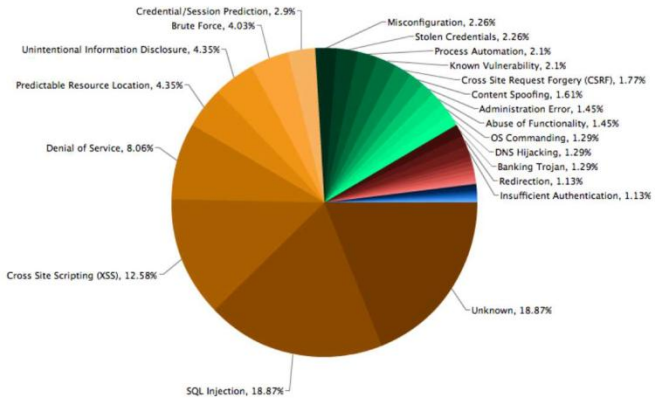

Fig. 1 Web Hacking Incident Database Report for 2011

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.
SQL Injection Attack (SQLIA) is considered one of the top 10 web application vulnerabilities of 2007 and 2010 by the Open Web Application Security Project. The attacking vector contains five main sub-classes depending on the technical aspects of the attack's deployment:

- Classic SQLIA
- Inference SQL Injection
- Interacting with SQL Injection
- DBMS specific SQLIA
- Compounded SQLIA

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 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;
• minimize the runtime overhead.

1.1 SQL Injection

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in the SQL query in such a way that part of the user's input is treated as SQL code [4]. It is a trick to inject SQL query or command as an input possibly via the web pages. They occur when data provided by user is not properly validated and is included directly in a SQL query. By leveraging these vulnerabilities, an attacker can submit SQL commands directly access to the database. There are two major SQL Injection techniques: i) *access through login page* and ii) *access through URL*. The first technique is the easiest in which it bypasses the login forms where users are authenticated by using passwords. This kind of technique can be performed by the attackers through: *'or' condition*, *'having' clause*, *multiple queries* and *extended stored procedure*. The second technique can be performed by the attackers through: *manipulating the query string in URL* and *using the 'SELECT' and UNION statements.* This kind of vulnerability represents a serious

SELECT * FROM users WHERE username = ' " & userName & " ' AND password = ' " & userPass & " ' "
If the username and password as provided by the user are used, the query to be submitted to the database takes the form;

SELECT * FROM users WHERE username = 'greg' AND password = 'secret'

If the user were to enter [' OR 1=1 --] and [ ] instead of [greg] and [secret], the query would take the form;

SELECT * FROM users WHERE username = ' ' OR 1=1 –' AND password = ' '

The query now checks for the conditional equation of [1=1] or an empty password, then a valid row has been found in the *users* table. The first ['] quote is used to terminate the string and the characters [--] mark the beginning of a SQL comment, and anything beyond is ignored. The query as interpreted by the database now has a tautology and is always satisfied. Thus an attacker can bypass all authentication modules gaining unrestricted access to critical information on the server. SQL injection potentially affects every database on all platform and web application. This attack can be used to gain confidential information, to bypass authentication mechanisms, to modify the database, and to execute arbitrary code. In certain circumstances the attacks happened on the database server itself. Even though DBMSs provide access

control mechanisms, these mechanisms are not adequate to deal with SQL injection attacks. For that reason, various techniques such as the use of stored procedures, prohibiting display of database server error messages and use of escape sequences for sanitizing user inputs are employed as a quick fix solution. Unfortunately, even these security measures are also inadequate against highly sophisticated SQL injection attacks.

1.2 Problem Domain

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. Thus a technique is required to prevent this type of attack more effectively without more using more over head time and cost.

1.3 Solution Domain

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. To authenticate our work we calculate overhead to database query costs. In addition, for empirical analysis, we provide a case study of our solution. We implement our solution, and show its effectiveness and scalability.

2. SQL INJECTION ATTACK TYPES

In this section, we present and discuss the different kinds of SQLIAs known to date. The different types of attacks are generally not performed in iso- lation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are count- less variations of each attack type.

Types of Attack Working Method

Tautologies SQL injection codes are injected into one or more conditional statements so that they are always evaluated to be true.

Logically Incorrect Queries

Using error messages rejected by the database to find useful data facilitating injection of the backend database.

Union Query Injected query is joined with a safe query using the keyword UNION in order to get information related to other tables from the application.

Stored Procedure Many databases have built-in stored procedures. The attacker executes these built-in functions using malicious SQL Injection codes.

Piggy-Backed Queries Additional malicious queries are inserted into an original injected query.

Inference

- Blind Injection

- Timing Attacks

An attacker derives logical conclusions from the answer to a true/false question concerning the database.

- Information is collected by inferring from the replies of the page after questioning the server true/false questions.

- An attacker collects information by observing the response time (behavior) of the database.

Alternate Encodings

It aims to avoid being identified by secure defensive coding and automated prevention mechanisms. It is usually combined with other attack techniques.[4]

3. VALIDATIONS THROUGH PARSE TREE

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.

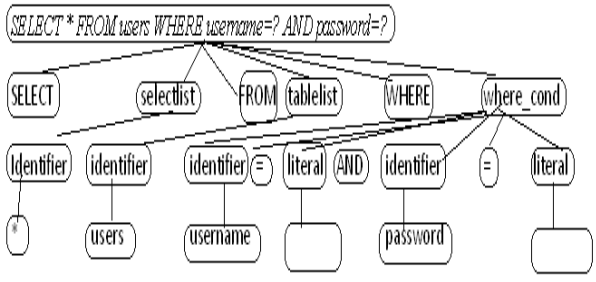

Fig 2: SELECT query with two user inputs

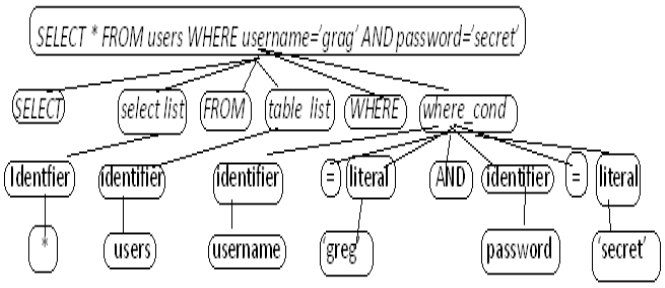

Fig 3 SELECT query with two user inputs inserted

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. 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.[5] 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 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.

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.

4. RELATED WORK

In this section, we list the work closely related to ours and discuss their pros and cons. The techniques related to SQL injection are classified and were evaluated by Halfond *et al.* [6], and we also made our classifications based on it. Interested readers can refer to [7] for the formal definition of SQL injection, and to [2, 7, 9, 10, 11] for the attacking and preventing techniques.

Framework Support Recent frameworks for web applications provide a functionality that can be used to prevent SQL injections. For example, Struts [12] supports a *validator*. A validator verifies an input from the user conforms to the pre-defined format of each parameter. If a validator prohibits an input from including meta-characters, we can avoid SQL injections. Since a validator does not transform the dangerous characters to safe ones, we can not prevent SQL injections if we want to include meta-characters in the input.

Prepare Statement SQL provides the `prepare` statement, which separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The `prepare` statement makes it harder to inject SQL queries because the SQL structure cannot be changed. Hibernate [14] enforces us to use the `prepare` statement. To use the `prepare` statement, we must modify the web application entirely; all the legacy web applications must be rewritten to reduce the possibility of SQL injections. Sa- nia is useful to check if a particular application needs to be re-written to prevent SQL injections.

Static Analysis Wassermann and Su [15] proposed an approach that uses a static analysis combined with automated reasoning. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that in- sert a tautology in the SQL queries, but cannot detect other types of SQL injections attacks.

JDBC Checker [18] statically checks the type correctness of dynamically generated SQL queries. JDBC Checker can detect SQL injection vulnerabilities caused by improper type checking of the user inputs. However, this technique would not catch more general forms of SQL injection at- tacks, because most of these attacks consist of syntactically correct and type-correct queries.

Dynamic Analysis Paros [19] is a free tool for testing for web application vulnerabilities without rewriting any scripts in the web application. This tool automatically scans for SQL injection vulnerabilities with pre-defined attack codes. Paros checks the contents of HTTP response messages to determine whether an SQL injection attack was successful or not.

5. SYSTEM ARCHITECTURE

In this section we provide the architecture of the system. Below given diagram shows the architecture of the system. Complete system is designed in modules and all module are work to gather to form a complete system. Each module is contains a complete functionality and form a complete a unit.

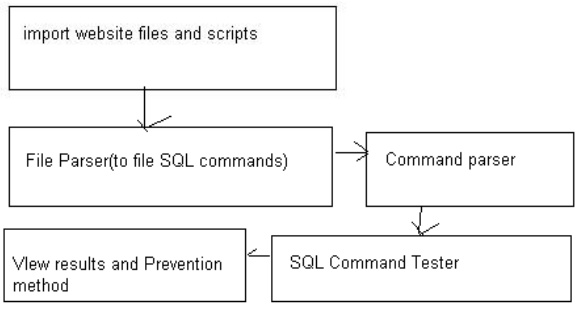

Fig 4 architecture diagram

Complete system is a joint effort of five sub systems the description and working is given below.
Import web site files and scripts: it is a way to input files on the system. There are two different modes to accept input to the system first using local system in this way user can input the file and scripts by selecting a local machine folder. Second by FTP (remote) files to access the remote files IP address, user name and password required to download the file over local machine to test.
File parser: after accept input data to the system required to classify the files where SQL commands are reside. Using this phase we extract file, line no and command where SQL commands are found.
Command parser: the found command tested over the parameters that are given on the literature survey.
SQL command Tester: here we classify the commands according to the injection type and list as results.
View results and Prevention: here a class script is written to prevent the sql injection over the system.

5.1 Implementation

Implementation required for this software and hardware on the development side system.
Hardware Interfaces Recommended
- 2.0 GHz Processor required (Pentium D and above)
- Minimum 2 GB RAM
- 25 GB hard disk space

***Software* Server Side**
- Operating System(Windows 2003 and above)
- Microsoft Visual Studio 2008 Frameworks 3.5
- MS SQL Server 2000

Project will be done in C#.NET as front end and SQL Server 2005 as back end. Microsoft .NET is software that connects information, people, systems and devices.
The .NET Framework is the programming model of the .NET environment for building, deploying and running Web- based applications, smart client applications and XML Web services. It manages much of the plumbing, enabling developers to focus on writing the business logic code for their applications. The .NET Framework includes the common language runtime and class libraries.

6. TESTING

To test our application we perform the unit testing in this method we test the functions of the prepared system. Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. This is usually at the class level, and the minimal unit tests include the constructors and destructors. These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected.

For that reason we are apply UNIT TEST for our project. And we conclude the test results by screen by screen (by classes).

S.No.	Case	Expected output	Actual output	Remark
1.	mainMenu Screen	It is a MDI form contains menu to all menu items are working properly	Same as expected	Pass

2.	Import file screen	This screen perform the download and listing of files, directories and sub directories	Same as expected	Pass
3.	analysis screen	That is a analysis representation screen by which we estimate which method of SLQ injection hacker break the security and how we prevent them	Same as expected	Pass

Table-1: Test cases:

RESULTS:

This application gives us a support to detect and prevent SQL Injection attack at run time using parse method. There are many methods available for detection and prevention SQLIA. But this method provide us all the processing at run time by which we can recover our interactive web page(designed in ASP.NET) from the SQLIA.

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 imple- mentation of our technique in a common web application platform, ASP.NET, C# and demonstrated its efficacy and effective- ness. 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 program- mer, throwing an exception when appropriate. We have made 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 PHP incorporate automated injection error logging.

REFERENCES

[1] Cesar Cerrudo, *Manipulating Microsoft SQL Server Using SQLInjection*, http://www.appsecinc.com/presentations/ Manipulating SQL Server Using SQL Injection.pdf

[2] Steve Friedl, *SQL Injection Attacks by Example* http://www.unixwiz.net/techtips/sql-injection.html

[3] 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

[4] J. V. William G.J. Halfond and A. Orso, "A classification of sql injection attacks and countermeasures," 2006.
[5] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In Proceedings of the 26th International

[6] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[7] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. *Annual Symposium on Principles of Programming Languages (POPL)*, pages 372-382, 2006.

[8] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form eld validation. World Wide Web, 3(4):205{214, 2000.

[9] C. Anley. Advanced SQL Injection In SQL Server Applica- tions. *White Paper, Next Generation Security Software Ltd.*,2002.

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

[11] C. Anley. (more) Advanced SQL Injection. *White Paper, Next Generation Security Software Ltd.*, 2002.

[12] S. McDonald. SQL Injection Walkthrough. http:// www.securiteam.com/securityreviews/5DP0N1P76E.html, May 2002

[13] K. Spett. SQL Injection: Are your web applications vulner- able? *SPI Labs White Paper*, 2004

[14] Struts. Apache Struts project. http://struts.apache.org/.

[15] 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.

[16] Hibernate. hibernate.org. http://www.hibernate.org/.

[17] G. Wassermann and Z. Su. An Analysis Framework

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buer overow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, pages 91{104, August 2003for Security in Web Applications. *In Proceedings of the FSE Workshop on Specification and Verification of Component- Based Systems (SAVCBS)*, pages 70–78, 2004

[19] 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

[20] 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 Engineer- ing (ICSE)*, pages 697–698, 2004.

[21] Paros. Parosproxy.org. http://www.parosproxy.org/.
[20]http://www.information-systems research.com/blog/2011/11/08/the-ever-rising-sqlinjectionattack

BIOGRAPHIES

Peeyush Mathur is M. Tech. in Computer Science & Engineering and having vast experience in teaching

Kajal Mathur is pursuing her M.Tech from Rajasthan Technical University, Kota(Rajasthan).

Pulkit Mathur is pursuing his M.Tech from Rajasthan Technical University, Kota(Rajasthan).

Puneet Mathur is pursuing his M.Tech from Rajasthan Technical University, Kota(Rajasthan).