International Journal of Innovative Research in Computer Science and Engineering (IJIRCSE)
www.ioirp.com ISSN: 2394-6364, Volume – 2, Issue – 5. May 2016

Detecting SQL Injection Attacks in Web Application Using REGEX and Query Result Size

Gowthami S
PG Student
Department of Information Technology
Kongu Engineering College, Erode
E-mail: mailtogowthamikarthik@gmail.com

K.R.Prasanna Kumar
Assistant Professor
Department of Information Technology
Kongu Engineering College, Erode
E-mail: krprasanname@gmail.com

Abstract - Nowadays, web applications are essential part in the real world environment. Nearly each and every major organization has a web presence. Most of these firms and organizations use web applications to provide various services to users. Many of these web applications make use of database driven content. The back-end database often contains privileged and sensitive information. As the scale of these applications grows, they are exposed to SQL injections. These attacks cause a serious threat to the data of web applications because it can provide attackers an unrestricted access to databases through a web front-end. In addition, attackers take advantage of deficiencies within the input validation logic of web elements. The proposed system reveals a unique scheme that automatically recasts web applications, rendering them safe against SQL injection attacks. This method first sanitize the code using regular expression and dynamically scrutinizes the developer-intended query result size for the input submitted and spots attacks by comparing this against the result of the actual query. The proposed system is implemented as a web application using asp.net and c# languages.

Keywords: parse; regular expression; sanitize; query result size; malicious; selectivity

I. INTRODUCTION

1.1 Web Application Security

Web applications become the essential part of day-to-day life. The application programs are stored on a remote server and brought over the internet through a browser interface. Web applications evolved from web sites or web systems. The Figure 1.1 shows the three-layered web application model. The first layer is a web browser where the user interacts, the second layer is the dynamic content generation technology like Java Servlet Pages (JSP) or Active Server Pages (ASP) and the third is the database containing confidential and sensitive data.

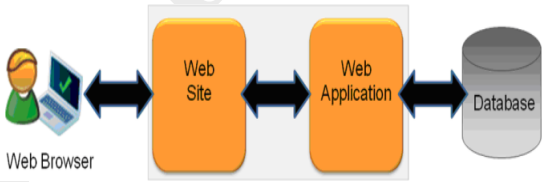

Figure 1.1 Basic Architecture of Web Application

Despite this necessity, much vulnerability still exists in web applications. Web applications provide customized content to the user after accepting the input. Therefore, it is important to protect web applications from the attacks which give unauthorized access to the system and private information. Intrusion detection systems are largely ineffective against SQL injection attacks. Firewalls and antivirus software are also ineffective in detecting SQL injection attacks. As the scale of the web applications increase, code injection vulnerabilities such as SQL injections, become major security challenge. According to top ten vulnerabilities submitted by OWASP community, injection flaws are considered to be the top one vulnerability exists in web application. Most of these vulnerabilities caused from lack of input validation that is, web applications use malicious input as the target sensitive operation without properly checking the input submitted the user and subsequently use that data to dynamically construct an SQL query to the database backing that application. These crimes target the confidentiality, integrity or availability of resources possessed by an application, its creators and its users.

The paper is organized as follows. In section 2 we define SQL Injection attack and present different SQLI attack types. In section 3 we review current tools against SQLI. In section 4 we present the proposed technique.

In section 5 we evaluate SQL Injection detection tools against all types of SQL injection attacks and deployment requirements. Conclusion and future work is provided in section 6.

II. BACKGROUND

2.1 SQL Injection Attack

Structured Query Language (SQL) is the special purpose programming language for managing databases that allows the storage, manipulation and retrieval of data. Web based forms must allow to access the database for the entry of data and for the response to the user, so the SQL injection attack can bypass the firewalls and endpoint defenses. Any web form, even a simple log-on form, might provide access to the database by means of SQL injection if coded improperly.

SQL injection is a code injection technique for exploiting web applications that use user-supplied data in SQL statements. SQL injection is currently the most common form of web application attack in that forms are not coded properly and there are as many hacking tools used to find vulnerabilities and to take advantage of them. In the hands of the very skilled hacker, a web code weakness can provide root level access of web servers. Even the firewalls and anti-virus software are unable to find this type of injection attacks.

Improperly coded forms will allow the attacker to use them as an entry point to the database at which point the data in the database may become accessible and also allows access to other databases on the same server or other servers in the network may be possible. Even though SQL injection has been a known problem for years, there are several factors causing the rate of risk to increase.

2.2 Variants of SQL Injection Attack

SQL injection attacks can be characterized based on the goal or intent, of the attacker. The main variants of this attack are as follows

Tautology
In this type of attack the attacker tries to use a conditional query statement to be evaluated always true. Attacker uses WHERE clause to inject and turn the condition into a tautology which is always true.

Example
SELECT * FROM Accounts WHERE accno=‘‘or 1=1 - - AND pass= ‘password’;

The result would be all the records in Accounts table because the condition of the WHERE clause is always true.

Illegal/Logical Incorrect Queries
When a query is rejected an error message is returned from the database including useful debugging information. This information helps attackers to make move further and find vulnerable parameters in the application and consequently database of the application.

Example
SELECT * FROM Accounts WHERE user=‘‘AND pass =convert (int, (SELECT TOP 1name FROM objects WHERE x type=‘u‘))

The attacker attempts to convert the name of the first user defined table in the metadata table of the database to _int‘.

Union Queries
In this type of attack, unauthorized query is attached with the authorized query by using UNION clause.

Example
SELECT * FROM Accounts WHERE accno=‘‘ UNION SELECT * FROM details-- AND pass=‘‘

The result of the first query in the example given above is null and the second one returns all the data in details table so the union of these two queries is the details table.

Piggy-Backed Query
In this attack, attacker tries to add the additional queries in to the original query string. In this injection the intruders exploit database by the query delimiter, such as - -,‖, to append extra query to the original query.

Example
SELECT * FROM Accounts WHERE accno=‘‘; drop table Accounts;- - AND pass=‘ _

The result of the example is that Accounts table would be dropped from database. In this type of attack, intruders change the behavior of a database of application.

Blind Injection
This is little difficult type of attack for attacker. During the development process sometime the developer hides some error details which help the attacker to compromise with database. In this situation the attacker face the generic page provided by developer in place of an error message.

Example
SELECT * FROM Accounts WHERE accno=1001 AND 1=1 - - AND pass=’’

In this injection it is always evaluated as true if there are no any error message, and the attacker realizes that the attack has passed user parameter is vulnerable to injection.

Timing attack
In this attack, the attacker gathers information about the response time of the database. This technique is used by executing the if-then statement which results the long running query or time delay statement depending upon the logic injected in database and if the injection is true then the ―WAITFOR‖ keyword which is along with the branches delays the database response for a specific time.

Example
SELECT * FROM Accounts WHERE accno=1001 AND ASCII (SUBSTRING ((SELECT TOP 1 name FROM sysobjects),1,1))>X WAITFOR DELAY _000:00:09‘- -AND transdate=’’

Attacker trying to find the first character of the first table by comparing its ASCII value with X. If there is a 9 second delay, the attacker realizes that the answer to his question is yes. So by continuing the process the name of the first table would be discovered.

Alternate encoding
In this type of attack, the regular strings and characters are converted into hexadecimal, ASCII and Unicode. Because of this, the input query is escaped from filter which scans the query for some bad character which results SQLIA and the converted SQLIA is considered as normal query.

Example
SELECT * FROM Accounts WHERE accno=1001; exec(char(0x8774675u8769e))-- AND pass=’’

The functions will get integer number as a parameter and return as a sample of that character. In the example it will return ―SHUTDOWN‖, so whenever the query is interpreted the SHUTDOWN command is executed.

Stored procedure
Stored procedure is the built in extra abstraction layer on the database defined by the programmer. By using the stored procedure the user can store its own function according to the need. It is extending the functionality of database and interacting with the operating system. Then the attacker tries to identify the underlying database in order to exploit the database information.

III. RELATED WORK

Based on the approaches used by the authors in the literature, we discuss the related work in the following categories.

A) ***Positive tainting and syntax-aware evaluation***

Halfond et al presented Web Application SQL Injection Preventer (WASP), a method for detecting and preventing SQLIAs on the server. WASP utilizes positive taint analysis and syntax-aware evaluation. Using positive taint, WASP fetches trusted values from a metastring library, and tracks them through the application to identify trusted parts of a query. Although this technique is not fully automated and requires a white-list, it is important to the proposed work because in proposed work, it uses such a list to detect attack patterns in our detection model.

B) String Analysis

Fang Yu et al developed a string analysis-based framework that, given attack patterns, automatically generates vulnerability signatures. These signatures are constructed via forward symbolic reachability analysis, followed by backward symbolic reachability. Some vulnerability found in the forward reachability analysis may be false positives.

C) Dynamic Analysis

Liu et al proposed SQLProb which is another dynamic analysis approach. It uses a customized MySQL proxy to collect queries, extract user input, generate parse trees, and validate user input. This technique can calculate the similarity between the given query and every previously collected query. In the data collection phase, all the queries are stored in the system. In the query evaluation phase, an incoming query generated by the application is sent to the evaluation modules. The drawback of this technique is that it is a program for MySQL servers, and therefore cannot be used on other database systems.

D) Hashing

Shaukat Ali adopted the hash value approach to improve the user authentication mechanism; SQL Injection Protector for Authentication (SQLIPA) was developed to test this framework. User name and password hash values are calculated at runtime when a particular user account is created. This technique was only tested on a few sample data.

E) Query Tokenization

Anjugam presented a lightweight method to prevent SQL injection attacks by applying query tokenization technique to convert SQL queries into number of useful tokens and then encrypting the table name, fields, literals and data on the query using AES algorithm. This approach avoids memory requirements to store the legitimate query in repository and facilitates fast and efficient accessing mechanism with database.

F) Flag Sequencing

Manveen Kaur proposed a flag sequencing method in which the query is converted into flags and flags are separated and converted to integer values. Then the structures of the input query and previously stored query are compared. There are also some complexities involved such as flag separation, flag to integer conversion and the searching process in link list. Complexity of flags to integer conversion is O(n) where n is the total number of literals in all flags of query.

G) Adaptive Algorithm

Ashish John dealt an adaptive algorithm to detect SQL injection attacks which combines two methods. Parse tree validation technique is used to find the SQL injection query by comparing query length and code conversion method is used for converting the user input to code like ASCII, binary, etc and searching for the availability of converted input in the database. The disadvantages of this method are (i) Code conversion to each

and every user input is more time consuming as well as the database size will also increase. (ii) Parse tree validation technique will raise false alarm even if legitimate user is having blank space in his/her input.

IV. PROPOSED APPROACH

The main objective of the proposed system is to estimate the query result size to prevent the SQL injection attack. In this method, the result size of substituted query is estimated. The proposed technique first uses the positive taint propagation that characterizes the sanitization process by modeling the way in which the application processes input values. Second, the query result size evaluation is done.

In this proposed system, the information needed to check the attack patterns are determined and estimation technique is applied to compare the substituted query result size with the actual query result. The Figure 4.1 shows the basic architecture of the proposed system. The user interacts with the user interface normally a web browser, the substituted input is analyzed for attack patterns using regular expression and then the query result size is estimated and compared with the size of real queries and output is sent to user interface.

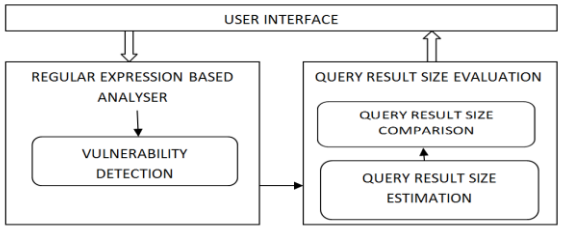

Figure 4.1 Basic Architecture of Proposed System

4.2 Module Description

The proposed system consists of the following modules

- Regular Expression based Query Analyzer
- Query Result Size Evaluation

4.2.1 Regular Expression Based Query Analyzer

Regular expressions are a powerful and simple way to describe a set of strings. For this reason, they are often chosen as the input language for text processing applications. For instance, in the lexical analysis phase of compilers, regular expressions are often used to specify and distinguish tokens to be passed to the syntax analysis phase. A trivial regular expression to detect SQL injection attacks is to watch out for SQL specific meta-characters such as the single-quote (') or the double-dash (--).

Examples of regular expression
(i) Regex for numeric input:

(\"[0-9]\")*)

This regular expression pattern is used to check whether the given input is a only numeric values or not. The input of this type contains only numbers from 0 to 9 in any order and any number of times.
(ii) Regex for date:

^(0[1-9]|[12][0-9]|3[01])[- / .](0[1-9]|1[012])[- / .](19|20)\d\d$

This regular expression pattern checks whether the input is in date pattern or not. The input in this type will be in dd/mm/yyyy format. If the input is mismatched, it reports error message.

Regular expressions for different input field are constructed and then the system checks the attack patterns against these regular expressions. If any rule mismatches, then the attack pattern is sent to the detection system. Then the system analyzes the attack pattern using internal database to classify the vulnerability. To evaluate the

query string, string is broken into sequence of tokens and checks whether those contains only trusted data. If all tokens pass the test, the query is considered as safe and it is passed to query result size evaluation. Consider the following relational Table 4.1,

Table 4.1 *Bankde*t Table in Database

ACCNO	NAME	ACC TYPE	DOB	BALANCE
1001	RAJ	SAVING	12.12.1993	10000
1002	MANO	CURRENT	24.09.1991	20000
1003	SRIJA	CURRENT	05.10.1994	30000
1004	RAM	SAVING	07.04.1987	40000

Consider the query,

SELECT * FROM Bankdet WHERE accno=1001; DROP TABLE bankdet

If an attacker obtains the table privileges and enters the value " 1001; DROP TABLE bankdet," the table will be dropped. To replace the vulnerable strings in the SQL query, split() function which is an inbuilt-function, is used. The split() function splits "SELECT * FROM Bankdet WHERE accno=1001 " from "DROP TABLE Bankdet". Now the output of the query will be,

1001 RAJ SAVING 12.12.1993 10000

4.2.2 Query Result Size Evaluation

In a relational database, relations are accessed by using SQL queries. Cardinality and selectivity are used to estimate the query result sizes. The count() function which is defined in the SQL functions can also be used to obtain the query result size. The selectivity is the measure of how much variety there is in the given column in relation to the total number of rows in a table. The following is an expression for selectivity estimation,

Selectivity= selected number of tuples / total number of tuples (4.1)

4.2.2.1 ELS Algorithm

Equivalence and Largest Selectivity (ELS) algorithm is used to estimate the query result size. This algorithm consists of two phases. First is the preliminary phase, which is performed before any result size is estimates. In this phase, predicates implied by the join selectivity are calculated. The second phase incrementally computes the query result sizes.

After estimating the query result sizes, result sizes of the substituted query and the real query are compared. If they match, the query is considered to be safe otherwise it is unsafe query.

Example,

Consider the following normal query,

Query 1: SELECT * FROM bankdet WHERE accno=1001;

Total number of tuples, ||B|| =10,
Selected number of tuples, B(accno=1001)=1,
Selectivity = 1 / 10 = 0.1
Consider the following attack query,

Query 2: SELECT * FROM bankdet WHERE accno=1000 OR 1=1—

Total number of tuples, ||B|| = 10,
Selected number of tuples= 10, WHERE clause become true because of " OR 1=1"
Selectivity= 1

The query result size of the normal query and the attack query varies i.e., the selectivity of the normal query is 0.1 and selectivity of attack query is 1. While comparing the result size of these two queries, they are not same. So the substituted query is considered as SQL injection query.

V. RESULT ANALYSIS

The proposed system is tested by giving different malicious inputs. The proposed system detects five different kinds of SQL injection attacks such as tautology, illegal/logically incorrect queries, blind injection attack, piggy-backed queries and union queries.

The proposed system is compared with the existing approach like SQLIPA algorithm. The following Table 5.1 shows the comparison between the proposed and SQLIPA approaches.

The web application was installed on the desktop computer having configuration mentioned above under Microsoft IIS 7.0. The proposed technique provides full protection from most varieties of SQL injection, but it has one limitation. This approach cannot prevent second order SQL injection attempts.

Table 5.1 Comparison between existing and proposed approach

Type of attack	SQLIPA	PROPOSED
Tautology	yes	yes
Illegal/logically incorrect queries	no	yes
Piggy-backed queries	no	yes
Union queries	no	yes
Blind injection	no	yes

VI. CONCLUSION

The important problem addressed in the existing system is the improper validation of input. To overcome this problem, the system identifies the vulnerabilities that stem from incomplete sanitization. The SQLIAs is detected and prevented by automatically analyzing the target operation. By using regular expression based analysis, it is possible to detect and prevent SQL queries that include injection vulnerabilities in the input fields. This method detects five different types of attacks. The query result size evaluation provides guarantee in securing the applications and eliminates well-known attacks. In future, the second order attacks may be prevented by extending the technique and also other web-based attacks may be concentrated.

REFERENCES

[1] Anjugam S, A Murugan (2014), 'Efficient Method for Preventing SQL Injection Attacks on Web Applications Using Encryption and Tokenization', IJARCSSE, Vol.4, No.4, pp.124-132.
[2] Ashish John, Ajay Agarwal, Manish Bhargwaj (2015), 'An adaptive algorithm to prevent SQL injection', AJNC, Vol.4, pp.12-15.
[3] Fang Yu, Muath Alkhalaf, Tevfik Bultan (2009), 'Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses', IEEE/ACM International Conf. on Automated Soft. Engg., pp.605-609.
[4] Halfond WG, Orso A, Manolios P (2006), 'Using Positive Tainting and Syntax-Aware Evaluation to counter SQL injection attacks', Proceedings of the 14th ACM SIGSOFT International Symp. on Foundations of Soft. Engg., pp.175-185.
[5] Jaskanwal Minhas, Raman Kumar (2013), 'Blocking of SQL Injection Attacks by Comparing Static and Dynamic Queries', IJCNIS, Vol.5, No.2, pp.1-9.
[6] Liu A, Yuan Y, Wijesekera D, Stavrou A (2009), 'SQLProb: A Proxy-based Architecture towards preventing SQL injection attacks', Proc. of the ACM Symp. on Applied Comp., pp.2054-2061.
[7] Manveen Kaur (2015), 'SQL Injection attacks, Its Prevention by Flag Sequencing Method', IKSPCEIS, Vol.6, No.2, pp.102-110.
[8] OWASP (2013), Top_Ten_Projects, http//:www.owasp.org/Index/Top_ten.
[9] Shaukat Ali, Azhar Rauf, Huma Javed (2009), 'SQLIPA: An Authentication Mechanism against SQL Injection', European Journal of Scientific Research, Vol.38 No.4, pp.604-611.
[10] Sruthy Manmadhan and Manesh T (2012), 'A method of detecting SQL injection attack to secure web applications', International Journal of Distributed and Parallel Systems, Vol.3,No.6, pp.45-54.
[11] Young-Su Jang, Jin-Young Choi (2014), 'Detecting SQL injection attacks using query result size', International Journal of Computers & Security, Vol.44, pp.104-118.