International Journal of Advanced Technology in Engineering and Science www.ijates.com
Volume No 03, Special Issue No. 01, March 2015 ISSN (online): 2348 – 7550

DETECTION AND PREVENTION OF TAUTOLOGY AND UNION QUERY BASED SQL INJECTION ATTACKS

Jyoti Agrawal[1], Mukesh Gupta[2]

[1,2] *Dept. of Computer Science, SKIT, (India)*

ABSTRACT

Web applications are pervasive and play a vital role as web applications are significant mode of communication. SQL injection is one of the most dangerous security vulnerability that is exploited in web application by attacker to get the access of databases. This paper proposes a method SQL idetection and algorithm to detect and prevent the tautology and union query based attacks at run time. To demonstrate efficiency of this method dataset is taken from NIST.

Keywords: Detection, Input Validation, Prevention, SQL Injection Vulnerability, SQL Injection Attacks

I INTRODUCTION

At present time the use of web applications is increasing rapidly. We are using web applications in daily life in the different ways such as shopping, mailing, downloading-uploading various videos and audios, virtual network (social networking sites) etc. Web applications store information in databases that is to be delivered to legitimate customer, supplier or other host. Since a number of dangerous security vulnerabilities are present in these web applications, these web applications become prone to attacks and an attacker in that way can make malicious attacks. One of these attacks is SQL injection attack that gives a chance to attackers to amend the behavior of system by exploiting these vulnerabilities. Therefore the requirement to identify these vulnerabilities is raised.

Many researchers and practitioners are working to detect SQL injection attacks from crucial web applications. For this purpose they are applying several input validation and sanitization methods in their approaches [1], [2], [3], [4], [5]. NTAGW ABIRA Lambert and KANG Song Lin have proposed a method [7] called queryParser method. This method tokenizes the original query and query with injection after tokenization the indices of tokens is stored in two arrays. Then the length of both array are compared if lengths are equal then there is no injection else there is injection. This method will produce false positives because if user enters invalid input which does not result in SQLIA for example for user input “CH01 ASE” this method will print there is an injection because the length of original query and query with injection is not same.

The proposed approach is able to detect and prevent the web applications from tautology and union query based SQL injection attack in a simple and manner without applying any sanitization and filtering.

The rest of this paper is outlined as follows. Section 2 describes SQL injection with tautology and union query based SQLI attacks, section 3 demonstrates our approach and section 4 presents empirical evaluation of our approach. At last section 5 talks about the conclusion and future work in this direction.

II SQL INJECTION

OWASP (Open Web Application Security Project) has reported that SQL injection is one of top10 vulnerabilities. SQL Injection (SQLI) attack is one in which an unauthorized user gets access to unprivileged data. Since the web applications suffer from improper input validation. This lack of proper user validation allows attacker to find injectable fields to exploit vulnerabilities. After succession in attack an attacker can manipulate the valid user's data without knowledge to that user.

2.1Tautology Attack

In tautology-based attack attacker injects code in one or more conditional statements so that they always evaluate to true. Consider a website's page in which SQL query is dynamically created and includes user input fields. The following query is used for fetching information about books:

"SELECT * FROM books WHERE authorname= ' " + authorname + " ' ";

In a general way this web page having one user input field authorname which have the valid entries as stored in its database.But an attacker can enter the malicious inputs in user input field as authorname: anything' or `x'=`x then resultant query will have following form:

"SELECT * FROM books WHERE authorname = 'anything' or `x'=`x'"

In this way resulting query will allow the attacker to access the complete table without actually knowing a valid authorname because in this WHERE CLAUSE is always true.

2.2Union Query Based Attack

In this type of attack an attacker uses a vulnerable parameter to modify the data set returned for a given query. This is done by injecting a UNION SELECT statement. This additional query allows an attacker to fetch the data from a specified table by getting the rights and privileges of authorized user. For example let an attacker can inject the string "' UNION SELECT creditcardno, pinno FROM creditcard" into the newsid field. Therefore resultant query is:

SELECT newstitle, newsbody FROM news WHERE newsid = '340' UNION SELECT creditcardno, pinno FROM creditcard

In this dynamic illegitimate query statement the first part returns two attribute values of newstitle and newsbody corresponding to newsid 340 along with this the second query returns data from the "creditcard" table because the result that is returned is the union of both original and injected query statement.

III OUR APPROACH

This approach is able to detect and prevent the web applications from those SQL injection attacks that results in stealing the crucial data from database. When a web application is vulnerable to attacks, in such a manner that a

malicious user can inserts some malicious input instead of original required input in user input field willing to fetch data from database for which that user is not authenticated. Then the SQL query statement formed by user input becomes an illegitimate SQL query statement and then the execution of this illegitimate statement will fetch data from database. Proposed approach will detect that there is an attack if the web application fetches data from database even if user entered wrong input or malicious input and then this approach will prevent these data from attacker's eye by not displaying the fetched data.

Our approach consists of implementation of a function called SQLIdet() to detect if there is an SQL injection or not. Major components of this approach are as in Fig.1.

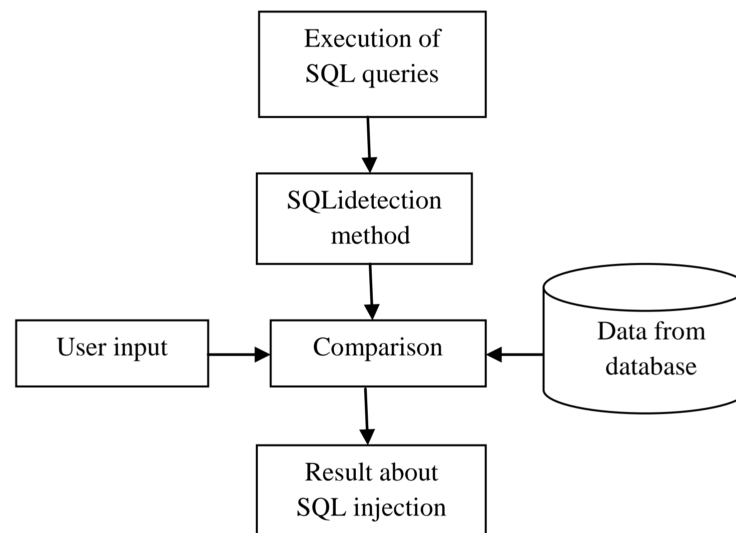

Fig.1: Block diagram of proposed method

This proposed technique follows following algorithm that has 3 basic steps which are as follows:

Inputs:

SQL_o: Original query

SQL_n: Query created for selecting user input fields

$\$result_o$: result variable of SQL_o

$\$result_n$: result variable of SQL_n

$\$U_i$: user inputs

$\$index_i$: user input field index of each U_i

Output:

Result of SQLI

Steps:

1. Execute SQL_o and Execute SQL_n
2. Call function SQLIdet() with parameters $\$result_o$, $\$result_n$, U_i and $\$index_i$
3. Execution of function
   i. Compute no. of rows in $\$result_n$

ii. If no. of rows>0 then

a. Fetch first row of $\$result_n$ from database

b. Compare fetched data of $\$index_i$ column with U_i

c. If both matches

d. Then perform required action

e. Else print “injection”

iii. Else return

4. End

To understand how actually this technique works consider an example in which a legitimate user enters “Emile Zola” in user input field “author” or any other valid author name. Resultant query for this input is:

“SELECT * from books
WHERE author=’Emile Zola’”

Then the query is executed and then proposed SQLidet() function is called in which this query’s result which is 3rd row of database and index of user input field that is author are passed. Then it fetches result in a variable and compares input entered by user with the fetched value of passed index. For taken example both the values input entered by user “Emile Zola” and row [author] = Emile Zola fetched from database matches. So there is no injection and required action is performed.

If user input is a crafted input then it performs the same process and when entered user input does not matches with database entry then it reports that there is an injection. For example an illegitimate user enters “anything’ or ‘x’=’x” then after executing the resultant query it selects complete database as it is a form of tautology attack. But when this function is called it compares “anything’ or ‘x’=’x” with each entry in author column fetched from database one by one and since this is an invalid user input it does not match and this function reports that there is an injection. In such a way this technique works at run time.

3.1 Empirical evaluation

To evaluate proposed approach consider one example of query present in test case 1940 of NIST benchmarks [6] the records and structure of table book of this test case is presented by Table 1 and Table 2 respectively. The original query is

“SELECT * FROM books WHERE Author = ’$q’”;

Where ‘q’ is the user input ‘author’ when an illegitimate user enters *anything’ or ‘x’=’x* an invalid input string willing to get unauthorized information. When SQLidet() method discussed in section 3 is applied on this test case then the result produced after applying this attack will be same as shown in Fig.2.

Since taken example is a tautology type SQLIA. Similar to this when this approach is applied to union query based attack then also the attack will be detected and web application will be prevented.

Table 1

Structure of table book

BookID	Name	Author
1	A la recherche du temps perdu	Marcel Proust
2	Ulysses	James Joyce
3	Germinal	Emile Zola
4	L'etranger	Albert Camus

Table 2

Records of table book

	Field	Type	Collation	Attributes	Null	Default	Extra
	BookID	tinyint(4)			No	None	auto_increment
	Name	varchar(255)	latin1_general_ci		Yes	NULL	
	Author	varchar(127)	latin1_general_ci		Yes	NULL	

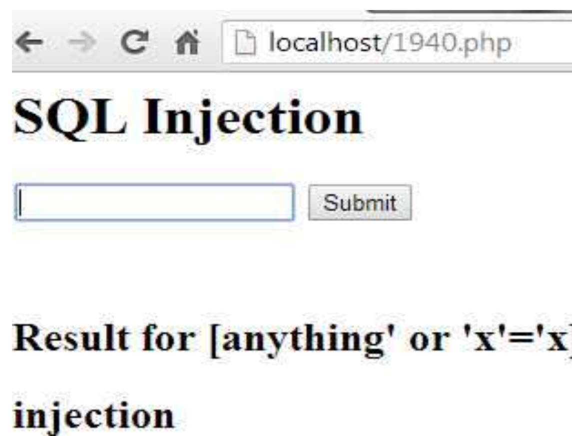

Fig.2: Result of proposed method

IV CONCLUSION AND FUTURE WORK

The method proposed in this paper is robust and more efficient as this method produces no false positive and no false negative. The proposed method is easy to understand and implement. This method requires no filtering and any sanitization approach to validate use input.

In future this technique can be enhanced to detect other SQL attacks and can also be extended to include different web applications attacks.

REFERENCES

[1] T. Scholte and W. Robertson, Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis, IEEE 36th International Conference on Computer Software and Applications (COMPSAC), 16 July 2012, 233-243.

[2] M. Ghafari, H. Shoja and M. Y. Amirani, Detection and Prevention of Data Manipulation from Client Side In Web Applications, IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, 2012, 1132-1136.

[3] M. Alkhalaf, T. Bultan and Jose L. Gallegos, Verifying Client-Side Input Validation Functions Using String Analysis, IEEE 34th International Conference on Software Engineering (ICSE), June 2012, 947-957.

[4] R. B. Brinhosa, C. M. Westphall, C. B. Westphall, D. R. dos Santos and F. Grezele, A Validation Model of Data Input for Web Services, The Twelfth International Conference on Networks, ISBN: 978-1-61208-245-5, 2013.

[5] W. Min and L. Kun, An Improved Eliminating SQL Injection Attacks Based Regular Expressions Matching, IEEE International Conference on Control Engineering and Communication Technology (ICCECT), 2012, 210-212.

[6] NIST, SAMATE Reference Dataset, http://samate.nist.gov/.

[7] N. A. Lambert and K. S. Lin, Use of Query Tokenization to detect and prevent SQL Injection Attacks, 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), vol.-2, July 2010, 438-440.

[8] E. Merlo, D. Letarte, G. Antoniol, Insider and Ousider Threat-Sensitive SQL Injection Vulnerability Analysis in PHP, IEEE 13th Working Conference on Reverse Engineering, 2006.