Journal of Engineering and Applied Sciences 13 (20): 8426-8431, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

Security Testing of Web Applications for Detecting and Exploiting Second-Order SQL Injection Vulnerabilities

Najla'a Ateeq Mohammed Draib, Abu Bakar Md Sultan, Abdul Azim B. Abd Ghani and Hazura Zulzalil
Department of Software Engineering and Information System,
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia (UPM), 43400 Serdang, Selangor, Malaysia

Abatract: SQL injection is considered one of the most serious issues affecting web application's security. It occurs when an attacker tries to access the back-end database of web applications by exploiting improper user input validation vulnerabilities. There are two types of SQL injection, namely, first-order SQL injection and second-order SQL injection. Most of the existing research works addressing this issue focus on detecting the first-order SQL injection with a common assumption that preventing first-order injection attack makes web applications secure against other SQL injection attacks. However, second-order injection attacks can occur in applications that are secured against first-order injection attacks. This is a dangerous security problem which can occasionally, lead to dire consequences. In this study, we present our work-in-progress that uses a static taint analysis and symbolic execution approach for detecting second-order SQL injection vulnerabilities. We first use static taint analysis to identify candidate vulnerabilities. Then, we use symbolic execution to generate those input vectors that make the program execution satisfy conditions and confirm the existence of SQL injection vulnerabilities. This is the first technique of which we are aware that generates input vectors that expose second-order SQL injection vulnerabilities. The initial analysis of our proposed approach shows some promising results.

Key words: Security testing, static analysis, second-order SQL injection, vulnerability detection, web applications, promising results

INTRODUCTION

Web applications are becoming among the most ubiquitous software applications in use. This is because they can be both an efficient and reliable solution to communicating and conducting business challenges and a resource of information. Typically, web applications are designed with hard time restrictions. Therefore, the majority of them contain unexpected security vulnerabilities that have made them main targets of cyber-attacks. Thereupon, most web attacks take advantage of these vulnerabilities to get unauthorized access to the back-end database which often contains confidential or sensitive information such as user's financial and medical data or company confidential information.

Among web application vulnerabilities, Structured Query Language Injection Vulnerabilities (SQLIVs) have consistently been top-ranked for the past years as reported by Anonymous (2017), SANS institute (Calbraith 2012; Anonymous, 2016) as a special type of SQL Injections (SQLIs), second-order SQL injection attack tends to be more serious, more difficult to be detected and has a greater impact on the backend database than first-order SQL injection attack. This is due to the ability of second-order SQL injection to be seeded first into the application's persistent storage which is usually, deemed as a trusted source, prior to its actual exploitation.

There have been a lot of efforts devoted to detecting SQLIVs and preventing their exploitations using static analysis/white box testing, dynamic testing/black box testing or runtime monitoring. Static analysis approaches utilize taint analysis and similar code analysis techniques to detect SQLIVs by tracking the flow of intruder/tainted input values throughout the application itself but are unable to track input values across databases (Huang *et al.*, 2004; Jovanovic *et al.*, 2006; Su and Wassermann 2006; Xie and Aiken 2006). However, the attacker can store malignant code into the database and triggers its execution at a later time by taking advantage

Corresponding Author: Najla'a Ateeq Mohammed Draib, Department of Software Engineering and Information System, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (UPM), 43400 Serdang, Selangor, Malaysia

of improper sanitization of data retrieved from the database which results in second-order SQL injection attack. Moreover, static analysis is not accurate as some runtime information is required. Due to this inaccessible information, static analysis keeps a conservative approach which leads to producing false positives in large numbers. Recently, a static analysis tool is introduced to detect second-order attacks (Dahse and Holz, 2014). However, as a static analysis tool, this tool suffers from reporting many false positives. Moreover, misinformation may occur, since, static analysis techniques do not carry out actual attack instances to discover a vulnerability.

Black box testing techniques cannot guarantee precision and completeness as they do not explore all possible program paths of applications. In addition, as assessed by Bau *et al.* (2010, 2012), Deepa and Thilagam (2016) many existing vulnerability scanners are limited to detecting first-order SQLIVs and are not capable of detecting second-order SQLIVs. This tendency is due to two main reasons, namely, they cannot confirm that the injected code is already in the storage and they may have trouble in linking the initial injection event with the triggering of the stored injected code.

For the runtime monitoring approach, some runtime monitoring approaches address second-order SQLIVs (Halfond and Orso, 2005; Lam *et al*., 2008) but their success and the accuracy of detection is dependent on the accuracy of statically building query model.

The majority of existing solutions only address the first-order SQLIA. They consider that if first-order attack is detected and prevented then second-order attack cannot occur any more. First-order vulnerabilities are considered not exploitable when the user input is sanitized properly. However, the attack can be launched later on by exploiting the second-order vulnerabilities that make use of the input.

The lack of effective mechanisms for addressing the detection of second-order SQLIVs associated with an increasing trend to reprocess submitted data and optimize its use increases the probability and risks of this kind of attack.

In this study, we propose an approach for automatically identifying second-order SQLIVs by using taint analysis and symbolic execution. Taint analysis is devoted to identifying candidate vulnerabilities in the source code of the web application. When candidate vulnerabilities are detected, we resort to symbolic execution and constraint solver to find input vectors that can traverse the vulnerable baths and expose the existence of false positives.

Background: In this study, a relevant background on second-order SQL injection attack and static taint analysis is provided.

SQL injection attack: SQL Injection Vulnerabilities (SQLIVs) are a security flaws that enable an attacker to compromise underlying databases of web applications resulting in unwanted extraction or insertion of data from or into a database. SQL Injection Attack (SQLIA) is a hacking technique in which the attacker exploits SQLIVs to inject SQL code fragments into vulnerable input parameters (HTTP requests) generating malicious SQL query which enables the attacker to gain an unauthorized access to the back-end database.

Practically, SQL injection can be introduced into vulnerable web applications using two main mechanisms based on the injection order: first-order SQL injection and second-order SQL injection (Halfond *et al*., 2006; Sharma and Jain, 2014). In first-order attack, the attacker inserts SQL commands into a vulnerable input field that flows directly from an entry point (e.g., $_GET) to a sensitive sink (e.g., mySQLI-query). The successful injection results are delivered immediately upon user-input submission.

Second-order SQLIA is a special type of SQLIA which is more serious and more difficult to be detected. In such attacks, the attacker first seeds SQL commands into the database and uses that input at a later stage in a sensitive sink for launching the attack. Unlike first-order SQLIA in second-order SQLIA, malicious code is not initiated immediately but instead it is first stored in the application back-end database and then later on retrieved and activated by the victim/attacker. Table 1 summarizes the main differences between these two mechanisms.

Table 1: Implementation and detection of first-order and second-order SQLIAs

Injection	Injection mechanism	Detection
First-order SQLI	The attacker enters a malicious input and causes the modified code to be executed instantly or in real-time	Easily filtered out anti-malware application and other detection techniques as the detection process does not by involve tracking taint data through the database
Second-order SQLI	The attacker initially injects malicious code into backend database and subsequently different context and time, triggers the malicious content execution	Difficult to be detected and prevented because the injection in point is different from that where the attack was actually triggered/launch-ed

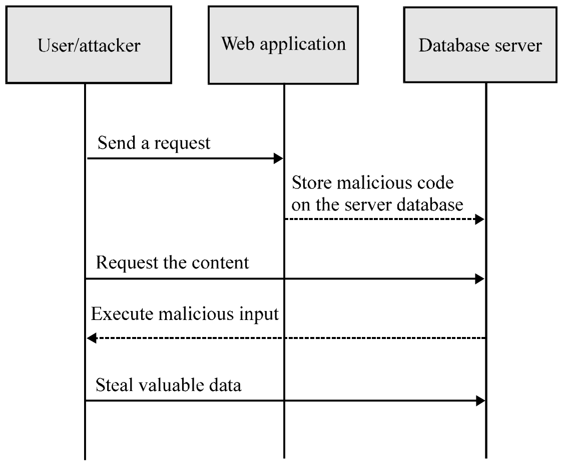

Fig. 1: A high level view of typical second-order SQL injection attack

Second-order SQL injection attack: Typically, second-order SQLIA stems from missing or improper sanitization of data flowing from web application's user to the database and then to sensitive sink (e.g., SQL query statement) (Anley, 2002; Mederios, 2016). The resulting security violations of such attacks can be disastrous it may include identity theft, loss of confidential or sensitive data, take control of data and destroy back-end database (Sharma and Jain, 2014). To illustrate, Fig. 1 presents a typical architecture of the second-order SQLIA.

In such an attack, the following scenario may take place (Anley, 2002). An attacker may register on a web site where an "Admin" account exists using a seeded username "Admin' #". Below is an example code for creating an account (Algrothim 1):

Algrothim 1; Code for creating an account:

```
$username =
mysqli_real_escape_string($db,$_POST['username'])
$password =
mysqli_real_escape_string($db,$_POST['password'])
$qry1 = mysqli_query($db,"INSERT INTO user
(username,password) VALUES
'$username','$password')")
```

The application correctly escapes the single quote in the input before storing it in the database as shown in the code. In this particular code by applying escaping, the payload will be stored into the database as "Admin' #" but will not cause string termination issues when building SQL insertion statement, thus, preventing its potentially malignant effects. Let's say that the application allows a user to modify password of his or her account. The modifying operation typically, involves ensuring that the user has the old password before changing to the new password. The code might be as follows (Algrothim 2):

Algrothim 2; Code:

```
$username =
mysqli_real_escape_string($db,$_POST['username'])
$oldpassword = mysqli_real_escape_string
($db,$_POST['oldpassword'])
$newpassword = mysqli_real_escape_string($db,$_
POST['newpassword'])
$qry2 = mysqli_query($db,"SELECT * FROM
user WHERE username = '$username' and
password = '$oldpassword'")
$array = mysqli_fetch_assoc($qry2)
$uname = $array['username']
$qry3 = mysqli_query($db,"UPDATE user SET
password = '$newpassword' WHERE username = '$uname' ")
```

"$uname" is the stored username "Admin'#" that was retrieved from the database for building up a new SQL statement. Given the username "Admin'#", the update statement executed by the database is as shown:

```
$qry3 = mysqli_query($db,"UPDATE user SET password =
'$newpassword' WHERE username =
'Admin' #' ")
```

Everything after the#character will be ignoredby the database so the actual query is

```
$qry3 = mysqli_query($db," UPDATE user SET password =
'$newpassword' WHERE username = 'Admin')
```

As a result, the password of "Admin" will be updated rather than the username "Admin'#" and thus, the attack is achieved successfully. This is a dangerous problem which can occasionally, lead to dire consequences. Second-order SQLIA is much more difficult to be detected and prevented. This is due to the different points of injection and attack launching. The developer may successfully 'escape' user input and deem it safe but later on when the data is reused to create different types of queries, the previously sanitized input may result in a second-order SQL injection attack.

Static taint analysis: Static analysis, basically, finds the root cause of a security problems in source codes. They statically check program texts to discover loopholes in the early stage of development, even before running the program. It is important to find errors early in the development as it not only minimizes the cost of fixing the errors but also, the coding approach of the developer is improved due to the quick feedback received. Another advantage of using static analysis is its ability to provide better coverage compared to dynamic analysis. Static taint analysis is a special type of data flow analysis and it is the most commonly data flow analysis technique for security analysis due to its ability to flag the tainted data entered in the program and detect if it reaches a sensitive sink. In second-order SQLI vulnerabilities context, a vulnerability is discovered whenever a possibly tainted variable is stored into a persistent storage of a web application and then at another stage this input is reused in a sensitive (sink) statement without being validated. A sink point/statement is the SQL statement that uses an input that is initially, supplied by the user. Although, static analysis is effective in finding vulnerabilities in source code, it tends to generate many false positives due to its undecidability (Landi, 1992).

MATERIALS AND METHODS

We now present the proposed approach of automatic detection and exploitation of second-order SQLIV. Our main goals are: to automatically identify vulnerable points to second-order SQLI in the source code and to generate input vectors that can reveal the real vulnerabilities and help developers to understand under which conditions these vulnerabilities can be exploited. This approach consists of two stages consecutive taint analysis for detecting the vulnerability, followed by a combination of symbolic execution and constraint solving for input vectors generation. Figure 2 shows the architecture of the proposed approach.

Vulnerability detection: The static analyzer first parses the source codes and generates Their Abstract Syntax Trees (AST). Then it does taint analysis based on the generated ASTs. Lastly, it generates trees that describe candidate vulnerable control-flow paths. Second-order SQLIA consists of two phases, one that seeds the malicious code into the database and another that triggers its execution to cause second-order attack. Therefore, detection method consists two phases, first phase to detect the inserting process and second phase to find out the triggering process.

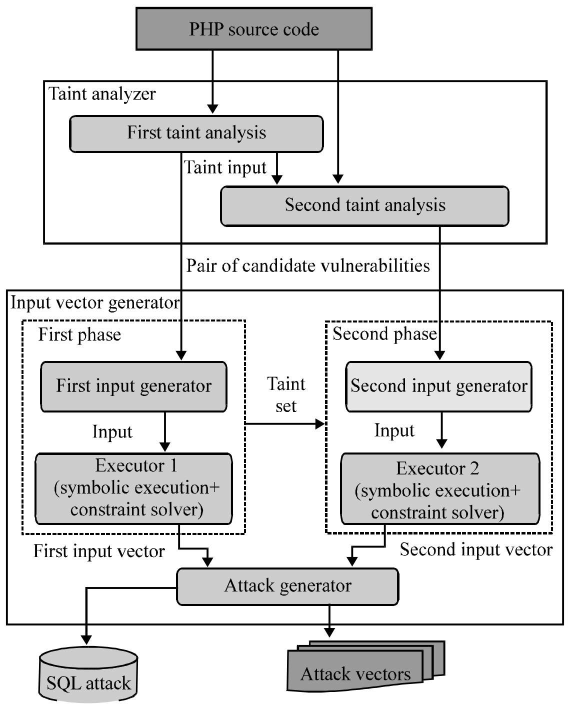

Fig. 2: Architecture including main modules

First static taint analysis phase: The goal of doing static analysis first is to identify vulnerable paths where taint data flows from source to insertion statement without appropriate validation. In this context, source is corresponding to input data supplied by the user. Insertion statements correspond to SQL database insertion statements (INSERT, UPDATE or REPLACE) that store that input into a field in the database table without appropriate sanitization. The information about the targeted table's name, column names and corresponding input values is determined by tokenizing the SQL query. In the case when the target column name is not specified, the parser uses database schema.

Second static taint analysis phase: The second static analysis of our approach is to take the information collected from first static analysis as input and checks for the existence of sensitive sink (mysql_query) that uses the same column name to generate SQL query without proper sanitization. Here, the source is the data retrieved from the column of database table and sensitive sink is the statement (e.g., SQL query) that does retrieves the values from database for building up a new SQL statement. Our analysis then issues a warning. For this phase of analysis, taint sources are elements (e.g., table name, column name, etc.) inferred by the first analysis. On the other hand, sensitive sinks are statements that retrieve this column values from the database.

Generating input vectors: The main aim of first stage of our method is to identify vulnerable paths to second-order SQLIA. However, taint analysis has two mature limitations: It does not consider infeasible paths which leads to produce false positives (not exploitable paths). In addition, taint analysis just reports information about the tainted data that gave rise to the vulnerability without reporting executable test cases or providing information about under which conditions the detected vulnerability would be exploited. Hence, in the second phase input vector generation engine will generate input vectors to confirm the existence of false positives and help programmers to understand vulnerabilities reported by the tool. The attack vector for exploiting second-order SQLI vulnerabilities consists of two inputs. The first input represents data supplied by the attacker which contains malicious code, associated with other user inputs needed to traverse the target path. The second input represents data provided by the victim along with other inputs needed to traverse the second target path.

First input vector generation: The algorithm takes the first target path as an input. This path represents the process of insertion to the database. Algrothim 3 presents the algorithm for generating first attack vector. In the first step (line 3) the code is instrumented in a way where we can get the execution path for any input. Then the algorithm generates new concrete inputs until a time limit expires (line 5) in each iteration the algorithm picks an input set (line 7). Then the algorithm executes the input set using symbolic execution and constraint solver. This is to traverse the target path and check which input set is able to reach the insertion statement. Whenever an input set reaches insertion statement (e.g., insertion query) the input set will be saved and the tainted value (which would be used to build up the insertion query) of the input set is mutated using a dataset of SQLI attack patterns in an attempt to create attack vectors by modifying the inputs.

Algrothim 3; Algorithm for creating first input attack vectors:

```
Parameters : P1
Result     : First input attack vectors
1. Input 1 = Ø
2. InputSet1´ = Ø
3. P1' = makeInstromentedCopy (P1)
4. while (! timeLimitExpired()) do
5. Inputs1 = inputs1 U generateNewInput(P1)
6. while (! covered (P1') AND ! inputListEnd()) do
7.        { inputSet1 = selectInput (input1)
8.        P = sembolicExecutoion & constraintSolver(p1' inputSet1)
9.        if (covered (P1´))
10.                 InputSet1 ' = mutate(inputSet1);}
11         Return (P, inputSet1 ')
```

Second input vector generation: The second phase works similarly as the first one except that it is simpler, since, the input vector can be generated without mutating. Mutation is not necessary because the attack can be triggered with non-malicious input string. Hence, we can have a satisfying assignment with backwards symbolic execution of the generated constraint. Algrothim 4 describes the algorithm for generating second input vectors.

Algrothim 4; Algorithm for creating first input attack vectors:

```
Parameters : P2
Result     : Second input vectors
1. Input 2 = Ø
2. InputSet2' = Ø
3. P2' = makeInstromentedCopy(P2)
4. while (! timeLimitExpired()) do
5. Inputs2 = inputs2 U generateNewInput(P2)
6. while (! covered(P2´) AND ! inputListEnd()) do
7.     { inputSet2 = selectInput (input2)
8.     P = sembolicExecutoion & constraintSolver(P2' inputSet2)
9.      if (covered (P2´))
10.     inputSet2 ' = mutate(inputSet2);}
11.  Return (P, inputSet2 ')
```

RESULTS AND DISCUSSION

This research is ongoing work. The initial analysis of the proposed approached shows some promising results. This research is expected to produce a new approach for detecting second-order SQL injection vulnerabilities in PHP-based web applications with less false positives. Furthermore, we expect this approach to benefit web application developers by enabling them to easily test their application's source codes and precisely detect vulnerabilities before deployment which in turn will benefit the users of these applications by protecting them from potential attacks. Furthermore, generating attack vectors helps the developers to understand how the detected vulnerabilities being exploited.

CONCLUSION

In this study, we presented our research plan towards detecting second-order SQLIVs using a combination of static taint analysis, symbolic execution and constraints solving. The candidate vulnerable paths are detected using taint analysis and then input vectors are generated to expose the false positives and to help the developer to understand under which constraint the vulnerabilities might be exploited.

SUGGESTIONS

The future work on this progressive work is to accomplish the research plan, develop a prototype to automate the process and conduct an exhaustive experiment to evaluate the proposed approach.

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 Higher Education to the Faculty of Computer Science and Information Technology at Universiti Putra Malaysia.

REFERENCES

Anley, C., 2002. Advanced SQL injection in SQL server applications. NGS Secure, Manchester, England, UK. http://www.cgisecurity.com/lib/advanced_sql_injection.pdf

Anonymous, 2016. Trustwave global security report. Trustwave Holdings, Chicago, Illinois, USA. https://www.trustwave.com/Resources/Global-Security-Report-Archive/

Anonymous, 2017. OWASP top 10-2017: The ten most critical web application security risks. OWASP, Maryland, USA. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Bau, J., E. Bursztein, D. Gupta and J. Mitchell, 2010. State of the art: Automated black-box web application vulnerability testing. Proceedings of the 2010 IEEE International Symposium on Security and Privacy, May 16-19, 2010, IEEE, Berkeley, California, USA., ISBN:978-0-7695-4035-1, pp: 332-345.

Bau, J., F. Wang, E. Bursztein, P. Mutchler and J.C. Mitchell, 2012. Vulnerability factors in new web applications: Audit tools, developer selection and languages. J. Web Des., 1: 1-15.

Calbraith, B., 2012. CWE/SANS top 25 most dangerous software errors: What errors are included in the top 25 software errors?. SANS Institute, USA.

Dahse, J. and T. Holz, 2014. Static detection of second-order vulnerabilities in web applications. Proceedings of the 2014 International Symposium on USENIX Security, August 20-22, 2014, USENIX, San Diego, California, USA., ISBN:978-1-931971-15-7, pp: 989-1003.

Deepa, G. and P.S. Thilagam, 2016. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. Inform. Software Technol., 74: 160-180.

Halfond, W.G. and A. Orso, 2005. AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks. Proceedings of the 20th IEEE-ACM International Conference on Automated Software Engineering, November 07-11, 2005, ACM, Long Beach, California, ISBN:1-58113-993-4, pp: 174-183.

Halfond, W.G.J., J. Viegas and A. Orso, 2006. A classification of SQL injection attacks and countermeasures. Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering Vol. 1, March 13, 2006, IEEE, New York, USA., pp: 13-15.

Huang, Y.W., F. Yu, C. Hang, C.H. Tsai and D.T. Lee *et al*., 2004. Securing web application code by static analysis and runtime protection. Proceedings of the 13th International Conference on World Wide Web, ACM, New York, USA., May 17-20, 2004, ISBN: 1-58113-844-X, pp: 40 52-10.1145/988672.988679.

Jovanovic, N., C. Kruegel and E. Kirda, 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. Proceedings of the Symposium on Security and Privacy, May 21-24, 2006, Berkeley/Oakland, CA., USA., pp: 263-269.

Lam, M.S., M. Martin, B. Livshits and J. Whaley, 2008. Securing web applications with static and dynamic information flow tracking. Proceedings of the 2008 ACM SIGPLAN International Symposium on Partial Evaluation and Semantics-Based Program Manipulation, January 7-8, 2008, ACM, San Francisco, California, USA., ISBN:978-1-59593-977-7, pp: 3-12.

Landi, W., 1992. Undecidability of static analysis. ACM. Lett. Program. Lang. Syst., 1: 323-337.

Medeiros, I.V.D.S., 2016. Detection of vulnerabilities and automatic protection for web applications. Ph.D Thesis, University of Lisbon, Lisbon, Portugal.

Sharma, C. and S.C. Jain, 2014. Analysis and classification of SQL injection vulnerabilities and attacks on web applications. Proceedings of the 2014 International Conference on Advances in Engineering and Technology Research (ICAETR=14), August 1-2, 2014, IEEE, Unnao, India, ISBN:978-1-4799-6393-5, pp: 1-6.

Su, Z. and G. Wassermann, 2006. The essence of command injection attacks in web applications. ACM. SIGPLAN Not., 41: 372-382.

Xie, Y. and A. Aiken, 2006. Static detection of security vulnerabilities in scripting languages. Proceedings of the 15th International Symposium on USENIX Security, July 31-August 4, 2006, USENIX, San Diego, California, USA., pp: 179-192.