SQLI-Dagger, a Multilevel Template based Algorithm to Detect and Prevent SQL Injection

Teresa K. George
Research Scholar
Cochin University of Science and Technology

Rekha James,PhD
Associate Professor
Cochin University of Science and Technology

ABSTRACT
SQL injection attacks are often found within the dynamic pages of a web application that exploit the security vulnerability of the database layers of an application. In this attack category a specifically crafted SQL command is entered in the form field of a web application instead of the expected information. SQL injection takes advantages of the design flaws in poorly designed web applications to poison SQL statements and bypass the normal methods of accessing the database content .In these types of Injection attempt the database server execute undesirable SQL Code to steal, manipulate or delete the content of a database. The proposed algorithm is implemented on an application which is placed on a proxy server kept between the Database server and a web server. It is working on multi-level template based approach, which is a model based approach to detect the illegal queries before they are executed on the database server. With the support of the query evaluation engine it can detect and block the injected query. Only the benign query is allowed to get the access to the back end database server. An alert message is generated if there is an Injection.

Keywords
Vulnerability, Exploit, injection, Benign Query, Detection, Prevention

1. INTRODUCTION
SQL injection attack compromises back end database servers of an online application and perform the critical information disclosure or even manipulate the databases. Privilege escalation and unauthorized access to the database are most common outcome of this attack. SQL Injection Attack can be easily carried out by a malicious user if the user inputs within the applications are not validated properly or the available vulnerability scanners are not effective to validate the interactive queries send to the database servers through the web server [1].
SQL Injection makes use of the publically available fields to gain entry to the database. As the threat of SQL injection become more advanced, the need for developing a defense against SQL injection is greater than ever. In the hands of a very skilled hacker, a web application code weakness can reveal a root level access of application servers by bypassing firewalls and endpoint defense [2]. Databases that use SQL include MS SQL server, Oracle, MySQL and Access are equally subjected to Injection attack if coded incorrectly.

The proposed algorithm is a model based approach, working on two different levels of code injection as per the specified template to detect the illegal queries before they are executed on the database server [3]. In this approach a specific identity for each legal query is created by the developer, using the predefined template and considering all possible types of standard query format. This unique identity for each Intended query is placed in a template repository with the support of an evaluation engine; it can effectively detect and alert the presence of an Injection attack [4].

2. MOTIVATION
Most of the clients at the online application are accessing the backend database server by means of web forms, logon screens. These user inputs are directed to the database servers and are executed in the form of a Structured Query Language statement. If those applications are having in appropriate input validations, then the hackers will get unauthorized access to the critical servers such as database server and the entire application which requires a high degree of confidentiality can be compromised [5]. As the user interactions are growing higher and better, the automated techniques of attacks are also easily available. Hence, the attack spectrum is growing exponentially.

3. WEB APPLICATION AND SQL INJECTION ATTACK
SQL injection vulnerabilities put a severe threat on the online application, because it serves as an open door to the hacker to explore and compromise backend Database. SQL injection appear in a small percentage of applications, yet are making huge impact on the business organization in terms of data theft or compromising the most important database Server [6].

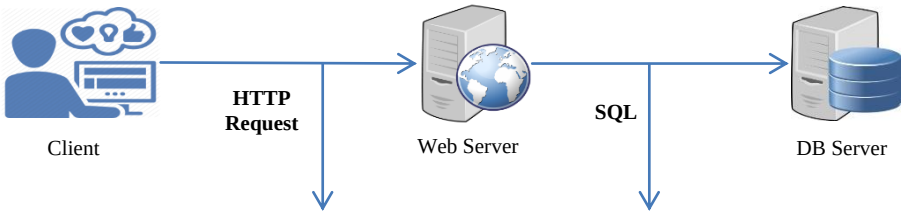

Figure 1: SQL Injection Attacks

User name: Arun " OR 1=1 OR '1'='1'
Password : “

SELECT * FROM administrators
WHERE username = Arun" OR 1=1 OR '1'='1'
AND password = '';

4. SQLI-DAGGER ALGORITHM

Web application usually consists of more than one static/dynamic pages; each page consists of one or more database request code. It is now obvious that each page can generate multiple database requests and so there will more than one legitimate query pattern. In our template based architecture, there is a template repository stores legal query with a unique identity [7]. The Query evaluation engine which is one of the core components in this template based architecture uses a SQLI-Dagger algorithm to parse the query to verify the presence of any type of injection attack. The query evaluation engine compares level by level and field by field between the standard query template JSON file and the input query template[8]. In any level if a contrast is found, then an error is reported. An alert message is send to the administrator, if the injection is detected and the injected query is blocked from further processing. Otherwise the query is considered as the *benign* query and forwarded to the backend database server for information access.

4.1.SQLI - Dagger Design specification for Detection

SQL query written by a user is considered as in the form of forest. Here, forest is a collection of independent queries written by user. A forest is an undirected tree which is composed of connected components in which each and every component can be represented as a tree. Each independent query is in tree like structure[9]. Each node in the tree contains sub-queries. Forest of independent query is represented as, F = [I1,I2,……,Ik] where I1,I2,……,Ik are independent queries.

Dagger Detection Algorithm for SQLI

```
Dagger Detection Algorithm (Forest F)
For each Tree I in F
        For each Vertex V in I
                Visited (V) = False;
        Next
        Traverse(Tree I)
Next
Traverse(Tree Node I)
        For each Vertex V in I
                If (!Visited (V)) Then
                        Visited (V) = True;
                        Boolean match =
                templateMatch (TreeNode.Value);
                        If (match==true) Then
Break;
                ElseIf(!IsNull(TreeNode.Child))
                                Return
                        queryText +=
                        TreeNode.Value;
                    Traverse(TreeNode.child);
                Else
                    Return queryText +=
TreeNode.Value;

                        End If
                 End If
        Next
End
```

Figure 2: SQLI_Dagger _the proposed algorithm

4.2.The template match procedure

The template match procedure is done as per the following template specification identified and after analyzing the different categories of probable legal queries which can be tested as per the below given template specification [10].

TemplateMatch(InputQuery)

1. String ValidQuery = getStandardQuery(InputQuery) //Return the standard query of the current node
2. Input_QuerytypeList[] = getQueryType(InputQuery)
3. Input_TableList[] = getTables(InputQuery)
4. Input_FunctionList[] = getFunctions(InputQuery)
5. Input_SplsymbolList[] = getSplsymbols(InputQuery)
6. Input_CommentList[] = getComment(InputQuery)
7. Input_OperatorList[] = getOperator(InputQuery)
8. Input_ColumnsList[] = getColumns(InputQuery)
9. Input_JoinsList[] = getJoins(InputQuery)
10. Input_S/mVariablesList[] = getS/mVariablesList(InputQuery)
11. Input_GlobalVariableList[] = getGlobalVariableList(InputQuery)
12. Input_KeywordList[] = getKeywordList(InputQuery)
13. Input_SubqueryNum=getNoOfChild(InputQuery); // Return the no childs of the current node from the tree
14. Intended_QuerytypeList = getQueryType(ValidQuery);
15. Intended_TableList[] = getTables(ValidQuery);
16. Intended_FunctionList[] = getFunctions(ValidQuery);
17. Intended_SplsymbolList[] = getSplsymbols(ValidQuery);
18. Intended_CommentList[] = getComment(ValidQuery);
19. Intended_OperatorList[] = getOperator(ValidQuery);
20. Intended_ColumnsList[] = getColumns(ValidQuery);
21. Intended_JoinsList[] = getJoins(ValidQuery);
22. Intended_S/mVariablesList[] = getS/mVariablesList(ValidQuery);
23. Intended_GlobalVariableList[] = getGlobalVariableList(ValidQuery);
24. Intended_KeywordList[] = getKeywordList(ValidQuery);
25. Intended_SubqueryNum = getNoOfChild(ValidQuery); // Return the no of childs of standard query of the current node
26. Boolean QuerytypeValid = Match(Input_QuerytypeList, Intended_QuerytypeList);
27. Boolean TablesValid = getMatch(Input_TableList, Intended_TableList);
28. Boolean FunctionValid = getMatch(Input_FunctionList, Intended_FunctionList);
29. Boolean SplsymbolValid = getMatch(Input_SplsymbolList, Intended_SplsymbolList);
30. Boolean CommentValid = getMatch(Input_CommentList, Intended_CommentList);
31. Boolean OperatorValid = getMatch(Input_OperatorList, Intended_OperatorList);
32. Boolean ColumnsValid = getMatch(Input_ColumnsList, Intended_ColumnsList);
33. Boolean JoinsValid = getMatch(Input_JoinsList, Intended_JoinsList);

34. Boolean S/mvariableValid = getMatch(Input_S/mVariablesList, Intended_S/mVariablesList);
35. Boolean GlobalvariableValid = getMatch(Input_GlobalVariableList, Intended_GlobalVariableList);
36. Boolean KeywordValid = getMatch(Input_KeywordList, Intended_KeywordList);
37. Boolean SubqueryValid = getMatch(Input_ SubqueryNum, Intended_SubqueryNum);
38. If(QuerytypeValid&&TablesValid&&FunctionValid&& SplsymbolValid&&CommentValid&&OperatorValid& &ColumnsValid&&JoinsValid&& S/mvariableValid&&GlobalvariableValid&&KeywordV alid&&SubqueryValid) Then
        Return True;
    End If
39. Boolean getMatch(List1,List2)
    SizeOfList1 = size(List1);
    SizeOfList2 = size(List2);
    Count = 0;
    If(SizeOfList1== SizeOfList2) Then
        For i=1 to SizeOfList1
            If(List1[i]!=List[i])
                Return False;
    Else
            Count++
        End If
    Next
Else
    Return False;
End If
If(Count==SizeOfList1)
    Return True;
End If

A sample screen shot of template format is given in **Figure 3.**

5. DISCUSSION AND RESULT

In Dagger detection algorithm each tree (independent query) in the forest is processed independently. For each tree first invoke the root node of the tree and check for injection. If any injection is detected, then the process is stopped. If that query is a valid query it returns the 'true' value. Next, if there is a child node for the current node, the child is checked and the process is repeated until a node is found with no child nodes. When we get a node with no children processed it. It is explained by using the example given in Figure 4. This query contains two independent queries, and each independent query contains sub-queries. Figure 5 shows the forest representation of above mentioned query. If we remove the root node we get the forest of two trees, these trees are the independent queries in the user query. Each red colour node represents the root of the tree, which means independent query. Each green and blue colored node are the sub-queries (child) in the independent query.

Figure 3: Sample Template format

```
SELECT CustomerID, CustomerName, City, Street, housenum
FROM Customers
WHERE CustomerID IN
        (SELECT a.CustomerID
        FROM Customers AS a INNER JOIN
        (SELECT Country, City, Street, houseno, count (*) AS cn
        FROM Customers
        GROUP BY Country, City, Street, housenum
        HAVING count (*) >1) AS b
        ON (a.Country = b.Country) AND
(a.City = b.City) AND  (a.Street = b.Street) AND (a.housenum = b.housenum))
        ORDER BY City, Street, housenum;
SELECT * FROM department WHERE deptno NOT IN( SELECT deptno FROM emp);
```

Figure 4. Sample query format for evaluation.

As shown in **Figure. 5**, first the red node is invoked.

Consider the following query:-

"SELECT CustomerID, CustomerName, City, Street, housenum

FROM Customers

WHERE CustomerID IN" ----------(1)

Query (1) is checked for any injection. If it is a valid query , then child nodes are checked. In this case, there are two child nodes. The first child node [Query (2)] is considered next.

"SELECT a.CustomerID

FROM Customers AS a INNER JOIN" ----------(2)

This is a recursive process and it will be continued until the node has no children. So the presence ofa subquery is checked in Query(2).

"SELECT Country, City, Street, housenum, count(*) AS cn FROM Customers

GROUP BY Country, City, Street, housenum

HAVING count(*) > 1" ----------(3)

Query (3) is a simple query, and has no children. Similarly all the nodes in the tree are processed. This Dagger detection process is repeated for all the trees in the forest.

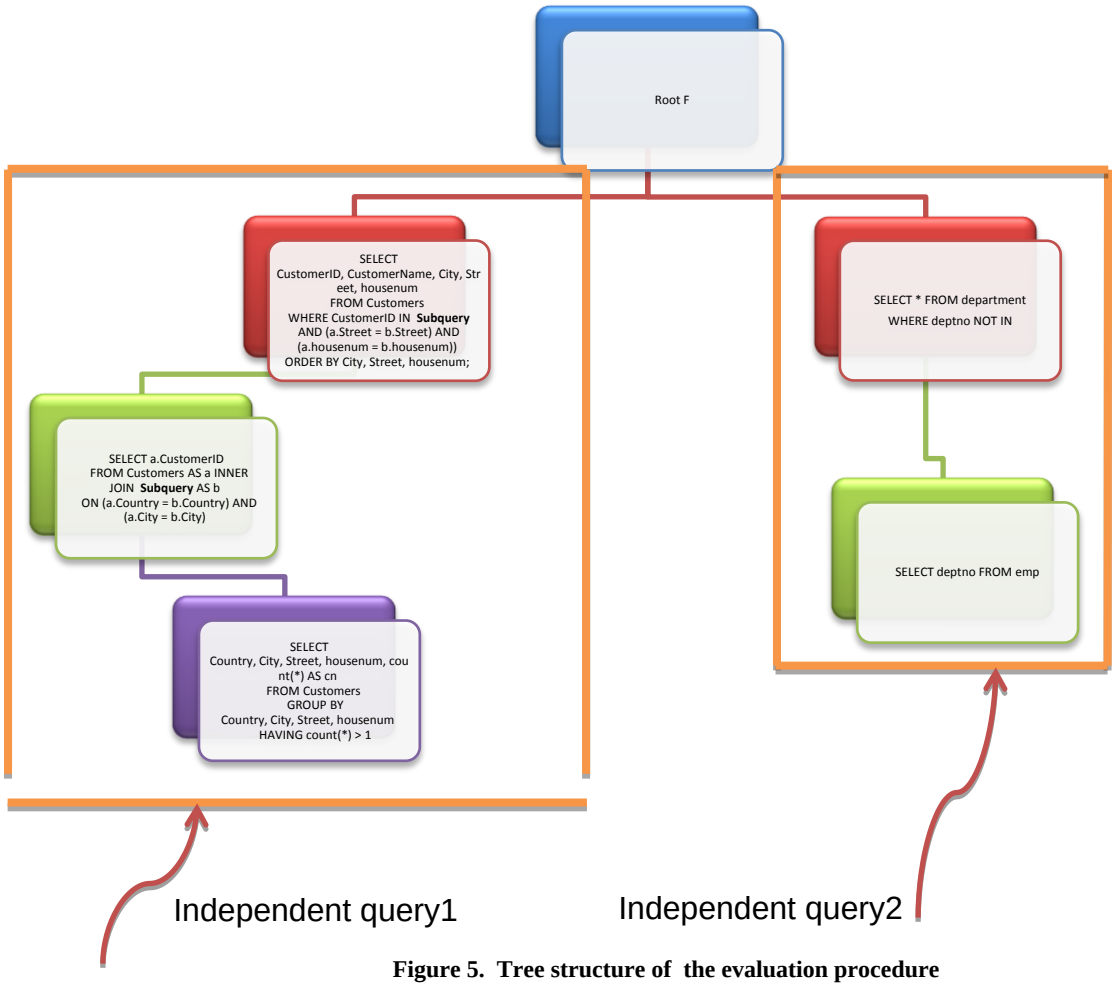

Figure 5. Tree structure of the evaluation procedure

6. CONCLUSION

There are many prevention and protection techniques available against SQL Injection vulnerabilities but still there are flaws and Injection attacks. Handling of SQL injection vulnerabilities effectively by a single tool is near to impossible due to the sophistication in the existing automated tools to penetrate in to the web application system for exploitation. The SQLI-Dagger algorithm, implemented using Java based application program is embedded on a proxy server. It detects the SQL injection vulnerabilities without any false positives. With the support of the template matching algorithm, the standard query identity for each legal query stored in the template repository of the application will be compared with the dynamic user queries entered through the web pages. As the Query templates placed in the template repository is in JSON format, accessing and parsing the query will be faster compared to the other available techniques and have lighter storage specification.

7. REFERENCES

[1] Su, Zhendong, and Gary Wassermann. "The essence of command injection attacks in web applications." *ACM SIGPLAN Notices*. Vol. 41. No. 1. ACM, 2006.

[2] Junjin, Mei. "An approach for SQL injection vulnerability detection."*Information Technology: New*

Generations,2009. ITNG'09. Sixth International Conference on. IEEE, 2009.

[3] Dharam, Ramya, and Sajjan G. Shiva. "Runtime monitoring technique to handle tautology based SQL injection attacks." *International Journal of Cyber-Security and Digital Forensics (IJCSDF)* 1.3 (2012): 189-203.

[4] Ruse, Michelle, Tanmoy Sarkar, and Samik Basu."Analysis & detection of SQL injection Vulnerabilities via automatic test case generation of programs." *Applications and the Internet*

[5] Kindy, Diallo Abdoulaye, and Al-Sakib Khan Pathan. "A detailed survey on various aspects of sql injection in web applications: Vulnerabilities, innovative attacks, and remedies." *arXiv preprint arXiv:1203.3324* (2012).

[6] Chapela, Victor. "Advanced SQL njection." *OWASP Foundation, Apr* (2005).

[7] Kemalis, Konstantinos, and Theodores Tzouramanis. "SQL-IDS: a specification-based approach for SQL-injection detection." *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008.

[8] Kosuga, Yuji, et al. "Sania: Syntactic and semantic analysis for automated testing against sql injection." *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007.

[9] Buehrer, Gregory, Bruce W. Weide, and Paolo AG Sivilotti. "Using parse tree validation to prevent SQL injection attacks." *Proceedings of the 5th international workshop on Software engineering and middleware*. ACM, 2005.

[10] Valeur, Fredrik, Darren Mutz, and Giovanni Vigna. "A learning-based approach to the detection of SQL attacks." *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Berlin Heidelberg, 2005.123-140.