Front. Electr. Electron. Eng. China 2008, 3(2): 162–166
DOI 10.1007/s11460-008-0047-x

RESEARCH ARTICLE

Zhefei ZHANG, Qinghua ZHENG, Xiaohong GUAN, Qing WANG, Tuo WANG

A method for detecting code security vulnerability based on variables tracking with validated-tree

Abstract SQL injection poses a major threat to the application level security of the database and there is no systematic solution to these attacks. Different from traditional run time security strategies such as IDS and firewall, this paper focuses on the solution at the outset; it presents a method to find vulnerabilities by analyzing the source codes. The concept of validated tree is developed to track variables referenced by database operations in scripts. By checking whether these variables are influenced by outside inputs, the database operations are proved to be secure or not. This method has advantages of high accuracy and efficiency as well as low costs, and it is universal to any type of web application platforms. It is implemented by the software code vulnerabilities of SQL injection detector (CVSID). The validity and efficiency are demonstrated with an example.

Keywords vulnerability detection, database security, SQL injection

1 Introduction

It is a hot research topic to determine system security status using formalized methods, which include detecting bugs in software systems, mining vulnerabilities in scripts, and identifying the security status of protocols. In this paper, based on previous work, we develop a new method of detecting specific vulnerabilities, SQL injection, in the scripts of Web application to secure database operations.

However, there still exists a continuing problem – how to prevent the system from SQL injection attack? One solution to this problem is detecting attacks between the scripts level and database level at run time. It is usually implemented by parsing the SQL strings, whose scripts are delivered into the database for a set of signatures. Compiling techniques [1] were used to parse the structure of the SQL strings and finite state machine techniques [2] were used to analyze the exact logic of the SQL strings. However, although the approach can effectively detect attacks at run time, it inevitably brings run time load to the Web server at the same time. Another solution is to directly analyze the code for logic vulnerabilities with code analysis techniques from software engineering, whereby software testing methods such as data flow analysis[3,4] and type system[5,6] are modified to detect SQL injection vulnerabilities. Typical approaches are the bounded model check [7] and context sensitive analysis [8]. One intrinsic advantage of these approaches is that the static analysis in scripts rather than run time detection focuses on the cause of the problem, while bringing in no extra run time load.

Based on previous work, we introduce a new concept of validated tree that reversely tracks the variables referenced by database operations to determine whether or not these variables have been influenced by inputs from outside. In this way, database operations in scripts can be saved and in addition to zero run time burdens, there are some other advantages. For instance:

1) There are definite reports of vulnerabilities and no positive false events that are confused in the results of IDS.

2) It can locate vulnerabilities in exact lines and track the cause to help programmers modify their codes.

3) It is independent of any specific type of Web application platforms, which may be configured with different script languages and database systems.

The remainder of this paper is organized as follows. In Sect. 2 the SQL injection will be briefly reviewed. Based on

Translated from *Journal of Xi'an Jiaotong University*, 2007, 41(4): 439–443 [译自: 西安交通大学学报]

Zhefei ZHANG (✉), Qinghua ZHENG, Xiaohong GUAN, Qing WANG, Tuo WANG
MOE Key Lab for Intelligent and Network Security, Xi'an Jiaotong University, Xi'an 710049, China
State Key Laboratory for Manufacturing Systems Engineering, Xi'an Jiaotong University, Xi'an 710049, China
E-mail: zfzhang@sei.xjtu.edu.cn

Xiaohong GUAN
Center for Intelligent and Networked Systems, Department of Automation, Tsinghua University, Beijing 100084, China

the mechanism of SQL injection attack, the reverse tracking of variables with validated-tree is developed in Sect. 3. In particular, the formalized algorithm description of the reverse tracking is introduced in Sect. 3.5, and a mathematical model to calculate the security rank of a Web application is developed in Sect. 3.6. Finally, the implementation of the software code vulnerabilities of SQL injection detector (CVSID) is briefly provided in Sect. 4.

For convenience and clarity, we will use the PHP language and the MYSQL database in all code sections to illustrate our approach.

2 Mechanism of SQL injection attack

Numerous scripts of Web applications are filled with code sections like

$result=mysql_db_query('sample_db',"SELECT * FROM table_name WHERE user='$user' and psw='$psw' ");

If an attacker set the value of the variable $user in the URL request as

admin' or 1=1 #

Then, the actual code executed on the Web server is

$result=mysql_db_query('sample_db',"SELECT * FROM table_name WHERE user= 'admin' or 1=1#' and psw='$psw' ");

Since the code after the comment mark # will be filtered by the database language parser at run time, the exact code executed in the database is

SELECT * FROM admin WHERE user='admin' or 1=1

As a result of the carelessness of a programmer, an attacker can easily enter into an authorized system. All of the security problems are attributed to variables without any constraint.

Because of the carelessness of programmers in coding, it is easy to find this kind of vulnerabilities on the Internet. They usually exist on campus, in the institute, government and other information post Web sites, and pose threats to the application level security.

3 Variables tracking with validated-tree

The cause of SQL injection vulnerabilities is that the variables referenced by database operations can be accessed without any verification by the attacker. The method we discussed in this paper tracks the exact variables from database operation functions like mysql_db_query() to the start of the program. Since the tracking process is like a growing tree, it is named validated-tree.

To describe our approach, we give a code section as an example here, which will be discussed in this paper thoroughly.

```
<?php
…
40: $col_1='name';
41: $col_2='time';
42: $first_name='wang';
43: $name=$first_name . $last_name;
44: $level='admin';
45: $id=filter($id);
46: $table_name='sample_table';
…
76: mysql_connect('dbhost','dbuser','test') or die ("Database connection failed");
77: $sql = "SELECT * FROM $table_name WHERE name=$name";
78: $result_1 = mysql_db_query ('sample_db', $sql);
79: $sql = "SELECT $col_1 , $col_2 FROM $table_name WHERE level=$level ";
80: $result_2 = mysql_db_query('sample_db', $sql);
81: $sql = "SELECT * FROM $table_name WHERE id=$id";
82: $result_3= mysql_db_query('sample_db', $sql);
…
?>
```

3.1 Validated point

The validated point is the starting point from which the tracking process begins, wherein generally the database operation function is included. The SQL strings are finally delivered into the database by this kind of functions in scripts. In the examples of PHP and MYSQL, the function mysql_db_query() at line 78 and line 80 are typical validated points. Each validated point has only one validated-tree as the counterpoint, and the parameter of the function is the root of the validated-tree.

3.2 Code sections of scripts

Dynamic section: the condition branches and loop blocks whose execution will be determined at run time and are related to some undetermined conditions are defined as the dynamic section. In addition, the functions defined by the user also belong to the dynamic section. Codes in this kind of sections cannot be used for variables tracking.

Static section: the codes except the dynamic section in scripts are defined as the static section.

3.3 Trusty evidence

Evidence: expressions which fit one of the following conditions are defined as evidence.

1) The parameters of validated point are evidence

2) From the current position of the expression to the start of the program, if the left part of the expression appears in the evidence set first, the right part of it is evidence.

Trusty evidence: if the evidence is in the static section and the element is in the set of variables of constant or string operation, it is defined as trusty evidence. It can be used to update the evidence set and replace the leaves of the validated-tree.

Uncertain evidence: if the evidence is in the dynamic section or the function appears, the evidence is uncertain evidence.

3.4 Validated-tree

The root of the validated tree is the parameters in the validated point. Current evidence can be used in replacing the leaf node on the validated tree, but the node derived from the uncertain evidence cannot be replaced again. To find out whether the validated point has injection vulnerability, we trace back along the tree to search for the evidence line by line, and fully complete the tree. When the tree stops to grow, the procedure ends.

3.4.1 Vulnerable point

For the code section, the point that needs to be validated in line 78 can first take the expression in line 77 as trustable evidence and broaden the root point $sql, then it finds the second evidence in line 46 to broaden the extended leaf node $table_name. The last evidence appears in line 43. The growing procedure of the validated-tree for this point is shown in Fig. 1.

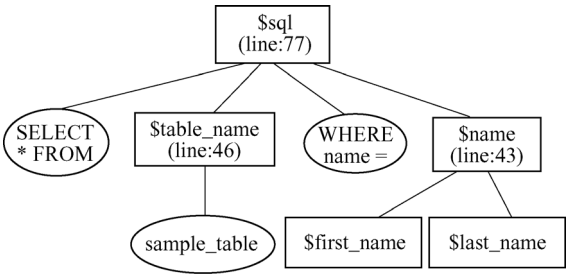

Fig. 1 Validated-tree generated by a vulnerable point

When we trace back the starting point of the program, it means the end of growth. After visiting all the leaves of the validated-tree from left to right, all the information about the database operation in the source code will be recovered. For example:

SELECT * FROM sample_table WHERE name= $first_name.$last_name

Here, two variables $first_name and $last_name are used to generate the SQL string without any constraint. It will allow attackers to inject arbitrary database operations with a skillful string. We will not elaborate on the injection attack technique in this paper. However, here we must notice that the essence to succeed in the injection attack is to inject a string into unconstraint variables at the client site.

Conclusion: when the validation is finished, if there are leaves in the tree which have variables deriving from trustable evidences, we can conclude that this point does have injection vulnerability. The validated point is called the vulnerable point.

3.4.2 Static point

For the point in the code section, line 80, the generation of its validated-tree is shown in Fig. 2.

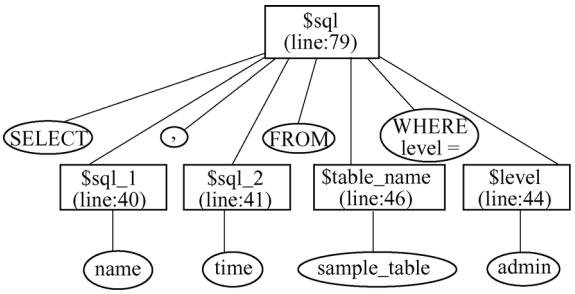

Fig. 2 Validated-tree generated by static point

After traversing all the validated-tree's leaves, we can obtain this string:

SELECT name , time FROM sample_table WHERE level='admin'

At this time, the SQL string is isolated from the outside input, thus the database operation from the point in line 80 cannot be injected.

Conclusion: when the validation is finished, if there are no variables in the leaves of the validated-tree, the database operation cannot be injected. This validated point is called the static point and cannot be injected.

3.4.3 Potential injection point

After visiting the leaves of the validated-tree, if there is an element deriving from the uncertain evidence, we are not able to tell whether there is injection vulnerability. According to the generation of the SQL string in the script, the main cause of the uncertain evidence is that the programmer has already constrained the contents of the user's input. In this case, the database operation may have been secured by the programmer when designing.

SQL strings are the case in point, which are generated from the validated point in line 82 in the code section and extend through the uncertain evidence in line 45 as follows:

SELECT * FROM sample_table WHERE id=filter($id)

Function filter() is a function to filter the client inputs. It is written by the programmer. However, its capacity in preventing all kinds of harmful strings is unknown to the validated-tree. Therefore, it is still a potential injection point.

3.5 Description of the variable tracking algorithm

The algorithm of variables tracking is given as follows:

Step 1 Scan the script file, create the set of validated point: $V=\{v_1, v_2, \ldots, v_m\}$. At the same time initialize $i=1$.

Step 2 If $i==m$, go to Step 11.

Step 3 Select v_i from V, take its parameter as the root point of the validated-tree, and set current validated position p as the line number of it.

Step 4 Scan the root from the left to the right, set the initial value of the leaf points. Set L with constant or variables. Set the initial value of extendable leaf points set L_E with variables.

Step 5 If $p>0$, carry out $p=p-1$; otherwise go to Step 9.

Step 6 Judge if the current line is evidence:

If it is not evidence, go to Step 5;

If it is trusty evidence, go to Step 7;

If it is uncertain evidence, go to Step 8.

Step 7 Scan the right of the trusty evidence, use constants and variables to replace the relevant elements in the set L, use variables to replace the relevant elements in the set L_E. Find out whether L_E is empty; if it is empty go to Step 9; otherwise go to Step 5.

Step 8 Replace the relevant elements in the set L with uncertain evidence, and mark the new ones as special elements created from uncertain evidence; delete relevant elements in L_E. Find out whether L_E is empty, if it is empty go to Step 9; otherwise go to Step 5.

Step 9 The validation towards v_i is finished, output the elements in L in order.

If L is labeled as special elements created from uncertain evidence, v_i has potential injection vulnerability;

If L does not include variables or have special elements, then v_i does not have injection vulnerability and it is a static state point.

If L includes variables and has no special elements, then v_i does have injection vulnerability and it is a vulnerable point.

Step 10 $i=i+1$; go to Step 2.

Step 11 End.

3.6 Security rank of codes

For the validated points for which we could not give a certain answer, we can find out the location and relationship among the database related variables, and we can also use the information that the validated-tree provides to give an security evaluation of the whole Web site.

The depth of the validated-tree indicates the logic complexity of the SQL string generation. The deeper the validated tree is, the more steps are needed to generate the SQL string. The width of the validated tree indicates the number of variables that are related to the database operation in scripts. It also indicates the complexity of the database operation itself. We can use this information to give a security rank of codes with a simple mathematical model as follows:

Symbol definitions: V: the number of vulnerable points; P: the number of potential injection points; p_w: the width of the validated-tree deriving from the potential injection point; p_h: the depth of the validated-tree deriving from the potential injection point; p_{pn}: the number of the node in the leaves generated by the uncertain evidence.

We can use an expression to calculate a security degree (SD) like this:

$$\begin{cases} \mathrm{SD}=V, & V>0, \\ \mathrm{SD}=\dfrac{\sum_{i=1}^{P} p_{w_i}}{\sum_{i=1}^{P} p_{h_i}+\sum_{i=1}^{P} p_{pn_i}}, & V=0. \end{cases}$$

If $V>0$, SD is the number of the vulnerability points. The scripts of the Websites can be injected in this case. If $V=0$, SD indicates the security risk of the scripts.

The value of $\sum_{i=1}^{P} p_{w_i}$ indicates the complexity of the DB operation. The more complex the DB operation is, the more variables have been referenced and the worse security the script has.

The value of $\sum_{i=1}^{P} p_{h_i}$ indicates the complexity of the SQL string generation logic. The deeper the validated tree is, the more SQL string generating steps are needed and the more difficult it is for the attacker to inject into the database. The value of $\sum_{i=1}^{P} p_{pn_i}$ indicates the potentials of the validated-tree deriving from the uncertain evidence. They hide a part of the actual depth of the tree.

Therefore, we can draw a conclusion as follows:

The bigger the value of the SD is, the worse the security of the script has. If $\mathrm{SD}>1$, the Web site has a high security risk of SQL injection. If $\mathrm{SD}<1$, the Web site is a simple application with DB operations. Thus it has fewer security problems in SQL injection.

4 Code vulnerabilities of SQL injection detector

CVSID is the software based on the validated-tree. As a tool to detect code security vulnerabilities, it includes four modules as follows:

Morphology analysis module: it is used to handle the included relationship among the source files, unfold source files, delete comments and abstract the elements in scripts such as variables, constants, keyword, etc.

Grammar analysis module: it uses the result generated from the morphology analysis module to parse expressions, functions, and identify static code sections or dynamic code sections. It can abstract all necessary information from the source file for the validated-tree.

Validated-tree growth control module: this module uses the result provided by the grammar analysis module to track the variables and execute all of the logic for the growth of the validated-tree.

Report generation modules: it is used to report the final result. The report includes the location of the vulnerabilities in scripts and the details of the related variables, etc.

We use open source Web projections in the test of CVSID. There is a famous article submission system which can be downloaded (http://www.aspsky.net) as an open source project and used by Web sites. The project includes 22 PHP files with 4088 lines. CVSID abstract 830 lines as evidences used for variables tracking, and locate 6 vulnerabilities in the project. For example, line 16 of the file show.PHP which is used to show articles has SQL injection vulnerabilities. The code section is as follows:

15: $sql = "SELECT * FROM $my_article_table WHERE id=$id";

16: $result = mysql_db_query($dbname, $sql);

Obviously, the variable $id can be used in SQL injection if the attacker sets the value of $id as

xx and ascii(mid(load_file('/etc/inetd.conf'), 1, 1)) =47#

With the logic relationship between the response of the server and the SQL request, the attacker can write a simple program to change the parameters of the mid() function used in testing the Web server to get any file contents on the server. It is equal to getting the reading authority of the server.

5 Conclusions

In this paper, we present a feasible algorithm to implement static detection of SQL injection. The reverse tracking of variables with the validated-tree does not take into account all of the variables and logic in the program, while it only focuses on the variables referenced by the database operation. As a result, it greatly simplifies the logic of the algorithm and tremendously raises the efficiency of the analysis. In addition, the procedure of variables tracking follows the thinking pattern of programmers: First, find the most sensitive functions which may lead to vulnerabilities; then track all related variables back. As for the uncertain evidence, this method is effective in finding out where the variables are related to the injection points and provide detailed relationships among variables to help the programmers to fix them manually. Herein, we recommend using automatic tools such as CVSID to locate logic vulnerabilities in scripts, and implementing filter functions with regular expressions to constrain the variables which would access the database directly or indirectly.

Significant results of SQL injection vulnerabilities mining at the code level have been achieved through the approach of reverse tracking of the variables with the validated-tree. If the problems of tracking variables in conditional branches or loop sections can be solved completely, the reverse tracking method can be implemented not only in SQL injection vulnerabilities detection, but can also be used in the buffer overflow and other vulnerabilities detection system, and a formalized mathematical method to determine system security status will finally become available.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 60574087), the Hi-Tech Research and Development Program of China (Nos. 2007AA01Z475, 2007AA01Z480, 2007AA01Z464) and the 111 International Collaboration Program of China.

References

1. Buehrer G, Weide B W, Sivilotti P A G. Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware. New York, NY: ACM, 2005, 106–113
2. Wassermann G, Su Z. An Analysis Framework for Security in Web Applications. In: Proceedings of the Workshop on Specification and Verification of Component-Based Systems, 2004
3. Fosdick L D, Osterweil L J. Data Flow analysis in software reliability. Computing Surveys, 1976, 8(3): 305–330
4. Gustafsson J, Lisper B, Sandberg C, et al. A tool for automatic flow analysis of C-programs for WCET calculation. In: Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems. IEEE Press, 2003, 106–112
5. Shankar U, Talwar K, Foster J S, et al. Detecting Format String Vulnerabilities with Type Qualifiers. In: Proceedings of the 10th USENIX Security Symposium, 2001
6. Walker D. A type system for expressive security policies. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, NY: ACM, 2000, 254–267
7. Huang Y W, Fang Y, Hang C, et al. Verifying web applications using bounded model checking. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks, 2004, 199–208
8. Pietraszek T, Berge C V. Defending against injection attacks through context-sensitive string evaluation. In: Proceedings of Recent Advances in Intrusion Detection (RAID), 2005, 124–145