Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

SQLDefend: AN AUTOMATED DETECTION AND PREVENTION TECHNIQUE FOR SQL INJECTION VULNERABILITIES IN WEB APPLICATIONS

E. E. Ogheneovo and P. O. Asagba
Department of Computer Science,
University of Port Harcourt,
Port Harcourt, Nigeria.
edward_ogheneovo@yahoo.com, pasagba@yahoo.com

Received: 16-10-12
Accepted: 07-11-12

ABSTRACT

SQL injection attacks (SQLIAs), one of the most foremost threats to Web applications is an attacking technique in which specially crafted input string result in illegal queries to a database. An SQL injection attack target interactive Web applications that employ database services. In this paper, we propose SQLDefend as a technique to detect and prevent SQLIAs. Our approach provides a full automated model. This model combines parser and decision tree. It is an algorithm that models string values using Context Free Grammars (CFGs) and then use decision tree to train the user input. We use parse tree validations to input strings. First the technique checks if the two queries match syntactically and then use rule-based decision tree classifier to classify user input. If the result meets the condition defined, then the query will be considered legitimate and thus accepted otherwise it will be rejected. Our result clearly shows no false positives and false negatives. The result also shows a lower runtime overhead in execution time and CPU usage. The technique is thus effective in preventing SQLIAs.

Key words: SQL Injection, Vulnerability, SQLDefend, Web Applications, Attacks

INTRODUCTION

Database-driven web applications have become widely deployed on the Internet and organizations use them to provide a broad range of services to their customers. These databases and their underlying databases, often contain confidential, or even sensitive, information, such as customer and financial records. This information can be highly valuable and makes web application an ideal target for attacks. In fact, in recent years there has been an increase in attacks against these online databases (Halfond et al., 2005). One of such attacks is called the SQL Injection Attacks (SQLIAs). Injection attacks constitute one of the largest classes of security problems (Bravenbor et al., 2007).

SQL Injection attacks (SQLIA) is a Web attacking vector considered to be one of the most common form of attacks in Web applications. OWASP (2010) rated SQLIAs as one of the top ten web application vulnerabilities. SQL injection attacks are one of the most foremost threats to web applications. It is an attacking technique which is used to pass SQL query through a web application directly to the database by taking advantage of insecure code’s non-validated input values (Muthuprasama et al., 2010). SQL injection attacks pose a serious security threat to web applications. They allow attackers to obtain unrestricted access to the databases underlying the applications and to the potentially sensitive information these databases contain (Halfond et al., 2006). Web applications that are vulnerable to SQL injection may allow an attacker to gain complete control of underlying databases. As a result, sensitive information about users of such web applications are exposed leading to malicious activities such as: password theft, identity theft, loss of confidential information, stealing of credit card numbers, denial-of-service attacks, and fraud (Asagba and Ogheneovo, 2011).

The root cause of SQLIAs is insufficient input validation. SQLIAs occur when data provided by a user is not properly validated and included in an SQL query (Halfond et al., 2005). In such a vulnerable application, an SQLIA uses malformed user input that alters the SQL query issued in order to gain unauthorized access to a database and extract or modify sensitive information (Bisht et al., 2010). Usually, web application is a three-tier architecture: the application tier at the user side, the middle tier which converts the user queries into the SQL format, and the backend database server which stores the user data as well as the user's authentication table (Wei et al., 2006; Ali et al., 2009). Whenever a user wants to enter into the web database through application tier, the user inputs his/her authentication from a login form. The middle tier server will convert the input values of username and password from user entry form into the format shown below.

SELECT * FROM user_account WHERE username='username' AND passwd='password'

If the query result is true then the user is authenticated otherwise it is denied. But there are some malicious attacks which can deceive the database server by entering malicious code through SQL injection which always return true results of the authenticated query. For example, the hacker enters the expression in the username field like " ' OR 1=1- -' ". So, the middle tier will convert it into SQL query format as shown below. This deceives the authentication server. The query result will be:

SELECT * FROM user_account WHERE username= 'OR 1=1- -'AND passwd='password'

Analyzing the above query, the result would always be true. This is because malicious code has been used in the query. In this query, the mark (') tells the SQL parser that the user name string is finished and like " ' OR 1=1--' " statement appended to the SQL statement would always evaluate to true. The (--) is comment mark in the SQL tell the parser that the statement is finished and the password will not be checked. So, the result of the whole query will return true and this authenticate the user without checking password. The login form is used to get the user name and password from the user. The user name field can take some extra values other than alphanumeric characters. It may support some special characters like %, $, |, #, etc.

A number of approaches to dealing with SQLIAs have been proposed, but none has been completely effective due to some drawbacks. These approaches either used taint method where an untrusted user input is tainted and checked for malicious queries or the query is dynamically checked at runtime. For one, these approaches incur high runtime overhead and in situations where static methods are used only, it means the programmer will have to manually check the query each time. Our approach is different from other techniques in that it uses parser and a machine learning tool called rule-based decision tree classifier for its methodology and it assumes all queries to be malicious until proved otherwise and that it will reduce runtime overhead and thus remove the possibilities of false positives and false negatives.

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

MATERIALS AND METHODS

Architecture of SQLDefend

This approach uses three phases: the query collection phase, the query validation phase. In the query collection phase, query is collected by user input validator and stores them in a repository. In the query validation phase, the generated query stored at the proxy is sent to the user input extractor. The query is then analyzed statically by first scanning the query at the lexical analysis stage where the query is grouped into various tokens and keywords. A parse tree is then generated for the query by the parser. The parse tree generated is then analyzed dynamically at runtime by comparing the statically generated query with the dynamic query. At this stage, the parse tree of the guest language is compared with the parse tree of the host language to see if they agree syntactically and to see if the parse trees produced by the two languages matches. If they match each other, it means the query is a benign (good) query and it is sent to the database. However, if the parsed queries of the host and the guest languages do not match, the query is malicious and will be rejected before it gets to the database. However, if the query is legitimate (benign) query, it will be passed to the database and the result of the query will be returned to the user. Our approach will be able to track the effects of string operations while retaining the syntactic and semantic structure of the input strings. Figure 1 below shows the architecture for the proposed model.

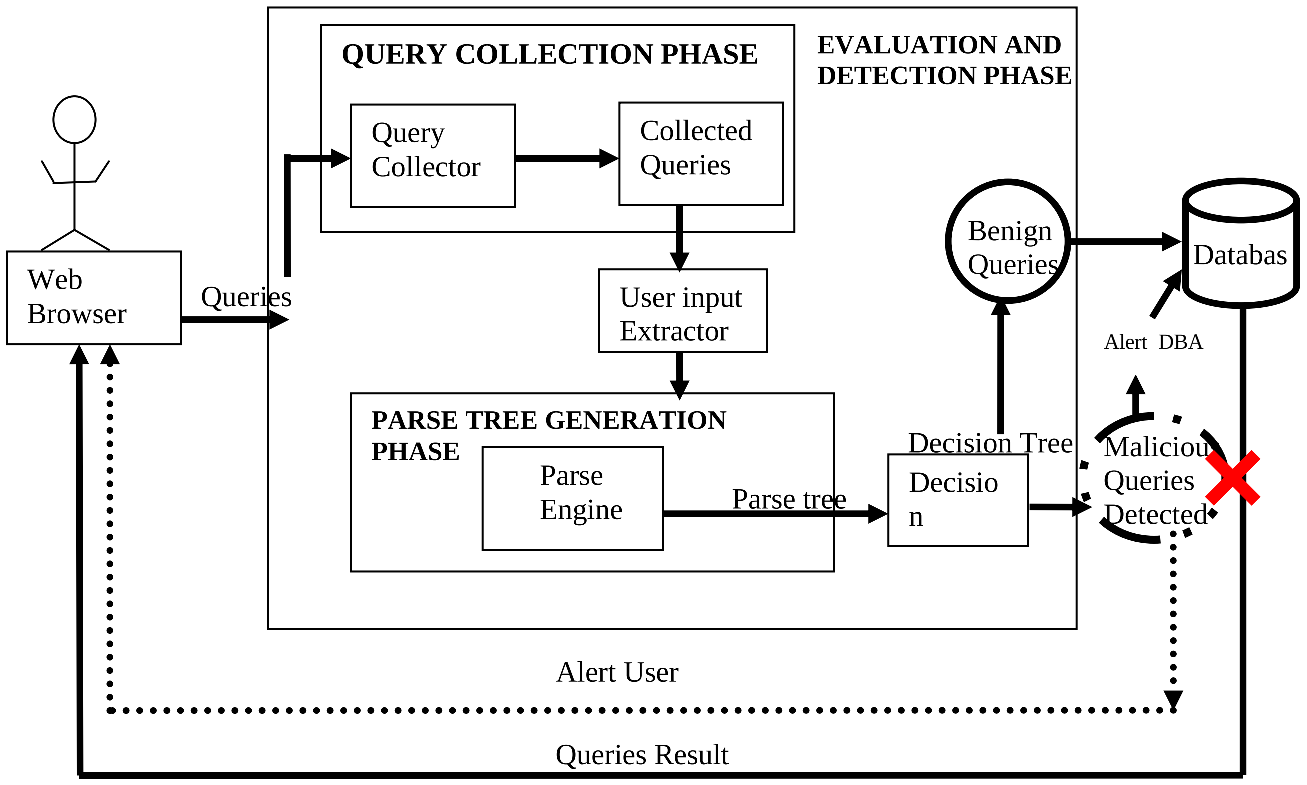


Fig. 1: Architecture for SQLDefend

Generation BNF for SELECT Statements

We generated a Backus-Naur Form (BNF) for select statements. The general BNF generated was then used to construct the structure of each select statement syntactically. The BNF of a select statement is shown in the figure below.

```
Input            ::=  sql [sql] EOF
<Select-stmt>    ::=  SELECT select_list from_clause
|  SELECT select_list from_clause where_clause
<select_list>    ::=  id_list | *
<id_list>        ::=  id | id, id_list
<from_clause>    ::=  FROM tbl_list
<tbl_list>       ::=  id_list
<where_clause>   ::=  WHERE bool_cond
<cond>           :: =  bcond OR bterm | bterm
<bterm>          ::=  bterm AND bfactor | bfactor
<bfactor>        ::=  NOT cond | cond
<cond>           ::=  value comp value ("--")
<value>          ::=  id | num | str_lit | (select-stmt)
<str_list>       ::=  'lit'
<comp>           ::=  = | != | < | > | <= | >=
```

Fig. 2: A BNF grammar for a select statement

In the figure 2, the Left-Hand-Side (LHS) represents non-terminal symbols while the Right-Hand-Side (HRS) represents terminal or non-terminal symbols of the production process.

Sample Parse Trees for Legitimate and Malicious Queries

In this section, we design sample parse trees for both legitimate and malicious queries. Parsing a statement requires the grammar of the language (quest language, e.g., MySQL, MS-SQL, etc.) that the statement was written. By parsing two statements and dynamically comparing their structures at runtime, we can determine if the two queries are structurally identical. When a malicious user successfully inject SQL query into a database, the parse trees of the intention query and the resulting SQL query do not match. Intended queries are the codes written by the programmer to query the database. The programmer supplied portion is the hardcoded 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. The programmer intends that the user supplied values to these empty leaves. In figure (a), the empty leaves are the placeholders represented by question mark ("?") which are empty leaves where the user is expected to supply his username and password; which are expected to be validated before they are passed into the database. These question marks are substituted for and they represent placeholder meta-character. A placeholder in an intention statement represents an expanding point, where each expansion must conform to the corresponding grammatical rule intended by the developer. Here, a placeholder is an intention grammar which helps to regulate the instantiation of a placeholder dynamically at runtime. Each intention rule is mapped to an existing non-terminal symbol (e.g., comp) or terminal symbol (e.g., identifier) of an SQL statement.

In our technique, we developed pre-defined queries and the user input parser using the syntactic structure of the query. The syntactic structure of the user queries are compared with the pre-defined queries generated at runtime in order to see if they are equal. This is to avoid the problem of grammar ambiguities so that only one type of parse tree is generated for a particular type of query. This we did by embedding the guest language (MySQL) inside the host language (Java). This is to ensure that the statement remains unambiguous. At the parser engine, the parser generated parse tree structures are compared at runtime and they are found to be syntactically the same, the queries are then sent to the decision engine for further verifications. In the decision

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

engine, the query will be further checked to see if it is legitimate or malicious. If legitimate, it will be parsed to the database to find the result of the query. The result once found will be returned to the web application. However, if the query is malicious, the decision trees will automatically classifier the query into the SQL injection attack type.

For example, the following SQL statement was used as one of our case studies.

SELECT * FROM user WHERE uname='?' AND password='?'

As shown in figure (a), the placeholders are represented with question marks (?) and are underlined. These are the fields where users are expected to supply their inputs. We represented this by question marks (?) because we want to make the placeholder empty since it is believed that different users have different username and passwords. In figure (b), parse tree of the SELECT statement is then drawn which indicate the programmer's intended query. This query is further checked by the decision engine and through its leaner's input data, the query is found to be legitimate (benign) and it is passed to the database. When another query is supplied, the parse tree is suspected to be different and it was classified as malicious and to further verify the query, it was passed to the decision engine where it is classified as malicious and is thus confirmed to be malicious. But to further know the exact nature and type of query, the decision engine classifier is used. The query is shown below.
SELECT * FROM user WHERE uname='eddy' AND password=passwd OR 1=1

Subsequently, the query is rejected and blocked from getting to the database. This parse tree is shown in figure (c). Similar explanation can also be giving for figures (d) and (e). In figure(d), user supplied an SQL SELECT statement.

SELECT * FROM usertable WHERE username='eddy' AND password='abc12'

However, when a comment was introduced into the query, the attacker is able to gain access into the database and get the information in the database. This is shown in the figure (e). As can be seen from figures (d) and (e), the parse trees are syntactically different. Thus the second query figure (e) will be blocked from entering the database.

SELECT * FROM usertable WHERE username='eddy' AND password='abc12'- - AND password='secret'

The parse trees showed below in figures (a-e) represents sample SELECT statements that shows how the parser will actually work whenever a query is injected into the database through the user input and password fields.

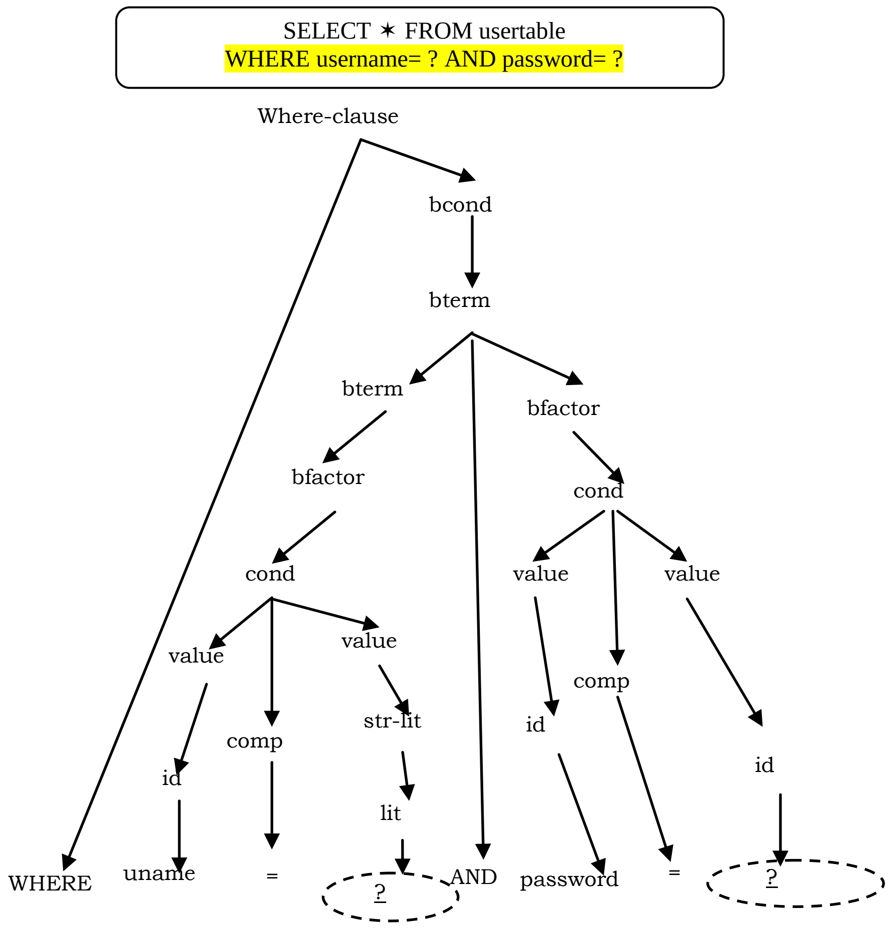
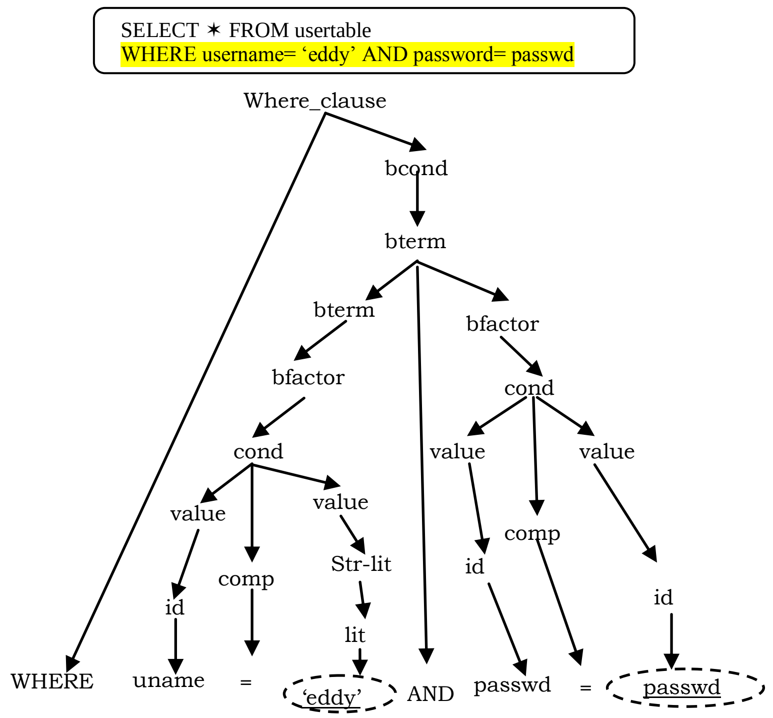


Fig. 3 (a): A parse tree for a select statement. The username and password are not supplied

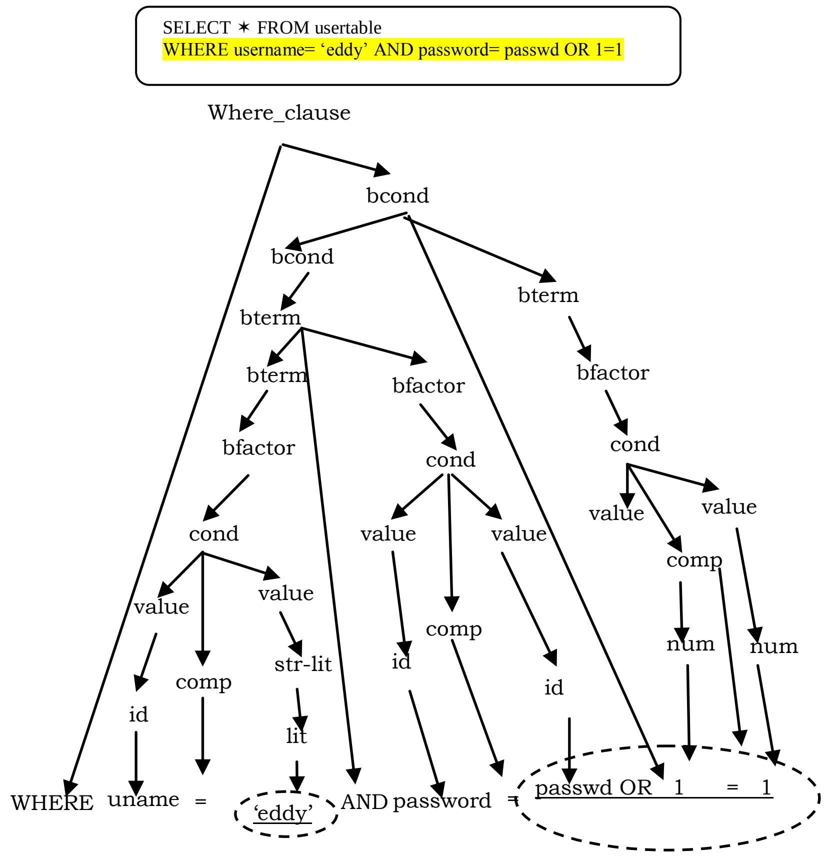
Figure 3(a) shows a parse tree for an SQL statement where the placeholders where the user is expected to supply his username and password. The placeholders are represented by question marks indicating that it is left open since any user can supply her username and password. The parse tree is drawn based on the production of the terminals and non-terminals representing the production on the SELECT statement by the Backus-Naur Form (BNF) in figure 3.

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*



```
SELECT ✶ FROM usertable
WHERE username= 'eddy' AND password= passwd
```

Fig. 3 (b) Benign select stmt



```
SELECT ✶ FROM usertable
WHERE username= 'eddy' AND password= passwd OR 1=1
```

Fig. 3 (c) Tautology query that is malicious

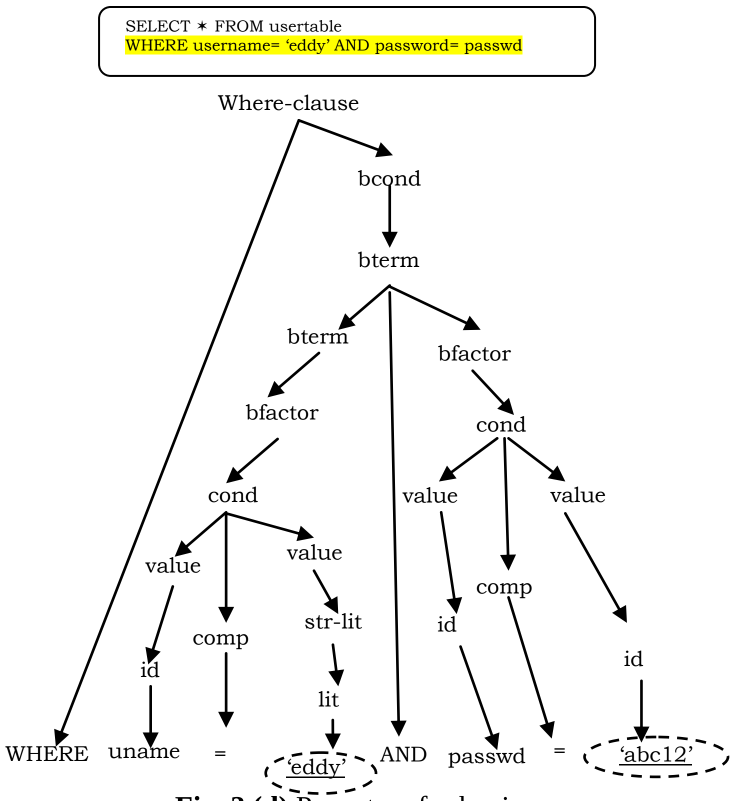


Fig. 3 (d) Parse tree for benign query

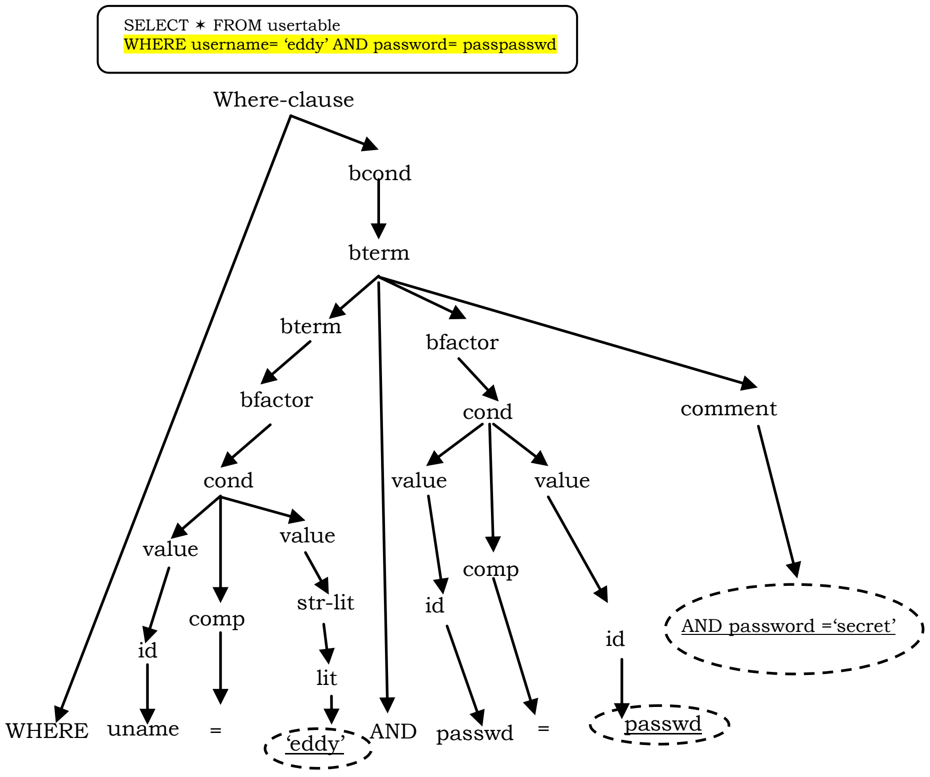


Fig. 3 (e) Parse tree for malicious query

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

Decision Tree Classifier

Combating the SQL injection attack, there are a variety of machine learning tools and techniques to detect and defend against attacks. These include: artificial neural network, Bayesian network, Genetic algorithm, Naïve-Bayes, Rule-based algorithm, and decision trees. A decision tree is an effective tool for guiding a decision process as long as no changes occur in the dataset used to create the decision tree (Abdelhalim and Traore, 2009). Decision trees provide unique insight into the problem of identifying malicious activities and can assist in the creation of technology-specific technique to defend against attacks. The main advantage of decision trees over many other classification techniques is that they produce a set of rules that are transparent, easy to understand, and easily incorporated into real-time technologies.

There are two ways to apply decision tree to classifying data. They are: data-based decision tree (DBDT) and rule-based decision tree classifier (RBDT). For data-based decision tree method, once there is a significant change in the data; restructuring the decision tree becomes a desirable task. A data-based decision tree classifier is a procedural approach of knowledge representation, which imposes an evaluation order on the set attributes. This poses a lot of challenges. There is therefore the need to use rule-based decision tree classifier as an alternative.

Also, generating a decision structure from decision rules can be done faster than generating it from training examples because the number of decision rules per decision class is often much smaller than the number of training examples per class. Finally, using rule-based decision tree methods can be done directly from the declarative rules themselves (Michalski and Imam, 1992). Some of the well-known rule-based methods for building decision tree are: RBDT-1 (Abdelhalim and Traore, 2009), RBDT-2 (Abdelhalim and Traore, 2010), and AQDT-1 (Machalski and Imam, 1992).

In our technique, we use the rule-based decision tree method for building the decision tree. Rule-based decision tree methods handle manipulations in the data through the rules induced from the data instead of the data itself. A declarative representation such as a set of decision rules is much easier to modify and adapt to different situation than to procedural one. This is simply due to lack of constraint on the order of evaluation (Imam and Michalski, 1992). It should be emphasized here that there is a major difference between building a decision tree from examples (data sets) and building it from rules. When building a decision tree from rules, the method assigns attributes to the nodes using criteria based on the properties of the attributes in the decision rules rather than statistics regarding their coverage of the data sets (Abdelhalim and Traore, 2009).

Tree Builder

The Tree Builder takes the rules supplied by the rule extractor and creates a decision tree from them. Once created, this tree is stored, and can be used subsequently without need for re-creation or modification. In our technique, we build the decision tree from rules rather than from the data sets. Our technique is based on RBDT-1, a method proposed by Abdelhalim and Traore (2009) and which has also been used to solve a number of machine learning problems. For instance, for a login module of a target application, the fundamental query will be one that retrieves a single record from the database representing the user's sign-in credentials. Let us consider a SELECT statement for our example.

SELECT user_id, user_category FROM user_credential_table

WHERE user_id=”?” AND password=”?”

In building a decision tree for the above SELECT statement, we selected the attributes that will be assigned to each node from the current set of rules attributes. These rule attributes are shown below.

$R_1 \leftarrow$ stmt_count = 1 (e.g., a single SELECT statement)
$R_2 \leftarrow$ stmt_type = plain_select_stmt
$R_3 \leftarrow$ stmt_expr = select_where_clause
$R_4 \leftarrow$ stmt_expr_count = 1
$R_5 \leftarrow$ stmt_expr_data-type = bool
$R_6 \leftarrow$ value_expr_count = 2
$R_7 \leftarrow$ value_expr_data-type = condition, condition
$R_8 \leftarrow$ value_expr_data-type = bool, bool
$R_9 \leftarrow$ parameter_count = 2
$R_{10} \leftarrow$ parameter_type = string, string

Fig. 4: A set of rules for a typical SELECT statement

Based on these rule attributes, we use what we called “rule extractor” to break an SQL SELECT statement into rules and then extract the attributes as shown below. However, to generalize our technique for all types of SQL statements for existing form of queries available in all database management system such as Microsoft SQL, MySQL, Oracle, Sybase, etc., with all their keywords, we generated the following rule attributes.

SQL Statement

- Number of distinct statement
- Statement type

Statement Expressions

- Number of distinct expressions
- Expression category
- Expression return data-type

Parameters

- Number of parameters
- Parameter data-type

Expressions

- Number of distinct
- Expression type
- Data type

Value Expression Arguments

- Number of distinct argument
- Type of each argument

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

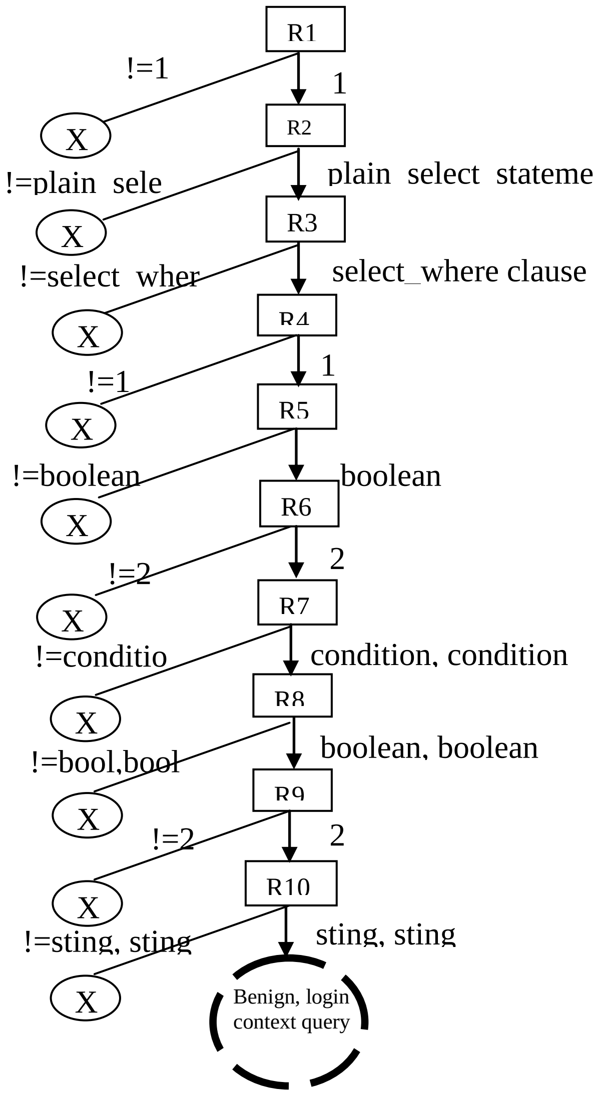


Fig. 5: Rule-based classifications of queries

Experimental Setup

We used real world applications from AMNESIA testbed (Halfond and Orso, 2005), which has been previously used by other techniques. We used this testbed since it allows us to have a common point of reference with other approaches that have used it for their evaluation. The AMNESIA testbed consists of both legitimate and malicious queries. It is a standard testbed used for evaluating code injection prevention techniques. It consists of seven applications: Bookstore, Classifieds, Portals, Employee Directory, Events, Checkers, and Office Talk. The AMNESIA testbed provides a set of subject Web application that are vulnerable to SQL injection attacks, along with test inputs that represent legitimate and malicious queries. They are available at http://www.gotocode.com and http://www.cc.gatech/~whalfond/testbed.html.

The purpose of these testbed is to facilitate the evaluation of SQL injection detection and prevention techniques. The AMNESIA testbed is shown in the table 1 below.

Table 1: Information about subject application

Subject	**LOC**	**DBIs**	**Servelet**
Bookstore	16,957	71	28
Portal	16,453	67	28
EmplDir	5,658	23	10
Classfieds	10,949	34	14
Events	7,242	31	13
Checkers	5,421	5	61
Office Talk	4,543	40	64

We also tested our technique by running it with WebGoat (http://www.owasp.org/software/webgoat.html) set of Web applications. WebGoat is a collection of applications designed to teach secure programming for Web applications, and has a range of vulnerabilities in it by design. Our application demonstrates command injection attacks, where user-supplied commands can be executed on the host by tempering with HTTP parameters. We specifically work on SQL injection attacks as an example of command injection attacks where supplying a malicious input in an HTML form results in a query being executed on the host that reveals secret data. Our technique blocked all SQL attacks and reported no false positives or false negatives. Table 2 below illustrates the list of vulnerabilities as well as injection attacks exploiting these vulnerabilities.

Table 2: Different types of attacks used in our evaluation

Attack Type	**Attack Description**	**Detected or Undetected**
Tautology	Injecting one or more conditional statements	Yes
Logically incorrect queries	Information gathering, extract data	Yes
Union queries	Return data from a different table	Yes
Piggy-backed queries	New queries within the original query without changing the logic of the first one	Yes
Stored procedure	Invoking stored procedure	Yes
Inference	Infer answers from applications response	Yes
Alternative encoding	Injecting modified control text	Yes

These vulnerability and attack types cover the most known SQLIA available in literature (Halfond et al., 2006). The combination of these attack types can be combined thus making new attacks possible. However, our technique can detect all forms of attacks irrespective of their nature or combinations.

Generation of Test Inputs

For each application in the testbed, there are two sets of inputs: LEGIT, which consists of legitimate inputs for the application, and ATTACK, which consists of attempted SQLIAs. This is shown in the table below.

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

Table 3: Set of legitimate and attacks used

Subject	Total No. of Attacks	Successful Attack	Legitimate Attack
Bookstore	6,154	1, 999	607
Portals	6, 403	3, 016	1, 080
EmplDir	6, 398	2, 066	658
Classifieds	5, 968	1, 973	574
Events	6,207	2, 141	900
Checkers	4,431	922	1,359
Office Talk	5,888	499	424

The result of this attack strings contained 30 unique attacks that had been used against applications similar to the ones in the testbed.

Effectiveness of SQLDefend
We use a Pentium® Dual-core 2.10GHz processor with 2GB RAM and 64-bit system architecture running Windows 7. We created sample database in MySQL server 5.0. The J2EE application bundled into Netbeans software was used. To match real world application, JSP (Java Server Page) was compiled into servlets. The web server, database server, and client were in same local machine.

RESULTS

Table 4: The number of false positives and false negatives detected

Subject	Total No. of Attacks	No. of Legitimate Accesses	False Positives	False Negatives
Bookstore	6,154	607	3	2
Portals	6, 403	1,080	5	3
EmplDir	6, 398	658	3	1
Classifieds	5, 968	574	2	2
Events	6,207	900	3	0
Checkers	4,431	1,357	6	3
Office Talk	5,888	424	1	1
Total	**41,449**	**5,602**	**23**	**12**

The table shows that out of 5,602 legitimate accesses, there are 23 false positives representing 0.0041%. Our result also shows 12 false negatives representing 0.00029%. These are quite high considering the damage effect they can cause. To ensure that this situation is brought under control, we enabled our second tool, which is the decision tree classifier. We run the application again, this time we discover no false positives. This shows that the decision engine was able to testbed queries accurately as legitimate and malicious. The table below shows the result of our experiment when the decision engine was enabled.

Table 5: No false positives and false negatives after using SQLDefend

Subject	Total No. of Attacks	No. of Legitimate Accesses	False Positives	False Negatives
Bookstore	6,154	607	0	0
Portals	6, 403	1,080	0	0
EmplDir	6, 398	658	0	0
Classifieds	5, 968	574	0	0
Events	6,207	900	0	0
Checkers	4,431	1,357	0	0
Office Talk	5,888	424	0	0
Total	**41,449**	**5,602**	**0**	**0**

The result in table clearly shows that there are no false positives. This is an improvement over previous techniques that used only parser as the only tool for detecting and preventing SQL injection attacks.

Classification Accuracy of Parser and Decision Tree

Our technique uses parser and decision tree classifier to detect and prevent SQL injection attacks without generating false positives and false negatives. However, there are some penalties to be paid. First, using parser, it is hard to predict accurately the structure of intended SQL. Secondly, there is additional runtime analysis overhead in terms of execution time which cannot be avoided due to the sequential nature of the analysis technique. Also, using decision tree, it is possible to experience over-fitting especially if the tree is too large; a situation which could cause noisy data. As a result, we try to avoid this problem by using rule-based approach instead of using data-based approach in building our decision tree. Also, decision tree is good for very large volume of data and if the data are too small, certain information may be lost which could lead to misclassification of data. Using the rule-based approach, we were able to correct this problem. Thus there was no misclassification of data based on our result since no malicious attacks were reported after our experiment and the issues of false positives and false negative were completely brought under control.

Complexity Analysis and Optimization

In this section, we discuss the time and space complexities in processing each query. That is, the time it takes to process each query and the storage space occupied by e query in the computer memory. We also discuss the worst case scenario in which it will take a query to be processed. We then consider the issue of queries that can be clustered such as having a SELECT + UPDATE queries that are concatenated. We discovered that such queries can have redundant information that could cause more memory utilization thus slowing down the machine and thus increasing the processing time. To eliminate this problem, we optimize all the 30 distinct set of queries as identified in AMNESIA testbed. We did this by further grouping the queries that have the same query structure. The result is shown in table 6 below.

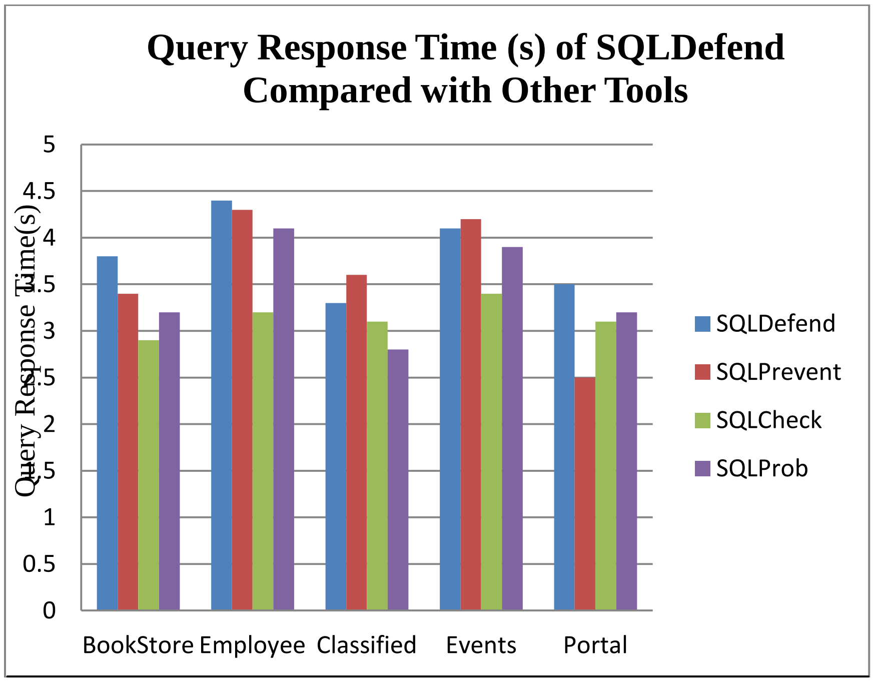
Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41-58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

Table 6 Percentage of reduction after query optimization

Subject	**No of queries Before optimization**	**No of queries After optimization**	**% of reduction**
Bookstore	320	63	19.6%
Portals	467	77	16.5%
EmplDir	295	36	12.2%
Classifieds	280	27	9.6%
Events	315	23	7.3%
Checkers	436	43	9.9%
Office Talk	214	24	11.2%

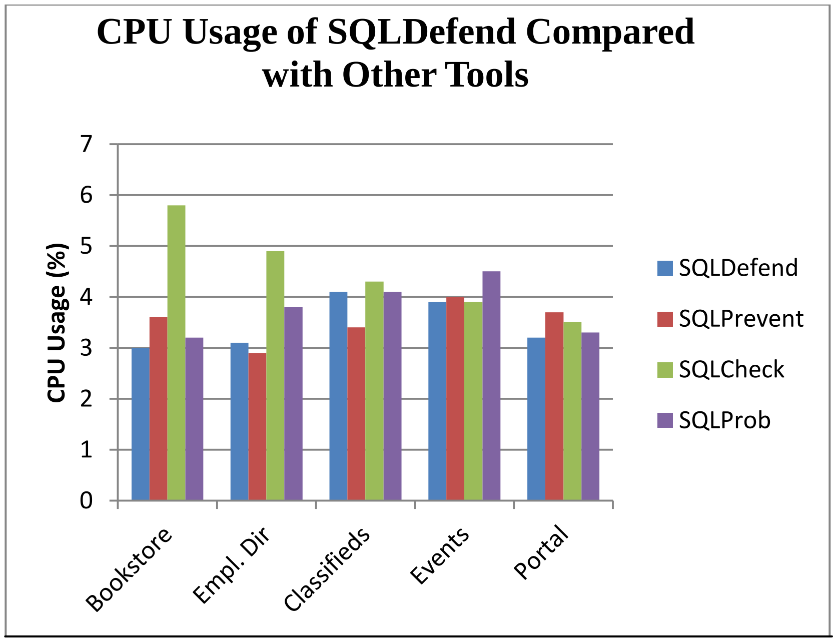
Time Complexity

We measured the time it takes a query to from the time a query is submitted to the time the result is returned to the user, which is known as the round trip time processing (RTTP) using ($O(n^3)$); where n is the number of keywords in a query. This time was measured in millisecond (ms) and we discover a significant improvement when we compared our result with previous techniques. The graph in figure 4.1shows the result of some other techniques we compared our results. The result validates the effort put into improving the performance of our technique.



Space Complexity

We also measured the space complexity of our technique and found out that there is significant improvement is our technique when we compared our results with. This is due to the fact that our algorithm performs better thus reducing the overhead incurred by the CPU when compared with previous techniques. The space complexity is measured using ($O(n^2)$). The graph in figure 4.2 shows the space complexity of SQLDefend when compared the other well-known techniques.



DISCUSSION

As seen in table 4, when only parser is used as the only tool for detecting and preventing SQL injection attack, there are 23 false positives out of 5,602 legitimate accesses representing 0.0041% of the total accesses. Though this percentage is very small, it could cause a lot of great trouble to a database if sensitive information is returned to a malicious user whose intention is to have access to sensitive information that could be used for theft such as credit card numbers. The table also shows that the number of false negatives is 12 representing 0.00029% indicating that when parser is used to detect and prevent SQL injection attacks, it produces false positives and false negatives.

However, to solve the problem of false positives where legitimate queries may be classified as malicious queries thus preventing genuine users from having access to a web site; and false negatives where malicious queries are classified as legitimate queries, we used a machine learning tool called decision tree classifier to further classify the queries correctly. The result in table 5 shows the outcome when the program was further tested. As seen in the table, there are no false positives and false negatives. This clearly shows that our technique is very effective in detecting and preventing SQL injection attacks.

As noted in the introduction of this paper, injection attacks are one of the largest classes of security problems till date. Code injection attacks continue to be a major threat to computer systems. In this paper, we proposed SQLDefend, a technique for detecting and preventing SQL injection attacks with the capabilities of securing Web applications against intruders and hackers. SQLDefend dynamically uses user input and analyze them syntactically by comparing the resulting parse tree with the dynamically generated in our system. It further check a query using decision tree classifier taking into account the context of every user input. Our approach is very modular and can be deployed to existing Web applications without extensive modifications.

Scientia Africana, Vol. 11 (No.2), December 2012. Pp 41- 58
© Faculty of Science, University of Port Harcourt, Printed in Nigeria *ISSN 1118 - 1931*

We conducted experiments and measured the overhead incurred by our technique. The result showed that our technique provide good protection against SQL injection attacks. Our experimental result shows that our approach provides a complete automated protection against SQL injection attacks with a minimal amount of overhead. To ensure that our technique keep the overhead to a minimum level, we explored a number of optimization using query reduction and thus calculating the percentage of query reduction in each of the subjects in AMNESIA testbed. We also measured the number of false positives and false negatives and our result shows that there are no false positives and false negatives.

REFERENCES

Abdelhalim, A. and Traore, T. (2009). *Converting Declarative Rules into Decision Trees.* In proceedings of the World Congress on Engineering and Computer Science, WCECS'09, Vol. 1, October 20-22, 2009, Sanfrancisco, USA.

Ali, S. Rauf, A. and Javed, H. (2009). *SQLIPA: An Authentication Mechanism Against SQL Injection*. In European Journal of Scientific Research, ISSN 1450 – 216X, Vol. 38, No. 4, pp. 604 – 611, http://www.eurojournal.com/ejsr.htm.

Asagba, P. O. and Ogheneovo, E. E. (2011). *A Proposed Architecture for Defending Against Command Injection Attacks in a Distributed Network Environment.* In Proceedings of the 10th International Conference of Nigerian Computer Society on Information Technology for People-Centred Development (ITePED 2011), Vol. 22, pp. 99-104.

Batory, D. Lofaso, B. and Smaragdakis, Y. (1998). *JTS: Tools for Implementing Domain-Specific Languages.* Int'l Conference on Software Reuse (ICSR'98), IEEE Computer Society, pp. 143-153.

Bisht, P., Madhusudan, P. and Venkatarishnan, V. N. (2010). *CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks*, ACM Transactions on Information and System Security, Vol. 13, No. 2, Article 14.

Boyd, S. W. and Keromytis, A. D. (2004). *SQLRand: Preventing SQL Injection Attacks*. In Proceedings of the 2nd International Conference of Applied Cryptography and Network Security (ACNS'04), Yellow Mountain, China, pp. 292 -302.

Bravenboer, M., Dolstra, E. and Visser, E. (2007). Preventing Injection Attacks With Syntax Embeddin*gs*. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE'07.

Bravenboer, M. (2008). *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*, PhD Thesis, Utrecht, University, Utrecht, The Netherlands, pp. 65, 92.

Buehrer, G. T., Weide, B. W. and Sivilotti, P. A. G. (2005). *SQLGuard: Using Parse Tree Validation to Prevent SQL Injection Attacks*. In Proceedings of the 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal, pp. 106 – 113.

Gould, C., Su, Z. and Devanbu, P. (2004).

Halfond, W. G. J. and Orso, A. (2005). *Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks*. In Proceedings of 3rd International Workshop on Dynamic Analysis (WODA'05), St. Louis, Missouri, pp. 1-7.

Halfond, W. G. J. and Orso, A. (2005). *AMNESIA: Analysis and Monitoring for Neutralizing SQL Injection Attacks.* In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, California, USA, pp. 174 – 183.

Halfond, W. G. J., Viegas, J. and Orso, A. (2006). *A Classification of SQL Injection Attacks and Countermeasures*. In Proceedings of IEEE International Symposium on Secure Software Engineering.

Halfond , W. G. J., Orso, A. and Manolios, P. (2006). *Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks.* In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering SIGSOFT'06/FSE-14, Portland, Oregon, U.S.A., pp. 175 – 185.

Huang, Y.-W., Yu, F, Hand, C., Tsai, C.-H., Lee, D,-T., and Kuo, S.-Y., 2004). *Securing Web Application Code by Static Analysis and Runtime Protection.* In Proceedings of the 13 International Conference on World Wide Web, New York, NY, pp. 40 – 52.

Imam, I. F. and Michalski, R. S. (1992). *Should Decision Trees be Learned from Exmaples or from Decision Rules?* In Proceedings of the 7th Int'l Symposium on Methodologies, Vol. 689, pp.395-404.

OWASP (2008). Open Web Application-Top-Ten-Projects.

Su, Z. and Wassermann, G. (2006). *The Essence of Command Injection Attacks in Web Applications.* In Conference Record of the 33rd ACM SIGPLAN—SIGACT Symposium on Principles of Programming Language POPL'06, New York, NY, pp. 372 – 382.

Sun, S.-T. and Beznosov (2008). *SQLPrevent: Effective Dynamic detection and Prevention of SQL Injection Attacks without Access to the Application Source Code.* Laboratory for Education and Research Secure systems Engineering, University of British Columbia, Vancouver, Canada, LERSSE-TR-2008-01.

Visser, E. (2002). *Meta-programming with Concrete Object Syntax.* In Generative Programming and Component Engineering (GPCE'02), Vol. 2487 of LNCS, Pittsburgh, PA, USA, pp. 299-315.

Wei, K., Muthuprasama, M. and Kothari, S. (2006), *Eliminating SQL Injection Attacks in Stored Procedures.* In Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06), pp. 191 – 198.