Mathematical and Computer Modelling 55 (2012) 58–68

Contents lists available at SciVerse ScienceDirect

Mathematical and Computer Modelling

journal homepage: www.elsevier.com/locate/mcm

A novel method for SQL injection attack detection based on removing SQL query attribute values

Inyong Lee [a], Soonki Jeong [b], Sangsoo Yeo [c], Jongsub Moon [d,*]

[a] Center for Information Security Technologies, Korea University, Seoul 136-713, Republic of Korea

[b] Graduate School of Information Security, Korea University, Seoul 136-713, Republic of Korea

[c] Division of Computer Engineering, Mokwon University, Daejeon 302-729, Republic of Korea

[d] Department of Electronics and Information Engineering, Korea University, Chochiwoneup, Yeonkigun, Choongnam 339-700, Republic of Korea

ARTICLE INFO

Article history:
Received 16 September 2010
Received in revised form 5 January 2011
Accepted 29 January 2011

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

ABSTRACT

SQL injection or SQL insertion attack is a code injection technique that exploits a security vulnerability occurring in the database layer of an application and a service. This is most often found within web pages with dynamic content. This paper proposes a very simple and effective detection method for SQL injection attacks. The method removes the value of an SQL query attribute of web pages when parameters are submitted and then compares it with a predetermined one. This method uses combined static and dynamic analysis. The experiments show that the proposed method is very effective and simple than any other methods.

1. Introduction

As networks and the internet have advanced, many offline services have moved online. Nowadays, most online services use web services. The ability to access the web anywhere and anytime is a great advantage; however, as the web becomes more popular, web attacks are increasing. Most web attacks target the vulnerabilities of web applications, which have been researched and analyzed at OWASP [1].

The SQL Injection Attack (SQL Injection Attack) does not waste system resources as other attacks do. However, because of its ability to obtain/insert information from/to databases, it is a strong threat to servers like military or banking systems.

Many researchers have been studying a number of methods to detect and prevent SQL injection attacks, and the most preferred techniques are web framework, static analysis, dynamic analysis, combined static and dynamic analysis, and machine learning techniques.

The web framework [2,3] uses filtering methods for user input data. However, because it is only able to filter some special characters, other detouring attacks cannot be prevented. The static analysis method [4–8] analyzes the input parameter type, so it is more effective than the filtering method, but attacks having the correct parameter types cannot be detected. The dynamic analysis method [9–11] scans vulnerabilities of web applications without modifying them; however this method is not able to detect all SQL injection attacks. A combined static and dynamic analysis method [12–16] can compensate for the weaknesses of each method and is highly proficient in detecting SQL injection attacks. The combined usage of a method of static and dynamic analysis is very complicated. A machine learning method [17,18] of a combined method can detect unknown attacks, but the results may contain many false positives and negatives.

* Corresponding author. Tel.: +82 2 3290 4750; fax: +82 2 3290 3998.
E-mail addresses: iylee@korea.ac.kr (I. Lee), soonki32@korea.ac.kr (S. Jeong), sangsooyeo@gmail.com (S. Yeo), jsmoon@korea.ac.kr, moon.jongsub@gmail.com (J. Moon).

0895-7177/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mcm.2011.01.050

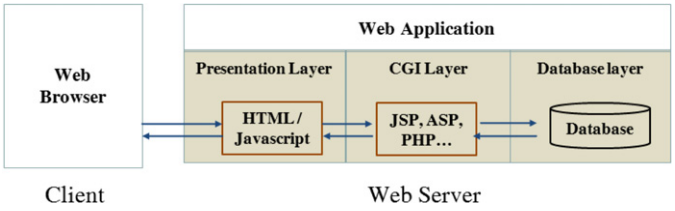

Fig. 1. Web application architecture.

This paper proposes a very simple and effective means to accurately detect SQL injection attacks by using a combination of SQL query parameter removal and combined static and dynamic analysis methods. The effectiveness of this method has been tested and validated using web applications.

The rest of this paper is organized as follows. Section 1 reviews the architecture of web application and SQL injection attacks. Section 2 discusses the related work. Section 3 proposes a method which uses a combination of SQL query parameters removal and combined static and dynamic analysis methods for the detection of SQL injection attacks. Section 4 elaborates the experiment and its results using the proposed method, and Section 5 ends with a conclusion.

1.1. Web application and SQL injection attacks

1.1.1. Web application architecture

Although a web application is simply recognized as a program running on a web browser, a web application generally has a three-tier construction as shown in Fig. 1 [12,15]. In Fig. 1, a presentation tier is sent to a web browser by request of the browser.

(1) Presentation Tier: This tier receives the user input and shows the result of the processing to the user. It can be thought of as the Graphical User Interface (GUI). Flash, HTML, Java script, etc. are all part of the presentation tier, which directly interacts with the user. This tier is analyzed by a web browser.
(2) CGI Tier: Also known as the Server Script Process, this is located in between the presentation and database tiers. The data inputted by the user is processed and the result is sent to the database tier. The database tier sends the stored data back to the CGI tier, and it is finally sent to the presentation tier to be viewed by the user. Therefore, data processing within the web application is performed at the CGI Tier and can be programmed in various server script languages such as JSP, PHP, ASP, etc.
(3) Database Tier: This tier only stores and retrieves all of the data. All sensitive web application data are stored and managed within the database. Since this tier is directly connected to the CGI tier without any security check, data in the database can be revealed and modified if an attack on the CGI tier succeeds.

1.1.2. An example of SQL injection attacks: tautologies

The SQL injection vulnerabilities are in between the presentation and CGI tiers, thus attacks occur between these tiers. Most of the vulnerabilities accidentally emerge in the development stage of the application program.

The data flows among the three tiers using both normal and malicious input data are shown in Fig. 2 as an example. This kind of attack is called a tautology and occurs at the user authentication step. When a genuine user enters their genuine ID and password, the presentation tier uses the GET and POST method to send the correct data to the CGI tier. The SQL query within the CGI tier connects to the database and processes the authentication procedure. The following is based on Fig. 2.

If a malicious user enters an ID such as 1' or '1 = 1'—, the query within the CGI tier becomes SELECT ∗ FROM user WHERE id = '1' or '1 = 1'—' AND password = '1111'. Because the rest of the string following—becomes a comment and '1 = 1' is always true, the authentication step is bypassed.

1.1.3. Other kinds of SQL injection attacks

(a) Illegal/Logically Incorrect Queries.
This attack derives the CGI tier replies error message by inserting a malicious SQL query such as query 1.
Query 1:
*SELECT * FROM user WHERE id='1111' AND password='1234' AND CONVERT(char, no) --';*
\ The purpose of this attack is to collect the structure and information of CGI.
(b) Union Queries.
This attack uses the "Union" operator which performs unions between two or more SQL queries. This attack performs unions of malicious queries and a normal SQL query with the "union" operator. Query 2 shows an example.
Query 2:
*SELECT * FROM user WHERE id='1111' UNION*
*SELECT * FROM member WHERE id='admin' --' AND password='1234';*

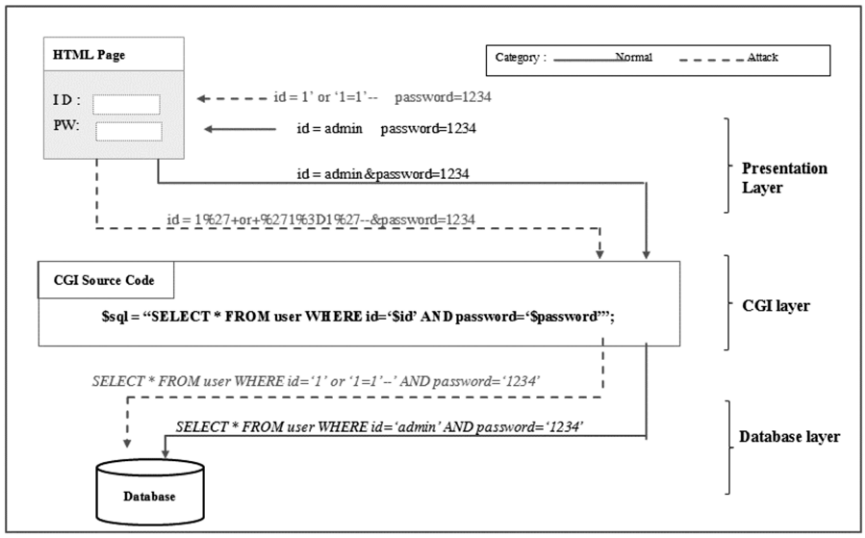

Fig. 2. SQL normal and SQL injection attack data flow.

All subsequent strings after—are recognized as comments, and two SQL queries are processed in this example. The result of the query process shows administrator's information of the DBMS.

(c) Piggy-Backed Queries.
This attack inserts malicious SQL queries into a normal SQL query. It is possible because many SQL queries can be processed if the operator "';" is added after each query. Query 3 is an example. Note that the operator "';" is inserted at the end of query.
Query 3:
*SELECT * FROM user WHERE id='admin' AND password='1234'; DROP TABLE user; --';*
The result of query 3 is to delete the user table.

(d) Stored Procedures
Recently, DBMS has provided a 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. An example is shown in query 4.
Query 4:
CREATE PROCEDURE DBO @userName varchar2, @pass varchar2,
AS
EXEC("SELECT * FROM user WHERE id='" + @userName + "' and password='" + @password + "');
GO
This scheme is also vulnerable to attacks such as piggy-backed queries.

SQL injection attacks are only malicious queries which change a normal SQL query into a malicious one and consequently allow anomalous database access and processing. Most web applications use filters to prevent these kinds of SQL injection attacks. However, there are many SQL injection attacks which can bypass data filters, which makes it difficult for the application to effectively defend the database from attacks. Therefore, a more effective means of detecting and preventing SQL injection attacks is necessary.

2. Related work

This section explains the protection methods for the SQL injection attack.

2.1. Web framework

The web framework uses a filtering method to remove special characters. Recently, some web frameworks have provided a wider variety of prevention methods than ever before. PHP provides Magic Quotes [3], which works when any combination of 4 special characters ', ", /, NULL exists in the data field of the POST, GET and COOKIES pages. It automatically adds a '\' in front of the special character to prevent SQL injection attacks. However, Magic quotes only works for the four special characters and therefore, other detouring attacking methods exist. Also, web applications must be rewritten in order to configure the Magic Quotes function.

The Validator [2] inspects the user input data with predefined rules. If the special characters used in attacks are not well predefined in Validator, it cannot protect against attacks. In addition, the setup procedure is very complicated.

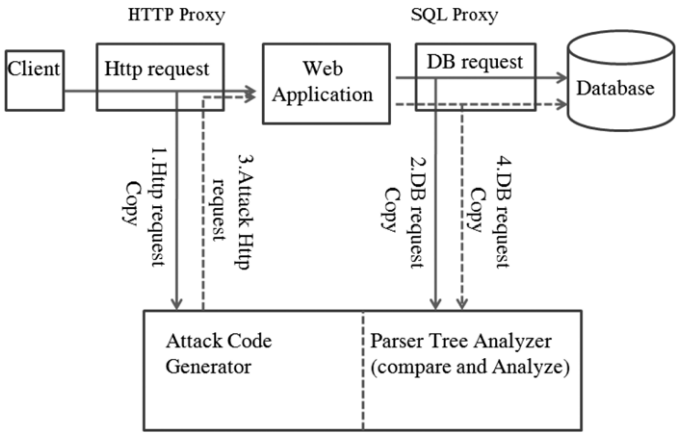

Fig. 3. Structure of Sania.

2.2. Static analysis

Static analysis analyzes the SQL query sentences of web applications to detect and prevent SQL injection attacks. It also requires rewriting of web applications. The focus of the static analysis method is to validate the user input type in order to reduce the chances of SQL injection attacks rather than detect them. JDBC-Checker [4] uses the Java String Analysis (JSA) library to validate the user input type dynamically and prevent SQL injection attacks. However, if malicious input data has the correct type or syntax, it cannot protect against the SQL injection attack. Also, the JSA library only supports the Java programming language. Wassermann [8] used a static analysis method which was combined with automated reasoning. This method assumes that there is no tautology in an SQL query generated dynamically, which was verified. Thus, this method is efficient in detecting SQL injection attacks, but other SQL injection attacks except for a tautology cannot be detected. Stephen [7] created a fix generation SQL query by collecting plain text SQL statements, SQL queries, and execution calls to validate user input types. This method does not directly prevent or detect SQL injection attacks, but by deleting vulnerabilities in the SQL query syntax in advance, it tries to prevent the attack. This method is only available for web applications written with Java, and requires the AST and ZQL libraries [19].

2.3. Dynamic analysis

Dynamic analysis analyzes the response from a web application after scanning it. A scan means to send every kind of input to the target and receive the response. Unlike static analysis, it can locate vulnerabilities from SQL injection attacks without making any modifications to web applications. Paros [10], which is an open source program, finds not only SQL injection attacks, but also other vulnerabilities within the web application. Paros is not effective because it uses predetermined attack codes to scan and determines the success or fail with the HTTP response. Sania [9] protects against SQL injection attacks by using the following procedures. (1) It collects normal SQL queries between client and web applications and between the web application and database, and analyzes the vulnerabilities. (2) It generates SQL injection attack codes which can reveal vulnerabilities. (3) After attacking with the generated code, it collects the SQL queries generated from the attack. (4) The normal SQL queries are compared and analyzed with those collected from the attack, using a parse tree. (5) Finally, it determines whether the attack succeeded or not. These procedures are shown in Fig. 3. Since it uses a parse tree, Sania is more accurate than the method which uses an HTTP response. Shin [11] proposed a method to build test input data to locate SQL injection vulnerabilities by making a white-box from both input flow analysis and input validation analysis.

The dynamic analysis method has advantages because no web application modifications are necessary. However, the vulnerabilities found in the web application pages must be manually fixed by the developers and not all of them can be found without predefined attack codes.

2.4. Combined method of static and dynamic analysis

A combined static and dynamic analysis method utilizes the advantages of both the static analysis and dynamic analysis method to detect SQL injection attacks. That is, it analyzes web pages and simultaneously generates SQL queries to test the results. SQLCheck [15] defined SQL injection attacks and proposed a sound and complete algorithm based on context-free grammars and compiler parsing techniques. AMNESIA [14] proposed a method to find hotspots in which SQL queries are executed inside web applications and all possible SQL queries are generated. The generated static SQL queries and all dynamic SQL queries from the user were analyzed and classified using the JSA library. Buehrer [12] compared static and

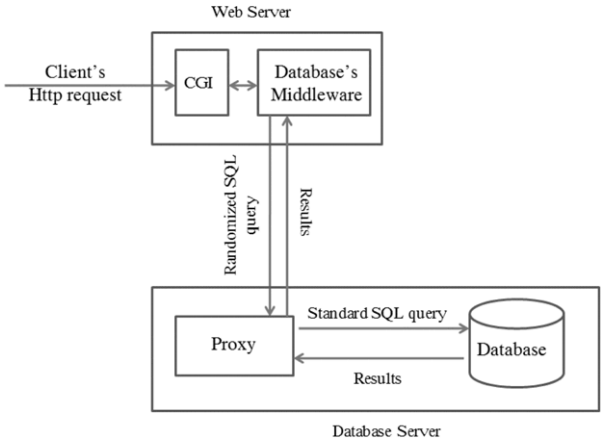

Fig. 4. Schematic diagram of instruction-set randomization.

dynamic SQL queries generated by the user using a parse tree to detect SQL injection attacks. Wei [16] proposed a method to compare and analyze a stored procedure in a web application with runtime user input of SQL queries using a control flow graph to detect SQL injection attacks.

2.5. Instruction-set randomization

The instruction-set randomization method inserts random values into the SQL query statements of a web application and checks for volatility in order to detect SQL injection attacks. SQLrand [20] places a proxy server between the web and database servers. It sends SQL queries with a randomized value to the proxy server. However if the random value can be predicted, this method is not effective. Fig. 4 shows a schematic diagram of this method.

2.6. SQL query profiling

Park [21] profiled the SQL queries of a web application and compared it with the dynamic SQL queries generated at runtime using the pairwise sequence alignment of amino acid formulate method to detect SQL injection attacks. This method has advantages because it can detect SQL injection attacks without rewriting the web application. However, the web application must be profiled whenever it is changed.

2.7. Machine learning method

Valeur [18] proposed an intrusion detection system with a machine learning method. The SQL queries generated in a web application were learned in order to generate the parameters of the detection model. Then, runtime SQL queries were compared to the generated model in order to check for discrepancies. If the model is not effectively trained, many false positive and negative results can occur. WAVES [17] used a web crawler to find the vulnerabilities in a web application and generate attack codes by utilizing a pattern list and attack techniques. Using the generated attack codes, the SQL injection attack vulnerabilities could be found. Since this method used a machine learning method, it is supposed to be more effective than traditional penetration testing. However, this method could not detect all vulnerabilities.

3. Proposed method

3.1. Proposed method

This section proposes a novel method to detect SQL injection attacks based on static and dynamic analysis. This method removes the attribute values of SQL queries at runtime (dynamic method) and compares them with the SQL queries analyzed in advance (static method). The symbols used in this proposed algorithm are shown in Table 1. The detection method is elaborated based on the example given in Section 2.2.

Applying Table 1 to the example shown in Section 2.2 results in the following:
I_f: admin, 1234, 1' OR '1=1'-- and 1234.
FQ: SELECT * FROM user WHERE id='$id' AND password='$password'.
DQ_t: SELECT * FROM user WHERE id='admin' AND password='1234',
DQ_f: SELECT * FROM user WHERE id='1' or '1=1'—' AND password='1111'.

Table 1
Symbols used in this algorithm.

Symbol	Description
$I_{\{t,f\}}$	Userinputdata{t:normalinputdata,f:abnormalinputdata}
f	Function which drops the value of the SQL query
FQ	Fixed SQL query in web application
$DQ_{\{t,f\}}$	GenerateddynamicSQLquerywithuserinput{t:normalSQLquery,f:abnormalSQLquery}
FDQ	Attribute indicating which value was removed from the fixed SQL query
$DDQ_{\{t,f\}}$	Attributeindicatingwhichvaluewasremovedfromthedynamic SQLquery{t:normalSQLQuery,f:abnormalSQLQuery}

The detection method proposed in this article uses the function f which deletes the attribute values in the SQL queries. The function is shown in formula (1) and the detail algorithm is shown in algorithm 2. The attribute values of the static SQL queries in the web application and those of the SQL queries generated at runtime will be deleted.

$$FDQ = f(FQ), \qquad DDQ = f(DQ). \tag{1}$$

In algorithm 1, the function, f, removes only the string values surrounded by ‘ after ‘“=” or within parenthesis. The attribute value of an SQL query consists either of the form *variable = ‘string value’* or *variable = numeric value*. In case that a function is used in an SQL query, the function head is either of the form “function name (numeric value)” or “function name (‘string value’)”. The value of the string is surrounded by ‘. The ‘which surrounds the string value is the operator’ but the value of ‘ in the SQL query is preceded by \. So the case where ‘ is preceded by \ is not considered. The function of Get_Token in the algorithm extracts and removes the first character in the input string and then returns the character. *Current_Quotation_State* is changed to the appropriate status in the function *Toggle Current_Quotation_State*in this algorithm.

```
Algorithm f(One SQL query)
Enumerate Quotation_Status = { Quot_Start, Quot_End}
Input String=One SQL query;
;Output_String=Null;
Current_Quotation_State=Quot_End;

Do while( not empty of Input String)
{
    Char=Get_Token(Input_String);
    If Char is a quotation character
    {
        Add Char to Output_String;
        If the preceding character is not back slash
            Toggle Current_Quotation_State;
    }
    Else
    {
        If Current_Status is Quota_End than
        {
            Add Char to Output_String;
        }
        Else
        {
            If the preceding character is \ (back slash) then
            Add Char to Output_String;
        }
    }
}

Return Output_String;
```

Algorithm 1: Algorithm which removes the attribute value in a SQL query.

The following examples show the result of function f. Bold characters are deleted and “consists of two concatenated ‘. DQ_1 is a normal query and DQ_2 is an abnormal query.

*FQ = SELECT * FROM user WHERE id='$id' AND password='$password'*
FDQ=f(DQ)
*= f(SELECT * FROM user WHERE id='* ***$id*** *' AND password='* ***$password*** *')*
*= SELECT * FROM user WHERE id=' ' AND password=' '*

DQ_1= *SELECT * FROM user WHERE id='admin' AND \ password='1234'*
DDQ_1=*f*(DQ_1)
= *f(SELECT * FROM user WHERE id='**admin**' AND password='**1234**')*
= *SELECT * FROM user WHERE id=' ' AND password=' '*

DQ_2= *SELECT * FROM user WHERE id='1' or '1=1'—' AND password='1234'*
DDQ_2=*f*(DQ_2)
= *f(SELECT * FROM user WHERE id='**1** ' or' **1=1**'—' **AND password=**'1234')*
= *SELECT * FROM user WHERE id=' ' or ' '—''1234'*

Formula (2) is applied regardless of whether an SQL query is normal or abnormal. Here, $\oplus$ is the symbol representing the exclusive OR operator. That is, two strings are logically exclusively ORed.

$$FDQ \oplus DDQ \begin{cases} = 0 & \text{Normal} \\ \neq 0 & \text{Abnormal.} \end{cases} \tag{2}$$

If we apply this formula to the above example, the following two results are obtained:

$FDQ \oplus DDQ_1 = 0$: Normal
$FDQ \oplus DDQ_2 \neq 0$: Abnormal.

Algorithm 2 is the generalization of the SQL injection attack detection algorithms proposed in this section. Lines 1–4 of this algorithm can be processed for the targeted web pages in advance.

N: *Total number of fixed SQL queries in web application*
FQ_i: *i 'th fixed SQL query in web application*
DQ_i: *Dynamic SQL query generated from* FQ_i
f: *Function to delete value of attribute in SQL query*
$FQ = \{FQ_1, \ldots, FQ_n\}$,
$FDQ = \{FDQ_1, \ldots, FDQ_n\}$,
// Static analysis
1. *For i=1 to N*
2. *Get* FQ_i
3. $FDQ_i = f(FQ_i)$
4. *End {For}*
5.
// Dynamic analysis (running time)
6. *While*(*Normal* & $\forall k \in N$)
7. *Get* DQ_k *from the web with* $I_{\{t,f\}}$
8. $DDQ_k = f(DQ_k)$
9. *If*($FDQ_k \oplus DDQ_k$) = 0 *then*
10. *Result = Normal*
11. *Else*
12. *Result = Abnormal*
13. *End {If}*
14. *End {While}*

Algorithm 2. Proposed SQL Injection Detection Algorithm.

3.2. Other examples applying proposed methods

Note that '' consists of two concatenated ' (single quotations). It is not a "(one double quotation) in this example.

(a) Illegal/Logically Incorrect Queries Attack.
*FDQ: SELECT * FROM user WHERE id='' AND password='';*

*DQ_f : SELECT * FROM user WHERE id='1111' AND password='1234' AND CONVERT(char, no) --';*
*DDQ_f : SELECT * FROM user WHERE id='' AND password='' AND CONVERT(char, no) --';*

$FDQ \oplus DDQ_f \neq 0$
(b) Union Queries.
*FDQ: SELECT * FROM user WHERE id='' AND password='';*

Table 2
Experiment results.

Web applications	Proposed algorithm		SQLCheck [14]		AMNESIA [12]	
	Detection/Attack	Detection rate (%)	Detection/Attack	Detection rate (%)	Detection/Attack	Detection rate (%)
Employee directory	247/247	100	3937/3937	100	280/280	100
Events	87/87	100	3605/3605	100	260/260	100
Classifieds	319/319	100	3724/3724	100	200/200	100
Portal	288/288	100	3685/3685	100	140/140	100
Bookstore	366/366	100	3473/3473	100	182/182	100

*DQ_f : SELECT * FROM user WHERE id='1111' UNION*
*SELECT * FROM member WHERE id='admin' --' AND password='1234';*
*DDQ_f : SELECT * FROM user WHERE id='' UNION*

*SELECT * FROM member WHERE id='' --'1234';*

$FDQ \oplus DDQ_f \neq 0$

(c) Piggy-Backed Queries.
*FDQ: SELECT * FROM user WHERE id='admin' AND password='1234'*

*DQ: SELECT * FROM user WHERE id='admin' AND password='1234'; DROP TABLE user; --';*
*DDQ: SELECT * FROM user WHERE id='' AND password=''; DROP TABLE user; --';*

$FDQ \oplus DDQ_f \neq 0$

(d) Stored Procedures.
*FDQ: SELECT * FROM user WHERE id='' AND password='';*

*DQ_f : SELECT * FROM user WHERE id='admin' AND password='1234'; SHUTDOWN;--;*
*DDQ_f : SELECT * FROM user WHERE id='' AND password=''; SHUTDOWN;--;*

$FDQ \oplus DDQ_f \neq 0$

As shown in the above examples, the proposed method is effective at detecting SQL injection attacks.

4. Experiment and evaluation

4.1. Experiment method

The proposed algorithm can be applied for real web applications. It can scan web applications in order to extract FQ_i and make a list to be compared with each generated DQ by each user. Because lines 1–4 are only for static analysis, when web pages are produced, the list can be constructed only once.

The web application cited for the experiment is GotoCode, which is also used in other many researches [22,14,9,15]. Furthermore, Paros 3.2.13 [10] was used to scan each web application and to collect all vulnerabilities for an accurate experiment.

4.2. Experimental results analysis

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

4.2.1. Detection/attack rate analysis

To compare the performance of our proposed method, SQLCheck and AMNESIA for detection rate of SQL injection, we used five types of web applications. At this experiment, we used Paros 3.2.13 as an attack tool. The detection frequencies versus the attacks frequencies were defined as the detection rate. As shown in Table 2, all three methods have the same performance. The result seems to reflect that all methods use rules of static and dynamic SQL queries.

4.2.2. Comparison of detection and prevention methods by attack types

Halfond [23] classified SQL injection attacks into various types and used them to compare the efficiency of methods for detection and prevention. This author used the method of Halfond to compare the efficiency of the proposed method with other SQL injection detection methods. The results are shown in Table 3.

Table 3
Comparison of detection and prevention methods for various SQL injection attacks.

Detection/Prevention method	Tautologies	Illegal/Incorrect queries	Union queries	Piggy-Backed queries	Stored procedures	Inference	Alternate encodings
AMNESIA [12]	●	●	●	●	×	●	●
CSSE [24]	●	●	●	●	×	●	×
IDS [18]	○	○	○	○	○	○	○
Java Dynamic Tainting [25]	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SQLCheck [14]	●	●	●	●	×	●	●
SQLGuard [13]	●	●	●	●	×	●	●
SQLrand [21]	●	×	●	●	×	●	×
Tautology-checker [8]	●	×	×	×	×	×	×
Web App. Hardening [26]	●	●	●	●	×	●	×
JDBC-Checker [4]	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Java Static Tainting [6]	●	●	●	●	●	●	●
Safe Query Objects [27]	●	●	●	●	×	●	●
Security gateway [28]	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SecuriFly [29]	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SQL DOM [30]	●	●	●	●	×	●	●
WAVES [17]	○	○	○	○	○	N/A	○
WebSSARI	●	●	●	●	●	●	●
Proposed method	●	●	●	●	●	●	●

Symbols: ●: possible, ○: partially possible, ×: impossible, N/A: Not Applicable.

The JDBC-Checker, Tautology-checker, WebSSARI and Java Static Tainting use the static analysis method. The static analysis method only analyzes static SQL queries implemented inside the web application and therefore the efficiencies of the methods differ. IDS and WAVES use a machine learning anomaly detection method, which needs a large amount of SQL injection data for learning. The detection rate of the method depends on the learned parameters. JDBC-Checker does not detect the attack, but reduces the chances of SQL injection attacks by checking the type of SQL queries.

Both the proposed algorithm and AMNESIA, SQLCheck and SQLGuard use both static and dynamic methods simultaneously. The proposed method compares static and dynamic SQL queries generated. It detects attacks by comparing the structure and the grammar of the queries. If a dynamically generated query has a different structure or uses a different grammar from that of a static query, it is detected. However, AMNESIA, SQLCheck and SQLGuard use static and dynamic SQL queries for a parse tree. As a result, these methods cannot detect stored procedure type attacks and because the time complexity is $O(n^3)$, it is impossible to detect the attack in real time. Furthermore, the grammar and structure used in various database management systems(DBMSs) differ, which makes the generation of parse trees dependent on the DBMS. On the contrary, the algorithm proposed in this paper does not use complex analysis methods such as parse trees. The time complexity of this algorithm is $O(1)$. It uses a very simple method which compares queries after the removal of attribute values. Therefore, it can be implemented in any type of DBMS and is able to detect SQL injection attacks including stored procedure type attacks.

The dynamic analysis method is not a solution for the detection and prevention of SQL injection attacks. It only finds the vulnerabilities of web applications. Therefore, it will not be considered for the comparison in this paper.

4.2.3. Comparison of additional detection factors

The comparison of additional detection factors is shown in Table 4. We divide the proposed method into three different implementation methods. When a profiling method is used, a proxy server is needed as an element. When a SQL query checking function is used, the developer needs to learn the method and modification of the source code is needed. When the SQL query list method is used, no additional elements are needed.

4.3. Advantage of our system over the other methods

There is a system such as WebSSARI that has the same performance as our proposal. The difference is apparently that WebSSARI uses the partial automatic method for attack prevention compared with our automatic method. And more, the operation of our system is very simple and the complexity time is a constant. Our system requires the pre-analyzed web page (statically, *FDQ*) and the dynamic analyzed web page (Dynamically, *DDQ*). The comparison is performed with only one pass exclusively OR operation for each character.

5. Conclusion

This paper proposed a novel method for detecting SQL injection attacks by comparing static SQL queries with dynamically generated queries after removing the attribute values. Furthermore, we evaluated the performance of the proposed method by experimenting on vulnerable web applications. We also compared our method with other detection methods and showed

Table 4
Analysis of additional elements of each detection and prevention method.

Detection method	Source code adjustment	Attack detection	Attack prevention	Additional elements
AMNESIA [14]	Not needed	Automatic	Automatic	N/A
CSSE [24]	Not needed	Automatic	Automatic	Custom PHP interpreter
IDS [18]	Not needed	Automatic	Report generation	IDS system training set
JDBC-checker [4]	Not needed	Automatic	Source code adjustment proposed	N/A
Java dynamic tainting [25]	Not needed	Automatic	Automatic	N/A
Java static tainting [6]	Not needed	Automatic	Source code adjustment proposed	N/A
Safe query objects [27]	Needed	N/A	Automatic	Developer learning
SecuriFly [29]	Not needed	Automatic	Automatic	N/A
Security gateway [28]	Not needed	Detailed manual	Automatic	Proxy filter
SQLCheck [15]	Needed	Partially automatic	Automatic	Key management
SQLGuard [13]	Needed	N/A	Automatic	N/A
SQL DOM [30]	Needed	Automatic	Automatic	Developer learning
SQLrand [20]	Needed	Automatic	Automatic	Proxy, Developer learning, Key management
Tautology-checker [8]	Not needed	Automatic	Source code adjustment proposed	N/A
WAVES [17]	Not needed	Automatic	Report generation	N/A
Web app. hardening [26]	Not needed	Automatic	Automatic	Custom PHP interpreter
WebSSARI [5]	Not needed	Automatic	Partially automatic	N/A
Proposed algorithm (SQL query check function)	Needed	Automatic	Automatic	Developer learning
Proposed algorithm (SQL query profiling)	Not needed	Automatic	Automatic	Proxy
Proposed algorithm (SQL query listing)	Not needed	Automatic	Automatic	N/A

the efficiency of our proposed method. The proposed method simply removes the attribute values in SQL queries for analysis, which makes it independent of the DBMS. Complex operations such as parse trees or particular libraries are not needed in the proposed method.

The proposed method cannot only be implemented on web applications but it can also be used on any applications connected to databases. Furthermore, it can be used for SQL query profiling, SQL query listing and modularization of detection programs.

Future work is needed for not only SQL injection attacks but also for other web application attacks such as XSS, based on the proposed method and machine learning algorithms.

Acknowledgements

This work was supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

References

[1] The Open Web Application Security Project, OWASP TOP 10 Project. http://www.owasp.org/.
[2] Apache Struts Project, Struts. http://struts.apache.org/.
[3] PHP, magic quotes. http://www.php.net/magic_quotes/.
[4] C. Gould, Z. Su, P. Devanbu, JDBC checker: a static analysis tool for SQL/JDBC applications, in: Proceedings of the 26th International Conference on Software Engineering, ICSE, 2004, pp. 697–698.
[5] Y. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, S.Y. Kuo, Securing web application code by static analysis and runtime protection, in: Proceedings of the 12th International World Wide Web Conference ACM, 2004, pp. 40–52.
[6] V.B. Livshits, M.S. Lam, Finding security errors in Java programs with static analysis, in: Proceedings of the 14th Usenix Security Symposium, 2005, pp. 271–286.
[7] S. Thomas, L. Williams, Using automated fix generation ot secure SQL statements, in: Proceeding of the 29th International Conference on Software Engineering Workshops, ICSEW, IEEE Computer Society, 2007, p. 54.
[8] G. Wassermann, Z. Su, An analysis framework for security in web applications, in: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems, SAVCBS, 2004, pp. 70–78.
[9] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, Y. Takahama, Sania: syntactic and semantic analysis for automated testing against SQL injection, in: Proceedings of the Computer Security Applications Conference 2007, 2007, pp. 107–117.
[10] Paros. Parosproxy.org. http://www.parosproxy.org/.
[11] Y. Shin, Improving the identification of actual input manipulation vulnerabilities, in: 14th ACM SIGSOFT Symposium on Foundations of Software Engineering ACM, 2006.
[12] G. Buehrer, B.W. Weide, P.A. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: Proceedings of the 5th International Workshop on Software Engineering and Middleware, 2005, pp. 105–113.
[13] G. Buehrer, B.W. Weide, P.A.G. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: Proceeding of the 5th International Workshop on Software Engineering and Middleware ACM, 2005, pp. 106–113.

[14] W.G. Halfond, A. Orso, AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 174–183.
[15] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 372–382.
[16] K. Wei, M. Muthuprasanna, S. Kothari, Preventing SQL injection attacks in stored procedures, in: Software Engineering Conference 2006. Australian, 2006, pp. 18–21.
[17] Y. Huang, S. Huang, T. Lin, C. Tasi, Web application security assessment by fault injection and behavior monitoring, in: Proceedings of the 12th International Conference on World Wide Web, 2003, pp. 148–159.
[18] F. Valeur, D. Mutz, G. Vigna, A learning-based approach to the detection of SQL attacks, in: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2005, pp 123–140.
[19] P.-Y. Gibello, Zql: A java SQL parser. http://www.gibello.com/code/zql.
[20] S. Boyd, A. Keromytis, SQLrand: preventing SQL injection attacks, in: Applied Cryptography and Network Security, in: LNCS, vol. 3089, 2004, pp. 292–302.
[21] J. Park, B. Noh, SQL injection attack detection: profiling of web application parameter using the sequence pairwise alignment, in: Information Security Applications, in: LNCS, vol. 4298, 2007, pp. 74–82.
[22] GotoCode. http://www.gotocode.com/.
[23] W.G. Halfond, J. Viegas, A. Orso, A classification of SQL-injection attacks and countermeasures, in: Proceeding on International Symposium on Secure Software Engineering, Raleigh, NC, USA, 2006, pp. 65–81.
[24] T.C. Pietraszek, V. Berghe, Defending against injection attacks through context–sensitive string evaluation, in: Proceeding of Recent Advances in Intrusion Detection, in: LNCS, vol. 3858, 2006, pp. 124–145.
[25] V. Haldar, D. Chandra, Franz, Dynamic Taint propagation for Java, in: Proceedings 21st Annual Computer Security Applications Conference, 2005, pp. 303–311.
[26] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically hardening web application using precise tainting information, in: Twentieth IFIP Iternational Information Security Conference, in: LNCS, vol. 181, 2005, pp. 295–307.
[27] W.R. Cook, S. Rai, Safe query objects: statically typed objects as remotely executable queries, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 97–106.
[28] D. Scott, R. Sharp, Abstracting application-level web security, in: Proceedings of the 11th International Conference on the World Wide Web, 2002, pp. 396–407.
[29] M. Martin, B. Livshits, M.S. Lam, Finding application errors and security flaws using PQL: a program query language, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, 2005, pp. 365–383.
[30] R. McClure, I. Krüger, SQL DOM: compile time checking of dynamic SQL statements, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 88–96.