Journal of Engineering and Applied Sciences 13 (17): 7162-7170, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

Evolutionary Search Method for Removal of SQL Injection Vulnerabilities

[1]K. Umar, [2]A.B. Sultan, [2]H. Zulzalil, [2]N. Admodisastro and [2]M.T. Abdullah
[1]Faculty of Computer Science and Information Technology, Bayero University, Kano, Nigeria
[2]Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Selangor, Malaysia

Abstract: Existing literature focuses more on describing SQL Injection Attacks (SQLIAs) and less on describing SQL Injection Vulnerabilities (SQLIVs), even though, the former is carried out to exploit the later. This study describe root causes of SQLIVs and illustrates how SQLIVs could be exploited using different types of SQLIAs. The study, also, presents proposal of a new method for automated detection and removal of SQLIVs. The new method employs grammar reachability analysis to define enhanced static source code analysis for detection of SQLIVs and employs Evolutionary Programming (EP) search strategy to automate source code modification for removal of SQLIVs. Preliminary experimental results show that the new method is feasible and promising.

Key words: SQL injection, vulnerabilities, attacks, exploitation, detection, removal

INTRODUCTION

Empirical evidences have shown that SQL Injection Vulnerabilities (SQLIVs) exist in web application if externally supplied input data containing malicious SQL commands can get into generation and execution of SQL query statements at runtime (Halfond *et al*., 2006; Garg and Singh, 2013; Kumar and Pateriya, 2012; Shar and Tan, 2013; Johari and Sharma, 2012; Anonymous, 2011, 2017). These kinds of vulnerabilities are exploited through several types of SQL Injection Attacks (SQLIAs) which can result in devastating consequences including basic information disclosure information theft, remove code execution, denial of service and total system compromise among others. Static analysis techniques are among most widely used approaches for detection and removal of SQLIVs. However, grammar reachability analysis has not been investigated for detection of SQLIVs. Similarly, Evolutionary Programming (EP) search has not been investigated for automating source code modification for SQLIVs removal.

This study presents the root causes of SQLIVs with the aid of an illustrative example. The study demonstrates how a vulnerable web application can be exploited using first order SQLIAs. Lastly, the study presents proposal of method for automated detection and removal of SQLIVs for web application.

SQL injection vulnerabilities: This study describes security flaws that easily leads to SQL Injection Vulnerabilities (SQLIVs) for web application. SQLIVs refer to software security loopholes or flaws which allows malicious SQL commands to be injected via. input data into an application (Halfond *et al*., 2006; Garg and Singh, 2013; Kumar and Pateriya, 2012; Shar and Tan, 2013; Johari and Sharma, 2012; Anonymous, 2011, 2017). In web applications, the injected malicious SQL commands change the structure and outcome of query statements that are generated and executed dynamically. Insecure coding practice can easily results in vulnerable application. This occurs where developer is either ignorant of techniques that ensure application's security or fails to implement them effectively. Obviously, most programming languages do provide built-in mechanisms for securing an application and preventing attacks. For example, PHP provides "mysql_escape_string()" function that validates input data for preventing injection of malicious SQL commands. Unfortunately, not all developers make effective use of language provided security mechanisms. Poor input data validation and poor out sanitization are among leading causes of SQLIVs. The former happens when an application allows inclusion of externally, supplied input data into generation of dynamic queries without sufficiently checking the safety and legitimacy of the input data (Garg and Singh, 2013; Shar and Tan, 2013; Johari and Sharma, 2012). Whereas, the later happens when likely output of dynamic query is not adequately checked prior to execution of the query on database server. Of cause, the former as well as the later are all forms of insecure coding practice.

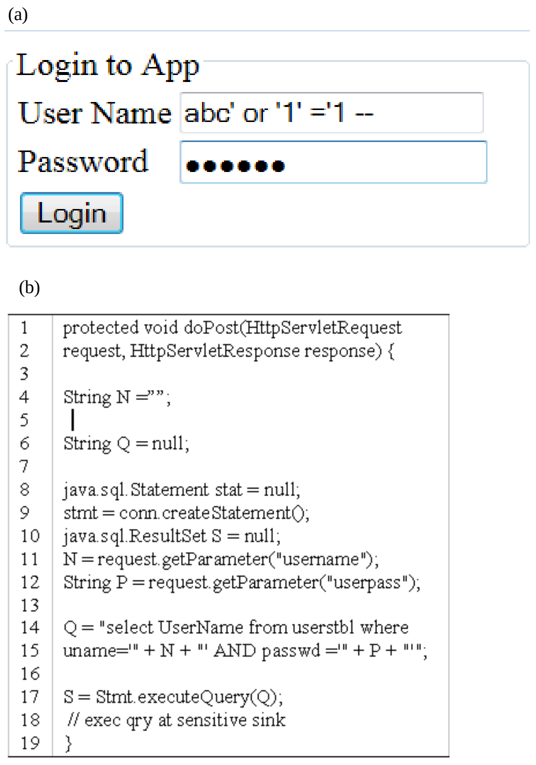
Corresponding Author: K. Umar, Faculty of Computer Science and Information Technology, Bayero University, Kano, Nigeria

Error message feedback that are generated by database server and displayed in client's browser can be used by an attacker to detect the type, version or structure of the underlying database. Dynamic query generation can also, lead to SQLIVs. This happens when an application includes un-validated and un-sanitized externally supplied input data into building and execution of query statements dynamically. In such applications, the "input data variables" in the dynamic query can be exploited by attacker using, virtually, all type of SQLIAs. The use of both system built-in and programmer-built stored procedures can also, lead to SQLIVs. This happens when input arguments to stored procedure are not properly validated or when output from stored procedure is not adequately sanitized. Generous privilege is often considered as insecure coding practice that can also, lead to SQLIVs. For instance, it is not uncommon practice to have an 'admin' or 'user' account with more privileges that necessary, i.e., the account has privileges for accessing resources that are not required by user of the account. Where an attacker is able to hijack such privileged account and bypass authentication, he/she would have all the privileges associated with the account. Good and secure coding practice is to give as few as required privileges to any account and to always change account privileges as the user's functions/responsibilities changes (Garg and Singh, 2013; Shar and Tan, 2013; Johari and Sharma, 2012; Anonymous, 2017).

MATERIALS AND METHODS

Example vulnerable web application: This study presents sample JSP web application that is used as a running example to illustrate root causes of SQLIVs as described in preceding study. The running example is used to illustrate how SQLIVs is exploited. The sample JSP web application consists of single webpage and perform basic user authentication. The application provides login form containing two data input fields. The first data input field is the "username" field. The second data input field is the "Password" field. Both fields are not validated and therefore are vulnerable to SQL injections. Hence, the application contains SQLIVs which can be exploited using any type of SQLIAs.

The login form containing two data input fields "username" and "Password" is shown in Fig. 1a. These fields are for collecting data from external source, i.e., user. The code fragment from JSP servlet that receives the input data and perform basic authentication is shown in Fig. 1b. Let us explain how the application works. In order to authenticate user, the application collects "username" and "password" through the login form. The collected


(a)

Login to App

User Name: abc' or '1' ='1 --

Password: ●●●●●●

(b)

```
1   protected void doPost(HttpServletRequest
2   request, HttpServletResponse response) {
3
4   String N ="";
5    |
6   String Q = null;
7
8   java.sql.Statement stat = null;
9   stmt = conn.createStatement();
10  java.sql.ResultSet S = null;
11  N = request.getParameter("username");
12  String P = request.getParameter("userpass");
13
14  Q = "select UserName from userstbl where
15  uname='" + N + "' AND passwd ='" + P + "'";
16
17  S = Stmt.executeQuery(Q);
18   // exec qry at sensitive sink
19  }
```

Fig. 1: Sample JSP web application for running example: a) Sample login form (with user input) and b) Code fragment from JSP servlet that receives the input data from from in (a) and perform basic user authentication)

data (username and password) are included in building and execution of dynamic query that checks "userstbl" table for existing record that matches the collected username and password. If match is found, the user is authenticated as legitimate and allowed to login. However, if no match is found, the login attempt is rejected through display of error message: "invalid username or password" and the user is not allowed into the application.

The authentication process that happens inside the application is here explained. Typically, the input data collected from user are stored in variables as in lines 11 and 12 of Fig. 1b. Thereafter, the variables may (or may not) be validated. Observe that the sample application performs no data validation for variable N (which stores username) and also, no data validation for variable P (which stores password). These variables are subsequently used in generation of dynamic query string at line 14. Finally, the dynamically generated query string is executed by database server at line 17. The result of the dynamic query execution is used for the authentication purpose. If matching record is found, the query would returns the record from database and consequently, the application authorize the login attempt. If no matching record is found, the query returns empty recordset and consequently, the login attempt would fail. Ideally input data sanitization or validation should happen before or at generation of query string (line 14) and presumably, prior to query execution (line 17). Absent of input data sanitization or validation is among root causes of SQLIVs. Hence, in the code fragment of Fig. 1b, both username field and password field are vulnerable to SQL injection and can be exploited by hacker.

RESULTS AND DISCUSSION

Illustration of root causes of SQL injection vulnerability: This study uses the running example of Fig. 1 to illustrate root causes of SQLIVs. To achieve this, first, we describe stages of generation and execution of dynamic query and then, explain root causes of SQLIVs alongside. In web application, generation and execution of dynamic query involves three basic stages that span over the code fragment of Fig. 1b. The three stages are described below.

Stage 1; Assign input data to variable: User submits input data through data entry fields on the login form. The submitted data gets assigned to variables inside the application as in lines 11 and 12. It is important to mention that the point at which submitted data is assigned to variable such as line 11 and line 12 in Fig. 1b is referred to as Application's Entry Point (AEP) (Medeiros *et al.*, 2014). The variables are used at a later stage for building dynamic query.

Stage 2; Input data validation: At this stage, the variables that stored input data should be validated or sanitized. Basically, sanitization or validation is to ensure that input data are safe for use inside an application. Once, validation is done or no validation is done (as in lines 11 and 12), the input data variables are used in building of dynamic query string. In the running example, dynamic query string is built at lines 14 and 15.

Stage 3; Query execution: In the last stage, generated dynamic query string is executed by database server. The query execution is done by invoking database server function with the query string as an argument. Notes that, the point at which dynamic query is executed such as line 17 in Fig. 1b is referred to as sensitive sink (Medeiros *et al.*, 2014). With reference to Fig. 1, ideally, validation of input data should happen at stage 2, before execution of dynamic query at stage 3. Unfortunately, the running example does not perform data validation at all. Thus, it is associated with some root causes of SQLIVs. It is obvious that the application fails to perform data validation for the two submitted input data, namely username and password. This kind of missing data validation for input fields constitutes typical example of insecure coding practice which results in SQLIVs. Moreover, the running example generates and executes dynamic query. As explained in preceding study inclusion of un-validated input data into building and execution of dynamic query results in SQLIVs. Unfortunately, the running example includes the input data collected via. username and password fields into building and execution of dynamic query without validation. Hence, besides insecure coding practice associated with the running example, it is apparent that, generation of dynamic query is another root cause of SQLIVs inherent in the running example web application. Having illustrated some root causes of SQLIVs and how they occur in the running example vulnerable web application in the following study, we demonstrate how SQLIV can be exploited using the 7 types of first order SQLIAs.

Exploiting vulnerable web application: This study describes how SQLIVs can be exploited using different types of SQL Injection Attacks (SQLIAs). The exploitation is demonstrated using 7 types of SQLIAs, namely, tautologies, logically incorrect queries, union queries, piggy-backed queries, stored procedures, inference and alternate encodings.

Tautology attack: This attack injects code that adds one or more conditional sub-expression into SQL query, thereby, making the query to always evaluate to true. The attack is carried out by injecting malicious SQL string through input data submitted to an application (Halfond *et al.*, 2006; Abawajy, 2013; Balasundaram and Ramaraj, 2012). In order to exploit the SQLIVs in the running example using tautology attack an attacker could submits the malicious string "*or 1 = 1-" for 'username' input field and submits blank in password field. The resulting dynamically generated SQL query string would be as shown in Algrothim 1. As shown in Algrothim 1, the injected malicious string (*or 1 = 1) adds a conditional sub-expression to the first part of the WHERE clause. This causes it to always evaluate to true and there by transforming the query to tautology behavior. The second part of the WHERE clause starting from "AND Userpass = """ is transformed into comment by the injected comment characters (--). The effect of this is that the query would evaluates to true and the application would grant access to the attacker, thus, causing authentication bypass (Halfond *et al*., 2006; Balasundaram and Ramaraj, 2012; Cecchini and Gan, 2013).

Logically incorrect queries attack: This attack reveals important information through display of error messages in the client's browser such as the type, version and structure of database underlying a web application. The attack is carried out by injecting malicious SQL string which causes syntax error, type conversion error or logical error. The database server responds by generation and display of appropriate error message in client's browser. An attacker infers useful information from the error messages (Halfond *et al*., 2006; Shar and Tan, 2013; Abawajy, 2013; Balasundaram and Ramaraj, 2012). In order to exploit the SQLIVs in the running example using this type of attack an attacker could submits the malicious string "convert(int, (SELECT top 5 name from userstbl where userType = '0')) --" in the "Username" field and submits blank for the "Password". The resulting dynamically generated SQL query would be as shown in Algrothim 2. In this example, a select query is injected through username field. The injected select query attempts to extract the top five first-names (fname) from users table where type of user is 0. However, the injected malicious string attempts to perform an illegal type conversion of a recordset into an integer. This would make the database server to throws an error. For instance with Microsoft SQL Server, the error would be as follows:

Algrothim 1; Example tautology attack SQL query:

"Microsoft OLE DB provider for SQL Server (0×80040E07) error converting nvarchar value 'AbdulKarim' to a column of data type int"

```
"SELECT UserName FROM userstbl WHERE
 uname=" " or 1=1 -- and passwd=""
```

Algrothim 2; Example logically incorrect attack SQL query:

```
SELECT UserName FROM userstbl WHERE uname = convert
(int,(select top 5 fname FROM userstbl WHERE userType='0')) --
AND passwd = ""
```

Clearly, from the above error message, the attacker could know that: the underlying database server is Microsoft SQL server and the first name in the table is 'AbdulKarim'. By repeatedly performing variations of the above attack an attacker could finger-print the database, discover vulnerable parameters and extract vital information (Halfond *et al*., 2006; Kumar and Pateriya, 2012; Shar and Tan, 2013; Balasundaram and Ramaraj, 2012).

Union queries attack: This attack is used to change the recordset returned by a given query, thus, extracting data in the process. This attack could make an application to return data from table different from the one intended by original query. The returned recordset becomes the union of results from the original query and the injected query. The attack is carried out by injecting a SQL query string that insert SELECT query into dynamically generated query using the UNION keyword. In order to exploit the SQLIVs in the running example using this type of attack, first, let us assume that an attacker has performed logically incorrect query attacks and discovered 'CreditCardsTbl' as a table in the database. The attacker could inject the malicious string "' UNION SELECT *from CreditCards --" through the username field and submits blank in the password field. The resulting dynamically generated SQL query would be as shown in Algrothim 3. In this example an addition select query is injected into the original query. The original query returns "empty recordset" because there is no "uname" equals to blank in the database. The injected query returns all records from the "CreditCards" table. The database combines, (i.e., unions) the results from the two queries (original and injected) and returns it to the application, thus, revealing credit card information to the attacker (Balasundaram and Ramaraj, 2012; Shahriar and Zulkernine, 2012; Lee *et al*., 2012).

Piggy-backed queries attack: This type of attack injects additional query without modifying the original developer intended query. Thus, the database server considers the injected query and the original query as two separate queries and are both executed alongside each other. The attack is carried out by injecting additional query as an attachment to the original query, i.e., the added query is "piggy backed" on the original query. Thus, the database server receive and execute multiple queries. In order to exploit the SQLIVs in the running example using this type of attack an attacker could submits the value "kbr" for the username and submits the malicious string"; exec (SHUTDOWN) - - " for password field. The resulting dynamically generated SQL query would be as shown in Algrothim 4. As shown in Algrothim 5, the injected SQL query is appended to the original query (separated by ";" mark). The original query, (i.e., first query) returns "empty recordset" because no user "kbr" with blank password exist in the database. The injected query (i.e., second query) would cause the database server to execute SHUTDOWN command. This attack would performing remote code execution leading to denial of service by shutting down the database server (Halfond *et al*., 2006; Balasundaram and Ramaraj, 2012; Shahriar and Zulkernine, 2012).

Algrothim 3; Example union query attack SQL query:

```
SELECT * FROM userstble WHERE uname="" UNION SELECT
* from
CreditCards -- AND passwd = ""
```

Algrothim 4; Example pigy-backed attacked SQL query:

```
SELECT * FROM userstble WHERE uname = kbr AND
passwd = “”; exec (SHUTDOWN) - -
```

Stored procedures: This type of attack execute built-in stored procedures as well as developer built stored procedures. The attack is carried out by injecting malicious string into arguments of a stored procedure and then triggering execution of the procedure (Halfond *et al*., 2006; Abawajy, 2013; Balasundaram and Ramaraj, 2012). In order to exploit the SQLIVs in the running example using this type of attack, first, let us consider the parameterized stored procedure shown in Algrothim 5. The stored procedure performs basic user authentication and returns a true/false value to indicate whether the user's credentials are authenticated correctly or not. Username and password are passed to the stored procedure via. the parameters @userID. and @Userpass, respectively. Let us further assume that the running example uses invokes the stored procedure for the authentication by passing username and password as parameters. An attacker could submit the value “kbr” for the username and submits the malicious string “ ; exec (SHUTDOWN); - -“ for password. This would cause the stored procedure to generate the dynamic query shown in Algrothim 6. As shown in Algrothim 6, the attack would result in two queries (as in piggy-backed query attack). The first query, (i.e., original query) is executed normally as intended by the developer. The second injected malicious query is executed alongside the first query.

Algrothim 5; Example parameterized stored procedure:

```
CREATE PROCEDURE DBO isAuthenticated
@userID varchar2, @Userpass varchar2
AS EXEC("SELECT UserName FROM userstbl
WHERE uname = '" +@userID+ "' AND passwd
= '" +@Userpass)
GO
```

Algrothim 6; Dynamic query from attacked stored procedure:

```
SELECT UserName FROM userstbl WHERE uname = kbr and
passwd = ”” ;
exec(SHUTDOWN) --
```

In this example, the injected query causes the database server to execute SHUTDOWN command and thus, resulting in denial of service (Halfond *et al*., 2006; Kumar and Pateriya, 2012; Balasundaram and Ramaraj, 2012).

Inference attack: This type of attack injects malicious code that modifies original query into form of action which is executed conditionally. After the attack is performed, the response and behavior of the website is carefully observed. From the website's responses an attacker could infer certain vulnerable parameters, database schema information and so on. There are two variations of this type of attack (Balasundaram and Ramaraj, 2012; Lee *et al*., 2012; Focardi *et al*., 2012).

Blind injection. This variation of inference attack asks the server true/false questions and infer answer from behavior of website after attack. If website continues to function normally after attack, it means the injected malicious string evaluates to true. If the observed behaviour changes after attack, it means the injected malicious string evaluates to false.

Timing attack: This variation of inference attack injects malicious string in form of an “IF, …, THEN, …,” structure with multiple branches. The branches contain SQL constructs that take a known amount of time to execute such as WAITFOR command. By measuring website's response time after different attacks an attacker could infer which branch was followed in the injected string and also, infer some information as to why such branch was taken. Let us demonstrate how the SQLIVs in our running example could be exploited using blind injection attack. Assuming an attacker wants to check whether the username field is vulnerable or not. The attacker needs to have a valid user name and password for this purpose. Let us assume that “Smith” is a valid user name with password “Kline”. In order to check the username field, the attacker needs to perform two separate attacks and observe response of each. The first attack submits the malicious string “Smith AND 1 = 1 - -” for username field and submits “Kline” for password field. The resulting dynamically generated SQL query would be as shown in Algrothim 7. The second attack submits the malicious string “Smith AND 1 = 0 - -” for username field and submits “Kline” for password field. The resulting dynamically generated SQL query would be as shown in Algrothim 8.

Algrothim 7; Dynamic query for first attack:

```
SELECT UserName FROM userstbl WHERE
uname = Smith AND 1 = 1 AND passwd = Kline
```

Algrothim 8; Dynamic query for second attack:

```
SELECT UserName FROM userstbl WHERE
uname = Smith AND 1 = 0 AND passwd = Kline
```

It can be seen from Algrothim 7 and 8 that the two dynamic queries resulting from first and second attacks are slightly different. Consequently, the attacker would observe the responses of the application after each attack.

If the application returns error message for the two attacks it means there is proper input data validation and username field is not vulnerable. However, if only the second attack returns an error message, it means there is no input data validation and username field is vulnerable (Balasundaram and Ramaraj, 2012; Lee *et al.*, 2012; Focardi *et al.*, 2012).

Alternate encodings attack: This type of attack injects malicious string written in alternate method of encoding such as hexadecimal ASCII and unicode character encoding. The attack allows an attacker to avoid (or dodge) detection and exploit vulnerabilities that might not otherwise be exploitable. To perform this type of attack an attacker converts malicious string into different encoding system before injecting into vulnerable field of an application (Halfond *et al.*, 2006; Lee *et al.*, 2012; Focardi *et al.*, 2012). In order to exploit the SQLIVs in the running example using this type of attack an attacker could submits the malicious string "; exec (char (0x73687574646f776e)) - - " for username field and submits blank for password field. The resulting dynamically generated SQL query would be as shown in Algrothim 9.

Algrothim 9; Example alternate encoded attack SQL query:

```
SELECT UserName FROM userstbl WHERE uname = ;
exec(char(0x73687574646f776e)) - - AND passwd =""
```

This example injected malicious string in an alternate encoded form. The string "0x73687574646f776e" is the hexadecimal encoding of "SHUTDOWN". Therefore, the function call char (0x73687574646f776e) would return the SQL command SHUTDOWN and consequently, the database server would execute the command. Consequently, the above attack would results in denial of service by shutting down the database server (Halfond *et al.*, 2006; Lee *et al.*, 2012; Focardi *et al.*, 2012).

Automated detection and removal of SQLIVs: Static analysis techniques are among most widely used approaches for detection and removal of SQLIVs during testing phase of web application's development. Although, there have been considerable number of static analysis techniques for SQLIVs detection (Almorsy *et al.*, 2012; Shar and Tan, 2013a, b; Razzaq *et al.*, 2009; Qu *et al.*, 2013; Gupta *et al.*, 2014; Fu and Qian, 2008; Gould *et al.*, 2004), the area of SQLIVs removal has not been adequately explored. In fact, very few static analysis techniques addressed removal of SQLIVs (Medeiros *et al.*, 2014; Thomas *et al.*, 2008; Al-Khashab *et al.*, 2011). Obviously, it would be highly desirable to have a tool that can take-in source code of vulnerable web application as input and produce reliably secure version which is ready for deployment to live environment. This kind of tool would reduce human efforts and costs associated with testing phase of web application's development and thus, facilitates improvement of software quality.

Consequently, we employ context free grammar reachability analysis to define enhanced static source code analysis for automated detection of SQLIVs. In addition, we employ Evolutionary Programming (EP) search strategy to automate source code modification for SQLIVs removal. The conceptual model of our method is depicted in Fig. 2. As shown in the figure, conceptually, the key goal of the method is to receive input of source code of vulnerable web application (WAuT) and produce output of modified (and secured) version of the web application (WAuT_new). For achieving the stated goal, the method consists of four main components, namely, grammar rules extractor component, EP search component, SQLIVs detector component and SQLIVs remover component. The components perform series of inter-dependent processing activities in such a way that output from one component serves as input to the subsequent component.

The processes performed by the components and sequence of interactions between them is diagrammatically represented using activity diagram of Fig. 3. With the aid of Fig. 2 an overview of activities

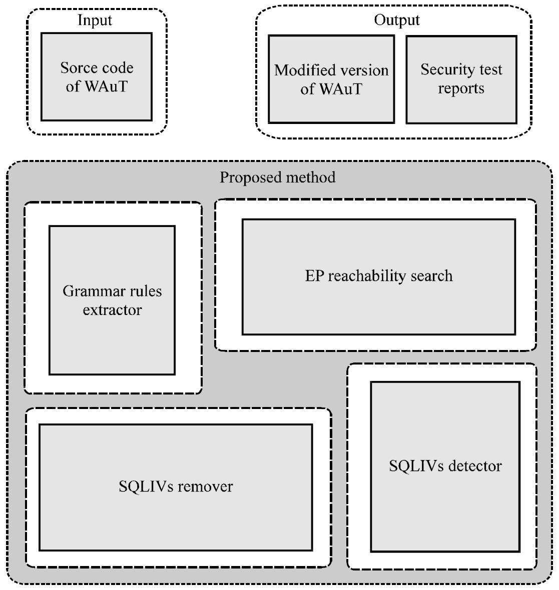


Fig. 2: Activity diagram for the new method

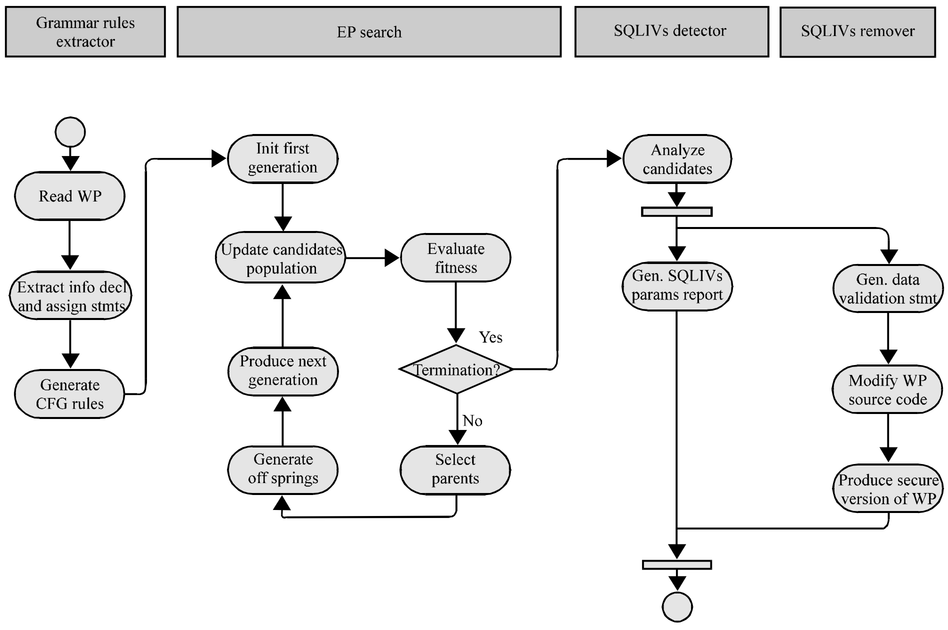


Fig. 3: Conceptual model of the method

performed by the method for automated detection and removal of SQL injection vulnerabilities is presented below.

Grammar rules extractor component: As shown in Fig. 3, the working of the method starts in the grammar rules extractor component. The component reads and analyzes source code file of subject web application. It uses string analysis based parsing to recognize all declaration and assignment statements from the source code and then extracts CFG production rules for each of the extracted statements.

EP search component: The collection of extracted CFG production rules serve as input to second component, i.e., EP search component which performs EP reachability search. The component uses EP search to evolve candidates which are represented as productions sequences. The EP search process begins by seeding first generation with randomly built candidate solutions. Fitness of candidates is evaluated to check for optimal solutions. Thereafter, if optimality is not reached, the component selects parent, produce offspring and combines them to produce population of next generation. The EP search process is repeated through generations. When optimal result is achieved, the EP search process stops and forward all optimal solutions to the third component for analysis.

SQLIVs detector component: The component performs grammar reachability analysis. This component analyzes all collected optimal solutions and sort then into two categories, namely, level 1 fitness optimal solutions (which reveals SQLIV parameters) and level 2 fitness optimal solutions (which reveals validated parameters). The component passes all level 1 fitness optimal solutions to the fourth component for source code modifications. In addition, this component generates report of vulnerable parameters as well as validated parameters.

SQLIVs remover component: This component constitutes part of the EP search variation operations and performs source code modification for SQLIVs removal. The component receives all level 1 fitness optimal solutions and generates appropriate data validation statement for each associated SQLIV parameter. Finally, the component produce modified version of a subject webpage containing insertions of the auto generated data validation statements at appropriate location.

CONCLUSION

The root causes of SQL injection vulnerabilities for web application were presented in this study. In addition, an illustrative example was used to demonstrate how SQL injection vulnerabilities can be exploited using several types of first order SQL injection attacks. Thereafter, a proposal of new method for automated detection and removal of SQL injection was presented. The new method leverages grammar reachability analysis to enhance SQLIVs detection through static analysis approach and also, leverages Evolutionary Programming (EP) search process to automate source code modification for SQLIVs removal.

RECOMMENDATIONS

In our future work, we plan to fully define and develop all the components and processes of the newly proposed method for automated detection and removal of SQLIVs for web application.

ACKNOWLEDGEMENTS

We acknowledge that, this research received support from the Fundamental Research Grant Scheme FRGS/1/2015/ICT01/UPM/02/12 awarded by Malaysian Ministry of Education to the Faculty of Computer Science and Information Technology at Universiti Putra Malaysia.

REFERENCES

Abawajy, J., 2013. SQLIA detection and prevention approach for RFID systems. J. Syst. Software, 86: 751-758.

Al-Khashab, E., F.S. Al-Anzi and A.A. Salman, 2011. PSIAQOP: Preventing SQL injection attacks based on query optimization process. Proceedings of the 2nd Kuwait Conference on E-Services and E-Systems, April 5-7, 2011, Kuwait, USA.

Almorsy, M., J. Grundy and A.S. Ibrahim, 2012. Supporting automated vulnerability analysis using formalized vulnerability signatures. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), September 3-7, 2012, IEEE, Essen, Germany, ISBN:978-1-4503-1204-2, pp: 100-109.

Anonymous, 2011. CWE-SANS top 25 most dangerous software errors. Common Weakness Enumeration (CWE), Virginia, USA. http://cwe.mitre.org/top25/

Anonymous, 2017. Category: OWASP top 10 project. OWASP, Maryland, USA. https://www. owasp. org/index.php/Category:OWASP_Top_Ten_Project

Balasundaram, I. and E. Ramaraj, 2012. An efficient technique for detection and prevention of SQL injection attack using ASCII based string matching. Procedia Eng., 30: 183-190.

Cecchini, S. and D. Gan, 2013. SQL injection attacks with the AMPA suite. Intl. J. Electron. Secur. Digital Forensics, 5: 139-160.

Focardi, R., F.L. Luccio and M. Squarcina, 2012. Fast SQL blind injections in high latency networks. Proceedings of the 2012 IEEE 1st AESS European Conference on Satellite Telecommunications (ESTEL), October 2-5, 2012, IEEE, Rome, Italy, ISBN:978-1-4673-4687-0, pp: 1-6.

Fu, X. and K. Qian, 2008. SAFELI: SQL injection scanner using symbolic execution. Proceedings of the 2008 International Workshop on Testing, Analysis and Verification of Web Services and Applications, July 21, 2008, ACM, New York, USA., ISBN:978-1-60558-053-1, pp: 34-39.

Garg, A. and S. Singh, 2013. A review on web application security vulnerabilities. Intl. J. Adv. Res. Comput. Sci. Software Eng., 3: 222-226.

Gould, C., Z. Su and P. Devanbu, 2004. JDBC checker: A static analysis tool for SQL/JDBC applications. Proceedings of the 26th International Conference on Software Engineering, May 23-28, 2004, IEEE, Washington, DC, USA., pp: 697-698.

Gupta, M.K., M.C. Govil and G. Singh, 2014. An approach to minimize false positive in SQLI vulnerabilities detection techniques through data mining. Proceedings of the 2014 International Conference on Signal Propagation and Computer Technology (ICSPCT), July 12-13, 2014, IEEE, Ajmer, India, ISBN:978-1-4799-3140-8, pp: 407-410.

Halfond, W.G., J. Viegas and A. Orso, 2006. A classification of SQL-injection attacks and countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering, March 13-15, 2006, Washington, DC., USA.

Johari, R. and P. Sharma, 2012. A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. Proceedings of the 2012 International Conference on Communication Systems and Network Technologies (CSNT), May 11-13, 2012, IEEE, Rajkot, India, ISBN:978-1-4673-1538-8, pp: 453-458.

Kumar, P. and R.K. Pateriya, 2012. A survey on SQL injection attacks, detection and prevention techniques. Proceedings of the 3rd International Conference Computing Communication and Networking Technologies, July 26-28, 2012, Coimbatore, India, pp: 1-5.

Lee, I., S. Jeong, S. Yeo and J. Moon, 2012. A novel method for SQL injection attack detection based on removing SQL query attribute values. Math. Comput. Modell., 55: 58-68.

Medeiros, I., N.F. Neves and M. Correia, 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. Proceedings of the 23rd International Conference on World Wide Web, April 07-11, 2014, ACM, New York, USA., ISBN:978-1-4503-2744-2, pp: 63-74.

Qu, B., B. Liang, S. Jiang and Y. Chutian, 2013. Design of automatic vulnerability detection system for Web application program. Proceedings of the 2013 IEEE 4th International Conference on Software Engineering and Service Science (ICSESS), May 23-25, 2013, IEEE, Beijing, China, ISBN:978-1-4673-4997-0, pp: 89-92.

Razzaq, A., A. Hur, N. Haider and F. Ahmad, 2009. Multi-layered defense against web application attacks. Proceedings of the 6th International Conference on Information Technology: New Generations ITNG'09, April 27-29, 2009, IEEE, Las Vegas, Nevada, USA., ISBN:978-1-4244-3770-2, pp: 492-497.

Shahriar, H. and M. Zulkernine, 2012. Information-theoretic detection of SQL injection attacks. Proceedings of the 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), October 25-27, 2012, IEEE, Omaha, Nebraska, ISBN:978-1-4673-4742-6, pp: 40-47.

Shar, L.K. and H.B.K. Tan, 2013a. Defeating SQL injection. Comput., 46: 69-77.

Shar, L.K. and H.B.K. Tan, 2013b. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. Inf. Software Technol., 55: 1767-1780.

Thomas, S., L. Williams and T. Xie, 2009. On automated prepared statement generation to remove SQL injection vulnerabilities. Inf. Software Technol., 51: 589-598.