Tribhuvan University

Institute of Science & Technology

Central Department of computer Science & information Technology

Kirtipur, Kathmandu

A Literature Review Report

On

Web Application Security by SQL Injection using Parse Tree Validation

Submitted By:

Ganesh B. Khatri

Roll no: 04

Submitted To:

CDCSIT

Kirtipur, Kathmandu

Abstract

An SQL injection attack targets interactive web applications that employ database services. Such applications accept user input, such as form fields, and then include this input in database requests, typically SQL statements. In SQL injection, the attacker provides user input that results in a different database request than was intended by the application programmer. That is, the interpretation of the user input as part of a larger SQL statement, results in an SQL statement of a different form than originally intended. The technique described here is a technique to prevent this kind of manipulation and hence eliminate SQL injection vulnerabilities. The technique is based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. The solution is efficient and reduces overhead to database query costs. In addition, it is easily adopted by application programmers, having the same syntactic structure as current popular record set retrieval methods.

Keywords: SQL Injection, SQL Parsing, Parse Tree, Dynamic Queries.

Central Department of Computer Science and IT

Tribhuwan University

Letter of Approval

This is to certify that this literature review report is prepared by Mr. **Ganesh B. Khatri** in partial fulfillment of the requirement for the degree of MSc.CSIT II year III semester. In our opinion, it is satisfactory in the scope and quality for the required semester.

……..……
Assist. Prof. Mr. Navraj Poudel
Head Of Department
CDCSIT, TU, Kirtipur
HOD

…….
Assist Prof Mrs. Lalita Sthapit
CDCSIT, TU, Kirtipur
Supervisor

……
Internal Examiner

1. INTRODUCTION

Web applications are often vulnerable to attacks, which can give attackers easily access to the application's underlying database. SQL injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer originally intended. SQL Injection Attacks (SQLIAs) have known as one of the most common threats to the security of database-driven applications. So, there is not enough assurance for confidentiality and integrity of this information. SQLIA is a class of code injection attacks that takes advantage of lack of user input validation. In fact, attackers can shape their illegitimate input as parts of final query string which accesses with database. Financial web applications or secret information systems could be the victims of this vulnerability because attackers, by abusing this vulnerability, can threat their authority, integrity and confidentiality. So, developers address some defensive coding practices to eliminate this vulnerability but they are not sufficient. For preventing the SQLIAs, defensive coding has been offered as a solution but it is very difficult. Not only developers try to put some controls in their source code but also attackers continue to bring some new ways to bypass these controls. Hence, it is difficult to keep developers up to date, according to the last and the best defensive coding practices. On the other hand, implementation of best practice of defensive coding is very difficult and needs special skills. These problems motivate the need for a solution to the SQL injection problem.

Researchers have proposed some tools to help developers to compensate the shortcoming of the defensive coding [1, 2, 3]. The problem is that some current tools could not address all attack types or some of them need special deployment requirements.

2. LITERATURE REVIEW

SQL injection has been the focus of a flurry of activity over the past 3 years. Although the security vulnerability has persisted for some time, recent efforts by hackers to automate the discovery of susceptible sites have given rise to increased invention by the research community [4]. The industrial community has also gone to lengths to make programmers aware and provide best practices to minimize the problem [5].

However, a recent study showed that over 75% of web attacks are at the application level and a test of 300 web sites showed 97% were vulnerable to web application attacks [6]. Still, It is believed that the problem of SQL injection is readily solvable.

It has similarities to buffer overflow security challenges, because the user input extends its position in a query [7]. The heart of the issue is the challenge of verifying that the user has not altered the syntax of the query. As such, much of the work casts the problem as one of user input validation, and focuses on analyzing string inputs [8]. Several technologies exist to aid programmers with validating input. A simple technique is to check for single quotes and dashes, and escape them manually. This is easily beaten, as users can simply adjust their input for the escaped characters. Programmers must then check for pre-escaped text, etc.

Boyd and Keremytis [4] developed SQLRand, a technique that modifies the tokens of the SQL language: Each token type includes a prepended integer. Any additional SQL supplied by the user, such as OR 1=1, would not match the augmented SQL tokens, and would throw an error. This approach is a useful strategy, and can effectively eliminate SQL injection. From a practicality stand-point, the programmer must generate an interface between the database tier and the middle tier which can generate and accommodate the new tokens. This is not a trivial endeavor. Secondly, and more importantly, these new tokens are static. As pointed out by the authors, many applications export SQL errors [9], and as such, the new tokens would be exposed. External knowledge of the new tokens compromises the usefulness of the technique.

3. DEFINITION OF SQLIA

Most web applications today use a multi-tier design, usually with three tiers: a presentation, a processing and a data tier. The presentation tier is the HTTP web interface, the application tier implements the software functionality, and the data tier keeps data structures and answers to requests from the application tier [10]. Meanwhile, large companies developing SQL-based database management systems rely heavily on hardware to ensure the desired performance [11].SQL injection is a type of attack which the attacker adds Structured Query Language code to input box of a web form to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to a web application's underlying database and destroy functionality or confidentiality.

3.1 SQL Injection Attack Process

SQLIA is a hacking technique which the attacker adds SQL statements through a web application's input fields or hidden parameters to access to resources. Lack of input validation in web applications causes hacker to be successful. For the following example, assume that a web application receives a HTTP request from a client as input and generates a SQL statement as output for the back end database server.

For example, an administrator will be authenticated after typing: username=admin and password=secret. Figure1 describes a login by a malicious user exploiting SQL Injection vulnerability [12]. Basically it is structured in three phases:

i) An attacker sends the malicious HTTP request to the web application

ii) Creates the SQL statement

iii) Submits the SQL statement to the back end database

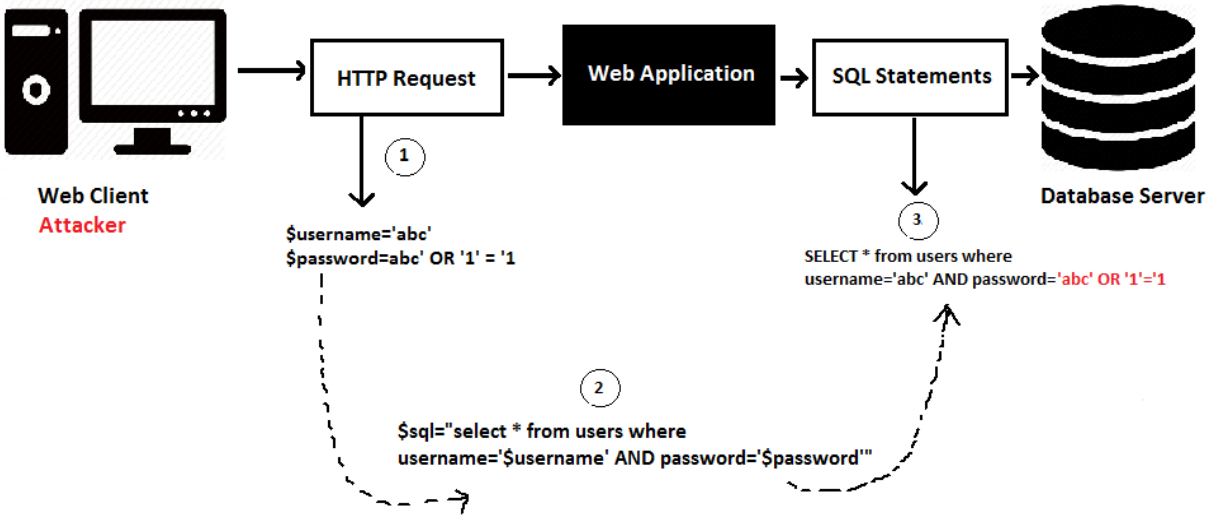

Figure 1: Example of a SQL Injection Attack

The above SQL statement is always true because of the Boolean tautology appended (OR 1=1) so, it will access to the web application as an administrator without knowing the right password.

3.2 Main Cause of SQL Injection

Web application vulnerabilities are the main causes of any kind of attack [13]. In this section, vulnerabilities that might exist naturally in web applications and can be exploited by SQL injection attacks will be presented:

3.2.1 Invalidated input

This is almost the most common vulnerability on performing a SQLIA. There are some parameters in web application which are used in SQL queries. If there is no any checking for them, then they can be abused in SQL injection attacks. These parameters may contain SQL keywords, e.g. INSERT, UPDATE or SQL control characters such as quotation marks and semicolons.

3.2.2 Generous privileges

Normally in database, the privileges are defined as the rules to state which database subject has access to which object and what operation are associated with user to be allowed to perform on the objects. Typical privileges include allowing execution of actions, e.g. SELECT, INSERT, UPDATE, DELETE, DROP, on certain objects. Web applications open database connections using the specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the accounts. The number of available attack methods and affected objects increases when more privileges are given to the account. The worst case happens if an account can connect to system that is associated with the system administrator because normally it has all privileges.

3.2.3 Error message

Error messages that are generated by the back-end database or other server-side programs may be returned to the client-side and presented in the web browser. These messages are not only useful during development for debugging purposes but also increase the risks to the application. Attackers can analyze these messages to gather information about database or script structure in order to construct their attack.

3.2.4 Variable Orphism

The variable should not accept any data type because attacker can exploit this feature and store malicious data inside that variable rather than is supposed to be. Such variables are either of weak

type, e.g. variables in PHP, or are automatically converted from one type to another by the remote database.

3.2.5 Client-side only control

If input validation is implemented in client-side scripts only, then security functions of those scripts can be overridden using cross-site scripting. Therefore, attackers can bypass input validation and send invalidated input to the server-side.

3.2.6 Stored procedures

They are statements which are stored in DBs. The main problem with using these procedures is that an attacker may be able to execute them and damage database as well as the operating system and even other network components.

3.2.7 Multiple statements

If the database supports UNION so, attacker has more chance because there are more attack methods for SQL injection. For instance, an additional INSERT statement could be added after a SELECT statement, causing two different queries to be executed. If this is performed in a login form, the attacker may add him or herself to the table of users.

3.3 SQL Injection Attack Types

There are different methods of attacks depending on the goal of attacker. For a successful SQLIA, the attacker should append a syntactically correct command to the original SQL query. Now the following classification of SQLIAs [14, 15] will be presented.

3.3.1 Tautologies

This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack is used to bypass authentication control and access to data by exploiting vulnerable input field which uses WHERE clause.

“SELECT * FROM users WHERE username = 'abc' and password ='abc' OR '1'='1’”

As the tautology statement (1=1) has been added to the query statement, so it is always true.

3.3.2 Illegal/Logically Incorrect Queries

When a query is rejected, an error message is returned from the database including useful debugging information. This error message helps an attacker to find vulnerable parameters in the application and consequently database of the application. In fact, attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose.

3.3.3 Union Query

By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application. Suppose for our examples that the query executed from the server is the following:

SELECT username, password FROM users WHERE username=’admin’ and password=’secret’

By injecting the following in password field:

1’ UNION SELECT * FROM users_test where '1'='1

New query will be

SELECT username, password FROM users WHERE username=’admin’ and password=’1’ UNION SELECT * FROM users_test where '1'='1' which will join the result of the original query with all the records from table users_test.

3.3.4 Piggy-backed Queries

In this type of attack, intruders exploit database by the query delimiter, such as ";", to append extra query to the original query. With a successful attack, database receives and executes multiple distinct queries. Normally the first query is legitimate query, whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker injects "delete from users_test where '1'='1" into the password input field instead of logical value. Then the application would produce the query:

select * from users where username='admin' and password='1';delete from users_test where '1'='1'.

Because of ";" character, database accepts both queries and executes them. The second query is illegitimate and can delete records from users_test table from the database. It is noticeable that some databases do not need special separation character in multiple distinct queries, so for detecting this type of attack, scanning for a special character is not impressive solution.

3.3.5 Stored Procedure

Stored procedure is a part of database that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, so, this part is as injectable as web application forms. Depending on specific stored procedure on the database, there are different ways to attack.

3.3.6 Alternate Encodings

In this technique, attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII, and Unicode. By this way, they can escape from developer's filter which scan input queries for special known "bad character". For example attacker uses char (44) instead of single quote that is a bad character. This technique with join to other attack techniques could be strong, because it can target different layers in the application so developers need to be familiar to all of them to provide an effective defensive coding to prevent the alternate encoding attacks. By this technique, different attacks could be hidden in alternate encodings successfully.

4. SQL PARSE TREE VALIDATION TECHNIQUE

A parse tree is a data structure for the parsed representation of a statement. Parsing a statement requires the grammar of the language that the statement was written in. By parsing two statements and comparing their tree structures, it can determine if the two queries are equal. When a malicious user successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query do not match. By intended SQL query, it means that when a programmer writes code to query the database, he/she has a formulation of the structure of the query. The programmer-supplied portion is the hard-coded portion of the parse tree, and the user-supplied portion is represented as empty leaf nodes in the parse tree. These nodes represent empty literals. What a developer intends is for the user to assign values to these leaf nodes. These leaf nodes can only represent one node in the resulting query, it must be the value of a literal, and it must be in the position where the holder was located.

For an example, SELECT * FROM users WHERE username=? AND password=?. The question marks are place holders for the leaf nodes the developer requires the user to provide. While many programs tend to be several hundred or thousand lines of code, SQL (structured query language)

statements are often quite small. This affords the opportunity to parse a query without adding significant overhead [16].

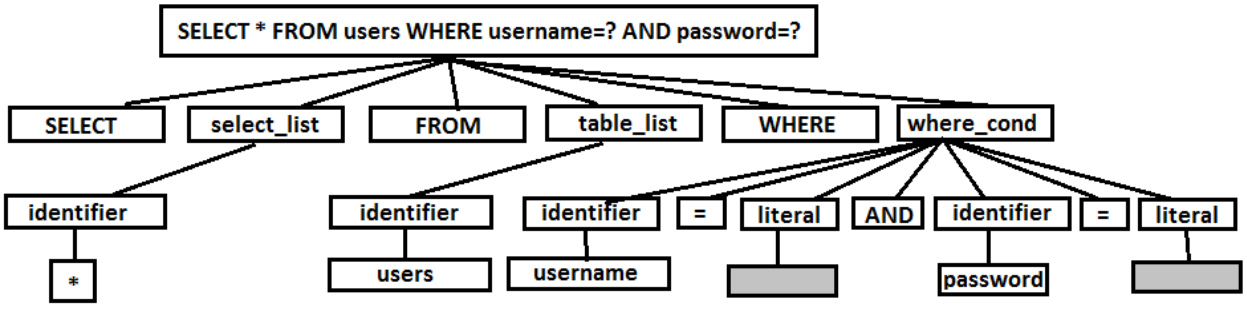

Figure 2. A SELECT query with two user inputs.

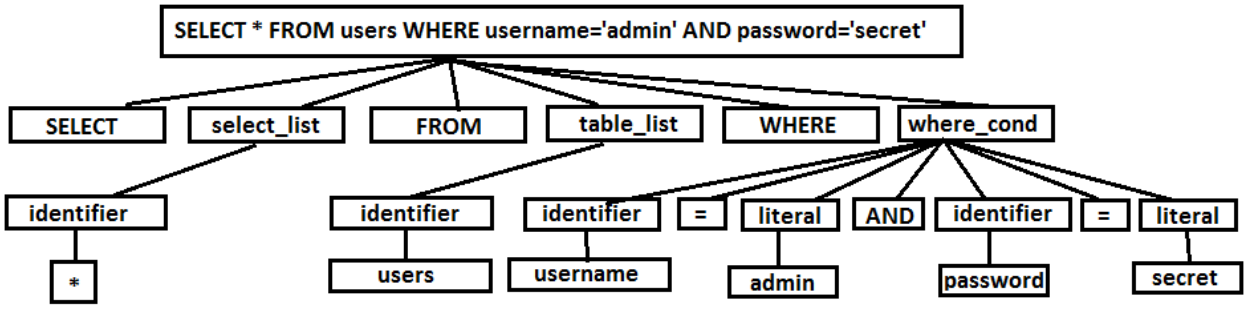

Figure. 3. The SELECT query with the user input inserted.

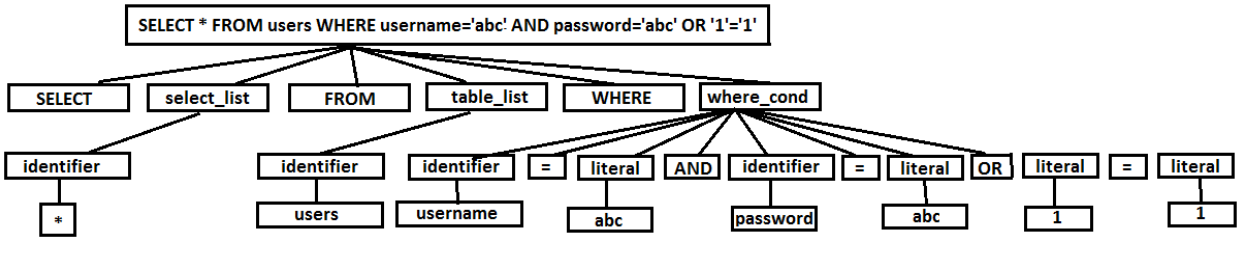

Figure. 4. The SELECT query with a tautology inserted.

4.1 Dynamic Queries

One significant advantage over static methods is that exact structure of the intended SQL query can be evaluated. Many times, this structure itself is a function of user input or depends on some runtime calculated values like dates, which does not permit static methods to determine its structure.

For an example, suppose a query retrieves all the items from a live database which are ordered between certain ranges of order dates supplied by the user. The query is of the form

```
function get_lod_columns($lods){
	$sql="SELECT distinct mt.item_barcode as 'Item Code', '' as RRP,";
	for($i=0;$i<count($lods);$i++){
		$l=$lods[$i]; $sql.=" sum(ht$i.ic$i) as '$l'";
		if($i!=count($lods)-1) $sql.=",";
	}
	$sql.=" from (SELECT item_id,item_barcode from tbl_item) as mt";
	for($i=0;$i<count($lods);$i++){
		$l=$lods[$i];
		$sql.=" LEFT JOIN (
		SELECT item_barcode,sum(rel_oi_quantity) as ic$i
		FROM tbl_item
		INNER JOIN tbl_item_offer ON tbl_item.item_id=tbl_item_offer.io_item_id
		INNER JOIN tbl_rel_order_item ON
	tbl_item_offer.io_id=tbl_rel_order_item.rel_oi_item_id
		INNER JOIN tbl_order ON tbl_order.order_id=tbl_rel_order_item.rel_oi_order_id
		WHERE date_format(subtime(timestamp(rel_df_date),maketime(recipient_dtc*24,0,0)),
	'%Y-%m-
		%d')='$l'
		GROUP BY item_barcode
		) as ht$i on mt.item_barcode=ht$i.item_barcode";
	}
	$sql.=" GROUP BY mt.item_id order by mt.item_barcode";
}
```

In this example, structure of the query depends on the input supplied. ie. list of LODS(load out dates) and hence this is a dynamic query because column names of the query are the dates supplied by the user.

4.2 Implementation

This approach is implemented in PHP. The core of this solution is a single class, PHPSQLParser, which provides parsing and string building capabilities. The programmer uses this class to dynamically generate, through concatenation, a string representing an SQL statement and incorporating user input.

The parsed representation returned by php-sql-parser is an associative array of important SQL sections and the information about the clauses in each of those sections.

For example, suppose a query is of the form

var sql=**”**SELECT username, password from users where username=’admin’ and password=’secret’**”;**

In the example, the given query has three sections: SELECT,FROM,WHERE. Each of these sections will be in the parser output. Each of those sections contain items. Each item represents a keyword, a literal value, a sub query, an expression or a column reference.

In the following example, the **SELECT** section contains two **items** which are column reference. The FROM clause contains only one table. Finally, the where clause consists of seven items, two column references, three operators, and two literals values (constants).

The parser output of the given query will be of the form

```
Array(
    [SELECT] => Array (
            [0] => Array (
                    [expr_type] => colref
                    [alias] =>
                    [base_expr] => username
                    [no_quotes] => Array (
                            [delim] =>
                            [parts] => Array (
                                    [0] => username
                                )
                        )
                    [delim] => ,
                )
            [1] => Array (
                    [expr_type] => colref
                    [alias] =>
                    [base_expr] => password
                    [no_quotes] => Array (
                            [delim] =>
                            [parts] => Array (
                                    [0] => password
                                )
                        )
                )
        )
    [FROM] => Array (
            [0] => Array (
                    [expr_type] => table
                    [table] => users
                    [no_quotes] => Array (
                            [delim] =>
                            [parts] => Array (
                                    [0] => users
                                )
                        )
                    [alias] =>
                    [hints] =>
                    [join_type] => JOIN
                    [base_expr] => users
                )
        )
```

```
    [WHERE] => Array (
            [0] => Array (
                    [expr_type] => colref
                    [base_expr] => username
                    [no_quotes] => Array (
                            [delim] =>
                            [parts] => Array
                                (
                                    [0] => username
                                )
                        )

                    [sub_tree] =>
                )
            [1] => Array (
                    [expr_type] => operator
                    [base_expr] => =
                    [sub_tree] =>
                )
            [2] => Array (
                    [expr_type] => const
                    [base_expr] => 'admin'
                )
            [3] => Array (
                    [expr_type] => operator
                    [base_expr] => AND
                )
            [4] => Array
                (
                    [expr_type] => colref
                    [base_expr] => password
                    [no_quotes] => Array (
                            [delim] =>
                            [parts] => Array (
                                    [0] => password
                                )

                        )
                )
            [5] => Array
                (
                    [expr_type] => operator
                )

            [6] => Array
                (
                    [expr_type] => const
                    [base_expr] => 'secret'
                )
        )

)
```

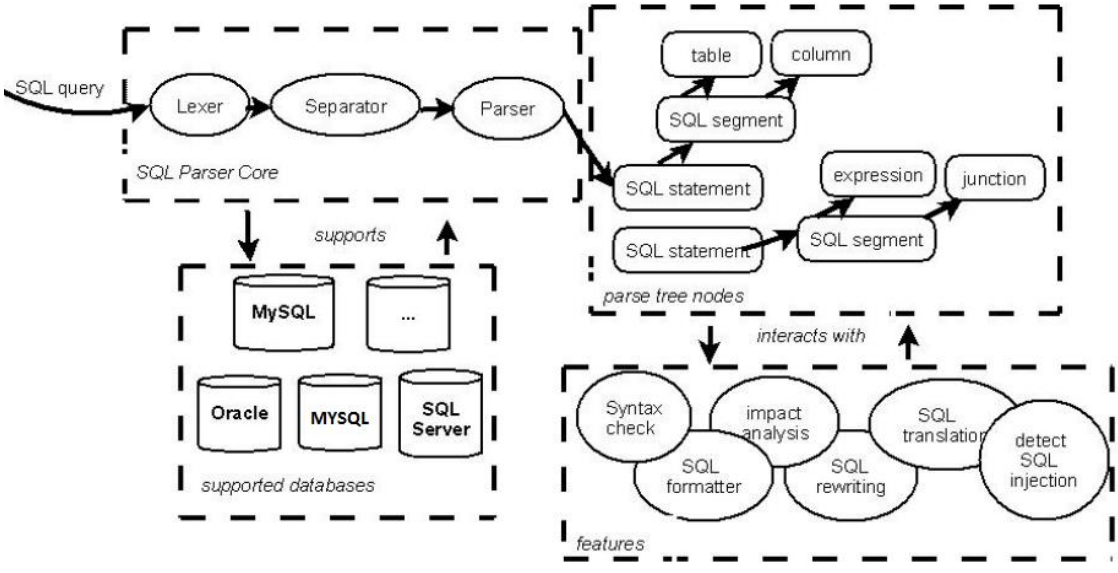

Figure 2: Architecture of General SQL Parser

PHP SQL Parser reads the input SQL text of the query. Lexer (lex parser) tokenizes the input SQL into a list of tokens. Separator turns the source tokens into a list of SQL statements. Each separate SQL statement includes a fragment of source tokens. The parser then creates a raw parse tree. Parser then translates the raw query parse tree into a formal parse tree in the form of an associative array [17].

This implementation makes use of the publically available SQL Parser for PHP (Hypertext Pre Processor).

Install SQL Parser

```
namespace PHPSQLParser;
require_once dirname(__FILE__) . './PHP-SQL-Parser/vendor/autoload.php';
$parser = new PHPSQLParser();
$sql = "SELECT username,password FROM users where username = '$uname' AND password = '$pswd'";
  $parse_tree = $parser->parse($sql);
  print_r($parse_tree);
```

5. CONCLUSION

Most all web applications employ a middleware technology designed to request information from a relational database in SQL. SQL query injection is a common technique hackers employ to attack these web-based applications. These attacks reshape SQL queries, thus altering the behavior of the program for the benefit of the hacker. It is evident that all injection techniques modify the parse tree of the intended SQL. It is illustrated by simply juxtaposing the intended query structure with the instantiated query. This approach can detect and eliminate these attacks and guarantees the security against any type SQL Injection. This approach has provided an implementation of this technique in a common web application platform, PHP, JAVA etc. This implementation minimizes the effort required by the programmer, as it captures both the intended query and actual query with minimal changes required by the programmer.

REFERENCES

[1] P. Bisht, P. Madhusudan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security Volume: 13, Issue: 2, 2010.

[2] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[3] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) Formal Demos, pp 697–698, 2004.

[4] S. W. Boyd and A. D. Keromytis. SQLRand: Preventing SQL injection attacks. In In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, 2004.

[5] O. Maor and A. Shulman. SQL injection signatures evasion. In http://www.imperva.com/application defense center/white papers/sql injection signatures evasion.html, 2004.

[6] W. Security. Challenges of automated web application scanning. In http://greatguards.com/docs/insightweb.htm, 2003.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P.Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, 1998.

[8] C. Brabrand, A. Moeller, M. Ricky, and M. Schwartzbach. Powerforms: Declarative client-side form field validation. In World Wide Web, 2000.

[9]. D. Litchfield. Web application disassembly with ODBC error messages. In http://www.nextgenss.com/papers/webappdis.doc.

[10] Bogdan Carstoiu, Dorin Carstoiu. Zatara, the Pluginable Eventually Consistent Distributed Database, Journal of AISS, Vol. 2, No. 3, pp. 56-67, 2010.

[11] Dorin Carstoiu, Elena Lepadatu, Mihai Gaspar, "Hbase – non SQL Database, Performances Evaluation", Journal of IJACT, Vol. 2, No. 5, pp. 42-52, 2010.

[12] F.Monticelli, PhD SQLPrevent thesis. University of British Columbia (UBC) Vancouver, Canada.2008.

[13] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String evaluation. Recent Advances in Intrusion Detection, Volume: 3858, Pp: 124-145, 2006.

[14] William G.J. Halfond, Jeremy Viegas and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures," College of Computing Georgia Institute of Technology IEEE, 2006.

[15] Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom, "Evaluation of SQL Injection Detection and Prevention Techniques". International Journal of Advancements in Computing Technology, 2011, Korea.

[16] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[17] Gudu Software General SQL Parser User Guide. Version: 1.0. Release Date: 19 September 2012.