A METHOD OF DETECTING SQL INJECTION ATTACK TO SECURE WEB APPLICATIONS

Sruthy Manmadhan[1] and Manesh T[2]
[1]Department of Computer Science & Engineering, Adi Shankara Institute of Engineering & Technology, Kalady
sruthym.88@gmail.com
[2]Department of Information Technology, Adi Shankara Institute of Engineering & Technology, Kalady
maneshpadmayil@gmail.com

ABSTRACT

Web applications are becoming an important part of our daily life. So attacks against them also increases rapidly. Of these attacks, a major role is held by SQL injection attacks (SQLIA). This paper proposes a new method for preventing SQL injection attacks in JSP web applications. The basic idea is to check before execution, the intended structure of the SQL query. For this we use semantic comparison. Our focus is on stored procedure attack in which query will be formed within the database itself and so difficult to extract that query structure for validation. Also this attack is less considered in the literature.

KEYWORDS

Arraylist, Attack, Parse Tree, Semantics, SQL injection, Web application.

1. INTRODUCTION

Nowadays, for most of the activities in our life, we depend on internet or web applications. There exists a natural trend that as the usage of a particular service increases; the attacker's interest on it also increases. The same thing happened in case of web applications. Of many kinds of attacks against web applications, SQL Injection Attack (SQLIA) is one of the top most threats against them[12]. So it is highly requires in the current scenario to have a good solution to prevent such attack to secure the information. This is the motivation behind this work.

SQL Injection targets the web applications that use a back end database. Working of a typical web application is as follows: User is giving request through web browsers, which may be some parameters like username, password, account number etc. These are then passed to the web application program where some dynamic SQL queries are generated to retrieve required data from the back end database.

SQL Injection attack is launched through specially crafted user inputs. That is attackers are allowed to give requests as normal users. Then they intentionally create some bad input patterns which are passed to the web application code. If the application is vulnerable to SQLIA, then this specially created input will change the intended structure of the SQL query that is being executed on the back end database and will affect the security of information stored in the database. The tendency to change the query structure is the most characteristics feature of SQLIA which is being used for its prevention also.

For better understanding let us have look at the following example. We all know that most of the applications that we are accessing through internet will have a login page to authenticate the user who is using the application. Figure 1 show such a login page. Here when a user is submitting his username and password, an SQL query is generated in the back end to check whether the given credentials are valid or not. Suppose the given username is 1 and password is 111, the query will be:

*Select * from login where user='1' and pass='111'*

This is the normal case and if any rows are selected by the query, the user is allowed to log in.

Now, figure 2 shows an attack scenario. That is an attacker wants to log in without correct username and password. Instead of entering valid username if he uses injection string like "hacker' OR '1'='1'—" as username and "something" as password, the query formed will be like this:

*Select * from login where user='hacker' or '1'='1' –' and pass='something'*

When this query is executed in the database, it will always return a true and the authentication will succeed.

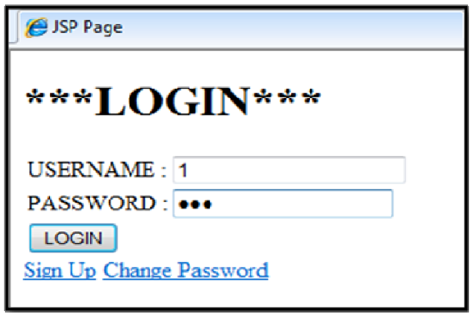

Figure 1. Example login – Normal case

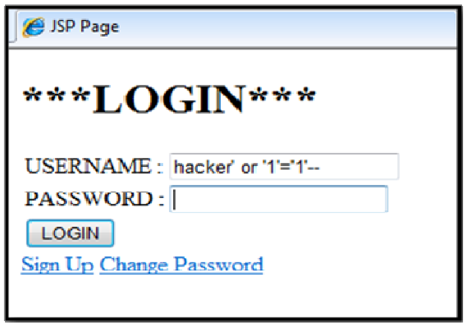

Figure 2 : Example login – attack case

2. LITERATURE SURVEY

The following formatting rules must be followed strictly. This (.doc) document may be used as a template for papers prepared using Microsoft Word. Papers not conforming to these requirements may not be published in the conference proceedings.

2.1. SQLIA Types

The SQLIA can be broadly classified into two: *first order* and *second order* attacks. First of these will have direct effect on the system whereas other doesn't have any direct harm.

Different types of first order attacks are listed below[1]:

Tautologies: The main intention of this attack is to bypass authentication. For this they attack the field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table to be returned so that he can login successfully without having a valid username and password. The attack shown in figure 2 is an example of tautology attack.

Illegal/Incorrect Queries: This is the first step of SQL injection attack. Here the intention of the attacker is to gather information about the type and structure of the back end database that is being used in the web application. This attack exploits very descriptive default error pages returned by the application servers.

Union Queries: This type of attack is mainly used to bypass authentication and to extract data by changing the data set returned for a given query. Format is 'UNION SELECT <part of injected query>', where the query after the UNION keyword is fully under control of the attacker so that he/she can retrieve data from any table which is not intended by the actual query.

Piggybacked Queries: This attack mainly aims at extracting data. Like the concept of piggybacked acknowledgement in computer networks where, acknowledgement of a packet is sent along with the next packet, here, the attacker tries to inject additional queries with original one.

Stored procedure Attack: This type of attack tries to execute stored procedures present in the database with malicious inputs. This is explained in next section.

Inference: Main aim of this kind of attack is to identify injectable parameters. The information can be inferred from the behavior of the page by asking the server true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally functioning page.

2.2. Related Works

Research on SQL injection attacks can be broadly classified into two basic categories: *vulnerability identification* approaches and *attack prevention* approaches. The former category consists of techniques that identify vulnerable locations in a Web application that may lead to SQL injection attacks. In order to avoid SQL injection attacks, a programmer often subjects all inputs to input validation and filtering routines that detects attempts to inject SQL commands. The techniques presented in [3, 4, 13] represent the prominent static analysis techniques for vulnerability identification, where code is analyzed to ensure that every piece of input is subject to an input validation check before being incorporated into a query (blocks of code that validate input are manually annotated by the user). While these static analysis approaches scale well and detect vulnerabilities, their use in addressing the SQL injection problem is limited to merely identifying potentially unvalidated inputs. The tools do not provide any way to check the correctness of the input validation routines, and programs using incomplete input validation routines may indeed pass these checks and cause SQL injection attacks.

Another approach to solve the problem is provided by the class of attack prevention techniques that *retrofit* programs to shield them against SQL injection attacks [5, 6, 7, 8, 9, 10, 11]. These techniques often require little manual annotation, and instead of detecting vulnerabilities in programs, they offer preventive mechanisms that solve the problem of defending the Web application against SQL injection attacks. Relying on input validation routines as the sole mechanism for SQL injection defense is problematic. Although they can serve as a first level of defense, they cannot defend against sophisticated attack techniques (e.g., those that use alternate encodings and database commands to dynamically construct strings) that inject malicious inputs into SQL queries.

A more fundamental technique to solve the problem of preventing SQL injection comes from the commercial database world in the form of PREPARE statements. These statements, originally created for the purpose of making SQL queries more efficient, have an important security benefit. They allow a programmer to declare (and finalize) the structure of every SQL query in the application. Once issued, these statements do not allow malformed inputs to influence the SQL query structure, thereby avoiding SQL injection vulnerabilities altogether. The following statement.

```
SELECT * FROM phonebook WHERE username = ? AND password = ?
```

is an example of a PREPARE statement. The question marks in the statement are used as "place-holders" for user inputs during query parsing and, therefore, ensure that these possibly malicious inputs are prevented from influencing the structure of the SQL statement. Thus, PREPARE statements allow a programmer to easily isolate and confine the "data" portions of the SQL query from its "code." Thus, PREPARE statements are in fact a robust and effective mechanism to defend against SQL injection attacks. However, retrofitting an application to make use of PREPARE statements requires manual effort in specifying the intended query at every query point, and the effort required is proportional to the complexity of the Web application.

Table 1. Comparison of related works.

Technique	**Tautology**	**Illegal**	**Piggy Back**	**Union**	**Stored Proced-ure**	**Inference**	**Alterna te encodin g**
SQL-DOM	*	*	*	*	X	*	*
SQLrand	*	X	*	*	X	*	X
AMNESIA	*	*	*	*	X	*	*
Tainting	*	*	*	*	*	*	*
SQLCheck	*	*	*	*	X	*	*
SQLGuard	*	*	*	*	X	*	*
CANDID	*	p	p	p	X	p	p

*-Prevention
p-Partial prevention
X-Prevention not possible

From this comparison, it is clear that stored procedure attacks are less considered in the literature. This paper focuses on this particular kind of attacks along with general prevention.

3. PROPOSED METHOD

This paper offers a technique, *dynamic query structure validation*, that automatically (and dynamically) mines programmer-intended query structures at each SQL query location, thus providing a robust solution to the retrofitting problem.

The idea is that the process of generation of queries in a dynamic web application can be represented as a function of user's inputs[2]. In this context, SQL injection is any situation in which the user's input is inducing an unexpected change in the output generated by the function.

Two parameters can be defined

Original_Query = Fun(input_i) i = 1 to n

input_i = input from user

Fun() = Function represented by web application

Benign_Query =Fun(input_benign_i) I = 1 to n

input_benign _i = "qqq" or any evidently non-attacking input

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 semantically equivalent, if they perform similar activities, once they are executed on the database server. So if it can be determined that both Original_Query and Benign_Query are semantically equivalent, then there is no possibility of SQL injection. This paper uses this semantic comparison to detect SQL injection. The semantic comparison is done by parsing each of the statements and comparing the syntax tree structure. If the syntax trees of both the queries are equivalent, then the queries are inducing equivalent semantic actions on the database server, since the semantic actions are determined by the structure of the Original_Query.

Steps include:

1. Generate a *Benign_Query* from the *Original_Query* generated by the application. This is done by replacing user inputs to the query with benign inputs.
2. Check the syntax of the *Benign_Query* to ensure its validity while doing the replacement.
3. Get the count of stacked queries in both original SQL query and generated *Benign_Query*.
4. Compare the count of stacked queries. If both counts are different, then we can directly report SQL injection attack and prevent that query from execution without going for semantic checking.
5. Now construct a syntax tree of both *Original_Query* and *Benign_Query* and compare them. Here, syntax trees are created using java *ArrayList* structure.
6. Compare the syntax trees. If they are equal, the query is valid and allow its execution. Otherwise, report injection and block the query.

These steps can be explained using an example: Consider a web application with two text boxes and a submit button. Let the text boxes be uid, and pwd. Consider the input from the user as "hacker' OR 1 = 1 –", and "something". Here the Original_Query generated from the web application is

*Original_Query = SELECT * FROM User WHERE UserName='hacker' OR 1 = 1 --' AND Password='Something'*

Here first the user inputs in the order "hacker' OR 1 = 1 –" and "something" will be replaced to produce the statement as shown below.

*SQL_Statement_Safe = SELECT * FROM User WHERE UserName='qqq' AND Password='qqq'*

Then, the syntax trees are created and compared. The syntax tree for the *Original_Query* using ArrayList will look like*:*

```
[select, [VAR, *],
    from,
    [VAR, login],
    where,
    [VAR, uname=qqq, AND, pwd=qqq]]
```

Now, the tree for *Benign_Query* generated will be look like:

```
[select, [VAR, *],
    from,
    [VAR, login],
    where,
    [VAR, uname=admin', OR, '1'='1'--, AND, pwd=somethng]]
```

While comparison we can identify that the tree structures are different and so it is an SQL Injection attack. So we prevent its actual execution.

3.1. Extension To Prevent Stored Procedure Attack

Stored procedures are an important part of relational databases. They add an extra layer of abstraction into the design of a software system. This extra layer hides some design secrets from the potentially malicious users, such as definitions of tables. By using stored procedures, one could make sure that all the data is always contained in the database and is never exposed. In these databases, the developer is allowed to build dynamic SQL queries ie. SQL statements are built at runtime according to the different user inputs. For example, in SQL Server, EXEC(varchar(n) @SQL) could execute arbitrary SQL statements. This feature offers flexibility to construct SQL statements according to different requirements, but faces a potential threat from SQL Injection Attacks.

Consider an example MySQL Stored procedure for Login.

```
DELIMITER $$

USE `sqlstor`$$

DROP PROCEDURE IF EXISTS `LoginCheckNew1`$$

CREATE DEFINER=`root`@`localhost` PROCEDURE `LoginCheckNew1`(IN uname VARCHAR(20),IN passwrd
VARCHAR(20))

BEGIN

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

        PREPARE stmt FROM @aaa;

        EXECUTE stmt;

        DEALLOCATE PREPARE stmt;

END$$

DELIMITER ;
```

Here, the procedure name is ‘LoginCheckNew1’ with two input arguments, uname and password. 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 LoginCheckNew1(?,?)}");
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 three arguments of the stored procedure. The last statement will execute and give the result.

The SQL injection attack is possible by injecting specially crafted user inputs to the stored procedure. For prevention, the method proposed in this paper is dynamic semantic equivalence checking. For doing that the query structure that is being formed within the procedure is required. But, in case of stored procedures, getting query structure before actual execution is difficult. To manage this, we are constructing one additional procedure which is similar to the one being considered, but, with one additional output argument ‘qry’ for getting the dynamic query structure which is required for semantic equivalence checking.

```
DELIMITER $$

USE `sqlstor`$$

DROP PROCEDURE IF EXISTS `LoginCheckNew1`$$

CREATE DEFINER=`root`@`localhost` PROCEDURE `LoginCheckNew1`(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 ;
```

For prevention, first execute this procedure with original arguments. Then the 'qry' variable will give the dynamic query structure that is being generated. For example, if the inputs given are ''1' or '1'='1'—' for uname and '' for password, then the result will be:

qry = *select * from login where id='1' or '1'='1'-- and pass=*

Now pass the original inputs and this query string to the above explained attack detection algorithm.

3.2. Test Results

For testing I used the test suite obtained from an independent research group, AMNESIA test bed[14]. It consists of some medium to large web applications. From that I selected one application, 'BookStore'.

Also two sets of URLs(Total: 3191) is used for testing, one set with attack URLs(3063) and other set with legitimate URLs(128).

Test results can be summarized in a table as follows:

Table 2. Test Results

	Bookstore- Without Prevention	**Bookstore With Prevention**	**Bookstore- With Prevention (Stored Proc)**
Total URLs	3191	3191	3191
Valid URL Requests	2901	2901	2901
SQLIA Detected	0	2777	2777
Undetected	2810	0	0
Syntax Errors	0	60	60
Others	91	64	64
Redirects	0	0	0
Error URL Requests	290	290	290
Omitted	0	0	0
Time	413s	327s	313

4. CONCLUSION

SQL injection vulnerability is one of the top vulnerabilities present in the web applications. In this paper we proposed an efficient approach to prevent this vulnerability. Our solution is based on the principle of dynamic query structure validation which is done through checking query's semantics. It detects SQL injection by generating a benign query from the final SQL query generated by the application and the inputs from the users and then comparing the semantics of safe query and the SQL query. The main focus is on stored procedure attacks in which getting query structure before actual execution is difficult.

REFERENCES

[1] Halfond, W., Viegas, J., & Orso, A. (2006). "Classification of SQLInjection Attacks and Countermeasures." *SSSE 2006.*

[2] Sandeep Nair Narayanan, Alwyn Roshan Pais, & Radhesh Mohandas. Detection and Prevention of SQL Injection Attacks using Semantic Equivalence. Springer 2011

[3] 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

[4] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. *In USENIX Security Symposium* (2006).

[5] Boyd, S. W., and Keromytis, A. D. Sqlrand: Preventing sql injection attacks. In *ACNS* (2004), pp. 292–302.

[6] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. *In ASE* (2005), pp. 174–183.

[7] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. *In SEC* (2005), pp. 295–308.

[8] Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. Using parse tree validation to prevent sql injection attacks. In SEM (2005).

[9] Prithvi Bisht, P. Madhusudan, V. N. VENKATAKRISHNAN. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACMTransactions on Information and System Security,*Vol. 13, No. 2, Article 14, Publication date: February 2010.

[10] Ke Wei, M. Muthuprasanna, Suraj Kothari. Preventing SQL Injection Attacks in Stored Procedures. *IEEE Software Engineering Conference, 2006. Australian.*

[11] Pietraszek, T. Berghe, C. V. 2006. Defending against injection attacks through context sensitive string evaluation. In *Proceedings of the Conference on Recent Advances in Intrusion Detection.* Springer, Berlin, 124–145.

[12] OWASP, O.W.(2010). OWASP Top 10 for 2010. *Category: OWASP Top Ten Project* http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (Apr. 14, 2011).

[13] 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.

[14] William G. J. Halfond, SQL Injection Application Testbed.

http://www- bcf.usc.edu/~halfond/testbed.html

Authors

Sruthy Manmadhan received B.Tech degree in Computer Science & Engineering from Adi Shankara Institute of Engineering and Technology, Mahatma Gandhi University with first rank in 2010.She is now doing her M.Tech at ASIET.

Manesh T received M.Tech degree in Computer Science & Engineering from NIT Suratkal. He is now working as Assistant Professor at Adi Shankara Institute of Engineering and Technology, Kalady.