A PROPOSED ARCHITECTURE FOR DEFENDING AGAINST COMMAND INJECTION ATTACKS IN A DISTRIBUTED NETWORK ENVIRONMENT

Asagba P.O.[1] and *Ogheneovo E.E.[2]
Department of Computer Science,
University of Port Harcourt, Port Harcourt, Nigeria.

ABSTRACT

Previous research works on SQL (Structured Query Language) injection attacks (SQLIAs) have focused on detecting and/or preventing a subset of the problem and are language dependent. The goal of this paper is to develop a model that can detect and prevent all forms of SQLIAs including stored procedures and alternate encoding which are very complex forms of the attacks. Our technique which we called SQLDefend is different from the previous approaches in that we combine a parser and decision trees. The parser will parse the query and the decision tree will automatically learn and recognise malicious queries irrespective of their types or even combination of queries. Although we would use Java programming language and MySQL for its implementation, our technique is language independent and can be implemented in any programming language.

Keywords: Distributed Network, Queries, SQLIAs, SQLDefend and Web applications

1.0 INTRODUCTION

The World Wide Web has experienced remarkable growth in recent times. Businesses, individuals and governments have found that web applications can offer efficient and reliable solutions to challenges of communicating and conducting commerce in the 21st century. Various corporate bodies whose business models completely focuses on the Web like Google, Yahoo, Amazon, etc., have taken Web interactions to newer heights. As many enterprise applications dealing with sensitive financial and medical data turn on-line, the security of such Web applications has come under close scrutiny (Wei et al., 2006). Nearly all Web applications are security critical, but only a small fraction of deployed Web applications can afford a detailed Security review (Nguyen-Tuong et al., 2005). Serious security vulnerabilities have been found in the largest commercial web applications. This has made these applications and their databases vulnerable to attacks thereby exposing these databases that contain confidential and sensitive information (Halfond and Orso, 2005). These attacks are called command injection attacks.

Command injection attack is an attacking technique in which an attacker alters dynamically generated contents of a Web page by entering code into an input mechanism, such as a form field that lacks efficient validation constraints. Whenever users visit such a Web page, their browsers interpret the code, which may cause malicious commands to execute the user's computers and across their networks. In recent years, there has been an increase in attacks against Web applications (Halfond et al., 2005). Theere are several forms of command injection attacks which include cross-site scripting, buffer overflow, path traversal, HTTP response splitting, shell command injection, and SQL injection attacks. Of these attack types, SQL injection attacks are the most prevalent command injection attacks (Bravenbor et al., 2007).

SQL injection attacks (SQLIAs) are one of the most foremost threats to Web applications (Halfond et al., 2006 and Bravenbor et al., 2007). They constitute the largest classes of security problems. According to OWASP Foundation, injection attacks, especially SQL injection, were the most serious type of web application vulnerability in 2008 (OWASP, 2008). The threats posed by SQLIAs go beyond simple data manipulation, escalate privileges, executes a denial-of-service (DoS) attack, or execute remote commands to transfer and install malicious software. Due to SQLIAs, part of or even an entire organizational IT infrastructure ca be compromised. Therefore, an effective and easy to display method for protecting existing Web applications from SQLIAs is very crucial for the security of today's organizations. Detecting or preventing SQLIAs is a topic of active research in computer security. State-of-the-pracice SQL countermeasures are far from effective. OWASP (2010) notes that almost all Web applications from SQLIAs is very crucial for the security of today's organizations. Detection or prevention of SQLIAs is a topic of active research in computer security. State-of-the-practice SQL countermeasures are far from effective. OWASP (2010) notes that almost all Web applications deployed today are still vulnerable to SQL injection attacks. Signature-based Web application firewalls which act as proxy servers filtering inputs before they reach Web applications and other network-level instruction detection methods may not be able to detect SQLIAs that employ evasion techniques (Maor and Shulman, 2005).

Some approaches have claimed to be 100% efficient. Halfond et al., (2005); Buehrer, et al., (2005); Su and Wassermann, (2006); Halder and Cortesi, (2010), proposed static and/or dynamic analysis techniques. Dynamic taint analysis (Nguoyen-Tuoug *et al*., (2005); Pietraszek and Berghe, (2005) or even machine learning models (Valuer et al, 2005). Runtime environment modification Pietraszek and Berghe, (2005) limit the adoption of these techniques in some real-world situations. Moreover, a common deficiency of existing SQLIA approaches based on analyzing dynamic SQL statements is in defining SQLIAs too restrictively, which leads to a higher than necessary percentage of false positives (FPs). False positives could have significant negative impact on the utility of detection and prevention mechanisms, because investigating them takes time and resources (Werlinger *et al*., 2008). There is therefore the need to develop an effective technique for detecting and preventing SQLIAs. The major goal of this research is to apply deterministic finite automata and parser techniques to develop a model that will detect and prevent SQL injection attacks. In order to be able to achieve this, the following main objective is defined: to accurately model a technique that can detect and prevent SQL injection attacks.

2.0 RELATED WORK

Analysis and Monitoring for Neutralizing SQL Injection Attacks (AMNESIA) was proposed by Halfond et al. (2005) as a technique in which they combine static analysis with statement generation and runtime monitoring of a query and declare it valid or malicious. AMNESIA checks query in different steps by first identifying the hotspots (these are application codes), then form a model for legitimate query in the form of Non-Deterministic Finite Automata (NDFA) and finally check the query with NDFA and declare it legitimate or malicious. Nodes in the automaton are terminal in the language, and special nodes represent external user input. At runtime, each full SQL query is matched against the automaton. If there is an injection, the query will almost certainly not be accepted by the automaton as additional terminals are present that do not occur in the automaton.

The model generates an alarm whenever malicious queries are discovered.

Buehrer et al. (2005) secure vulnerable SQL statements by comparing the parse tree of an SQL statement before and after input and only allowing an SQL statement to execute if the parser trees match. They conducted a study using one real world web application and applied their SQLGuard solution to each application. However, their solution though effective in stopping SQL attacks, required the developer to rewrite all of their SQL code to use their custom libraries with code generation. There is also the problem of computational overhead in term of time and space complexities. WebSSARI approach proposed by Huang et al. (2004) secured Web application codes using a combined static analysis and runtime monitoring to secure potential vulnerabilities. Their technique statically analyzed source code, finds potential vulnerabilities, including SQLIVs, and inserts runtime guards into the source code to sanitize inputs. Input from external source is used to detect vulnerable data. The runtime protection is used to determine whether input is valid or not.

Su and Wassermann (2004) propose SQLCheck model which statically analyzed SQLIA by generating finite state automata. They use this approach to model set of valid SQL commands for each data access. This approach is based on Context-Free-Grammars (CGFs) for validating data. They use this approach by wrapping user input in special markers, e.g., (|a|). The grammar of the guest language is then augmented to accept the markers by using some symbols in the grammar, for instance, so that it accept (|'a'|) in SQL whenever a string literal is accepted. This way, an injection attack would then fail to parse. For example, SELECT*FROM customer WHERE userid =John AND passwd=(|"OR 'a'='a' "|), there is no production that allows an arbitrary condition inside the markers. However, it is wrong to assume that markers will not be leaked since Web applications can "echo" SQL queries to the user if an error occurs. Thus, it may be quite easy for the user to trick the Web application into revealing its markers.

Bravenboer et al. (2007) uses syntax embeddings to prevent SQL Attacks. This they called SpringBord. In their work, they made grammatical structure of an SQL query explicit such that any resulting sentence can conform to it. They parsed the guest language fragments as a pre-processing step and generated code so that resulting sentence is in conformity with the SQL query. They constructed sentences structurally with respect to the Context-Free Grammar (CFG) of the guest language. The approach embedded the syntax of the guest language (e.g., SQL) in the host language (e.g., Java, PHP). They used this combine syntax for writing programs. The system architecture consists of assimilator that they used to parse source field and transform the guest code to invoke an API that helps in managing the composition, escaping, and serialization of the guest code sentences. The API generator was used to produce an API of the specified host language. This generated API thus helps to prevent injection attacks by always checking lexical values against the syntax for the lexical category in the syntax definition. Thus it is impossible for a malicious user to provide an input that matches the programmers intended input. Therefore, the technique is generic in nature to both in the guest and host languages. However, the technique has a lot of drawbacks. For one, it cannot detect injection attacks and cannot be used to redesign generic codes. Also, the technique does not protect against semantic injection attacks. The quotations and anti-quotations may not necessarily replace most current patterns of attacks without much refactoring of the host program. The technique allows only the conversion of strings to literals and not keywords, i.e., it cannot capture variable attacks.

SQLRand technique proposed by Boyd and Keromytis (2004) uses secret keys to counter SQL injection by randomizing and de-randomizing every SQL keyword.

The technique modifies the token of the SQL language. Each token type includes a prepped integer. Any additional SQL query supplied by the user such as ("OR 1=1"), would not match the augmented SQL tokens, and would throw an error. This is a proxy mechanism that filters SQL statement and it is introduced between the application and the database which helps to parse every query by looking at the syntactic structure of the query to see if it conforms to the syntax of the guest language. Since an attacker does not know the current syntax of the query, a command injection attack will produce a parsing error and will therefore be rejected or stopped from getting to the database.

JDBC-Checker proposed by Gould et al. (2004) statically type-check the correctness of dynamically generated SQL queries. The technique was not initially developed for detecting and preventing general SQLIAs, but it can be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker was able to detect improper type-checking of inputs, which is one of the major causes of SQLIA vulnerabilities in codes. However, the technique could not detect or prevent more general forms of SQLIAS because most queries consist of syntactically and type-correct queries.

3.0 METHODOLOGY

3.1 Developing SQLIA Model Features

In developing a model to counter SQL injection attacks, we discussed how the attack works. SQL injection in web applications works using the dynamically-generated SQL queries. it occurs when data provided by a user is not properly validated and is 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). normally, web application is a three-tier architecture: the application tier at the user side, the middle tier which converts the user queries into SQL format, and the backend database server which stores the user data as well as the user's authentication table (Ali et al, 2009). Whenever a user wants to enter into the web database through the 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 WHERE
user_name='username' AND
passwd='password'

If the query result is true, then the user is authenticated otherwise it is denied. But attackers can exploit weakness in security lapses of a database and enter malicious code which will always return true results. For example, the attacker can enter the expression "' OR 1=1-- '". So the middle tier will convert it into SQL query format as shown below. This then deceives the authentication server. The query result will be:

SELECT * FROM user WHERE
user_name=' 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 "' OR 1=1-- '" statement appended to the SQL 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 the password.

The set operator UNION is also frequently used in SQL tacks. The goal is to manipulate a SQL statement into returning rows from another table. For instance, a Web form may execute the following query to return a list of available products.

SELECT product_name FROM all-products
WHERE product_name like '%chairs'

An attacker can attempt to manipulate the SQL statement to execute as:

```
SELECT product_name FROM all_products
WHERE product_name like '%chairs'
        UNION
        SELECT username FROM dba-users
WHERE username like '%'
```

In this case, the list that will be returned to the Web form will include not only the selected products, but also all the database users in the application.

Therefore, in order to properly detect SQLIAs, we would consider a set of 300 benign queries and 300 malicious queries and from combination of types of queries. The 300 malicious queries would be collected from all known types of SQL injection attacks and we will test run them on our model. The result will be checked for false positive for benign queries and false negative for malicious queries. The decision engine would be used to train automatically learn and recognise SQL injection attacks after the parser has compared the resulting parse trees.

3.2 Designing SQL Injection Detection and Prevention Model

The model as seen in figure 1 uses three phases: the query collection phase, the query generator phase, and the query evaluation and detection phase. In the collection phase, query is collected by the query collector and stores them in a repository. In the parse tree generator phase, the generated query stored at the proxy is the sent to parser engine. The query is then analysed statically and a parse tree is generated and compared with the dynamically generated parse tree of the host language. If there is a match then the decision engine will decide whether to accept the query or not. Once the query has been validated, it is allowed to get to the database and the result of the query or web page if found is displayed on the web application for the user.

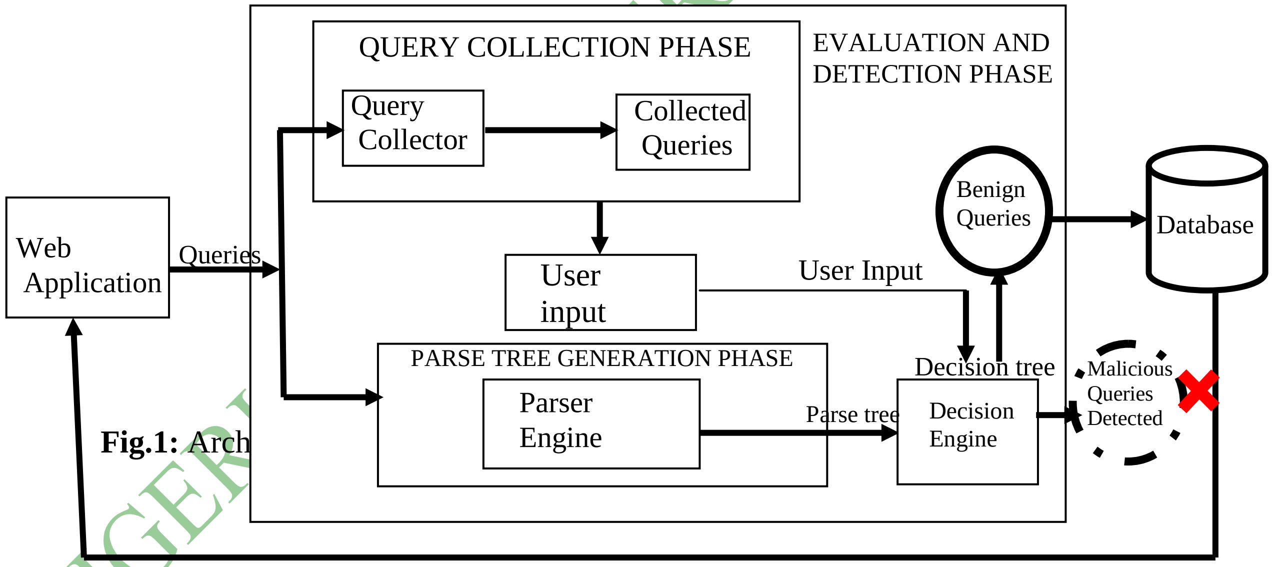

Fig.1: Arch

4.0 EXPECTED OUTCOME

Our approach combines AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) proposed by Halfond et al. (2005) and SQLCheck proposed by Su and Wassermann (2006). Our model is going to be grammar-based and machine-learning-based in which decision tree would be applied. Grammar-based violation detection uses the grammatical structure of SQL commands or scripting languages to detect vulnerabilities. There are two types of the grammatical structure: finites state machines (FSMs) and parse trees. This research work will be parser-based in which a parse tree would automatically be generated and compared with the parser of the host language to determine whether such a query is legitimate and should be allowed to enter into the database or it is malicious and should be disregarded. Also decision trees will be used to train user inputs to automatically learn and recognize SQL injection attacks or malicious queries. The query is then passed to the database to search for necessary information. Once the information is seen, the web page where the information is seen will be returned to the user for proper action.

5.0 CONCLUSION

Since web applications have become one of the most important communication channels between services provides and clients, sophisticated hackers target victims for commercial or personal gains. The increasing frequency and complexity of web-based attacks have raised awareness of web application administrators of the need to effectively protect their web applications. In this paper, we attempt to justify reasons why the proposed intelligent malicious SQL query detector would be able to detect and report malicious queries and also to protect databases from being attacked by hackers using input commands it is our hope to develop a model that can be near perfection by having low false positives and false negatives.

6.0 REFERENCES

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

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.

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.

Buehrer G.T., Weide B.W. and Sivilotti P. A.G. (2005). Using Parse Tree Validation to Prevent SQL Injection Attacks. In SEM'05: ACM Proceedings of the 5th International Workshop on Software Engineering and Middleware, New York, NY, pp. 106 – 113.

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

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.

Halfond W.G.J. and Orso A. (2005). Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," 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., Orso A. and Manolios P. (2008). WASP: Protecting Web Applications Using Positive Tainting and

Syntax-Aware. Software Engineering, IEEE Transactions, January 31, 2008, pp. 65-81.

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 13th International Conference on World Wide Web, New York, NY, pp. 40 – 52.

Pietraszek T. and Berghe C.V. (2005). Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, (RAID'05), pp. 124 – 145.

Maor O. and Shulman A. (2005). SQL Injection Signatures Evasion. White Paper of Imperva International, http://www.impevra.com/application_defence_centre/white-papers/sql_injection_signatures_evasion.html, Accessed September 12, 2010.

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

OWASP (2010). Open Web Application-Top-Ten-Projects.

Nguyen-Tuong et al. (2005). Automatically Hardening Web Applications Using Precise Tainting. SEC'05.

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.

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.

Werlinger R., Hawkey K., Muldner K., Jaferian P. and Beznosov K. (2008). The Challenges of Using an Intrusion Detection System: Is it Worth the Effort? In Proceedings of the 4th Symposium on Usable Privacy and Security (SOUPS), Pittsburgh, July 23 – 25, 2008, pp. 107 – 116.