Volume 3, Issue 6, June 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

SQL Injection Prevention Using Tokenization: Technique and Prevention Mechanism

Gaurav Shrivastava*, Kshitij Pathak
Department of Information Technology
Mahakal Institute of Technology, Ujjain, India

Abstract— SQL Injection Prevention Using Tokenization: A model exclusive of tokenization technique is used to prevent SQL Injection Attack by blocking the malicious input query in query execution phase. SQL Injection Prevention Using Tokenization Model detects SQLIA by applying tokenization process on input query. Tokenization process is applied by detecting spaces, single quotes and double dashes etc. This process converts the input query into the fruitful tokens and these tokens are then converted into hierarchical form. After applying tokenization, model validates each token by analyzing the value of left and right child of individual token. As soon as SQLIA detected it permanently block the input query. This model is seems to be able to detect and prevent all types of SQL Injection Attacks and does not trap in the case of appending set operators and Additional query attacks. It increases database security as well as contributes to maintain the confidentiality of sensitive data of web applications.

Keywords—: SQL Injection Prevention; SQLIA; Tokenization etc.

I. INTRODUCTION

Within the past decade, the growth of the Database industry and the Internet has revolutionized the way many people interact with information and databases. This rapid propagation and the cost effectiveness of new key technologies are creating large opportunities for developing large-scale distributed applications. However, this phenomenal growth has also brought about security concerns since some of the data now being made available on the Internet is sensitive. Web application's database often contains the crown jewels of an organization and they are subject to a wide range of attacks [1]. In recent time the use of web applications are increases rapidly. As soon as the services of Internet are rising; all web applications are depended on the Internet. Online banking, university, college admissions, online shopping, and various government e-portals are the examples of frequently used web applications. So, we can say that these activities are the key component of today's Internet Infrastructure. Database is the key of any web application. These rapid uses of web applications also receive the attention of attackers to steal the sensitive information from the databases. Database attacker appraises different ways to gain the knowledge of database. These different ways are known as database attacks. [2]. Table 1 shows the different database attacks as well as their corresponding percentages of adopting them. [3]

TABLE 1 ATTACK METHOD & PERCENTAGES

Attack method	Percentage
Denial of Service	25%
SQL Injection	24%
Cross Site Scripting (XSS)	8.9%
Brute Force	4.8%
Predictable Resource Location	3.8%
Stolen Credentials	3.7%
Unintentional Information Disclosure	3%
Banking Trojan	2.8%
Credential/Session Prediction	2.1%
Cross Site Request Forgery (CSRF)	1.9%

By analysing the facts of Table 1 Pie chart shown below can be drawn. Fig 1 shows that among all types of database attacks, SQL Injection is one of the most serious types of attack which is frequently used to detach the conscious information from the database. Hence, by analysing the facts and the figures a conclusion can be made that SQL Injection is one of the most serious type of attack which is frequently used to carry off the judicious information from the database. A survey held in 2010 shows web application vulnerabilities and SQL Injection attack ranked among top five[4].

© 2013, IJARCSSE All Rights Reserved

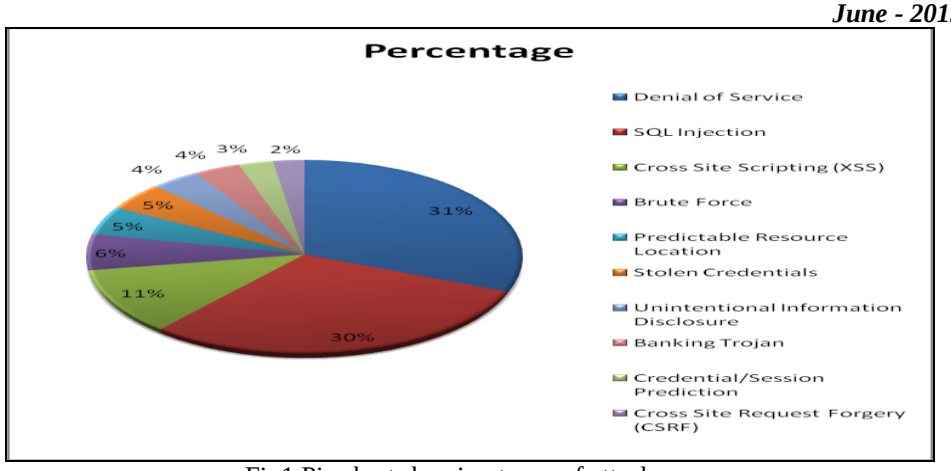


Fig1 Pie chart showing types of attacks

The term "SQL injection" dates back to 1998, while its first public use was in the year 2000 [5]. Since then SQLIA has become one of the most common attack employed over the internet [6]. It occurs when a malicious user modifies the semantic or syntax of a legitimate query by inserting new SQL keywords or operators consequently generating unexpected results not intended by web applications [7]. SQL injection is a type of code injection attack in which the attacker adds SQL code to input query. This query inserted into login box of a web form to gain access on database or make changes to data. SQL injection vulnerability allows an attacker to insert commands directly to web applications which can destroy functionality or confidentiality of database. SQL injection has become a almighty attacks that target web applications. It allows attackers to gain unauthorized access to the back-end database. [8] This type of attack is also known as SQLIA. Various approaches have been proposed in the field of detection and prevention of SQLIA. The major problem with the earlier work done is that it cannot horse-trade with UNION; UNION ALL etc set operators [9]. When a part of SQL commands contain such a set operators it cannot deal with such types of set operators [10]. To secure data from this type of attack SQL Injection Prevention Using Tokenization Model is proposed which block the malicious input query at entry point. The proposed solution aims to make this model secure and effective which emit out the fruitful results. The main purpose of this work is to prevent database from the most serious web vulnerabilities containing UNION, UNION ALL etc set operators. This solution completely based on query tokenization technique to prevent SQLIA.

II. SQL INJECTION PREVENTION USING TOKENIZATION

SQL Injection Prevention Using Tokenization Model is design and develops to prevent all types of SQL Injection Attacks. This model performs a strict checking against all types of SQLIA and does not trap in the case of UNION, UNION ALL set operator. SQL Injection prevention using tokenization model first of all tries to hit the bull's-eye WHERE clause in input query. If the input query contains WHERE clause so, there is a probability of having SQL Injection Attack because SQLIA cannot be possible without using WHERE clause [11]. If input query does not contain WHERE clause, definitely there is no SQL Injection Attack because malicious input can only be added after WHERE clause [40]. This model extracts WHERE clause from the input query then put the remaining portion in a temporary variable for further checking of SQLIA. After extracting WHERE clause this model apply first level of tokenization on the remaining portion (after where clause) of input query. In first level of tokenization, this model go through for "AND" or "OR" operator. Input query may contain several set operators so there may be several individual conditions can generate and for each individual condition, second level of tokenization is applied. Useful tokens are generated by performing second level of tokenization on each individual condition. These tokens are stored in a tree form data structure. Here validation of input query is performed by comparing the value of left and right child of each token. If the value of left and right child satisfies root condition, it means input query contains some malicious data and having SQLIA. If the value of left and right child does not satisfy root condition, it means current token is free from SQLIA. Above process is applied to each individual token. Whenever SQL Injection Attack is found an exception is raised and input query is blocked. When all the tokens passed the validation checking and no SQLIA found, this model passes the query from the database and display a massage success.

QUERY TOKENIZATION:

Query tokenization technique converts the input query into the useful tokens. These tokens are generated by detecting single quote, double dashes and space in an input query. All string before a single quote, before double dashes and before a space constitutes a token. Tokenization process executes in following four essential steps:

STEP I: PROCESS THE INPUT QUERY: Example of input query: Select*from Table where user_id = ''or1 = 1/--

STEP II: DETECT SINGLE QUOTE, DOUBLE DASHES AND SPACE IN INPUT QUERY: Here spaces, single quote and double dashes are detected to apply tokenization. Fig 2 shows how tokens are formed by detecting spaces single quote and double dashes in input query.

© 2013, IJARCSSE All Rights Reserved

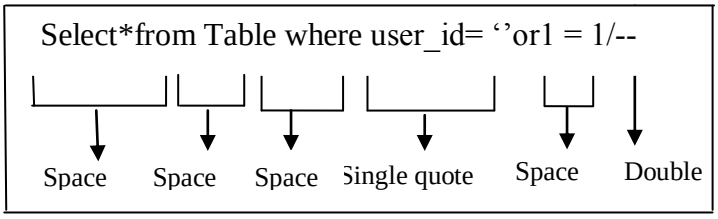


Fig. 2 Input query with spaces and single quote

STEP III: BREAK THE INPUT QUERY INTO USEFUL TOKENS: Tokenization will give following tokens: Token l: Select*from, Token2: table, Token3: where, Token4: user_id=, Token5:‘’ Token 6: or, Token 7: 1=1, Token 8: /--

STEP IV: STORE THE TOKENS IN A TREE TABLE: These tokens are stored in a tree form data structure. Table 2 shows this structure of token number 7 in a tree table.

TABLE 2 TREE TABLE

Root	Right child	Left child
=	1	1

III. PROPOSED MODEL

Figure 3 shows a block diagram of SQL Injection prevention using tokenization model. This diagram shows the actual functioning of each block and the flow of procedure to detect and prevent SQLIA.

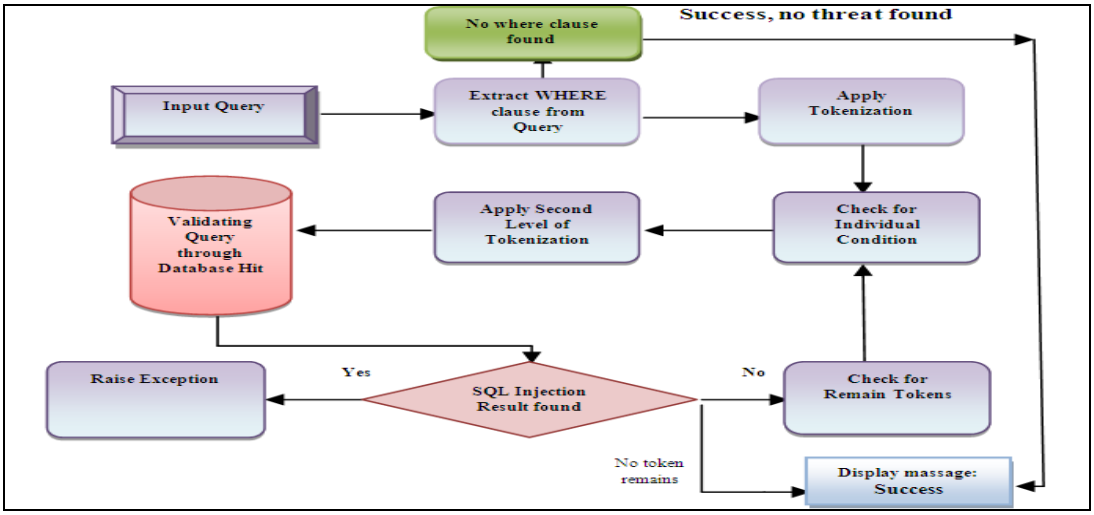


Fig.3 Block Diagram of Proposed Model

Over all working of this model is take place in five essential steps which are given below:

STEP 1: EXTRACT THE WHERE CLAUSE FROM THE INPUT: In a first step this model extract where clause from query and put remaining query (after where clause) in a temporary variable.

STEP 2: APPLY TOKENIZATION TO GENERATE THE USEFUL TOKENS: In this step this model performs tokenization process and convert the remaining query into useful tokens by detecting space, single quote and double dashes.

STEP 3: CONVERT THE TOKENS INTO HIERARCHICAL FORM: after performing tokenization process this model convert the tokens into hierarchical form. Fig. 4 shows pictorial representation of token in hierarchical form. Suppose Token is 2=2. This token converted in following manner

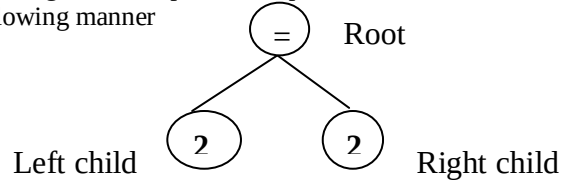


Fig.4 Hierarchical Structure

STEP 4: VALIDATE EACH TOKEN BY COMPARING THE VALUE OF LEFT AND RIGHT CHILD: In this step developed model performed validation of each token by comparing the value of left and right child to the root condition. Example: for the token 2=2 model performed checking whether 2 is equal to 2 or not.

© 2013, IJARCSSE All Rights Reserved

STEP 5: DETECT AND PREVENT SQLIA BY SHOWING THE STATUS FIELD VALUE "FAIL": This is the last and final step, in this step developed model detect the SQL Injection Attack by analyzing the value of O_STATUS field. If the value of left and right child of any token satisfied the root condition the background algorithm initialize O_STATUS FIELD BY 1 otherwise initialize O_STATUS FIELD BY 0. If the model finds that O_STATUS indicates 1, it immediately block the input query and set status field "FAIL" and display a massage in a massage field "SQL Injection found at that place". Otherwise set status field "SUCCESS" and display a massage "No threat Detected". It means O_STATUS indicates 0 and input query is injection free and passes it from database as well as.

IV. PROPOSED ALGORITHM

In this section, proposed algorithm for SQL Injection Prevention Using Tokenization Model is shown below. This is a customized algorithm based on the architecture of proposed model. The algorithm for SQL Injection Prevention Using Tokenization Model satisfies all necessary conditions step by step and result shows the practical implementation of this work.

ALGORITHM

//P Input Query ,X Query After Where Clause ,O_STATUS For Checking SQL Injection ,OP Operator, xi Subpart of X, Y Expression Check For Input Query P

STEP 1: CHECKING OF WHERE CLAUSE

If where clause found
Insert x into L_QUERY
Else
Display "there is no where clauses in input query"
GOTO STEP 6

STEP 2: CHECKING OF 'AND','OR' OPERATOR

If x contains ''AND'' or ''OR'' operator in OP
a) Break x after OP
b) SAVE value in Xi
Else GOTO STEP 3

STEP 3: TOKENIZATION PROCESS

Apply tokenization process on x
a) Replace ' ' by *
b) If multiple * found apply following
replacement procedure
Replace *** by SINGLE *
Replace ** by SINGLE *
Checking of first and last position of x
If * not found
Add * at first and last position
GO TO STEP 4

STEP 4: CONVERT INTO TREE

Find the infix expression of Y from L_QUERY
Store expression by breaking it into ROOT, LEFT_CONDITION & RIGHT_CONDITION
GO TO STEP 5

STEP 5: CHECKING OF SQLIA

Check the validity of expression using P_CHECK_HIT
If LEFT_CHILD = RIGHT_CHILD
Set O_STATUS =1
Else
Set O_STATUS = 0
If O_STATUS = 0
Display "SOL INJECTION FOUND AT THIS PLACE"
Else
Display "SUCCESS, NO THREAT FOUND"
Else
Display ''retrieving condition please insert correct query''

STEP 6: EXIT

V. EXPERIMENTAL VALIDATION

Table 3 shows a Query Set by which adversary can perform SQL Injection Attack. This Query Set contains 32 different queries which belong to different-different classes of SQLIA. These queries are collected from the different sources by which adversary tries to steal the sensitive information from the database. This Query Set is used to show the real working of this model.

© 2013, IJARCSSE All Rights Reserved

TABLE 3 QUERY SET

S.NO	QUERY SET	SQLIA TYPE
1	SELECT * FROM ENTITY_MASTER WHERE 4 in ("3","5",ent_id) and 1 = 1';	AND/OR/IN Attack
2	SELECT * FROM ENTITY_MASTER WHERE 4 in ("3","4",ent_id)';	
3	SELECT * FROM ENTITY_MASTER WHERE ENT_ID IN ("1","2",ent_id)';	
4	SELECT * FROM ENTITY_MASTER WHERE 1 NOT IN (2,3)';	
5	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 OR 1 != 2';	
6	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 OR 10 > 20 OR 1 = 1';	
7	SELECT * FROM ENTITY_MASTER WHERE ent_id LIKE ent_id';	LIKE Attack
8	SELECT * FROM ENTITY_MASTER WHERE "A" LIKE "A'";	
9	SELECT * FROM ENTITY_MASTER WHERE ent_id LIKE "%100%'";	
10	SELECT * FROM ENTITY_MASTER WHERE ent_id LIKE ent_id';	
11	SELECT * FROM ENTITY_MASTER WHERE "ABC" NOT LIKE "%DEF%'";	
12	SELECT * FROM ENTITY_MASTER WHERE "123" NOT LIKE "456'"	
13	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION SELECT * FROM ENTITY_MASTER WHERE 1 < 2'UNION ALL SELECT * FROM	Injected UNION/UNION ALL /INTERSECTION Attack
14	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION SELECT * FROM ENTITY_MASTER WHERE 1 >2';	
15	SELECT * FROM ENTITY_MASTER WHERE 10 > 2 UNION SELECT * FROM ENTITY_MASTER WHERE 1<=1';	
16	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION ALL SELECT * FROM ENTITY_MASTER WHERE 2 IN ("2","2","1");	
17	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION ALL SELECT * FROM ENTITY_MASTER WHERE 1 IN ("2","2","1");	
18	SELECT * FROM ENTITY_MASTER WHERE 1 <2 UNION SELECT * FROM ENTITY_MASTER WHERE 1 LIKE "%0%" OR 1 = 1';	
19	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION ALL SELECT * FROM ENTITY_MASTER WHERE 1 IN ("2","2","11") AND 20 = 20;	
20	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 INTERSECT SELECT * FROM ENTITY_MASTER WHERE 1 IN ("2","2","11") AND 25 = 25;	
21	SELECT * FROM ENTITY_MASTER WHERE 1 <2 MINUS SELECT * FROM ENTITY_MASTER WHERE 1 LIKE "%0%" OR 1 = 1';	
22	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 UNION ALL SELECT * FROM ENTITY_MASTER WHERE 1 IN ("2","2","1");	
23	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 INTERSECT SELECT * FROM ENTITY_MASTER WHERE 1 IN ("2","3","11") AND 2 = 2;	
24	SELECT * FROM ENTITY_MASTER WHERE 1 > 2 MINUS SELECT * FROM ENTITY_MASTER WHERE 1 LIKE "%0%" OR 100 = 100';	
25	SELECT * FROM ENTITY_MASTER WHERE 10 >= 10';	Bypass Authentication Attack
26	SELECT * FROM ENTITY_MASTER WHERE 1 <= 2';	
27	SELECT * FROM dual WHERE 1 = 1;	
28	SELECT * FROM dual WHERE A = A;	
29	SELECT * FROM ENTITY_MASTER WHERE ENT_ID = ENT_ID';	
30	SELECT * FROM ENTITY_MASTER WHERE "ABC" != "XYZ'";	
31	SELECT * FROM ENTITY_MASTER WHERE ENT_ID = ' ';SHUTDOWN;	Unauthorized Remote Execution Attack
32	SELECT * FROM ENTITY_MASTER WHERE ENT_ID = "100014";drop table ENTITY_MASTER	Injected Additional Query

Our Query Set consist of 32 queries and out of these 32 queries 6 queries belongs to Bypass Authentication Attack, 6 queries belongs to AND/OR/IN Attack, 12 queries belongs to UNION/UNION ALL/INTERSECTION Attacks, 6 queries belongs to LIKE Attack, 1 query belongs to Injected Additional Query Attack and one query belongs to Unauthorized execution of remote procedure Attack. Table 4. Shows the result of above hypothetical analysis. This table consists of three fields SQLIA types, occurrence of each type out of 32 query set and corresponding occurrence percentage in query set of each type.

TABLE 4 SQLIA OCCURRENCES IN QUERY SET

S.No.	SQLIA Type	Occurrence out of 32 sets	Corresponding % In Query Set
1	AND/OR/IN Attack	6	18.75%
2	LIKE Attack	6	18.75%
3	Injected UNION/UNION ALL Attack	12	37.50%
4	Bypass Authentication Attack	6	18.75%

© 2013, IJARCSSE All Rights Reserved

5	Unauthorized Remote Execution Attack	1	3.125%
6	Injected Additional Query Attack	1	3.125%

Pie chart corresponding above hypothetical analysis is shown in fig. 5 Pie chart shows that 37% of total attack is covered by Injected UNION / UNION ALL Attack, 19% of total attack is covered by LIKE Attack, 3% of total is covered by each Unauthorized Remote Execution Attack and Injected Additional Query Attack and 19% of total attack is covered by Bypass Authentication Attack. Hypothetical analysis demonstrates that among all the types of SQL Injection Attack, Injected UNION query attack is the first choice of adversary.

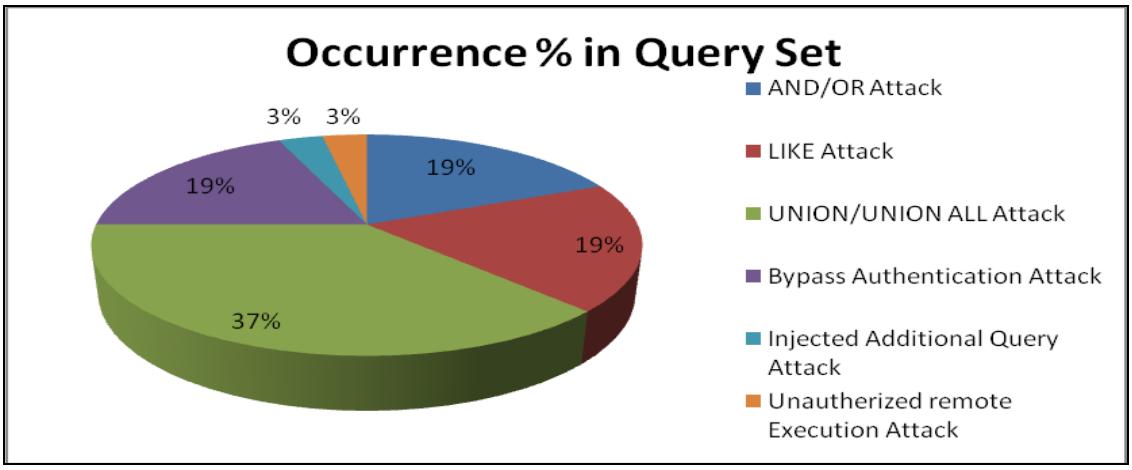


Fig.5 Pie chart showing % of attacks in Query Set

The above hypothetical analysis shoes that among all the types of SQL Injection Attack, Injected UNION query attack is the first choice of adversary. The strong reason to choose UNION, UNION ALL Attack by any adversary is that, set operators like UNION, UNION ALL, and INTERSECTION etc. are not trapped by most of the existing models. So query containing such type of set operators may pass easily from the database.

VI. EVALUATIONARY RESULTS

This model store the input query and tokens in to three types of tables i.e. Query Master , Query Detail and Tree table.

QUERY DETAIL TABLE: Query detail table store the input query by converting it into s.no, id and clause field. Fig 6 shows query detail view.

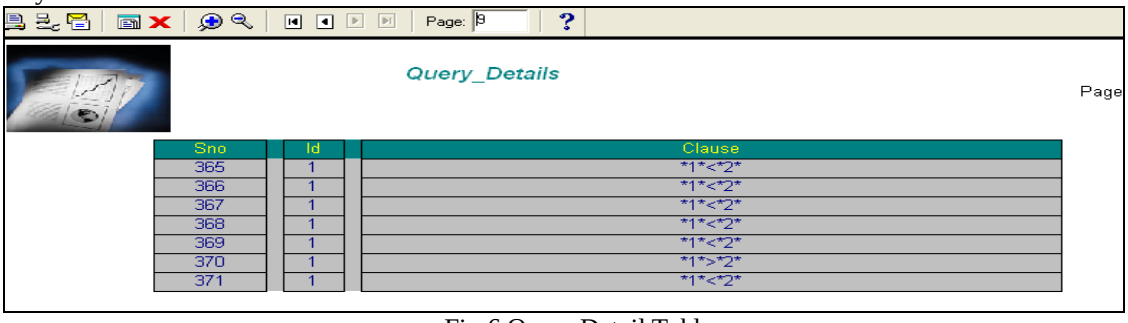


Fig.6 Query Detail Table

QUERY MASTER TABLE: Query master table store the input query in form of s.no and query. It store the remaining portion of the input query after where clause. Here spaces in between the token are also replaced by *. Fig.7 Shows query master.

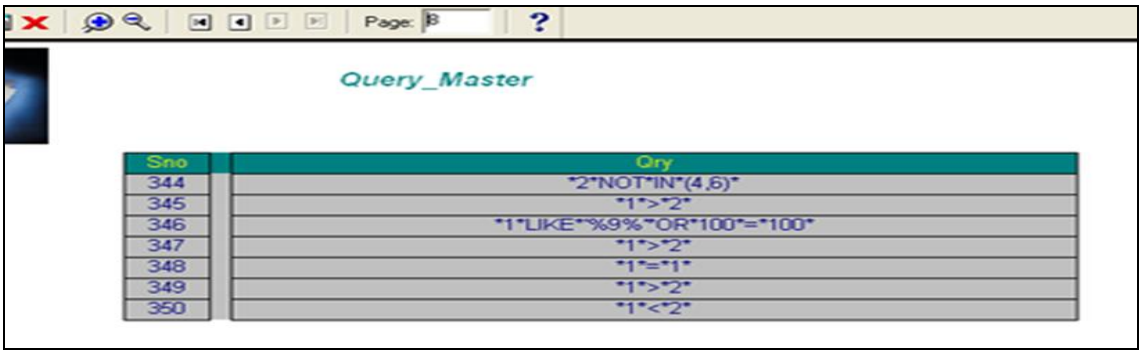
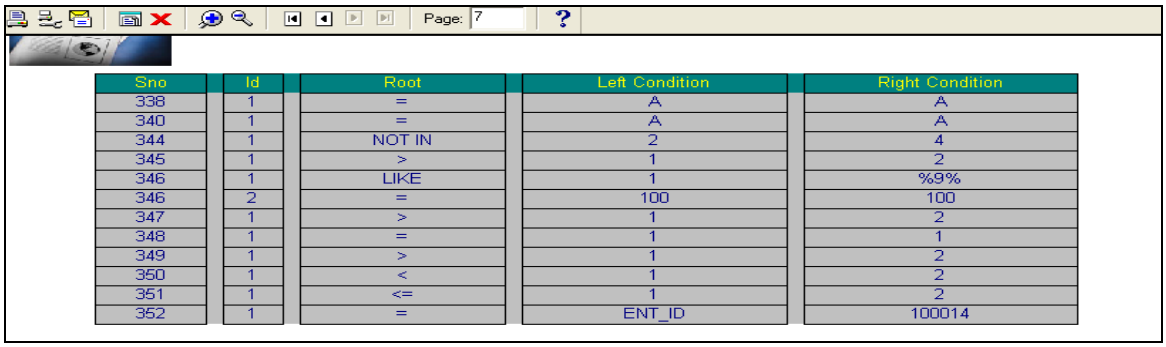


Fig.7 Query Master Table

© 2013, IJARCSSE All Rights Reserved

TREE TABLE: This is the most useful table of this model. Here tokens are stored by breaking them into left child, right child and root condition and also provided a unique id number. Fig 8 shows tree table view.



Sno	Id	Root	Left Condition	Right Condition
338	1	=	A	A
340	1	=	A	A
344	1	NOT IN	2	4
345	1	>	1	2
346	1	LIKE	1	%9%
346	2	=	100	100
347	1	>	1	2
348	1	=	1	1
349	1	>	1	2
350	1	<	1	2
351	1	<=	1	2
352	1	=	ENT_ID	100014

Fig.8 Tree Table

If input query contains SQL Injection Attack, status field shows query status FAIL and massage field display the massage "SQL Injection found at that place". Fig 4.5 shows validation process and its result. Fig 9 shows that a query containing UNION Attack is inserted into enter query field and after pressing validate button, developed model performed detection process and find SQL Injection Attack at 1<2. As soon as this model detect SQLIA it permanently block the input query and set the status field "FAIL" and display the massage "SQL Injection found at 1<2".

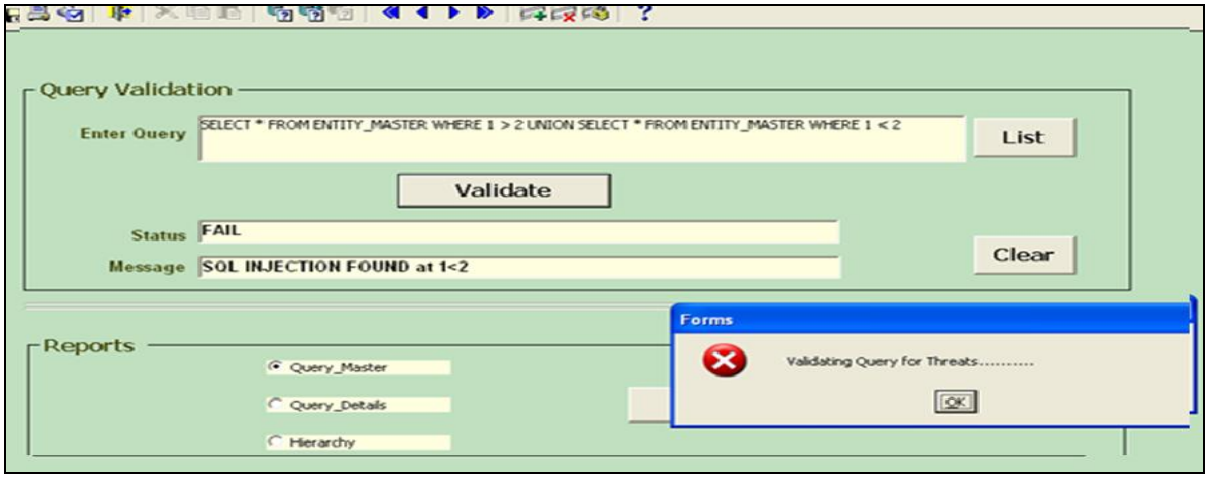


Fig.9 Validation Process

VII. COMPARATIVE ANALYSIS

Table below shows the comparative analysis between the existing SQL Injection Attack prevention models and SQL Injection Prevention Using Tokenization Model. This comparison is made on the basis of different types of SQL Injection Attacks like Bypass Authentication Attack, AND/OR/IN Attack, Unauthorized Remote Execution of Procedure Attack, Injected Additional Query Attack and Injected Union & Union ALL Query Attack. This comparative analysis also compares the strength of both existing and developed model to stand strongly against some or all types of SQLIA. Existing TransSQL model seems to be strong against SQLIA but it trapped in the case of Injected UNION Query Attack, which is one of the most powerful and prominent attack among all the type of SQLIA. Developed model eliminates the drawbacks of existing model and providing a strong protection shield against all types of SQL injection Attacks. Table 5 shows outcomes of existing model as well as SQL Injection Prevention Using Tokenization Model.

TABLE 5 COMPARATIVE ANALYSES BETWEEN EXISTING & DEVELOPED MODEL

SQL Injection Types	**SQLIA prevention technique**	
	Existing TransSQL [9] Model's Outcomes	**SQL Injection Prevention Using Tokenization Model's Outcomes**
1. Bypass Authentication	Prevented	Prevented
2. Unauthorized Knowledge of Database	Prevented	Prevented
3.Unauthorized Remote Execution of Procedure	Prevented	Prevented

© 2013, IJARCSSE All Rights Reserved

4.Injected Additional Query	Prevented	Prevented
5. Injected Union & Union ALL Query	Not Prevented	Prevented

Table 6 shows the comparative analysis between the existing TransSQL[9] Model and developed SQL Injection Prevention Using Tokenization Model. This comparison is made on the basis of different types of SQL Injection Attacks and their corresponding occurrence probability as well as their prevention probability by both models.

TABLE 6 COMPARATIVE ANALYSES ON THE BASIS OF PREVENTION PROBABILITY

TYPES OF SQLIA	**TransSQL Model [9]**		**SQL Injection Prevention Using Tokenization Model**	
	Occurrence Probability	**Prevention Probability**	**Occurrence Probability**	**Prevention Probability**
Bypass Authentication Attack	19%	19%	19%	19%
AND/OR/IN Attack	19%	19%	19%	19%
LIKE Attack	19%	19%	19%	19%
Unauthorized Remote Execution	3%	3%	3%	3%
Injected Additional Query Attack	3%	3%	3%	3%
Injected Union & Union ALL Query Attack	37%	0%	37%	37%

Fig 10 shows the experimental analysis chart which indicates that the prevention probability of Injected UNION/UNION ALL Attack by TransSQL Model is very low. On the other hand all types of SQL Injection Attacks can be prevented efficiently by SQL Injection Prevention Using Tokenization Model.

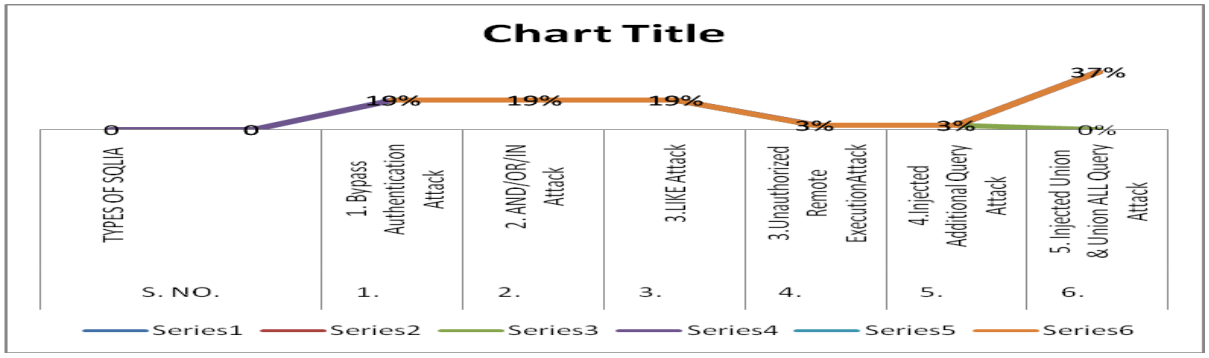


Fig.10 Graph showing results

VIII. CONCLUSION AND FUTURE WORK

SQL Injection Attack is one of the most powerful web application vulnerability of present time. In modern age most of the task is performed through web applications, it becomes important to develop a tool which can not only prevent SQL Injection Attack but can also show its strength to detect and prevent different kinds of SQLIA. Many models have been proposed in this direction and all they have their own advantages and drawbacks. This research work proposed a novel approach for prevention of SQL Injection Attack by using tokenization technique. Proposed approach prevents SQL Injection Attack by analysing the input query at entry level. Proposed model evaluate input query by applying tokenization on it and convert the input query in to fruitful tokens. After performing token generation process, model evaluate each token individually to find whether the token contain SQL Injection Attack or not. If SQLIA detected, proposed model immediately block the input query otherwise input query passed successfully. By testing proposed model on various Query Sets, a conclusion can be made that SQL Injection Prevention Using Tokenization Model is performing well against all type of SQL Injection Attack and does not trap in the case of appending set operators and Additional query attacks. In future the same method can be extended by adding, different levels of authentication within the same application.

ACKNOWLEDGMENT

I WANT TO EXPRESS A SINCERE ACKNOWLEDGEMENT TO **PROF. KSHITIJ PATHAK (**HEAD OF INFORMATION TECHNOLOGY DEPARTMENT, M.I.T., UJJAIN **)** FOR PROVIDING ALL THE NECESSARY REQUIREMENTS, MORAL SUPPORT AND FOR HIS VALUABLE GUIDANCE. HIS FRUITFUL SUGGESTIONS, VALUABLE COMMENTS AND SUPPORT WERE OF IMMENSE HELP FOR CONTINUATION THIS WORK.

© 2013, IJARCSSE All Rights Reserved

References

[1] Joshi, J., Aref, W., Ghafoor, A., and Spafford, E. "*Security Models for Web-based Applications"* Communications of the ACM, Feb. 2001.

[2] V. Nithya,R.Regan,J.vijayaraghavan.,"*A Survey on SQL Injection attacks, their Detection and Prevention Techniques*". International Journal Of Engineering And Computer Science ISSN:2319-7242 Volume 2 Issue 4 Page No. 886-905. April, 2013.

[3] "The *Web Hacking Incident Database Semiannual (WHID) Report from July to December* 2010", Trust wave Holdings, Inc, 2010"

[4] Danchev, Dancho (2007-01-23). "Mind Streams of Information Security Knowledge: Social Engineering and Malware". *Ddanchev.blogspot.com. Retrieved* 2011-06-03.

[5] M.Kiani, A. Clark, and G.Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks". *The Third International Conference on Availability*, Reliability, and Security, IEEE Computer Society, 2008.

[6] Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. "A static *analysis framework for detecting sql injection vulnerabilities*". In Proceedings of the 31st Annual International Computer Software and Applications Conference – Volume 01, COMPSAC '07, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.

[7] S. Friedl. "*Sql injection attacks by example*, 2005.

[8] S Ali, A. Rauf, and H. Javed. "*Sqlipa: An authentication mechanism against sql injection*". European Journal of Scientific Research, 38(4):604–611, 2009.

[9] Kai-Xiang Zhang, Chia-Jun Lin, Shih-Jen Chen, Yanling Hwang, Hao-Lun Huang, and Fu-Hau Hsu, "*Trans SQL: A Translation and Validation-based Solution for SQL-Injection Attacks*", First International Conference on Robot, Vision and Signal Processing, IEEE, 2011.

[10] Gaurav shrivastava, Kshitij pathak, "*SQL Injection Attacks: Technique and Prevention Mechanism*" International Journal of Computer Applications Digital Library ISSN: 0957-8887, may 2013.

[11] 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), 2006.

[12] Liang Guangmin, "*Modeling Unknown Web Attacks in Network Anomaly Detection*" Computer Engineering Department Shenzhen Polytechnic, Shenzhen 518055, China Email: gmliang@oa.szpt.net Third 2008 International Conference on Convergence and Hybrid Information Technology.,2008.

© 2013, IJARCSSE All Rights Reserved