Comparison Between Various Detection and Prevention Techniques for SQL Injection Attacks

Anurekh kumar, Shobha bhatt
Student M.Tech IS , Assistant proffesor,
Computer Science Engineering Department ,
Ambedkar Institute of Advanced Communication & Research Technology, Delhi, India

***Abstract* - In this paper, we present a detailed review on using dynamic queries, there are lots of chances that a user may inject in the query some extra statements that can result in a different database request. Thus SQL injection gives information can be stolen from the database. Most applications are designed in a way that the request of data from database is done through user inputs. An attacker can inject in the original SQL query and obtain, change, or view data for which he does not have permission. The aim of our research is to develop a method that detects and prevents SQL injection attacks by checking whether user inputs cause changes in the query's intended result. We proposed a method to detect SQL injection attacks by using Query tokenization that is implemented by the QueryParser method. When attacker is making SQL injection he should probably use a space, single quotes or double dashes in his input. Our method consists of tokenizing original query and a query with injection separately, the tokenization is performed by detecting a space, single quote or double dashes and all strings before each symbol constitute a token. After tokens are formed they all make an array for which every token is an element of the array. Two arrays resulting from both original query and a query with injection are obtained and their lengths are compared to detect whether there is injection or not. As a result, the access to data can be granted or denied once the lengths of the arrays are the same or different respectively.**

***IndexTerms* - Tokenization, SQL Injection Attacks.**

I. INTRODUCTION

SQL Injection Attacks are a threat to any database driven applications and sites. Reference [I] pointed out that when a network and host level entry point is fully secured; the public interface exposed by an application becomes the only source of attack.SQL Injection Attack can be used by people who want to get access to the database and steal, change or delete data for which they do not have permission. Reference [2] has said that researchers have proposed different techniques to provide a solution for SQLIAs (SQL Injection Attacks), but many of these solutions have limitations that affect their effectiveness and practicability. Our research discusses the methodology of detecting and protect against SQL injection; this methodology consists of tokenizing both the original query and the one with injection separately, afterwards every token constitutes an index for the array and finally two arrays are formed. Once the comparison of the lengths of both arrays has been performed, it can be concluded whether there is or there is no injection: if the length of the two arrays is the same, no injection otherwise there is injection. In fact for dynamic queries, the user inputs may cause the generated array length not to conform to the one of the original query, consequently the system concludes that there is an SQL injection and finally access to data is denied.Our approach is implemented by a QueryParser method that facilitates the query tokenization and this reveals whether there is injection or not. The QueryParser method will be discussed in the following section.
After introduction, the section II with the name (Related work) follows. This section discusses different ways SQL injections can be done. After section II, the next section named (Our approach) follows, it discusses our method in detail. Lastly we end our work with sections Conclusion and References.

II. RELATED WORK

According to reference [3], a SQL Injection attack occurs when input from a user includes SQL keywords so that the dynamically-generated SQL query changes the intended function of the SQL query in the application.
In fact for all types of SQL injection, there is no way someone can perform injection without inserting a space, single quotes or double dashes in a query. The way a user can perform injection without single quotes is when the user input is of type number; for instance, the following query injection can be done without using single quotes:
Select*from student where userid= 100; The injection can be done like this:

Select*from student where userid=100or 1=1;

The absence of space between '100' and 'or' does not prevent the query to retrieve information about all students, however the space between 'or' and the fIrst 'I' is compulsory since its absence results in a syntax error.

For user input of type character, it is necessary to use single quotes. In the following example the use of single quotes is necessary, otherwise there is a syntax error.

Select*from student where userid='CHOOI'; The injection can be done like this:
Select*from student where userid='CHOOI'or 1=1;

The single quotes for 'CHOOI' and a space between 'or' and the first 'I' are necessary.
The injection can also be done by inserting double dashes in a query. In SQL, double dashes are used to add comments in a query, so the attacker can insert double dashes in a query and make the part after it as a comment. This is emphasized by researchers saying that the characters '- -' mark the beginning of a comment in SQL, and everything after that is ignored [4].
This following example illustrates where the attacker use of double dashes as a SQL injection.
Select*from student where userid=' CHOO1 ';-' and marks>50;
The ';-'is an attacker injection.
Briefly if the attacker attempt to make an injection attack against any query, he (she) should necessarily use in his input single quotes, spaces or dashes otherwise the injection cannot succeed or the query can fail with a syntax error. Reference [5] shows a table that contains some examples of SQL Injection attack types.
If we compare the two arrays described respectively in figure 3 and figure 4, we observe that their lengths are different. Indeed, the array in figure 4 has indexes ranging from 0 to 8, whereas the indexes of array in figure 3 are ranging from 0 to 5. Thus, lengths are different and according to our method there is injection. The main idea of our research resides on the criteria for which a token is formed. In our methodology the query parser method uses two queries and for every query it detects a space, single quote ('), or double dashes. The detection of these symbols creates a token consisting of strings before each of them.

III. PROPOSED WORK

We proposed a method that we called QueryParser whose execution must be done before the query is run in the application.
The QueryParser method conveys two queries (the original query and the query with injection) each of which is considered as a text string. The tokenization is firstly made for every string and is performed by detecting either space, single quote ('), or double dashes. Secondly, the QueryParser reads the string element by element and when it reaches a space, single quote, or double dashes, it creates a token for the string before each of them. Finally, when the tokenization is finished, every token is considered as the element of the array and thereafter the QueryParser compares the length for the two arrays: their difference in length results in the injection, whereas their equality in length leads to absence of injection.

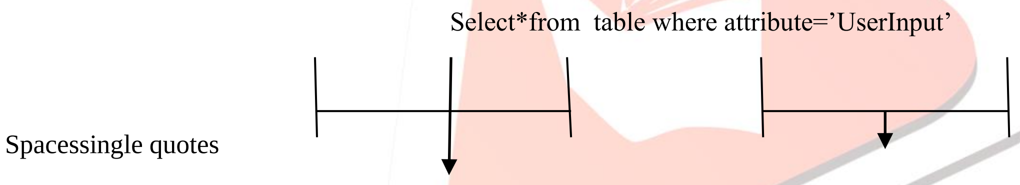


Figure 1: Origional query with spaces and single quotes

	1	2	3	4
Select*from	table	where	attribute=	userInput

Figure 2:Resulting array after tokenization of the query

Query with injection
Select*from table
Where attribute='UserInput' or ' 1 '=' 1 '

Figure 3: Injected query with spaces and single quotes

The corresponding array is:

0	1	2	3	4	5	6	7	8
Select*from	Table	where	attribute=	UserInput	or	1	=	1

Figure 4: Resulting after tokenization of the query with injection attacks.

IV. Proposed Model

SQLIA is a server type of web vulnerability, which impacts badly on web applications. In this section, a novel model for SQLIA prevention is proposed. As mentioned in previous section, several models are proposed for prevention of SQLIA, but they are not applicable for all type of injection attacks. SQLIA prevention via double authentication through tokenization is an approach to control SQLIA. This paper proposes double authentication process on both relational and hierarchical databases by applying tokenization approach on both databases. This task is performed via three steps.
Step 1: Query Forwarding
Step 2: Tokenization process on query
Step 3: Comparison of array index
Below figure shows the proposed architecture of SQLIA prevention through double authentication via tokenization by using above three essential steps

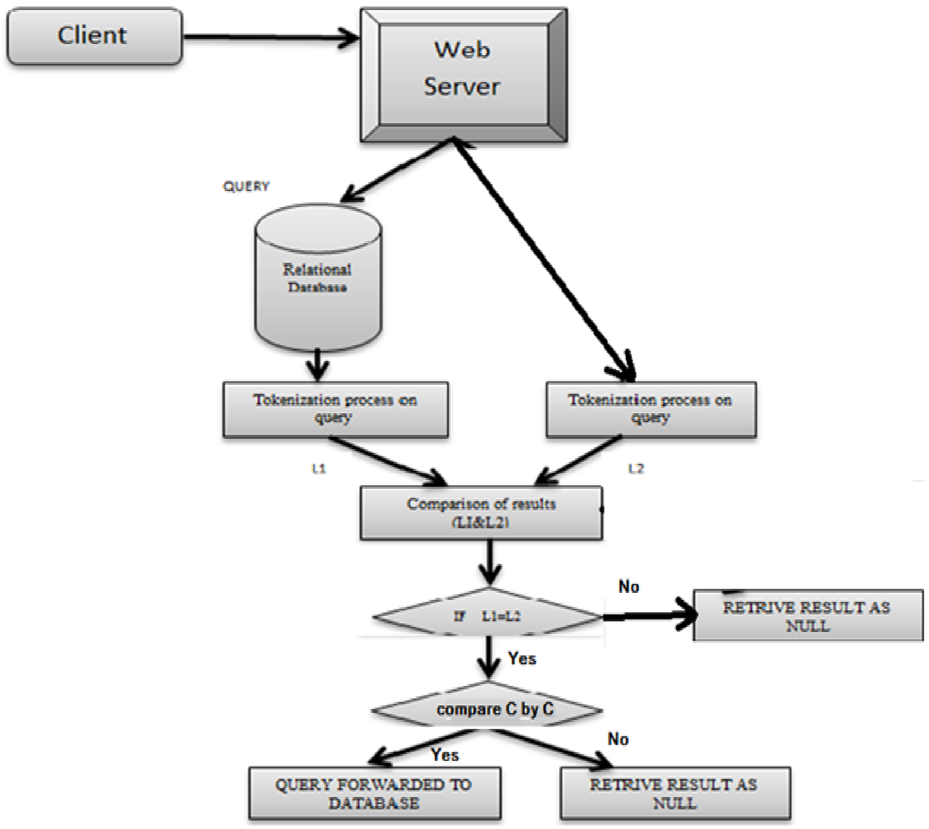


Figure 10: Proposed Architecture

Step 4: Compare character by character if L1=L2.
Step 1: Query Forwarding- When a query comes from a user via user interface, the input query is forwarded to both relational databases.
Step 2: Tokenization Process on Query- the input query is divided into various tokens on the basis of space, single quotes and double dashes between them. Once the tokens are decided, they are stored in array. Tokenization process is applied on both databases.
Step 3: Comparison of Array Index- In this step, the array length of both the arrays are compared. If the length of L1 and L2 are same then compare character by character which Step 4, if C1=C2, there is no injection present in the query and the query is proceed further to main database for retrieving result. But if the lengths L1andL2are different, then injection exists and query is not forwarded to the database. The result is a NULL value.

V. IMPLEMENTATION OF PROPOSED WORK

Below illustrates the code for the query parser with two queries: the original query (query) and the query with injection (query2). In this we take two query one is original and other is injected query.

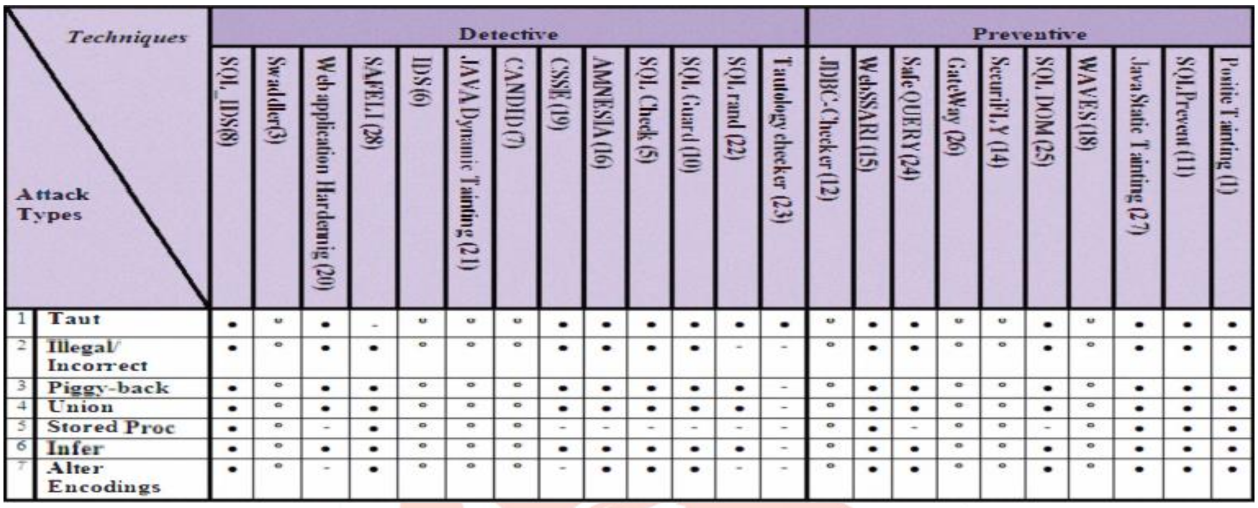
```
package sqlinjection;
import java.util.regex.*;
public class Major {
   public static void main(String[] args) {
String query="SELECT * FROM STUDENT WHERE STUDID='ABC' != 'XYZ';";
String query2="SELECT * FROM STUDENT WHERE STUDID='EMP' != '001';'--'";

                         String[] tokens = query.split("[\\s']|(--)");
                         String[] tokens2= query2.split("[\\s']|(--)]");
                         for(String token: tokens)
             System.out.println(token);
            for(String token: tokens2)
             System.out.println(token);
            if(tokens.length != tokens2.length)
                 {
              System.out.println("There is Injection");
                         }
              else
                         {
```

```
                System.out.println("No Injection");
                }}}
```

After the code is run, the result shows that there is injection if tokens lengths of both query original and injected query are different else no injection. Thus, without using our method the attacker should have got all the information.

VI. Comparison of SQLI Detection/Prevention Techniques with Respect to Attack Types

Tables 4 summarize the results of this comparison. The symbol "." is used for technique that can successfully stop all attacks of that type. The symbol "-" is used for technique that is not able to stop attacks of that type. The symbol "0"refers to technique that stop the attack type only partially because of natural limitations of the underlying approach. As the table shows the Stored Procedure and Alternate Encoding are critical attacks which are difficult for some techniques to stop them. Stored Procedure is consisting of queries that can execute on the database. However, most of tools consider only the queries that generate within application. So, this type of attack make serious problem for some tools.



Techniques / Attack Types		Detective													Preventive									
		SQL_IDS(8)	Swaddler(3)	Web application Hardenig (20)	SAFELI (28)	IDS(6)	JAVA Dynamic Tainting (21)	CANDID(7)	CSSE (19)	AMNESIA (16)	SQL Check (5)	SQL Guard (10)	SQL rand (22)	Tautology checker (23)	JDBC-Checker (12)	WebSSARI (15)	Safe QUERY (24)	GateWay (26)	SecuriFLY (14)	SQL DOM (25)	WAVES (18)	Java Static Tainting (27)	SQLPrevent (11)	Positive Tainting (1)
1	Taut	●	○	●	-	○	○	○	●	●	●	●	●	●	○	●	●	○	○	●	○	●	●	●
2	Illegal/ Incorrect	●	○	●	●	○	○	○	●	●	●	●	-	-	○	●	●	○	○	●	○	●	●	●
3	Piggy-back	●	○	●	●	○	○	○	●	●	●	●	●	-	○	●	●	○	○	●	○	●	●	●
4	Union	●	○	●	●	○	○	○	●	●	●	●	●	-	○	●	●	○	○	●	○	●	●	●
5	Stored Proc	●	○	-	●	○	○	○	-	-	-	-	-	-	○	●	-	○	○	-	○	●	●	●
6	Infer	●	○	●	●	○	○	○	●	●	●	●	●	-	○	●	●	○	○	●	○	●	●	●
7	Alter Encodings	●	○	-	●	○	○	○	-	●	●	●	-	-	○	●	●	○	○	●	○	●	●	●

VII. CONCLUSION

To make SQL injection attack, an attacker should necessary use a space, double quotes or double dashes in his input. The method to detect one of the above symbols has been discussed. Our method consists of tokenizing original query and a query with injection and after if it is found that additional symbols have been used in user input, the injection is detected. Our approach consists of tokenizing the original query and the query with injection, and after tokens are obtained they constitute arrays' elements. By comparing lengths of the resulting arrays from the two queries injection can be detected. The work presented in this paper has been implemented using java codes.

VIII. References

[I] R. Ezumalai and G. Aghila. Combinatorial Approach for Preventing SQL Injection Attacks. IACC, 2009.
[2] MeiJunjin. An approach for SQL Injection vulnerability detection. IEEE,2009.
[3] Ke Wei, M. Muthuprasanna and Suraj Kothari. Preventing SQL Injection Attacks in Stored Procedures. IEEE, 2006.
[4] Nuno Antunes and Marco Vieira. Detecting SQL Injectionvulnerabilities in web services. IEEE,2009.
[5] William GJ. Halfond, Alessandro Orso, Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks, 14th ACM SIGSOFT international symposium on Foundations of software engineering 2006, ACM.
[6] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID:Preventing SQL Injection Attacks using Dynamic Candidate Evaluations, 2007, Alexandria, Virginia, USA, ACM.
[7] Marco Cova, Davide Balzarotti. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), (Queensland, Australia), September 5-7, 2007, pp. 63-86.
[8] Xin Jin, Sylvia Losborn. Architecture for Data Collection in Database Intrusion Detection System. Secure Data Management. Pages 96-107.Springer Berlin /Heidelberg. 2007.
[9] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.
[10] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.