A Robust Method for Prevention of Second Order and Stored Procedure based SQL Injections

Anju Muraleedharan
M.Tech Student, Dept. of Computer Science
Adi Shankara Institute of Engg and Tech, Kalady, India

Neetha K N
Asst.Professor, Dept. of Computer Science
Adi Shankara Institute of Engg and Tech, Kalady, India

ABSTRACT
Today's interconnected computer network is complex and is constantly growing in size . As per OWASP Top10 list 2013[1] the top vulnerability in web application is listed as injection attack. SQL injection[2] is the most dangerous attack among injection attacks. Most of the available techniques provide an incomplete solution. While attacking using SQL injection attacker probably use space, single quotes or double dashes in his input so as to change the indented meaning of the runtime query generated based on these inputs. Stored procedure based and second order SQL injection are two types of SQL injection that are difficult to detect and hence difficult to prevent. This work concentrates on Stored procedure based and second order SQL injection. It uses a Similarity analysis technique to detect injection. The runtime generated query is checked against a query model for similarity analysis and if both are similar then the runtime query is free from injection else query is vulnerable and the further processing of the query is blocked.

Keywords
SQL Injection, Web application, Stored procedure, Second order injection.

1. INTRODUCTION
Over the past decade WEB-BASED services and applications have increased in both popularity and complexity. Daily tasks such as banking, shopping, bill payment, travel, social networking etc. are all done through web. Due to their wide use for personal and/or corporate data, attackers are attracted towards it. To compromise a database, SQL injection is one of the techniques used by attackers. It allows attackers to insert SQL characters or keywords into a SQL statement via unrestricted user input parameters to change the intended query's logic, so that attackers can obtain unauthorized access to a database.

2. SQL INJECTION
Structured Query Language (SQL) is a textual language which is used to communicate with relational Database. SQL statements can be used to modify the structure of databases and manipulate its contents by using various commands. The typical unit of execution of SQL is the 'query', which is a collection of statements that returns a single 'result set'. SQL injection is a technique that exploits a vulnerability that occurs in the database layer of an application. It happens when user input is either filtered incorrectly or is not strongly typed and thereby unexpectedly executed. In short it occurs when the data provided by user is not properly validated and is directly used to generate runtime SQL query. Thus an attacker is able to submit SQL commands that can directly access database. SQLIA compromises the confidentiality and integrity of user's sensitive data. [2,3].

Login page is common point of attack for attackers. On that page legitimate user is filling form with his username and password in order to gain access in his secure area with personal details.

Code for checking the username and password from database is as follows : **SELECT * from users WHERE name=' ' and password= ' ';** Instead of providing genuine Username, attacker uses the following code to manipulate the original query. [3,5,8] **' or '1' ='1'--'**

Now the meaning of the manipulated query will be **SELECT * FROM users WHERE name = ''or '1'='1'—'' and password='null';** The term, ' or '1'='1' --' does two things. First, it causes the first term in the SQL statement to be true for all rows of the query; second, the -- causes the rest of the statement to be treated as a comment and therefore ignored from further processing. As a result all details in the database from table users up to the limit the web page can list, are returned.

Attack techniques are the ways in which an attacker carries out attacks using malicious code. Various SQL injection attacks techniques [3,5,7,12] includes Tautology, Piggy-backed query, Logically Incorrect/Illegal query, Union query, Stored Procedure, Inference Attack, Alternate Encodings and Second Order Injection.

3. RELATED WORK
Related works [12] on SQL injection can be classified into two major divisions : Injection detection and Injection prevention. Former aims at identifying vulnerable locations in the application . It includes all types of input validation and filtering techniques to detect injection attempts. Techniques proposed in [13,14,15] explains some static analysis techniques for injection detection. Main disadvantage of using these techniques is its limited accuracy in identifying potentially not validated inputs. They lacks way to check the correctness of the input validation techniques, and programs using incomplete input validation techniques may pass these checks and cause SQL injection. Prevention techniques provide a way to prevent SQL injection. Techniques proposed in [16,17,18] explains some of the available prevention mechanisms. They can act as a first level of defense against the attack, but they cannot defend against sophisticated attack techniques (Stored procedure based injection, Second order injection) that inject malicious inputs into SQL queries.

Stored procedure based attack is a type of SQL injection that try to execute stored procedures stored inside the database. A stored procedure is a set of codes directly stored inside database. Typically, stored procedures are written in SQL. As stored procedures are stored on the server side, they are always available to all clients. Stored procedure is capable of accepting input parameters and a single procedure can be used by several clients using different data. Main issue in preventing injection in stored procedure is that it is difficult to extract the runtime query from stored procedure as it is directly stored inside the database. Earlier stored procedures where considered as a solution for SQLIAs. But they are also vulnerable to injection attacks.

Table1: Existing Methods for stored procedure

Technique	***Prevention mechanism***
Preventing SQL Injection Attacks in Stored Procedures	Static analysis and runtime validation
Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks	Positive tainting
SQLStor	Dynamic query structure validation

Second order injection is a type of SQL injection in which the attacker submits some crafted input in the request. The application then stores this input for future use (usually in the database), and responds to the request. The attacker then submits a second (different) request. To process the second request, the application calls the stored input and processes it, causing the attacker's injected SQL query to execute.

The below scenario provides an example for second order injection.Consider an update password screen. It will ask the user for providing these information: username, old password and new password. The SQL query for updating the password in this application is as follows:

UPDATE user set password='newpass' where name='admin' and password='oldpass';

Instead of a normal user an attacker is trying to update the password. First attacker will create a dummy user as **admin'--** After that attacker will try to update the password of this newly created user. Then the background query will become,

UPDATE user set password='newpass' where name='admin'-- ' and password='pass';

Here the crafted username leads to injection. Everything after -- is ignored by the SQL engine and the query get reduced to **UPDATE user setpassword='newpass' where name='admin';** Thus the password of the user admin can be updated successfully without even knowing the actual password that is the old password of the user.

4. PROPOSED METHOD

This work offers a technique, dynamic similarity analysis[11], that validates programmer-intended query structures at each SQL query location, thus providing a simple and efficient solution to the problem. The idea requires that the application will not allow the user to enter any part of SQL query directly. Two statements are said to be similar, if they perform similar activities, once they are executed on the database server. So if it can be determined that both Runtime Query and Reference Query are similar, then by definition the Runtime Query is bound to have an expected behaviour and there is no possibility of SQL injection. Here similarity implies a particular activity like comparison, retrieval etc and not the lexical equality. The proposed technique for SQL injection detection can be explained as follows: First the runtime query and user inputs are extracted from the application. User inputs are then passed on to a pre-processing function which pre-process them. This is done with the help of a regular expression . The regular expression consists of the blacklisted characters in SQL that may lead to injection. User inputs are filtered using this regular expression. Then a reference query is generated from the runtime query. This is done by replacing the user inputs with safe inputs. Next step is to generate a parse tree of both runtime and reference query model. Parse tree is a tree like structure which specifies the syntax of the query. It has the entire query as root and its components as leaves. Once both trees are generated they are compared to check whether both are similar.

This work mainly focuses on injection through stored procedure and second order injection as it is difficult to detect and prevent them. Most of the available techniques are not able to provide a solution for these issues. Main issue in preventing injection in stored procedure is that it is difficult to extract the runtime query from stored procedure as it is directly stored inside the database. And the main issue with prevention of second order injection is that point of attack is different from point of injection. Thus it is very difficult to detect second order injection and hence to prevent.

This technique can be applied successfully to all kinds of injections discussed above. The first order injections types prevented by similarity analysis includes Tautology, Piggy-backed query, Logically Incorrect/Illegal query, Union query, and Inference Attack.

Extension to prevent Second order injection: As explained earlier second order injections are difficult to prevent as the point of injection is different from point of attack. Hence more care should be taken in order to detect and prevent the same. Both attack points should be validated carefully. Point of injection as well as point of attack are checked using similarity analysis technique in order to prevent second order injection.

Extension to prevent injection through stored procedure: As discussed the main issue in preventing injection through stored procedure is that it is difficult to extract the runtime query from stored procedure as it is directly stored inside the database.

A sample stored procedure is given below

```
DELIMITER $$
USE `demo`$$
DROP PROCEDURE IF EXISTS `LoginChk`$$
CREATE DEFINER= `root`@`localhost`
PROCEDURE `LoginChk` (IN uname
VARCHAR(20), IN passwrd VARCHAR(20))
BEGIN
SET @aaa=CONCAT('select * from user where
name=',uname,' ',' and password=',passwrd);
        PREPARE stmt FROM @aaa;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END$$
DELIMITER ;
```

Here, the procedure name is 'LoginChk' with two input arguments, uname and passwrd. According to the inputs given by users, the query will be formed as a string and executed through 'EXECUTE' statement.

Now, the way of calling this procedure from the web page is as follows:

1. String uname = request.getParameter("username");
2. String pwd = request.getParameter("password");
3. CallableStatement calstat = con.prepareCall("{call LoginChk(?,?)}");
4. calstat.setString(1, uname);
5. calstat.setString(2, pwd);
6. ResultSet rs = calstat.executeQuery();

First two statements are for accepting input arguments. The third statement will create an object of 'CallableStatement' for calling stored procedure. The next two statements will set the values of the arguments of the stored procedure. The last statement will execute and produce required result.

The mechanism of SQLIA is same for both application layer program and stored procedure, but the same detection technique will not work for stored procedures, because of its limited programmability. SQL injection attack is possible by injecting specially crafted user inputs to the stored procedure. For prevention, the method proposed in this work is dynamic similarity analysis. For doing that query structure being formed within the procedure is required. It is very difficult to get the query structure out of the stored procedure for similarity analysis. In order to obtain the query structure, an additional procedure is constructed , which is similar to the one being considered, but, with one additional output argument 'qry' for getting the dynamic query structure . This runtime is then passed on to the similarity analyser and checked for any vulnerability. This technique provides a two stage checking for the detection of SQL injection. At the first stage the user inputs are checked for any blacklisted characters . In the second stage, the runtime generated queries are validated using similarity analysis. Sample procedure that returns the query structure is given below.

```
DELIMITER $$

USE `demo`$$

DROP PROCEDURE IF EXISTS `LoginChk1`$$

CREATE DEFINER=`root`@`localhost` PROCEDURE
`LoginChk1`(IN uname VARCHAR(20), IN passwrd
VARCHAR(20),OUT qry TEXT)

BEGIN

SET @aaa=CONCAT('select * from login where
id=',uname,' ',' and pass=',passwrd);

SET qry=@aaa;

END$$

DELIMITER ;
```

Thus we have extracted the runtime query. Newly developed stored procedure will return the runtime query for validation. This query is then passed on to the similarity analyser for detailed analysis. It will process the query in 5 steps and block the query in case of any issues.

An automated technique is being used for checking the presence of SQL injection. It has two sections. Purpose of the first section is to identify vulnerable points in the application. These are the points from which SQL queries are passed on to the database for execution. Once the vulnerable points are identified, all user inputs that pass through this points are monitored. The second part works on the SQL query generated using these user inputs and checks the query using above explained similarity analysis technique.

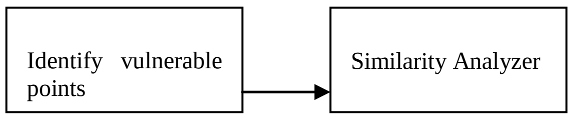

Fig 1: Proposed system

5. RESULTS AND DISCUSSION

The proposed solution was developed using Java as front end and MySQL as backend. Application server used was Apache Tomcat. Proposed technique was tested using test suite obtained from an independent research group [11], AMNESIA test bed developed by University of Southern California. This test bed provides a set of web applications developed by third party that are vulnerable to SQL Injection Attacks. It also includes safe as well as vulnerable set of test inputs. The purpose of the test bed is to evaluate detection and prevention techniques. The test bed consists of a set of applications . It includes seven web applications that accept inputs from user and use this input to generate queries to an underlying database which will lead to SQL injection.

Our attack list contained attacks from the AMNESIA test bed, which includes both attack and non-attack inputs for each application. Attack inputs were based on different vectors of SQL code injections. Overall, the attack suite contained 30 different attack string patterns (such as tautology-based attacks, UNION SELECT–based attacks, that were constructed based on real attacks obtained from sources US/CERT and CERT/CC advisories).

For testing the proposed method the application selected from test suite was Book Store.

Table 2: Applications from the AMNESIA Test Suite

Application	LOC	Servlets	SCL
Employee Dir	5658	10	23
BookStore	16959	28	71
Events	7242	13	31
Classifieds	10949	14	34
Portal	16453	28	67

The column SCL reports the number of SQL Command Locations, which issue either a sql.executeQuery (mainly SELECT statements) and sql.executeUpdate (consisting of INSERT, UPDATE or DELETE statements) to the database. Also two sets of URLs(Total: 3520) is used for testing, one set with attack URLs(3026) and other set with legitimate URLs(494). Test results can be summarized in a table as follows:

Table 3: Test Results

	Bookstore-Without Prevention	**Bookstore-With Prevention**	**Bookstore-With Prevention(Stored Proc)**
Total URLs	3520	3520	3520
Valid URL requests	3159	3159	3159
SQLIA detected	0	3026	3026
Undetected	3026	0	0
Error URL requests	361	361	361

AMNESIA test suite was unable to replicate any type of second order injection and hence second order SQL injection was tested using the standard patterns obtained from OWASP and other online sites. As part of testing 50 attack patterns were tested. Out of this 20 patterns were attack patterns and the proposed method was able to prevent all these patterns from accessing the underlying database.

Table 4: Test Results - Second order injection

	Application without prevention	**Application with prevention**
Total input patterns	50	50
Valid patterns	25	25
SQLIA detected	0	20
SQLIA Undetected	20	0
Syntax Errors	5	5

Thus the proposed method provides a robust technique to prevent all types of SQL Injection - Tautology, Piggy-backed query, Logically Incorrect/Illegal query, Union query, Stored Procedure, Inference Attack, Alternate Encodings and Second Order Injection.

6. CONCLUSION

SQL injection is one of the dangerous vulnerability in web application that can lead to loss of confidentiality, integrity and authentication. Existing techniques available for preventing SQL injection uses a combination of static as well as dynamic methods. The proposed method provide a novel dynamic methodology for detecting and preventing SQL injection. Similarity analyzer is a simple but efficient technique to detect and prevent SQL injection.

As a future enhancement the technique can be extended so as to detect blind injection, which is another form of SQL injection attack. The technique can also be automated by developing a tool which accepts application code and identify the vulnerable points in it and updates code to prevent injection. In order to validate application source code, byte code form will be needed. Main steps for identification of injection includes converting the source code into byte code format and analyze this byte code for any vulnerabilities. When any vulnerable points are identified, it automatically updates the code so as to prevent any form of injection. As the tool automatically identifies and prevents vulnerability, it will be very useful for the end user.

7. REFERENCES

[1] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[2] C. Anley. (more) Advanced SQL Injection. White paper, Next Generation Security Software Ltd., 2002.
[3] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.
[4] https://www.owasp.org/index.php/What_are_web_applications%3F
[5] W.G.J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures,” Proc. Int’l Symp. Secure Software Eng. (ISSSE 06), IEEE CS, 2006; www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas. orso.ISSSE06.pdf.
[6] Ke Wei, M. Muthuprasanna and Suraj Kothari (Iowa State University). ‘Preventing SQL Injection Attacks in Stored Procedures’ .Software engineering conference 2006.
[7] Justin Clarke, “SQL Injection Attacks ” 2nd Edition,2012.
[8] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios College of Computing – Georgia Institute of Technology.' Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks', 2006 ACM
[9] S Mamadhan, T Manesh, V Paul, SQLStor: Blockage of stored procedure SQL injection attack using dynamic query structure validation, IEEE ,Nov 2012
[10] Sandeep Nair Narayanan, Alwyn Roshan Pais, & Radhesh Mohandas. Detection and Prevention of SQL Injection Attacks using Semantic Equivalence. Springer 2011.
[11] http://www-bcf.usc.edu/~halfond/testbed.html
[12] Lwin Khin Shar and Hee Beng Kuan Tan, Defeating SQL Injection, IEEE Computer Society, March 2013
[13] Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM .Etienne Janot, Pavol Zavarsky Concordia University College of Alberta, Department of Information Systems Security
[14] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. In USENIX Security Symposium (2006).
[15] Mcclure, R. A. and Kr¨Uger, I.H. 2005. SQL DOM: Compile time checking of dynamic SQL statements.In Proceedings of the 27th International Conference on Software Engineering (ICSE’05).ACM, New York, 88–96.
[16] Boyd, S. W., and Keromytis, A. D. Sqlrand: Preventing sql injection attacks. In ACNS (2004), pp. 292–302.
[17] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In ASE (2005), pp. 174–183.
[18] Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. Using parse tree validation to prevent sql injection attacks. In SEM (2005)