International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org, editorijaiem@gmail.com
Volume 2, Issue 6, June 2013 ISSN 2319 - 4847

A Novel Approach to detect SQL injection in web applications

Kuldeep Kumar[1], Dr. Debasish Jena[2] and Ravi Kumar[3]

[1&2] IIIT Bhubaneswar, Bhubaneswar-751003
[3]InstaSafe Technologies Pvt. Ltd, Bangalore-560076

ABSTRACT

The recent increase in the growth and use of the Internet for a wide-range of Web-based applications such as e-commerce, e-banking, etc., has increased popularity of web based applications. This increase has made the Internet a potential target for different forms of attacks. The increasing frequency and complexity of web-based application attacks have raised awareness of web application administrators of the need to effectively protect their web applications from being attacked by malicious users. SQL injection attack is a class of command injection attacks in which specially crafted input string result in illegal queries to a database has become one of the most serious threats to Web applications today. An SQL injection attacks targets interactive Web applications that employ database services. In this paper, we proposed a method based on grammatical structure of an SQL statement and validation of the user input. This method uses combined static and dynamic analysis. The experiments show that the proposed method is effective and simple than other methods.

Keywords: SQL injection attack, SQL query, A combined dynamic and static method, Web application

1. INTRODUCTION

Web applications have been playing a critical role in many areas such as financial transactions, commercial business, cyber community services. The ability to access the web anywhere and anytime is a great a advantage; however, as the web becomes more popular, web attacks are increasing. Web applications usually take inputs from users through form fields, cookies, or some other standard channels, and use these input data in further processing operations, such as querying databases, generating web pages, or executing commands. Because the input data are from the remote users and may contain malicious values, they need to be validated before use. Once a web application fails to do so, attacks can exploit the vulnerabilities to launch particular attacks. Examples of popular input validation attacks are SQL injections, cross site scripting, and command injections. These attacks can cause many serious problems, such as leak of sensitive information and corruption of critical data.
Most web attacks target the vulnerabilities of web applications, which have been researched and analyzed at OWASP [1]. Many researchers have been studying a number of methods to detect and prevent SQL injection attacks, and most preferred techniques are web framework, static analysis, dynamic analysis, combined static and dynamic analysis, and machine learning techniques.

The web framework [2] uses filtering methods for user input data. However, because it is only able to filter some special characters, other detouring attacks cannot be prevented. The static analysis method [3] involves the inspection of computer code without actually executing the program. The main idea behind static analysis is to identify software defects during the development phase. Static analysis is applied to find potential violations matching a vulnerability pattern, so it is more effective than the filtering method. But attacks having the correct parameter types cannot be detected. The main limitation of the method is that it cannot detect SQL injection attacks patterns that are not known beforehand, and explicitly described in the specifications. The dynamic analysis [6] can be seen as the next logical step of static analysis. It inspects the behavior of a running system and does not require access to the internals of the system; however this method is not able to detect all SQL injection attacks. A combined static and dynamic analysis method [8, 15] can compensate for the weaknesses of each method and is highly proficient in detecting SQL injection attacks. The combined usage of a method of static and dynamic analysis is very complicated. A machine learning method [11, 12] of a combined method can detect unknown attacks, but the results may contain many false positives and negatives.
In this paper, we present a novel runtime technique to eliminate SQL query injection. We make the observation that maximum SQL injections alter the structure of the query intended by the programmer. By capturing the SQL queries at runtime and separate the value of SQL query attribute of web pages when parameters are submitted and then compare it

with a predetermined one. And also checks the validity of user input. Our method aims to satisfy the following three dimensions:
- eliminate the possibility of the attack.
- minimize the effort required by the programmer.
- minimize the runtime overhead.

The rest of the paper is organized as follows. Section 1 review the architecture of web application and SQL injection attacks. Section 2 reviewing and discussing related work. Section 3 proposes a method which uses a combination of SQL query parameter separation and combined static and dynamic analysis methods for detection of SQL injection attacks. Section 4 describes results and discussion. Finally, we conclude the paper in Section 5.

1.1 Web Application and SQL Injection attacks

1.1.1 Web application architecture

A web application is typically client/server software that handles user requests coming from clients such as web browsers. To serve the user requests, it often requires accessing system resources such as databases and files at the server end. Figure 1 shows a simplified architecture of a web application. The system resources are a part of trusted environment and often contain security critical data.

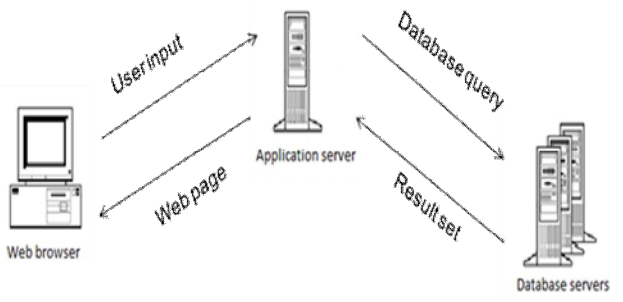

Figure 1 Web application architecture

Hence the resources need appropriate protection to maintain their confidentiality and integrity. It cannot be directly used to access the system resources. Checks should be performed to validate the user input before it can be used in the trusted environment. Lack of validity checks or inadequate checks can result in exploitable vulnerabilities.

1.1.2 SQL Injection Defined

SQL injection attack is a kind of attack where an attacker sends SQL (Structure Query Language) code to a user input box in a web form of a web application to gain unlimited and unauthorized access. The attacker's input is transmitted into an SQL query in such a way that it forms an SQL code [2].

1.1.3 SQL Injection Types

These are the classification of SQL injection types according to OWASP [1]:

(a) Tautology:
This attack bypasses the authentication and access data through vulnerable input field using "**where**" clause by injecting SQL tokens into conditional query statements which always evaluates to true.
SELECT * FROM <tablename> WHERE userId = '1' or '1=1'—' AND password = 'abc';

(b) Illegal/Logically Incorrect Queries:
The error message sent from database on being sending wrong SQL query may contain some useful debugging information.
SELECT * FROM <tablename> WHERE userId ='111' AND password = 'abc' AND CONVERT(int, (SELECT name FROM sysobjects WHERE xtype = 'u'))

(c) Union queries:
The “Union” keyword in SQL can be used to get information about other tables in the database. And if used properly this can be exploited by attacker to get valuable data about a user from the database.
SELECT * FROM <tablename> WHERE userId = ’111’ UNION SELECT creditCardNumber from CreditCardTable WHERE userId=’admin’—‘ AND password = ‘abc’;

(d) Piggy-backed Queries:
This is the kind of attack where an attacker appends “;” and a query which can be executed on the database.
SELECT * FROM <tablename> WHERE userId = ‘111’ and password = ‘abc’; DROP TABLE <tablename>;

(e) Stored Procedure:
It is an abstraction layer on top of database and depending on the kind of stored procedure there are different ways to attack. The vulnerability here is same as in web applications. Moreover all the types of SQL injection applicable for a web application are also going to work here.

(f) Blind Injection:
It’s difficult for an attacker to get information about a database when developers hide the error message coming from the database and send a user to a generic error displaying page. It’s at this point when an attacker can send a set of true/false questions to steal data.
SELECT name FROM <tablename> WHERE userId=’111’ and ‘1 = 0’ –‘ AND password = SELECT name FROM <tablename> WHERE id=’111’ AND ‘1 = 1’ –‘ AND password = ;
Both the queries will return an error message in case the web application is secure, however if there is no validation for input then the chances of injection exist. If attacker receives an error after submitting the first query, he might not know that, was it because of input validation or error in query formation. After that on submission of the second query which is always true if there is no error message then it clearly states that id field is vulnerable.

(g) Timing Attacks:
In this kind of attack timing delays are observed in response from a database which helps to gather information from a database.
SELECT account FROM <tablename> WHERE userId = ‘111’ AND ASCII(SUBSTRING(SELECT top 1 name FROM <tablename>),1,1)) > X WAITFOR 5 –‘ AND password=’’;

(h) Alternate Encodings:
This technique is used to modify injection query by using alternate encodings, like – Unicode, ASCII, hexadecimal. In this way attacker can escape the filter for “wrong characters”. It could be dangerous if used in combination with other techniques as it can target different layers of a web application. All different kinds of SQL injection attack can be hidden using this method through alternate encodings.
SELECT name FROM <tablename> WHERE id = ’’ and password = O; exec(char(O x73687574646j776e));
The actual character is returned by the char function used here which takes hexadecimal encoded characters as input. During execution this encoded string gets converted into shutdown command for database.

2. Related work

Various techniques have been proposed for the confrontation of the threat of SQL injection attacks. In this section, the characteristics of the best known techniques are briefly discussed and their principal weaknesses are highlighted.

2.1 Web framework
A web framework is a software framework that is designed to support the development of dynamic websites, web applications and web services. It uses a filtering method to remove special characters. Recently, some web frameworks have provided a wider variety of prevention methods than ever before. PHP provides Magic Quotes [2], which works when any combination of 4 special characters ‘, ’’, /, NULL exists in the data field of the POST, GET and COOKIES pages. It automatically adds a ‘\’ in front of the special character to prevent SQL injection attacks. However, Magic quotes only works for the four special characters.SQL injection attacks with other symbols are not detected.

2.2 JDBC-Checker
Carl Gould et al., in [3] propose a Static Analysis method in order to detect SQL injection vulnerability. They uses Java String Analysis (JSA) library to validate the user input type dynamically and prevent SQL injection attacks. However, if

malicious input data has the correct type or syntax, it cannot protect against the SQL injection attack. Also, the JSA library only supports the Java programming language.

2.3 Zhendong Su and Gary Wassermann's Approach [4]
This approach uses a static analysis method which was combined with automated reasoning. This method assumes that there is no tautology in an SQL query generated dynamically, which was verified. Thus , this method is efficient in detecting SQL injection attacks, but other SQL injection attacks except for tautology cannot be detected.

2.4 SAFELI
Fu et al. suggested a Static Analysis approach to detect SQL Injection Vulnerabilities. The main aim of SAFELI [5] approach is to identify the SQL Injection attacks during compile-time. It has a couple of advantages. First, it performs a White-box Static Analysis and second, it uses a Hybrid-Constraint Solver. On one hand where the given approach considers the byte-code and deals mainly with strings in case of White-box Static Analysis, on the other through Hybrid-Constraint Solver, the method implements an efficient string analysis tool which is able to deal with Boolean, integer and string variables. Its implementation was done on ASP.NET Web applications and it was able to detect vulnerabilities that were ignored by the black-box vulnerability scanners. This approach is an efficient approximation mechanism to deal with string constraints. However, the approach is only dedicated to ASP.NET vulnerabilities.

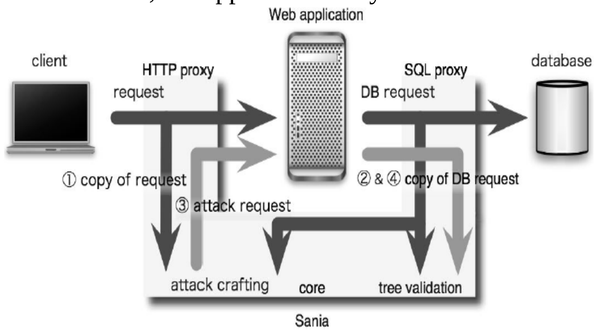

Figure 2 Structure of SANIA

2.5 SANIA
Yuji et al., in [6] propose a Dynamic Analysis method in order to detect SQL injection vulnerability. SANIA protects against SQL injection attacks by using the following procedures
(1) It collects normal SQL queries between client and web applications and between the web application and database, and analyzes the vulnerabilities.
(2) It generates SQL injection attack codes which can reveal vulnerabilities.
(3) After attacking with the generated code, it collects the SQL queries generated from the attack.
(4) The normal SQL queries are compared and analyzed with those collected from the attack, using a parse tree.
(5) Finally, it determines whether the attack succeeded or not.
These procedures are shown in figure 2. Since it uses a parse tree, SANIA is more accurate than the method which uses an HTTP response.

2.6 SQL-IDS Approach
This approach has been suggested by Kemalis and Tzouramanis in [7] and it uses a novel specification-based methodology for detecting exploitations of SQL injection vulnerabilities. The method proposed here does query-specific detection which allows the system to perform concentrated analysis at almost no computational overhead. It also does not produce any false positives or false negatives. This is a very new approach and in practice it's very efficient; however, it is required to conduct more experiments and do comparison with other detection methods under a flexible and shared environment

2.7 AMNESIA
This approach has been suggested by Halfond W. G, Orso. A , in [8] and it uses a model-based approach to detect illegal queries before their execution into the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. A primary assumption regarding the applications which the method targets is that the application developer creates queries by combining hardcoded strings and variables using operations such as concatenation, appending and insertion. The main drawback of AMNESIA is that it requires the modification of the web application's

source code for the successful collaboration with the security monitor officer. Figure 3 shows a schematic diagram of this method.

2.8 SQLrand Approach

SQLrand [9] is an approach proposed by Boyd and Keromytis in which randomized SQL query language is used, pointing a particular CGI (Common Gateway Interface) in an application. A proxy server was used in between the SQL server and Web server. It sends SQL queries with a randomized value to the proxy server. The SQL queries received from the client were de-randomized and request was sent to the server. This technique has two main advantages: security and portability. However, if the random value can be predicted, this method is not effective. Figure 4 shows a schematic diagram of this method.

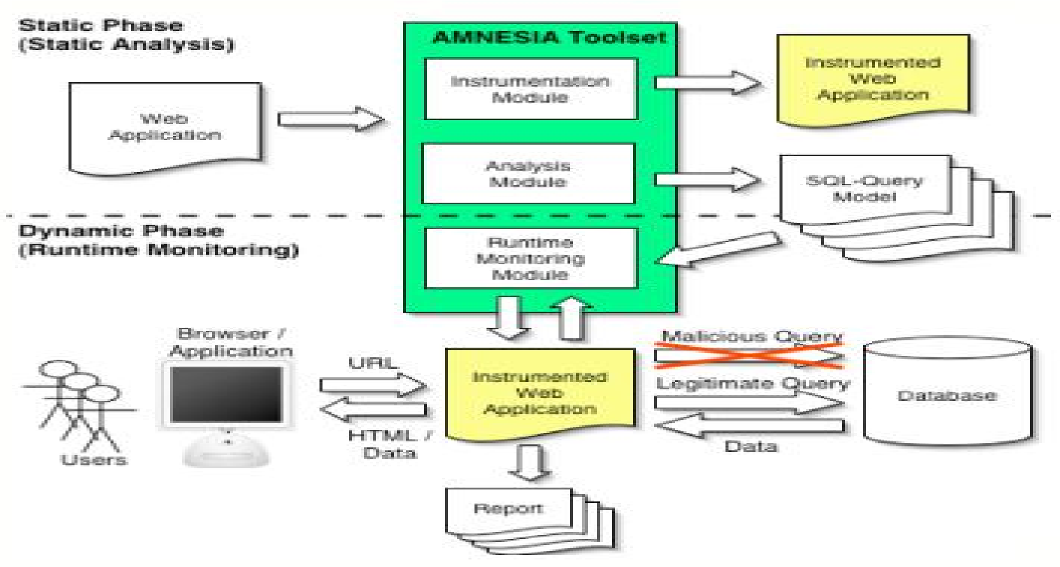

Figure 3 Schematic diagram of AMNESIA

2.9 Parse Tree Validation Approach

A parser tree framework was adopted by Buehrer et al. [10] . The original statement was compared with parsed tree of a particular statement dynamically at runtime. The execution of the statement was stopped unless there was a match found. A student's web application was used for this method using SQLGuard. Although the technique was found out to be efficient, it contained two major deficiencies: listing of input only and additional overheard computation [13].

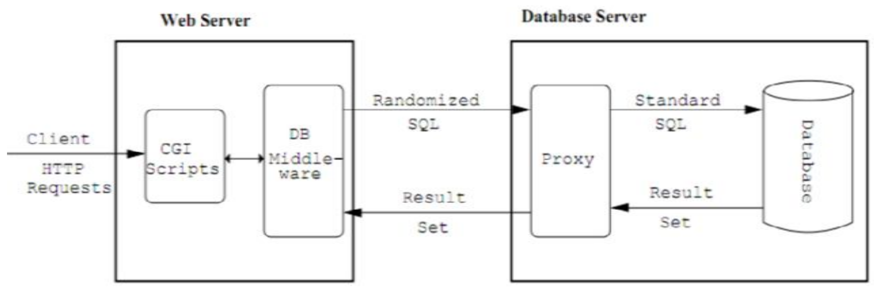

Figure 4 Schematic diagram of SQLrand.

3. Proposed method

3.1 Proposed method

This section proposes a novel method to detect SQL injection attacks based on static and dynamic analysis. This method separate the attribute values (user input) of SQL queries (dynamic queries) at runtime (dynamic method) and compares their structure with the SQL queries (template queries) analyzed in advance(static method). And also checks for the validity of the separated input from dynamic queries. Figure 5 shows the architecture of proposed model. The symbols used in this proposed algorithm are shown in Symbol Table 1.

ALGORITHM:

```
Step 1: Identify untrusted input and  security sensitive operation in the web application

Step 2: wrap every untrusted input and security operation with sql_Validator() Function

$query=select * from user_table where id='123' and password='abc'

 If(sql_Validator($query,$idf))

       then execute($query)

 else

      Block execution and redirect to error page

Step 3:  Separate the input from dynamic query,     so dynamic query is divided into two parts

Input:   id = 123, Password = abc

dynamic query after removing input:

X=select * from user_table where id='' and password =''

Step 4: Validation phase

 a)  Validate the structure of dynamic query: Compare the structure of dynamic query with template query

Template query:

 select * from user_table where id='' and  password=''

Dynamic query without input:

 X=select * from user_table where id='' and password=''

If (checksum(X) == checksum  (template query))

        Then structure is valid return(1)
 Else
       SQL injection attack return(0)

b)  Validate the user input which parsed from query check whether attack pattern present in input

If (no attack pattern is present in input)
       then input is valid return(1)
Else
      Input is malicious return(0)
```

```
Step 5:

 If (structure of query AND input is valid)
        Then No SQL injection attack is there and return (1)

 Else
        return(0)
```

Algorithm 1: Base Algorithm

Symbol Table 1: Symbols used in algorithms

Symbol	Description
TQ	Template Query
DQ	Dynamic Query with user input
CTQ	Checksum of template Query
X	Dynamic Query With-out user input
Attribute_Separtor()	Separate user input from Dynamic Query
Validator()	Validate the structure of Query X and also validate user input

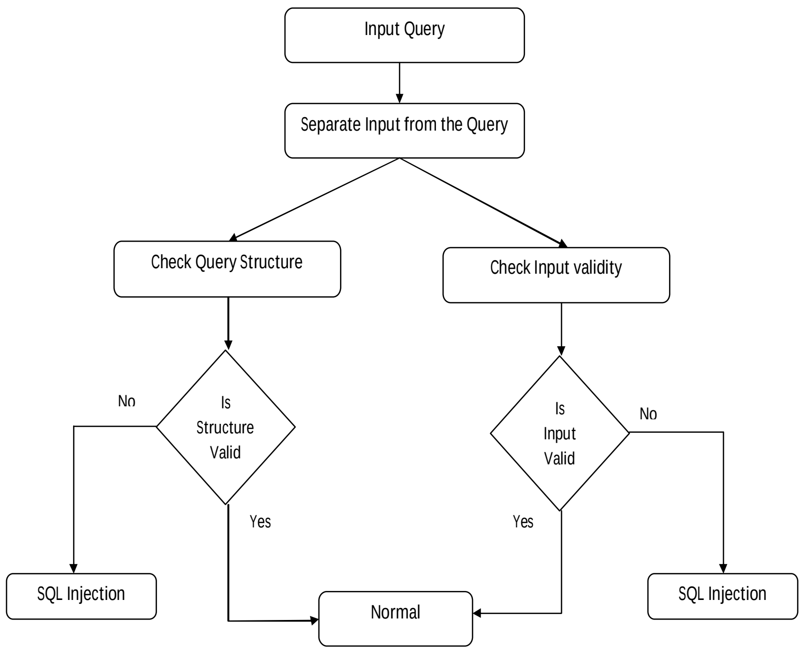

Figure 5 Architecture of proposed model

```
Algorithm Attribute_Separator($query)
Quotation_Status = { Quotation_Start,  Quotation_End};
Input_String=SQL query;
Output_Query_String=Null;
Attribute_String=Null;
Current_State=Quotation_End;
Do while( not empty of Input_String
{
   Char=Get_Token(Input_String);
If Char is a quotation character
{
   Add Char to Output_Query_String;
    If (the preceding character != back slash)
    Toggle Current_State;
  }
Else
{
  If Current_State  is Quotation_End then
      {
       Add white_space to Attribute_String;
       Add Char to Output_Query_String;
      }
 Else If Current_State  is Quotation_start then
    {
     Add Char to Attribute_String;
     }
  Else
{
     If the preceding character is \ (back slash) then
     Add Char to Output_Query_String;
       }
  }
}
```

Algorithm 2: Algorithm for separate
the attribute value from SQL Query

```
Algorithm Validator($Output_Query_String,$Attribute_String,$idf)
{
String[] blackList ={"alter","begin","cast","create","cursor”,”href”,"declare","delete","drop","exec","execute","fetch"
,"insert”, "kill" , "open","select","sys","sysobjects","syscolumns","table","update","<script","</script","--
","/*","*/","@@","@",”&#x27”, ”_&#39”};
/*
//set of template queries
TQ1: select * from <tablename> where userId=’’ ;
TQ2: select * from <tablename> where userId=’’ and password=’’;
TQ3: insert into <tablename> values (‘’, ‘’);
Etc…
*/
TQ={TQ1,TQ2,TQ3……………………..TQN}
// checksum of template queries
{CTQ1,CTQ2,CTQ3………CTQi…….CTQN}
//for Checking the query Structure
For each $Output_Query_String
If (checksum($Output_Query_String_i ) == CTQi)
    return(1);
  else
    return(0);
// for checking the user input validation
 Check whether user input contain attack Attribute_checker(Attribute_string)
{
 $Attribute_String=toLower($Attribute_String)
Do While (not empty of Attribute_String)
{
 word=getWord($Attribute_String);

 for i=0 to length.Blacklist
         {
            If (!strcmp(Blacklist[i],word))
            return(1);//attack pattern is found
          }
      }//end  do while
   return(0);
 }
```

Algorithm 3:Validate the user input and Structure of SQL Query

4. Results and Discussions

4.1 Examples applying proposed methods

A. Tautologies

```
Template Query:
   TQi = SELECT * FROM <tablename>  WHERE userId = ’’ AND password = ’’;
 Dynamic Query :
    DQi = SELECT * FROM <tablename>  WHERE userId= ’1’ OR ‘1=1’—‘ AND password=’abc’;
  Attribute_Separator (DQi):
   X =  SELECT * FROM <tablename>  WHERE userId=’ ’ or ‘ ‘—‘’abc’
    Validator(X,idf)
//query structure
     if(checksum(X)!=CTQi)
        Invalid query
// Input validity
         Input is valid no attack pattern is present in input
```

B. Illegal/Logically Incorrect Queries Attack.

```
Template Query:
       TQi = SELECT * FROM <tablename>   WHERE userId=’’ AND password=’’;
 Dynamic Query :
       DQi = SELECT * FROM <tablename>   WHERE userId=’1111’ AND password=’abc’ AND  CONVERT(char,
no) --’;
Attribute_Separator (DQi):
      X=SELECT * FROM <tablename>   WHERE userId=’’ AND password=’’ AND CONVERT(char, no) --’;
 Validator(X,idf)
//query structure
   If(checksum(X)!=CTQi)
        Invalid query
//  Input validity
         Input is valid no attack pattern is present in user input
```

C. Union Query:

```
 Template Query:
       TQi = SELECT * FROM <tablename>    WHERE userId=’’ AND password=’’;
  Dynamic Query :
       DQi = SELECT * FROM <tablename>    WHERE userId=’1111’ UNION
      SELECT * FROM <tablename>    WHERE userId= ‘admin’ --’ AND password=’abc’;
Attribute_Separator(DQi):
 X=SELECT * FROM <tablename>    WHERE userId=’’ UNION SELECT * FROM member WHERE userId=’’ --
’’abc’;
 Validator(X,idf)
```

```
//query structure
If(checksum(X)!=CTQi)
        Invalid query
//Input validity
         Input is valid no attack pattern is present in input
```

D. Second order Injection:

```
Template Query:
   TQi = INSERT INTO <tablename>     (UserID,Name,Contact) VALUES (‘’,’’,’’);
Dynamic Query :
   DQi = INSERT INTO <tablename>     (UserID,Name,Contact) VALUES(‘123’,’ <a
href="http://www.hacker.com">vishal </a>’,900000900);
Attribute_Separator (DQi):
 X=INSERT INTO <tablename>     (UserID,Name,Contact) VALUES(‘’,’’,’’);
 Validator(X,idf)
//query structure
If (checksum(X) == $CTQ_i$)
        Valid query
//Input validity
        Input is  invalid because attack pattern  is present in user input.
```

E. Piggy-Backed Query:

```
 Template Query:
        TQi = SELECT * FROM <tablename>     WHERE userId=’’ AND password=’’;
Dynamic Query :
      DQi SELECT * FROM <tablename>     WHERE userId=’admin’ AND password=’abc’; DROP TABLE
<tablename>   ; --’;
Attribute_Separator (DQi):
SELECT * FROM <tablename>     WHERE userId=’’ AND password=’’; DROP TABLE <tablename>   ; --’;
Validator(X, idf)
//query structure
If (checksum(X)!=$CTQ_i$)
        Invalid query
// Input validity
         Input is valid no attack pattern is present in input
```

F. Alternate encoding Injection

```
Template Query:
  TQi = SELECT * FROM <tablename>  WHERE userId = ’’ AND password = ’’;
 Dynamic Query :
   DQi = SELECT * FROM <tablename>  WHERE userId= ’ 1 &#39 OR &#39  1=1 &#39  —‘ AND
password=’abc’;
  Attribute_Separator (DQi):
  X = SELECT * FROM <tablename>  WHERE userId=’ ’ AND password=’’;
  Validator(X, idf)
//query structure
If (checksum(X) == $CTQ_i$)
        Valid query
//  Input validity
         Input is invalid because attack pattern is present in input
```

4.2 Results Analysis

This section compares the detection rate of the proposed method with other researchers’ methods under the same conditions.

4.2.1 Comparison of detection and prevention methods by attack types

The JDBC-Checker and Tautology-checker, use the static analysis method. The static analysis method only analyzes static SQL queries implemented inside the web application and therefore the efficiencies of the methods differ. IDS and WAVES use a machine learning anomaly detection method, which needs a large amount of SQL injection data for learning. The detection rate of the method depends on the learned parameters. JDBC-Checker does not detect the attack, but reduces the chances of SQL injection attacks by checking the type of SQL queries. Both the proposed algorithm and AMNESIA and SQLCheck use both static and dynamic methods simultaneously. The proposed method compares static and dynamic SQL queries generated. It detects attacks by comparing the structure, grammar of the queries and check for the validity of the user input. If a dynamically generated query has a different structure or uses a different grammar from that of a static query, it is detected. If somehow malicious input don't affect the grammars or structure of the query then at the time of validating the user input, malicious input is detected. However, AMNESIA and SQLCheck use static and dynamic SQL queries for a parse tree. As a result, these methods cannot detect stored procedure and query structure bypassing type attacks and because the time complexity is O(n3), it is impossible to detect the attack in real time. So our algorithm proposed in this paper does not use complex analysis methods such as parse trees. The time complexity of this algorithm is O(1). It uses a very simple method which compares queries after the separation of attribute values and also validates user input. Therefore, it can be implemented in any type of DBMS and is able to detect SQL injection attacks including stored procedure and query structure bypassing type attacks. The comparisons are shown in Table 2.

Table 2: Comparison of detection and prevention methods for various SQL injection attacks.

Detection/ Prevention method	Tautologies	Illegal/Incorrect Queries	Union Queries	Piggy-Backed Queries	Stored procedures	Queries structure bypassing
AMNESIA	YES	YES	YES	YES	NO	NO
SQLCheck	YES	YES	YES	YES	NO	NO
SQLrand	YES	No	Yes	YES	NO	NO
JDBC-Checker	N/A	N/A	N/A	N/A	N/A	N/A
SQLGuard	YES	YES	YES	YES	NO	NO
Tautology-Cheker	YES	NO	NO	NO	NO	NO
Proposed Method	YES	YES	YES	YES	YES	YES

5. Conclusion

Web applications employ a middleware technology designed to request information from a relational database in SQL. SQL 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. That is, effective injection techniques modify the structure of the intended SQL query or craft the malicious input without altering the structure of input query. We have illustrated that by simply compare query structure with the template query and, we can detect and eliminate SQL injection vulnerabilities and also eliminate simple XSS which directly based on the insertion of keywords. This implementation minimizes the effort required by the programmer, as it captures both the template query and actual query with minimal changes required by the programmer.

Future work should focus on evaluating the techniques precision and effectiveness in practice. Empirical evaluations will be performed which allow comparing the performance of the different techniques when they are subjected to real-world attacks and legitimate inputs. As well as precision, accuracy and add on will be incorporated for the prevention of SQL related vulnerability such as XSS and much complex SQLIA such as Internal Network Attack.

References

[1.] "The Open Web Application Security Project, OWASP TOP 10 Project." http://www.owasp.org/.
[2.] "PHP, magic quotes." http://www.php.net/magic_quotes/.

[3.] C. Gould, Z. Su, P. Devanbu, JDBC checker: " A static analysis tool for SQL/JDBC applications", in Proceedings of the 26th International Conference on Software Engineering, ICSE, 2004, pp. 697–698.
[4.] G. Wassermann, Z. Su, "An analysis framework for security in web applications", In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems, SAVCBS, 2004, pp. 70– 78.
[5.] Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., and Tao, L., "A Static Analysis Framework for Detecting SQL Injection Vulnerabilities" , in Proceedings 31st Annual International Computer Software and Applications Conference 2007 (COMPSAC 2007), 24-27 July (2007), 87-96.
[6.] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, Y. Takahama, "Sania: syntactic and semantic analysis for automated testing against SQL injection" , in Proceedings of the Computer Security Applications Conference 2007, 2007, pp. 107–117.
[7.] Kemalis, K. and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL injection Detection", SAC'08. Fortaleza, Ceará, Brazil, ACM (2008), 2153 2158.
[8.] W.G. Halfond, A. Orso, "AMNESIA: analysis and monitoring for neutralizing QL-injection attacks", in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 174–183.
[9.] S. Boyd, A. Keromytis, "SQLrand: preventing SQL injection attacks in Applied Cryptography and Network Security", in: LNCS, vol. 3089, 2004, pp. 292–302.
[10.] G. Buehrer, B.W. Weide, P.A.G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks", in Proceeding of the 5th International Workshop on Software Engineering and Middleware ACM, 2005, pp. 106–113.
[11.] Y. Huang, S. Huang, T. Lin, C. Tasi, "Web application security assessment by fault injection and behavior monitoring", in Proceedings of the 12th International Conference on World Wide Web, 2003, pp. 148–159
[12.] F. Valeur, D. Mutz, G. Vigna, "A learning-based approach to the detection of SQL attacks", in Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2005, pp 123–140.
[13.] Buehrer, G., Weide, B.W., and Sivilotti, P.A.G., "Using Parse Tree Validation to Prevent SQL Injection Attacks", in Proceedings of 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal (2005) 106–113.
[14.] W.G. Halfond, J. Viegas, A. Orso, "A classification of SQL-injection attacks and countermeasures", in Proceeding on International Symposium on Secure Software Engineering, Raleigh, NC, USA, 2006, pp. 65–81.