Detection of SQL Injection Attacks by Combining Static Analysis and Runtime Validation

Witt Yi Win, and Hnin Hnin Htun

***Abstract*—**SQL injection attack is a particularly dangerous threat that exploits application layer vulnerabilities inherent in web applications. These vulnerabilities might cause unexpected results such as authentication bypassing and information leakage. This paper proposes a simple and effective detection method for SQL injection attacks. This method uses combined static analysis and runtime validation. In static analysis, we find the legitimate queries that could be generated by the application and transform them into static query patterns. The resulting static query patterns are separately stored in the respective tables to reduce the runtime validation overhead. At runtime validation, the runtime query is also transformed as a runtime query pattern and is compared with the predetermined static query patterns.

***Keywords*—**World Wide Web security, SQL Injection, static analysis, runtime validation, Database Security

I. Introduction

IN today's world, Web applications play a very important role not only in individual life but also in any country's development. Web applications have gone through a very rapid growth in the recent years and their adoption is moving faster than that was expected few years ago. With the aid of different Web applications, billions of transactions are done online. Though hundreds of people use these applications, the security level is weak in many cases, which makes them vulnerable to get compromised. A less secure Web application design may allow crafted injection and malicious update on the backend database. This trend can cause lots of damages and unauthorized users theft of trusted users' sensitive data. Sometimes, the attacker may gain full control over the Web application and totally destroy or damage the system [1].

Currently, the SQL injection attacks (SQLIAs) is one of the most dangerous and common threats to databases and Web applications. Attackers can gain direct access to the underlying databases of a web application and the power to leak, modify, or even delete information that is stored on them by SQLIAs. Insufficient input validation is the root cause of SQLIAs. An attacker can submit input with embedded SQL commands directly to the database by leveraging inadequate input validation. This kind of attack represents a serious threat to any web application that reads inputs from the user and makes SQL queries to an underlying database using these inputs. Most web applications of the corporations and government agencies work like this way and could be vulnerable to SQL injection attacks [2].

Defensive coding has been offered as a solution for preventing the SQLIAs but it is very difficult. Although developers try to put some controls in their source code, attackers continue to bring some new ways to bypass these controls. It is difficult to keep developers up to date, according the last and the best defensive coding practices. On the other hand, implementing of defensive coding is very difficult and need to special skills and also erring. These problems motivate the need for a solution to prevent and detect SQL injection attacks [3].

Witt Yi Win is with the University of Technology (Yatanarpon Cyber City), Myanmar (e-mail: wittyiwin09@gmail.com).

Hnin Hnin Htun is with the University of Technology (Yatanarpon Cyber City), Myanmar (e-mail: hninhninhtun@gmail.com).

II. Types of SQL Injection Attack

SQL Injection Attacks refer to a class of code-injection attacks in which data provided by user is included in the SQL query and part of the user's input is treated as SQL code. In this section, we present and discuss the different kinds of SQLIAs known to date. Depending on the specific goals of the attacker, the different types of attacks are generally not performed in isolation; many of them are used together or sequentially.

A. Tautologies

A SQL tautology is a statement that is always true. The most common usages are to bypass authentication pages and extract data such as query 1.

Query 1:

```
SELECT * FROM employee WHERE name = ' ' or 1=1 -- '
AND password = '12345';
```

The single quote (') symbol indicates the end of string and two dashes comment the following text of the query. Boolean expression 1=1 is always true. As a result, the user will be logged on as the first user stored in the table.

B. Logically Incorrect Queries

This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. Query 2 shows an example.

Query 2:

SELECT * FROM employee WHERE name = ‘ ’ UNION SELECT SUM(username) from users -- ’ and password= ‘ ’ ;

The query tries to execute the column username from users table and it tries to convert the username column into integer, which is not a valid type conversion. Hence, the database server returns an error message which contains name of the database and information of the column field.

C. Union Queries

This attack uses the “UNION” operator, which performs union between two or more SQL queries. As a result of this attack, database returns a dataset which is union of the results of original query and the injected query. An example is shown in query 3.

Query 3:

SELECT emp_id FROM employee WHERE name = ‘’ UNION SELECT cardNo FROM creditCard WHERE accNo = 10032 -- AND password = ‘ ’ ;

The original first query returns the null set, whereas the second query returns data from the “creditCard” table.

D.PiggyBacked Queries

In this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. This type of attack can be extremely harmful such as query 4.

Query 4:

SELECT * FROM employee WHERE name = ‘guest’ and password = ‘1234’; DROP TABLE employee; -- ;

After completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The result of executing the second query would be to drop the employee table.

E. Stored Procedures

DBMS has provided stored procedures method with which a user can store his own function that can be used as needed. To use the function, a collection of SQL queries is included. Query 5 shows an example.

Query 5:

CREATE PROCEDURE DBO @userName varchar2, @pass varchar2, AS EXEC (“SELECT * FROM user WHERE id= ‘ “+@userName+”’ and password= ‘ “+@pass+’”); GO

This scheme is also vulnerable to attacks such as piggy-backed queries.

III. Related Work

Over the past decade, a lot of work has been accomplished by the research community in providing new techniques to detect and prevent SQLIAs. In this section, we discuss state-of-the-art in SQLIA detection and prevention techniques.

Yadav et al. [4] introduced an authentication scheme for preventing SQL Injection attack using Advance Encryption Standard (AES) using hash function for stored password and user id. Encrypted user name and password are used to improve the authentication process with minimum overhead. The server has to maintain three parameters of every user: user name, password, and user’s secret key. This paper proposed a protocol model for avoiding SQL Injection attack using AES (PSQLIA-AES) with hash function. This modeling explicitly captures the particular subtle incidents triggered by SQLIA adversaries and corresponding state transitions. This technique has a limitation also. This technique can be implemented in the beginning of website development. Reengineering of website will have to be done to implement this technique on the existing website.

Dharam et al. in [5] proposed a Runtime Monitoring framework that is used in the development of runtime monitors. The framework uses two pre-deployment testing techniques, such as basis-path and data-flow to identify a minimal set of all legal/valid execution paths of the application. Runtime monitors are then developed and integrated to perform runtime monitoring of the application, during its post-deployment for the identified valid/legal execution paths. The results of their study show that runtime monitor developed for the application was successfully able to detect all the tautology based attacks without generating any false positives. The important limitation of the proposed technique is that it can detect only tautology based SQLIAs.

Frankl et al. in [6] introduced a tool named ASSIST (Automatic and Static SQL Injection Sanitization Tool) for protecting Java-based web applications which could come from applications developed as JSPs or Servlets. This paper presents the technique of automatic query sanitization to automatically remove SQL injection vulnerabilities in code. By using a combination of static analysis and program transformation, this technique automatically identifies the locations of SQL injection vulnerabilities in code and instruments these areas with calls to sanitization functions. This automated technique can be used to relieve developers from the error-prone process of manual inspection and sanitization of code. But, there are several sources of imprecision which may lead to false positives and false negatives in this technique.

Kadirvelu et al. [7] proposed an intelligent dynamic query intent evaluation technique to learn and predict the intent of the SQL queries provided by users and to compare the identified query structure with the query structure which has been generated with user input in order to detect possible attacks by unethical users automatically. This type of evaluation is helpful in reducing the need for the user to have

more consciousness when SQL queries are written. The main advantage of this system is that it applies the decision tree classification algorithm which is enhanced with temporal rules to find the unethical users intelligently at the query execution points where the database manager of the system can be informed of the new possible query execution points with an intent for attacks, and thereby preventing the SQL injection attacks.

Salama et al. [8] proposed a framework based on misuse and anomaly detection techniques to detect SQL injection attack. The main idea of this framework is to create a profile for legitimate database behavior extracted from applying association rules on XML file containing queries submitted from application to the database. It then used data mining technique for fingerprinting SQL statements and uses them to identify SQLIA. This set of fingerprints is then used to match incoming database transactions. If the set of fingerprints in the legitimate set is complete, any incoming transaction whose fingerprint does not match any of those in the legitimate set is very likely to be an intrusion. As a second step in the detection process, the structure of the query under observation will be compared against the legitimate queries stored in the XML file thus minimizing false positive alarms.

Hidhaya et al. [9] developed a method to detect the SQL injection. It used a Reverse proxy and MD5 algorithm to check out SQL injection in user input. Using grammar expressions rules to check for SQL injection in URL's. This system does not do any changes in the source code of the application. The detection and mitigation of the attack is fully automated. By increasing the number of proxy servers the web application can handle any number of requests without obvious delay in time and still can protect the application from SQL injection attack. In future work, the focus will be on optimization of the system and removing the vulnerable points in the application itself.

IV. Proposed Method

This section proposes a method to detect SQL injection attacks based on static application code analysis and runtime validation. We use Java String Analyzer (JSA) that is a tool for performing static analysis of Java programs. The tool takes the .class file to analyze, which we call the application classes. In the application classes, one or more string expressions are selected as hotspots. A hotspot is defined as a point in the application code that issues SQL queries to the underlying database. In our analysis, we identify all the "execute" methods of the Statement class in Java as hotspots. The result of the analysis is NDFA that expresses, at the character level, all the possible values the considered string can assume at the hotspot. We perform a depth first traversal of each hotspot's NDFA and group characters into SQL tokens and string tokens to produce static query (SQ). SQL tokens consist of either SQL keywords or operators. Any token that is not a SQL keyword, SQL operator or delimiter is recognized as a string token. We use the generic label "var" for the variable string related to some user input. We initially create a table namely Token in Master database. We store SQL keyword, logical operators and relational operators in Token table. The string tokens that are not initialized in the table are marked as new tokens and stored in the table as shown in Table 1. Each static query is then transformed into the static query pattern (SQP) using Token table as shown in the following example.

SQ: SELECT * FROM employee WHERE name = 'var' and password = 'var';
SQP: $SAsFA_1WA_2R_1VL_1A_3R_1V$

TABLE I
Token Table

Token ID	Token Type	Token
S	Keyword	Select
D	Keyword	Delete
I	Keyword	Insert
As	Keyword	*
F	Keyword	From
W	Keyword	Where
K_1	Keyword	--
R_1	Relational Operator	=
L_1	Logical Operator	And
L_2	Logical Operator	Or
V	Variable	Var
A_1	Attribute	employee
A_2	Attribute	name
A_3	Attribute	password
A_4	Attribute	emp_id

In order to reduce the runtime validation overhead, we separately store the obtained static query patterns. We also create four tables for SELECT, INSERT, UPDATE and DELETE operations in Master database. These operations are four basic SQL operations commonly used by most applications. Then, we store the static query pattern for each hotspot in the corresponding table, according to the query operation as shown in Table 2.

TABLE II
Select Query Table

Query No	Static Query Pattern
1	$SAsFA_1WA_2R_1VL_1A_3R_1V$
2	$SA_4FA_1WA_2R_1V$

During runtime, when the application reaches a hotspot, we invoke the validation function and pass the runtime query (RQ) as a parameter. The validation function parses the runtime query into a sequence of SQL tokens, delimiters and string tokens. Then the function coverts the runtime query into the runtime query pattern (RQP) using Token table. The user input that is not included in the Token table is defined as "var". To detect SQL injection attack, the runtime query pattern is logically exclusively ORed with the static query pattern.

If we use every static query pattern to check a runtime query, the process is expensive. Therefore, we retrieve the static query patterns from Master database according to the query operation of the runtime query. For example, if the runtime query is "SELECT" query, we use the static query patterns in Select Query table for comparison. By restricting the number of static query patterns, we can reduce the runtime scanning overhead. The runtime query pattern is logically exclusively ORed with each of the corresponding static query patterns. If the result of logically exclusively OR is zero, the system allows the query to be executed on the database. If the result is not zero, the system blocks the query before it is executed on the database and alerts the user. The process of the validation function is described in algorithm 1. The convert_pattern( ) function in algorithm1 converts the runtime query into the runtime query pattern. The getFirstChar ( ) function returns the first character of the runtime query pattern and we use it to get static query patterns from Master database.

```
Algorithm Validation (RQ)
Begin
    Legitimate = false;
    RQP = convert_pattern (RQ);
    RQ_Op = getFirstChar (RQP);
    Get SQPs from Master database using RQ_Op and assign
    to Array N;
    While (not empty of N and legitimate == false)
        if ( (a SQP of N ⊕ RQP) == 0 ) then
          legitimate = true;
        else
          legitimate = false;
        end if
    end while
    return legitimate;
End
```

Algorithm I: Proposed SQL Injection Detection Algorithm

The following are the examples applying proposed method.

RQ_1: SELECT * FROM employee WHERE name = 'admin' and password = 'admin';
RQP_1 : $SAsFA_1WA_2R_1VL_1A_3R_1V$
RQ_2: SELECT * FROM employee WHERE name = 'test' and password = 'test ' or 1=1 ;
RQP_2 : $SAsFA_1WA_2R_1VL_1A_3R_1V\ L_2V\ R_1V$

SQP_1 : $SAsFA_1WA_2R_1VL_1A_3R_1V$
SQP_2 : $SA_4FA_1WA_2R_1V$

$SQP_1 \oplus RQP_1 = 0$

So, the RQ_1 is the legitimate query.

$SQP_1 \oplus RQP_2 \neq 0$

$SQP_2 \oplus RQP_2 \neq 0$

So, the RQ_2 is the attack query.

V. Evaluation

The system is tested on five real world applications, namely Portal, Event Manager, Employee Directory, Bookstore and Classifieds. The applications are taken from gotocode.com. For each application, there are two sets of inputs: LEGIT consists of legitimate inputs for the application, and ATTACK consists of attempted SQLIAs. The applications are deployed on Tomcat server with MySQL database. The queries are tested from the list of queries containing sets of both legitimate and attack queries and the request is provided to application using the wget command.

The proposed technique introduces two types of overhead. The first overhead is due to the static analysis of the application source code to construct static query patterns and the second due to runtime validation. As the static analysis is an offline process, the users do not experience the delay induced to this one-time operation (until next code modification). Table 3 shows the time required for the static analysis to complete when executed on different applications having varying number of queries.

TABLE III
Execution Time for Static Analysis

Web Application	No. of Queries	Execution Time(ms)
Portal	67	215
Event Manager	31	122
Employee Directory	23	80
Bookstore	71	249
Classifieds	34	194

To determine the runtime overhead of the proposed system, we conduct timing experiments on the bookstore application. We compare the runtime of the original bookstore application to that of the version that was instrumented by the proposed method. We run the LEGIT set on them and measure the difference in execution time as overhead. The applications are installed at localhost to prevent network delay. We found that the runtime overhead on the bookstore application is no more than 3%.

The total number of SQL injections and the total number of detections by our system defining the detection rate is stated in Table 4.

TABLE IV
Detection Rate

Web Application	No.of SQL Injection Attacks	No. of Detections	Detection Rate
Portal	435	435	100%
Event Manager	178	178	100%
Employee Directory	238	238	100%
Bookstore	410	410	100%
Classifieds	378	378	100%

VI. Conclusion

In this paper, brief study of various SQL Injection attacks is described and different methods for Detection and Prevention of these attacks are also discussed. The main goal of SQL injection attack is to inject some malicious script to the database to gain an unauthorized access to the system. Web applications are basically vulnerable to such type of attacks where the user provides input information and this information gets stolen by the attacker because of the lack of validation at the input side. We have proposed a method for detecting SQL injection attacks by comparing runtime query pattern with static query patterns. Furthermore, we have evaluated the performance of the proposed method by experimenting on vulnerable web applications. The proposed method can be implemented not only on web applications but also on any applications connected to databases.

References

[1] K.D.Abdoulaye and P.A.Khan, "A Detailed Survey on Various Aspects of SQL Injection: Vulnerabilities, Innovative Attacks, and Remedies", ACCEPTED VERSION for INFORMATION Journal, 2011

[2] W.G.J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL Injection Attacks", workshop on Dynamic Analysis (WODA 2005), St. Lousis, MO, USA

[3] T. Atefeh, I. Suhaimi, M. Maslin, "SQL Injection Detection and Prevention Techniques", International Journal of Advancements in Computing Technology Volume 3, Number 7, August 2011

[4] S. Y. Prasant, Dr pankaj Yadav, Dr. K. P. Yadav, "A Modern Mechanism to Avoid SQL Injection Attacks in Web Applications", IJRREST, Volume-1 Issue-1, June 2012

[5] D.Ramya and S.G.Sajjan, "Runtime Monitoring Technique to handle Tautology based SQL Injection Attacks", International Journal of Cyber-Security and Digital Forensics (IJCSDF), Nov.2012

[6] M.Raymond and F.Phyllis, "Preventing SQL Injection through Automatic Query Sanitization with ASSIST", Fourth International Workshop on Testing, Analysis and Verification of Web Software, EPTCS 35, pp. 27–38, 2010

[7] K.Selvamani and A.Kannan, "ISQL-IDPS: Intelligent SQL-Injection Detection and Prevention System", European Journal of Scientific Research ISSN 1450-216X Vol.51 No.2, pp.222-231, 2011

[8] E. S. Shaimaa, I. M. Mohamed, M. E. Laila, K. H. Yehia, "Web Anomaly Misuse Intrusion Detection Framework for SQL Injection Detection", IJACSA, Vol.3, No.3, 2012

[9] S. F. Hidhaya, A. Geetha, "Intrusion Protection against SQL Injection Attacks Using a Reverse Proxy", 2012

[10] M.Jayeeta and S.Gargi, "Analysis of SQL Injection Attack", Special Issue of International Journal of Computer Science & Informatics **(**IJCSI), ISSN **(**PRINT**)** : 2231–5292, Vol.- II, Issue-1, 2