A Parse Tree Model for Analyzing And Detecting SQL Injection Vulnerabilities

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

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 brought about the increased popularity of web based applications. This upsurge 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 developed a model based on grammatical structure of an SQL statement using parse tree to test a query by dynamically generating a parse tree and comparing their structures at runtime. We were able to determine if their structures match or not. If they match, the query is parsed signifying that it is legitimate, otherwise it is suspicious and possibly malicious. Our result shows that the parser detected and prevented malicious SQL queries although there were a couple of false positives and false negatives representing 0.01% of legitimate attacks. This result is good enough because achieving 100% security precision may be too difficult. However, we hope to improve on this result in our future research.

Keywords: SQL injection attacks, parse tree, web applications, attacker.

1.0 Introduction

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, online stores, social network services, e-governance, etc., has brought about the increasing popularity of web based applications. This upsurge has made the Internet a potential target for different forms of attacks [10]. 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.

Attacking Web applications by injecting SQL commands was first described as early as 1998 [17]. Since 2002, over 50% of total cyber vulnerabilities were input vulnerabilities [19]. SQL injection attack is a class of command injection attacks in which specially crafted input string result in illegal queries to a database [20]. This has become one of the most serious threats to Web applications [14]. An SQL injection attacks targets interactive Web applications that employ database services. Such applications accept user input in database requests, typically SQL statements. In SQL injection attacks, the attackers provide user input that results in a different database request than was intended by the application programmer [21].

SQL injection vulnerabilities (SQLIVs) account for 20% of the total cyber vulnerabilities since 2002 [18]. An SQLIV allows input to an SQL statement to change the structure of the statement and allows malicious users to gain unauthorized access to information in a database. As the trend of providing Web-based services continues, the prevalence of SQLIVs is likely to increase [11], [12]. Another concern facing the software development industry is that the number of developers inexperienced in software security outnumbers the number of ixperienced software security practitioners [3]. The implication is that significant portion of developers fixing SQLIVs will not be experienced with solving security issues [19]. The Open Web Application Security Project [22] report places injection attacks including SQLIAs as the most likely and damaging.

The most widely deployed defense technique today is to train the programmer and web-developers about the security implications of their code and to teach them corrective measures and good programming practices. However, rewriting or revising all or most of the existing legacy codes is quite a difficult task as it requires lots of hard work, commitments and this will incur additional cost to any organization that may want to embark on such projects. There is therefore the need to develop an automated technique that will guarantee the detection of these vulnerabilities and a fool-proof elimination of SQL injection attacks.

Many techniques have been proposed, these are either static or dynamic. These techniques have failed to address the full scope of the problem. There are many types of SQLIAs and countless variations of these basic types. Therefore, many of the proposed solutions only detect or prevent a subset of the possible SQLIAs [11]. In this paper, we would develop an automatic technique to counter SQL attacks and/or prevent attacks. Our approach combines both the static and dynamic approaches of AMNESIA proposed by [12] and SQLCheck proposed by [24].

2.0 SQL Injection Attacks (SQLIAs)

SQL injection is an attacking technique which is used to pass SQL comments through a web application directly to the database by taking advantage of insecure code's non-validated input values. An SQL Injection Attack (SQLIA) is a subset of the unverified or unsanitized input vulnerability and occurs when an attacker attempts to change the logic, syntax, or semantic of a legitimate (benign) SQL statement by inserting new SQL keywords or operators into the statement [21]. SQL injection in web applications works using the dynamically-generated SQL queries. The root cause of SQLIAs is insufficient input validation. SQLIAs occur when data provided by a user is not properly validated and is included in an SQL query [13]. 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 [5] SQL flaw can lead to e.g., unauthorized access, data manipulation, or information disclosure.

Normally, 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 [1]. Whenever a user wants to enter into the web database through application tier, the user inputs his or 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 authentication 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:

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

Analyzing the above query, the result would always be true. It 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.

SQL injections can be very dangerous for the integrity of web applications. With SQL injection, an attacker can access a database, change information stored in it, delete information, and can even have full control of a database. SQL injection attacker uses multiple statement method to insert his SQL command into the general query string. SQL injection are very prevalent, and ranked as the second most common form of attack on web applications for 2006 in CVE (Common Vulnerability and Exposures). The percentage of these attacks among the overall number of reported attacks rose from 5.5% in 2004 to 14% in 2006 [26]. The 2006 SQLIA on CardSystems solutions that exposed several hundreds of thousands of credit card numbers is an example of how such attack can victimize an organization and members of the general public. Analysts have found several application programs whose sources exhibit these vulnerabilities. Several reports suggest that a large number of applications on the web are indeed vulnerable to SQL injection attacks [20] and the number of the attacks is on the increase.

The most common type of SQL injection attacks is SQL manipulation. The attacker attempts to modify the existing SQL statement by adding elements to the WHERE clause or extending the SQL statement with set of operators like UNION, INTERSECT, or MINUS, etc. The classic SQL manipulation is during the login authentication. Virtually all Web applications usually check user authentication before granting users access to the database. Usually, when a user submits a query to a database; the web application check user authentication by executing SQL statement. For instance, the following query may be executed:

SELECT * FROM users WHERE username = 'eddy' and passwd = 'password'

The attacker may attempt to manipulate the SQL statement to execute as follow.

SELECT * FROM users WHERE username = 'eddy' and passwd = 'password' OR 'a' = 'a'

In this case the always true for every row and the attacker will automatically gain access to the application.

3.0 Methodology

The architecture of the proposed model is shown in figure 1. Usually, when a user input an SQL statement or query through a web application, the query is collected by the query collector and the user extractor extracts the query for processing in the parse tree generation phase when the query has been evaluated. The extracted query is then sent to the parse engine where it is compared with the one that is dynamically generated at runtime by our model to see if the structures are syntactically the same. If they are, the query will be allowed into the database otherwise it will be rejected and blocked

.

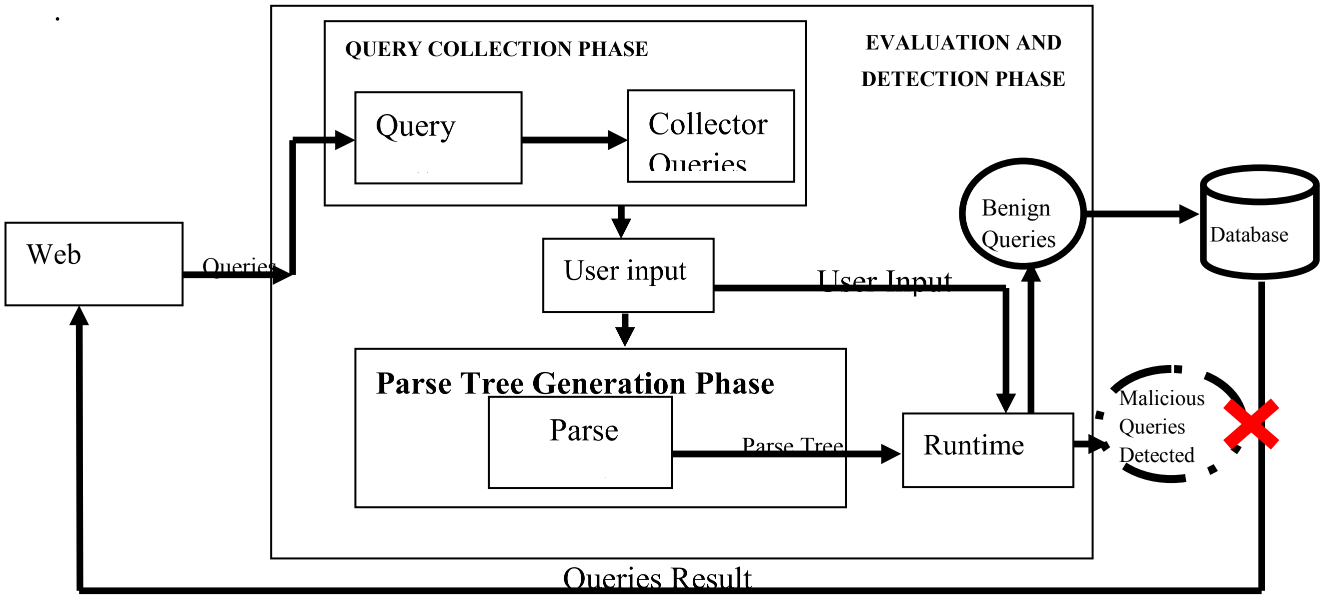

Fig. 1: Architecture for proposed model

A parse tree is a data structure for representing a parsed statement. 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 intended 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 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.

3.1 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 2 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 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.

We collected various SQL statements through web application. We also find the combination of these queries using the UNION, HAVING, ORDER BY clauses so as to have more complicated queries. We also collected queries which are stored procedures and alternate encoding which are very complex forms of queries. This is done to ensure that we have all the various forms of queries represented so that our technique will not be limited to solving only a subset of injection attacks. We also ensure that any possible combination of queries that an attacker can combine and use in future attacks are countered since it is a well known fact that just as security experts are finding ways to counter injection attacks, hackers will also be looking for new ways to hack well secured web sites. This we did by ensuring that certain query combinations are well verified and wherever keywords like UNION, HAVING, ORDER BY, LIKE, etc., are used in query combinations are first categorized as suspect and are well verified by the parser engine.

3.2 Parse Trees for SQL Statements

A parse tree is a data structure for representing a parsed statement. 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 3(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. In our technique, we combine the security of using Windows API. We did this by embedding the syntax of the guest language (MySQL) into the syntax of the host language. 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 [4], [25]. At the parser engine, the parser generated parse tree structures are compared at runtime and they are found to be syntactically the same, the query is then determined to be 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 classify 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 3 (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 3 (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. 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 3 (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 shown below in figures 3(a-e) represent 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.

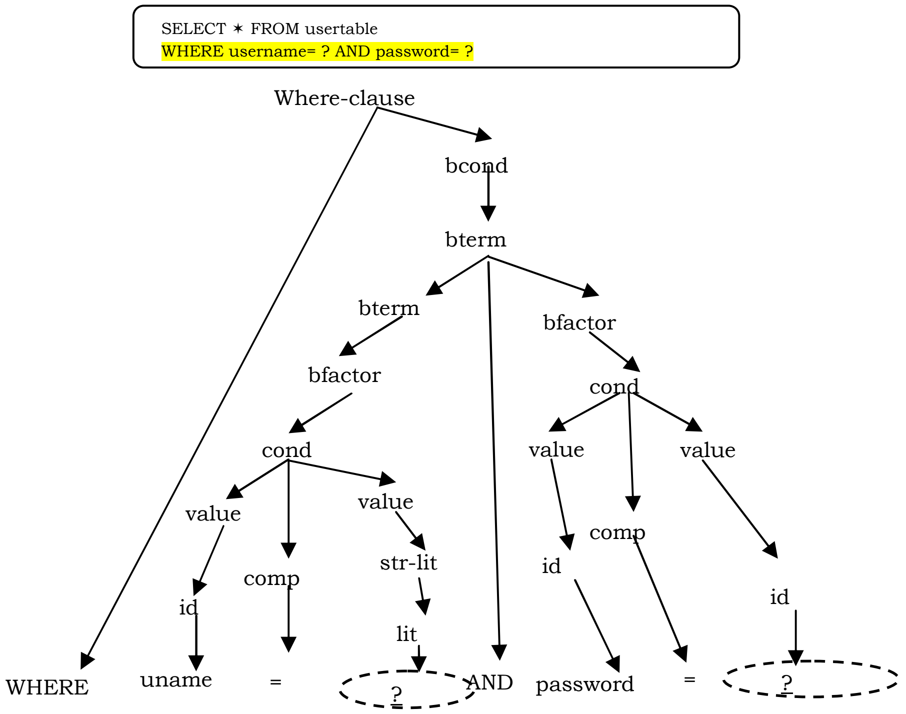

Fig. 3(a): A parse tree for a select statement. The username and password are not supplied

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 palceholders 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 2.

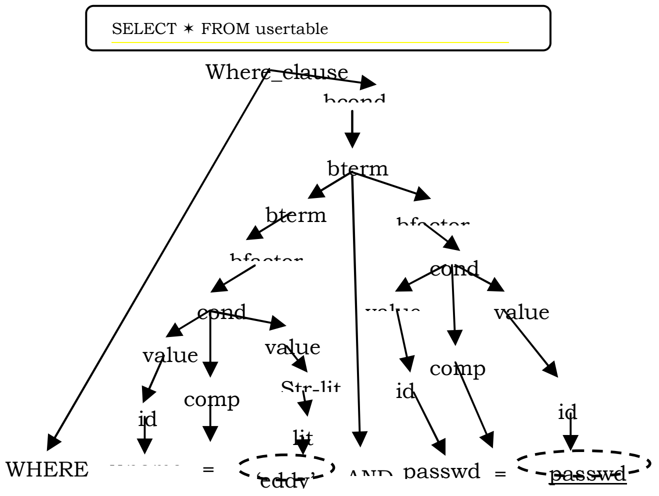

Fig. 3 (b) Benign select statement

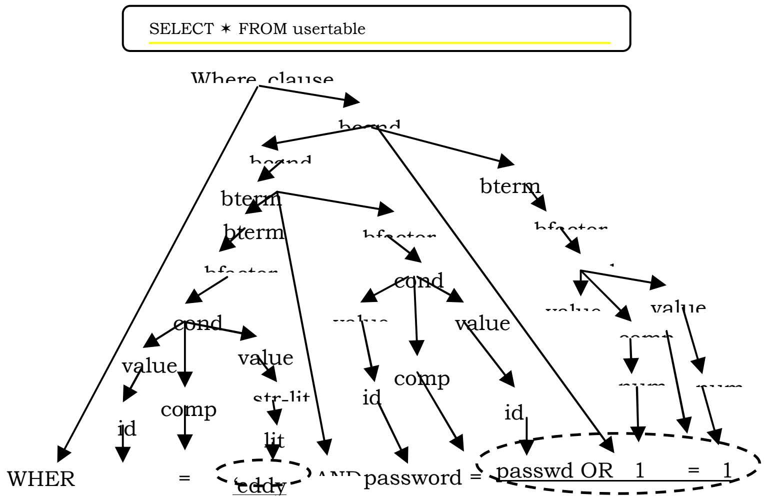

Fig. 3 (c) A malicious query

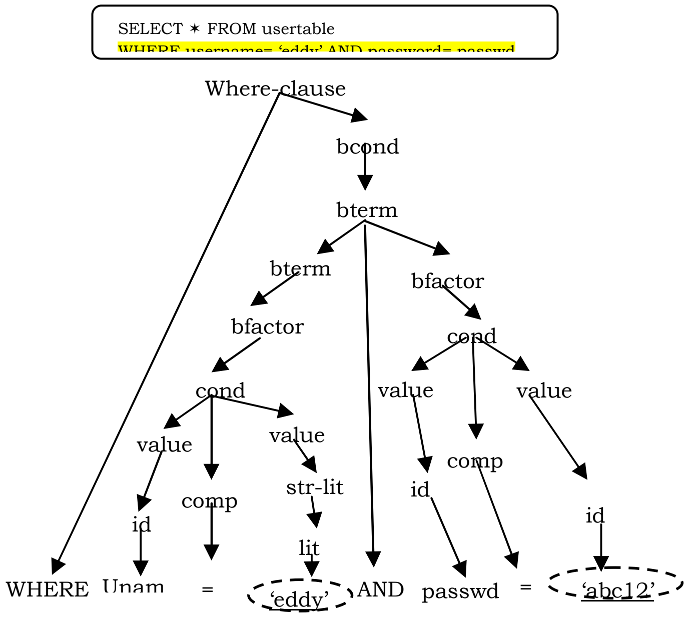

Fig. 3 (d) A benign query

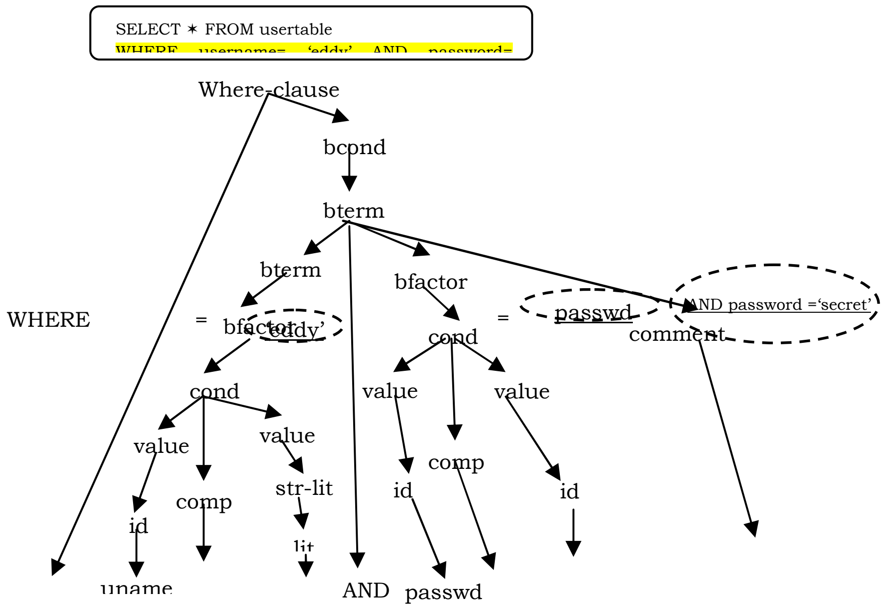

Fig. 3 (e) A malicious query

4.0 Experimental Setup

We used real world applications from AMNESIA testbed [12] 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 publicly 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
Classifieds	10,949	34	14
Events	7,242	31	13
Checkers	5,421	5	61
Office Talk	4,543	40	64

Our application demonstrates command injection attacks, where user-supplied command 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. The table below illustrates the list of vulnerabilities as well as injection attacks exploiting these vulnerabilities.

4.1 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 2 below.

Table 2: 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.

4.2 Evaluation

In our experiment, to ensure that our results are correct, we first disabled the decision engine. We then tested our technique against all legitimate and malicious queries. After testing, no false negatives were found but there are couples of false positives for each subject, which was tested.

The result shows that with the use of only parser as a tool, parser produces false positive but it produces no false negatives.

The table below illustrates the outcome of our experimental result

Table 3: 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 above shows that out of 41,449 total numbers of attacks, there are 23 false positives. This is approximately 0.0041% of total attacks. This is quite high. The reason for this is that if any of these attacks is very dangerous, it could cause serious damage to any individual or organization. Although, this result is good enough considering the fact that virtually all parser-based approaches used in the past have suffered from this same problem. In future, we hope to introduce another tool called decision tree classifier, a machine learning approach that will automatically classify queries into their respective groups (i.e., legitimate, malicious, and unclassified). This tool will be used in combination with the parser to correct the problem which the parser suffers from.

4.2.1 Discussion of Results

As seen in table 3, 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 0.41% 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 positives is zero (0) indicating that when parser is used to detect and prevent SQL injection attacks, it is very effective in curbing queries that are malicious in that it completely prevent them.

4.2.2 Complexity Analysis and Optimization

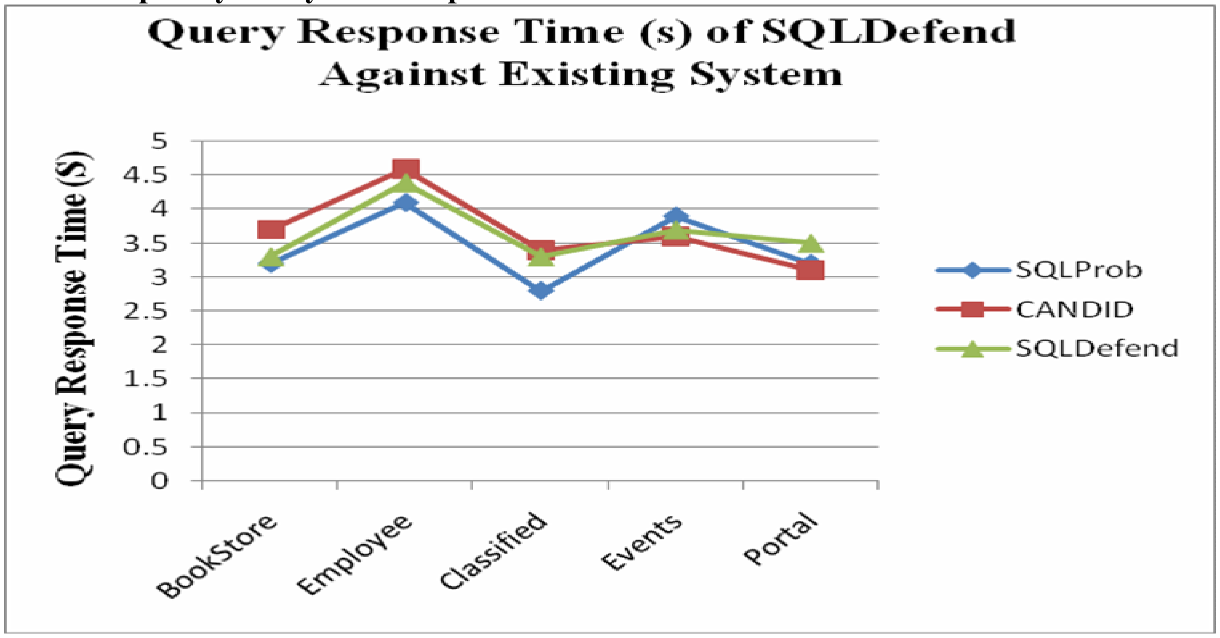

Fig. 4a

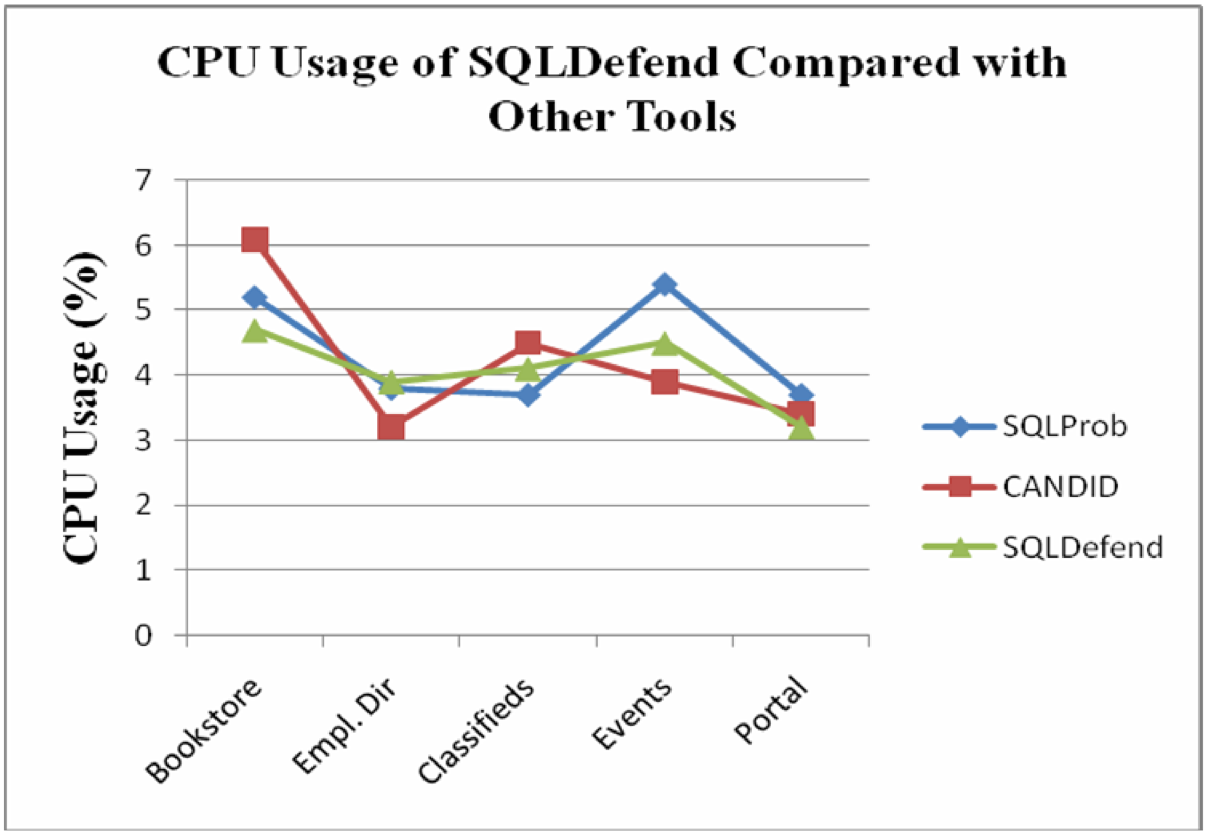

Fig. 4 b

5.0 Related Work

In [12] the authors used AMNESIA technique to secure vulnerable statement by combining static analysis with statement generation and runtime monitoring. They used static analysis of Java programs to compute a finite-state machine model that captures the lexical structure of SQL queries issued by a program. They analyzed the vulnerable SQL statement, then generate a general acceptable SQL statement model, and allow or deny each statement based on how it compares to the model at runtime. In the study they conducted, they used five real world Web applications and applied AMNESIA to each of the applications. SQL injection attacks cause SQL queries issued by the program to deviate from this model and were detected. Although the technique is effective because it detects injection attacks and it avoids runtime taint-tracking, it suffers some drawbacks. Their solution uses exceptions to indicate potential attacks which could cause overhead on the part of the developers. Also, the conservative nature of its static analysis and its inability to distinguish different courses of inputs can lead to a higher rate of false positives.

In [7], the authors proposed SQLGuard technique for detecting injection attacks. They use SQLGuard to secure vulnerable SQL statements by comparing the parse tree of an SQL statement before and after user input and only allow SQL statements to execute if the parse trees match. In their study, they used one real-world Web application for each application. They technique was able to stop all the SQLIAs after testing it and it generated no false positives. However, their technique had some overheads. First, the developer must rewrite all the SQL codes to use their custom libraries. This is quite a difficult time, consuming and costly task on the part of the application developers. There is also the problem of computational overhead due to dynamic statement validation by removing vulnerability and allowing all inputs. Therefore, SQLGuard is not flexible enough, because the source code of the application must be modified in many positions. This is a very tedious task on the part of the programmer which may be very difficult to achieve.

SQL Document Object Model (SQL DOM) technique was proposed by [16]. This is an API dependent stored procedure technique for detecting injection attacks. SQL DOM analyzes the database schema at compile time and writes codes to customize the SQL query construction classes. The resulting DOM is a tree-like structure based on a generic template, mapping the possible variations of SQL queries according to tables and column definitions. They used three (3) main classes, SQL statements, table columns and where conditions. These classes have strong-typed methods mapping the data types in the database schema. This enables them to validate data types automatically. The constructor of column classes escape strings (i.e., replace each quote by a double quote) at runtime to sanitize them. Although the approach was able to prevent application layer injection attacks, it however had some limitations. It has some overheads for developer training and code rewriting, as query-generating code needs to be rewritten. Its full-object criterion lead to additional cost. Also, since the technique uses stored procedures, it remains unprotected. The technique does not execute queries (it only generates them). While this could improve database integration and perhaps further reduce the attack surface, the technique neither describes its string sanitization strategy nor elaborates on exception handling and thus did not address how the SQL DOM would behave if a null value is passed on as a criterion.

In [24] the authors proposed SQLCHECK technique to prevent SQLIAs. Their approach employs context-free grammars for data validation. Data that is dynamically added to foreign code statements has to fulfill specifically constructed grammars. By tracking dynamically added values through the application's processes, SQLCHECK can identify un-trusted values before the query is parsed to the database. These values are parsed by the constructed grammar to validate their correctness. They analyzed the parse tree of the query, generated customs validation code, and then wrap the vulnerable statement in the validation code. They used five real-world Web applications in their study and applied their technique to each of the applications. Their wrapper stopped all of the SQLIAs in their attack set without generating false positives. However, the technique assumes the client will not be able to produce the magic marker symbol. This is very dangerous to assume since Web applications can "echo" SQL queries to the user if an error occurs, the user may trick the Web application into revealing its markers [6]. Also, the technique is still subject to denial-of-service attack. This is because, at runtime, it can only flag errors and prevents them from escalating into a full security compromise.

6.0 Conclusion and Future Work

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, online stores, social network services, e-governance, etc., has brought about the increasing popularity of web based applications. This upsurge has made the Internet a potential target for different forms of attacks. In this paper, 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 generated at runtime. We embedded the guest language (MySQL) into the syntax of the host language (Java). This is to avoid the problem of grammar ambiguities so that only one type of parse tree is generated for a particular query.

Our result shows that the parser was able to detect malicious queries, although it also produces false positives and false negative. This is approximately 0.01% of legitimate attack. This is quite high. The reason for this is that if any of these attacks is very dangerous, it could cause serious damage to any individual or organization. Although, this result is good enough considering the fact that virtually all parser-based approaches used in the past have suffered from false positives. In future work, we hope to introduce another tool called ***decision tree classifier***, a machine learning approach that will automatically classify queries into their respective groups (i.e., legitimate, malicious, and unclassified). This tool will be used in combination with the parser to correct the problem which the parser suffers from. This way we hope the problem of false positives and false negative will be solved.

References

[1] 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.

[2] Anley, C. (2002). Advanced SQL Injection, http://www.ngssoftware.com/papers/Advanced_sql_injection.pdf/, Accessed September 13, 2010.

[3] Barnum, S. and McGraw, G. (2005). *Knowledge for Software Security, Security and Privacy* Magazine, IEEE, Vol. 3, No. 2, pp. 74-78.

[4] 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.

[5] 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.

[6] Bravenboer, M., Dolstra, E. and Visser, E. (2007). Preventing Injection Attacks With Syntax

Embeddings. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE'07.

[7] 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.

[8] Das, D. Sherma, U., and Bhattacharyya, D. K (2010). *Approach to Detection of SQL Injection Attack Based on Dynamic Query Matching*, International Journal of Computer Applications (0975 – 8887), Vol. 1, No. 25, pp. 28–34.

[9] Fayol, E. M. (2005). *Advanced SQL Injection in Oracle Databases.* Technical Report, Argeniss Information Security, Black Hat Briefing, Black Hat, U.S.A.

[10] Halder, R. and Cortesi, A. (2010). *Obfucation-based Analysis of SQL Injection Attacks.* In Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT'10), Athens, Greece, pp. 254 – 265.

[11 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.

[12] 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.

[13] 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.

[14] Kim, H. K. (2010). *Frameworks for SQL Retrieval on Web Applications Security*. In Proceedings of the International MultiConference of Engineers and Computer Scientists, Vol. I, IMECS'10, Hong Kong.

[15] Mackay, C. A. (2005). *SQL Injection Attacks and Some Tips on How to Prevent Them.* Technical Reorp, The Code Project, January 2005, http://www.codeproject.com/cs/database/SQLInjectionAttacks.asp.

[16] McClure, R. A., Kruger, I. H. (2005). SQLDOM***:*** *Compile Time Checking of Dynamic SQL Statements*, ICSE'05, St. Louis, Missouri, USA, ACM, pp. 88 – 96.

[17] Nguyen-Tuong, A. S., Guarnieri, S., Greene, D., Shirley, J, and Evans, D. (2005). *Automatically Hardening Web Applications Using Precise Tainting*. In Proceedings of the 20th IFIP International Information Security Conference (SEC), pp. 295-308.

[18] NIST, National Vulnerability Database (2007) http://www.nvd.nist.org/, Accessed September 13, 2010.

[19] Ogheneovo, E. E. and Asagba, P. O. (2011***).*** *A Proposed Architecture for Defending Against Command Injection Attacks in a Distributed Network Environment*, Journal of Research in Physical Science, Vol. 7, No. 1, pp. 67-72.

[20] Ogheneovo, E. E. and Asagba, P. O. (2011). *A Machine Learning Technique for Detecting and Preventing SQL Injection Attacks*. Int'l Journal of Computer Science, Vol. 3, No. 1, pp. 111-123.

[21] Ogheneovo, E. E. and Asagba, P. O. (2012). SQLDefend: An Automated Detection and Prevention

Technique for SQL Injection Vulnerabilities in Web Applications, SCIENTIA AFRICANA, Int'l Journal of Pure & Applied Sciences, Vol. 11, No. 2, pp. 41-58.

[22] Open Web Application Security Project, OWASP 2010, http://www.owasp.org/images/0/0f/OWASP_T10_2010_rcl.pdf, Accessed September 22, 2010.

[23] Spett, K. (2003). *Blind SQL Injection.* White Paper, SPI Dynamics, Inc., http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf.

[24] 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.

[25] 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.

[26] Common Vulnerability and Exposures (CVEs), http://cve.mitre.org, accessed September 20, 2011.