International Journal of Engineering & Technology, 7 (4.1) (2018) 14-17

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Comparison of Security Testing Approaches for Detection of SQL Injection Vulnerabilities

Najla'a Ateeq Mohammed Draib*, Abu Bakar Md Sultan, Abdul Azim B Abd Ghani, Hazura Zulzalil

Dept. of Software Engineering and Information System, Faculty of Computer Science and Information Technology
Universiti Putra Malaysia, 43400 UPM Serdang, Selangor, Malaysia
* *Corresponding author E-mail:n.draib@gmail.com*

Abstract

Structured query language injection vulnerability (SQLIV) is one of the most prevalent and serious web application vulnerabilities that can be exploited by SQL injection attack (SQLIA) to gain unauthorized access to restricted data, bypass authentication mechanism, and execute unauthorized data manipulation language. Hence, testing web applications for detecting such vulnerabilities is very imperative. Recently, several security testing approaches have been proposed to detect SQL injection vulnerabilities. However, there is no up-to-date comparative study of these approaches that could be used to help security practitioners and researchers in selecting an appropriate approach for their needs.
In this paper, six criteria's are identified to compare and analyze security testing approaches; vulnerability covered, testing approach, tool automation, false positive mitigation, vulnerability fixing, and test case/data generation. Using these criteria, a comparison was carried out to contrast the most prominent security testing approaches available in the literature. These criteria will aid both practitioners and researchers to select appropriate approaches according to their needs. Additionally, it will provide researchers with guidance that could help them make a preliminary decision prior to their proposal of new security testing approaches.

Keywords: *SQL injection; vulnerabilities; Detection approaches, Software security test; Web applications.*

1. Introduction

Owing to their convenience and being easily accessible, web applications have become very popular and widely accepted in various fields of human endeavor. These applications often process and store sensitive data for many users which, besides their popularity, have made them attractive and ideal targets for malicious users. Typically, web applications are designed with hard time restrictions, and therefore, are often deployed with varied degrees of unexpected security vulnerabilities that are exploitable by hackers through different types of attacks. These hacking attempts ordinarily result in unauthorized and, often, harmful transactions with the application, as well as its' underlying database[1, 2].
Among web application attacks, SQL injection attacks (SQLIAs) have consistently been top-ranked for the past few years, as reported by OWASP [3], SANS institute [4], and Trustwave [5]. SQL injection vulnerability usually takes place when a web application does not properly sanitize the user input. In this respect, it is crucial that the security auditors ensure that the implemented code is, to a possible extent, free from these kinds of vulnerabilities before deployment. SQLIA basically takes advantage of the vulnerabilities found in the input validation and the improper handling of submitted requests in the server side program which interacts with the database server. In such attacks, the attacker usually injects 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 [6, 7]. The resulting security violations of these attacks can be disastrous; they could include identity theft, loss of confidential or sensitive data, taking control of data, and fraud.

A practical method that through which one can deal with SQL injection vulnerabilities (SQLIVs) is to apply appropriate application security testing techniques responsible for detecting and mitigating these vulnerabilities before the applications are deployed. There have been a lot of efforts devoted to detecting SQLIVs and preventing their exploitations using different types of security tests on web applications: static security analysis, dynamic security test, and hybrid security test.
To our knowledge, the literature does not have that sort of criteria that can help practitioners and researchers select an appropriate SQLIVs detection tool/approach. Additionally, there is no comparative analysis in current literature with respect to vulnerability coverage in terms of injection input mechanism, false positive mitigation and test cases/data generation. Therefore, web application practitioners and researchers may face some challenges in selecting the appropriate approach for detecting SQLIVs. Consequently, many SQLIVs may go undetected in the final product, and thus creating a potential for SQLAs.
The main purpose of this paper is to provide a comparison framework to help practitioners select the right tool/approach for SQLIV detection and prevention, to keep researchers up to date with the techniques and tools of detecting and preventing SQLIVs, and to identify possible directions for future research. For this purpose, 13 prominent tools and approaches are analyzed based on 6 criteria. These criteria is collected from different sources including [8–10].

This paper is structured as follows: Section 1 introduces the paper. Section 2 presents background information relevant to SQL injection attacks and web application security testing. Section 3 discusses the six criteria used for comparing and analyzing the con-

Copyright © 2018 Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

sidered approaches. In section 4, we compare the approaches, and then in section 5, we close with some directions and implications for future research.

2. Background

In this section, a relevant background on web application security testing and SQL injection attack is provided.

2.1. Web Application Security Testing

Web applications are popular and publicly exposed to attacks that can target their vulnerabilities and compromise their security. These vulnerabilities have been researched and analyzed at OWASP [3]. Among these vulnerabilities SQL injection vulnerability stands on top.

Generally web security testing is a common procedure that is used to determine whether or not an information system is protecting data and maintaining functionality as intended. Web applications security testing is a common technique for validating web applications. The main objective of security testing is to verify that the overall web application defense against undesired access of unauthorized users is effective. Taking this objective into account, we highlight three types of security tests to deal with code injection problems in web applications:

Static security analysis: involves the inspection of either source or binary code to find software bugs that could lead to a code injection attack without actually executing the program.

Dynamic security testing: observes the behavior of a running system in order to detect and prevent a code injection attack.

Hybrid security test: a combination of static analysis and dynamic testing approaches. In this approach the information collected in static analysis phase are used in the dynamic testing phase to guide the detection process.

2.2. SQL Injection Attack

SQLIV is a security flaw that enables an attacker to compromise underlying databases of web applications resulting in unwanted extraction or insertion of data from or into a database. 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 unauthorized access to the back-end database.

Practically, SQL injection can be introduced into vulnerable web applications using two main input mechanisms: first-order SQL injection, and second-order SQL injection [11,12] .

In first-order injection, 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 results of a successful injection are immediately delivered upon user-input submission. The second-order SQL injection, also known as persistent or stored SQL injection, is a special type of SQL injection (SQLI), 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. Table 1 summarizes the main differences between these two mechanisms.

Table 1: Comparison between first-order and second-order SQLIs adapted from [13].

Injection Type	First-order SQLI	Second-order SQLI
Injection Mechanism	The attacker enters a malicious input and causes the modified code to be executed instantly or in real-time	The attacker initially plants attack code into back-end database and, subsequently in a different context and time, triggers the malicious content execution.
Detection	Easily filtered out by anti-malware application and other detection techniques as the detection process does not involve tracking taint data through the database	Difficult to be detected and prevented because the injection point is different from that where the attack was actually triggered/launched.
Effect on the database	Medium	Very high
SQL generation	Application level	Database level

Unlike first-order SQLI, the malicious code in second-order SQLI 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. Figure 1 presents a typical architecture of the second-order SQLIA. Typically, second-order SQLI vulnerability stems from missing or improper sanitization of data flowing from web applications' user to the database and then to sensitive sink (e.g. SQL query statement)[14,15].

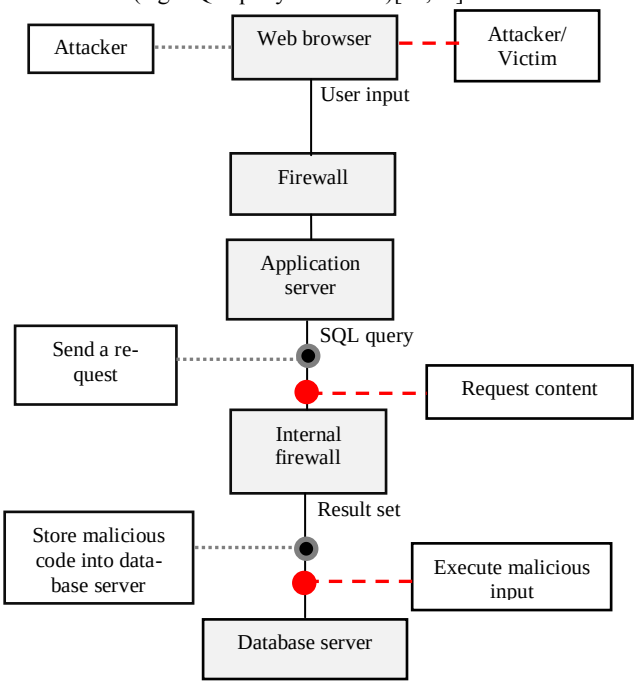

Fig. 1: Second-order SQL injection mechanism in Web application

3. Comparison Framework

There have been studies that presented a set of criteria to compare security testing approaches [8-10]. However, the selected approaches that are considered for these studies are from different type of security risks Injection, XSS, buffer overflow, remote file inclusion, etc. Furthermore, these studies only apply the coverage criteria for first-order SQLIA and ignore persistence injection attacks. Our proposed comparison framework focuses on security testing approaches which address SQLIVs in web applications. We adopt six criteria to compare security testing approaches for SQLIV detection that have not been addressed in the literature yet.

Vulnerability Covered: A common and important criterion for choosing an approach is based on what particular vulnerability can be tested. As we focus on SQLIVs, we test the tools based on which input injection mechanism that it covers either first-order SQLI, second-order SQLI, or both.

Testing Approach: This criterion indicates whither security testing of an application is performed using static analysis, dynamic testing or hybrid testing approach.

Tool Automation: One of the important steps to choosing a testing approach is to determine how much automation this approach supports (fully automated, semi- automated, or manual).
False Positive Mitigation: Vulnerability detection approaches often suffer from generating false positives. This criterion identifies whether or not the approach uses any methods to reduce false positive alarms.
Vulnerability Fixing: This criterion determines whether the tool provided automatically fixes the detected vulnerabilities or not.
Test case generation: This criterion identifies whether test cases/data are generated to test the vulnerable code or not.

4. Approaches Comparison

This section compares the SQLIVs detection approaches based on the aforementioned criteria. The selected approaches are analyzed and then a brief description about each one is given in Table 2. The list of the selected approaches for the present study is found very exhaustive, so we concentrated on approaches that we do consider more significant and whose literature is more recent to SQLIV detection and prevention.

Table 2: Approaches Comparison Summary

Approach	Swt	VC	TL	TA	FPM	VF	TC/TD Gen.	Detection Technique
[16]	ARDILLA	FO SQLI	DT	FA	Yes	NO	Yes	Input Generation, Dynamic Taint Propagation, And Input Mutation.
[17]	Pixy	FO SQLI	SA	FA	No	No	No	Flow-Sensitive, Interprocedural, And Context-Sensitive Data-Flow Analysis
[18]	WAP	FO SQLI	SA	FA	Yes	Yes	No	Taint Flow Analysis Plus Data Mining Classifier Prediction
[19]	QED	FO SQLI	HA	S-A	Yes	No	Yes	Combination Of Taint Analysis, Model-Checking, Dynamic Taint Tracking, And Runtime Detection
[20]	SAFELI	FO SQLI	SA	S-A	Yes	No	Yes	Symbolic Execution And Hybrid Constraint Solver
[21]	Webssari	FO SQLI	HA	S-A	No	No	No	Information Flow Analysis And Runtime Inspection
[22]	No Name	FO + SO SQLI	SA	FA	Yes	No	No	Context-Free Grammars And Taint Tracking
[23]	RIPS	FO + SO SQLI	SA	FA	No	No	No	Static Analysis Based On Block And Function Summaries
[24]	Stranger	FO SQLI	SA	FA	Yes	No	No	Taint Analysis And Automata-Based String Analysis
[25]	MUSIC	FO SQLI	HA	S-A	No	No	Yes	Mutation-Based Testing
[26]	No Name	FO SQLI	SA	S-A	No	No	No	Interpocedural Analysis Based On Code Property Graphs
[27]	Joanaudit	FO SQLI	SA	FA	No	No	No	Static Analysis To Extract Minimal Slice Relevant To Security
[28]	SQLIVDT	FO + SO SQLI	DT	S-A	No	No	Yes	Server Response Analyzing

Swt: Software Tool
VC: Vulnerability Covered
FO: First-Order, SO: Second-Order
TL: Testing Level
SA: Static Analysis
DT: Dynamic Testing
HA: Hybrid Approach
TA: Tool Automation
FA: Fully Automated, S-A: Semi-Automated
FPM: False Positive Mitigation
VF: Vulnerability Fixing
TC/TD Gen.: Test Case/Data Generation

Based on the above analysis of web application security testing approaches for the detection of SQLIVs, we summarized the primary observations of this comparative study in the following points:

- The majority of the analyzed approaches are not capable of handling second-order SQLIVs. They only focus on the detection of first-order SQLIVs since they only analyze SQL queries generated at application level and ignore those generated at database level. This may be due to two common assumptions: first, when first-order vulnerability is detected and prevented, second-order vulnerability is not exploitable any more. Second, when successfully 'escape', malicious input is deemed safe. A serious weakness with these assumptions, however, is that the attack can be launched later on in different time and context by exploiting the second-order vulnerabilities that make use of that data to create different SQL query.
- Most of the approaches do not generate executable test cases/data, but only report the data slice that gives rise to vulnerability. Nonetheless, providing test cases or test data sets helps the auditor to better understand how the detected vulnerabilities might be exploited. Moreover, test cases can be considered to have the aim of discovering false positives and unprevented vulnerabilities.
- Almost all analyzed approaches but WAP [15], only report the detected vulnerabilities and leave their removal as a burden on the programmer. WebSSARI [21] and Stranger [24] give suggestions of sanitization functions and filters to prevent malicious input, but they do not actually remove SQLIVs from the source code.
- Most of the approaches are static analysis based approaches. Although static analysis is effective in finding vulnerabilities in source code, it tends to generate many false positives. This mandates the need for complementary techniques to verify exploitations of program vulnerabilities and provide precise results.
- Most of the approaches do not mitigate false positives and consequently suffer from generating many false positive alarms.

Although none of the approaches provide a full solution that addresses SQLIV detection, some of these approaches such as WAP [18] and SAFILI [20], seem promising and could be extended to cover more injection mechanisms and to introduce more precise results.

SAFELI is a static analysis tool for identifying SQLIVs in the byte code of ASP.NET web applications at compile time. SAFELI utilizes a symbolic execution engine to analyse the source code. First, the source code is instrumented for symbolic execution. At each SQL query submission, a hybrid string equation is constructed to figure out the corresponding initial values that could lead to a SQLIA. The equation is then solved by a Hybrid-string Solver where the solution provided is used to generate test cases exploiting SQLIVs. If a defect is encountered, an attack is

replayed by the tool to developers. However, SAFELI has few drawbacks:

- It can detect SQLIVs only on Microsoft based products.
- This approach also has a problem of detecting second-order SQLIVs.
- It creates load on the web server.
- SAFELI only discovers the vulnerabilities and puts their removal burden on the developers.

The WAP [8] approach is based on combined taint analysis with data mining to reduce the rate of false positives. It uses the taint flow analysis to track the spread of untrusted input through an application generating tree describing candidate vulnerable control-flow paths (from an entry point to a sensitive sink). It then uses data mining to predict the existence of false positives within results of taint analysis by providing a machine learning algorithm with samples of code which humans classified as vulnerable or not vulnerable, thus allowing the system to effectively minimize the number of false positives based on certain ”symptoms” in the code. However, this approach has the following limitations:

- This approach uses basic methods of taint analysis (actual taint analysis) but does not add any improvement to the actual taint analysis process.
- The taint analysis technique in this approach does not track tainted data through the database which makes it incapable of detecting second-order SQLIVs.

5. Conclusion

This paper attempted to provide a framework that compares and analyses proposed approaches to SQL injection vulnerability detection. For carrying out this work, we first identified six criteria, which are vulnerability covered, testing approach, tool automation, false positive mitigation, vulnerability fixing, and test case/data generation. We then analysed the selected approaches via these criteria. We concluded that the results of the study can help both practitioners and researchers with an overview of prominent work in the literature and assist them with regards to making decisions as to which approach would be appropriate for their particular efforts. Additionally, it may provide some sort of a guideline for researchers interested in developing new approaches. Some open issues for future work have also been identified. In future work, the authors will analyse the possibility of using static analysis and concolic execution for identifying SQLIVs in web application code and generating input vectors that can be used to confirm their existence.

Acknowledgement

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

[1] G. Deepa and P. S. Thilagam, “Securing web applications from injection and logic vulnerabilities: Approaches and challenges,” Inf. Softw. Technol., vol. 74, pp. 160–180, (2016).
[2] Y.-F. Li, P. K. Das, and D. L. Dowe, “Two decades of Web application testing—A survey of recent advances,” Inf. Syst., vol. 43, pp. 20–54, (2014).
[3] OWASP, “OWASP Top 10 - The Ten Most Critical Web Application Security Risks,” Owasp, p. 22, (2017).
[4] B. Calbraith, “SANS Institute,” SANS Institute, (2012). [Online]. Available: https://www.sans.org/top25-software-errors/. [Accessed: 25-Feb-2017].
[5] Trustwave, “Trustwave Global Security Report,” (2018).
[6] W. G. J. Halfond, A. Orso, D. A. Kindy, and A. S. K. Pathan, “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.
[7] Y. Xie and A. Aiken, “Scalable error detection using boolean satisfiability,” Symp. Princ. Program. Lang., pp. 351–363, (2005).
[8] H. Shahriar and M. Zulkernine, “Automatic Testing of Program Security Vulnerabilities,” 2009 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf., no. July, pp. 550–555, (2009).
[9] F. T. Alssir and M. Ahmed, “Web security testing approaches: Comparison framework,” in Advances in Intelligent and Soft Computing, (2012), vol. 144 AISC, no. VOL. 1, pp. 163–169.
[10] S. M. Srinivasan and R. S. Sangwan, “Web App Security: A Comparison and Categorization of Testing Frameworks,” IEEE Softw., vol. 34, no. 1, pp. 99–102, (2017).
[11] W. G. J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures,” (2008).
[12] C. Sharma and S. C. Jain, “Analysis and classification of SQL injection vulnerabilities and attacks on web applications,” in 2014 International Conference on Advances in Engineering and Technology Research, ICAETR 2014, (2014).
[13] A. Mohammed, A. B. Sultan, A. Azim, B. A. Ghani, and H. Zulzalil, “Detecting and Exploiting Second-order SQL Injection Vulnerabilities of Web Applications,” in Sixth International Conference on Computer Science and Computational Mathematics, (2017), no. Iccscm, pp. 88–92.
[14] C. Anley, “Advanced SQL injection in SQL server applications,” White Pap. Next Gener. Secur. Softw., (2002).
[15] U. D. E. Lisboa, “Detection of Vulnerabilities and Automatic Protection for Web Applications,” (2016).
[16] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of SQL Injection and cross-site scripting attacks,” 2009 IEEE 31st Int. Conf. Softw. Eng., pp. 199–209, (2009).
[17] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting Web application vulnerabilities,” (2006) IEEE Symp. Secur. Priv., pp. 260–263, (2006).
[18] I. Medeiros, N. Neves, and M. Correia, “Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining,” IEEE Trans. Reliab., vol. 65, no. 1, pp. 54–69, (2016).
[19] M. Martin and M. S. Lam, “Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking,” USENIX Secur. Symp., pp. 31–43, (2008).
[20] K. Qian, “SAFELI – SQL Injection Scanner Using Symbolic Execution,” pp. 34–39, (2008).
[21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” Proc. 13th Int. Conf. World Wide Web, pp. 40–52, (2004).
[22] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” ACM SIGPLAN Not., vol. 42, no. 6, p. 32, (2007).
[23] J. Dahse and T. Holz, “Static Detection of Second-Order Vulnerabilities in Web Applications,” 23rd USENIX Secur. Symp. (USENIX Secur. 14), pp. 989–1003, (2014).
[24] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An Automata-Based String Analysis Tool for PHP,” Internation Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 154-157, (2010).
[25] H. Shahriar and M. Zulkernine, “MUSIC: Mutation-based SQL injection vulnerability checking,” in Proceedings - International Conference on Quality Software, (2008), pp. 77–86.
[26] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “Efficient and Flexible Discovery of PHP Application Vulnerabilities,” Proc. - 2nd IEEE Eur. Symp. Secur. Privacy, EuroS P 2017, pp. 334–349, (2017).
[27] J. Thomé, L. K. Shar, and L. Briand, “Security slicing for auditing XML, XPath, and SQL injection vulnerabilities,” 2015 IEEE 26th Int. Symp. Softw. Reliab. Eng. ISSRE 2015, pp. 553–564, (2016).
[28] Z. Djuric, “A black-box testing tool for detecting SQL injection vulnerabilities,” in 2013 Second International Conference on Informatics & Applications (ICIA), (2013), pp. 216–221.