Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 4, April 2015, pg.657 – 662

RESEARCH ARTICLE

METHODS FOR DETECTION AND PREVENTION OF SQL ATTACKS IN ANALYSIS OF WEB FIELD DATA

Thiyagarajan A[1], Dr.S.Uma[2], Ambat Vipin[3], Najeem Dheen A[4]

[1]PG Scholar Department of Computer Science and Engineering, HIT Coimbatore
[2]Head of the Department PG Department of Computer Science and Engineering, HIT Coimbatore
[3]PG Scholar Department of Computer Science and Engineering, HIT Coimbatore
[4]PG Scholar Department of Computer Science and Engineering, HIT Coimbatore

PG DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
HINDUSTHAN INSTITUTE OF TECHNOLOGY, COIMBATORE, TAMILNADU, INDIA

Abstract: ***A large number of vulnerability analysis approaches in web based applications detect and report on different categories of vulnerabilities. On the other hand, there is no single approach provides a generic technology independent management of web-based vulnerabilities. In view of the fact that the process of manual code reviews are extremely time-consuming, more chances for error and moreover very costly, the requirement for automated solutions has become essential. In order to overcome these security complications, it is of paramount significance to recognize the typical software defects. Similar to the majority of security vulnerabilities, SQL Injection Attacks (SQLIAs) have become known as one of the most severe attacks to the security of database-driven applications. At the same time developers place new checks in place, in the meantime attackers continue to discover new ways to circumvent these checks. In this paper, a field study is presented on two of the most extensively spread and vital web application vulnerabilities: SQL Injection and XSS. It investigates the source code of security patches of extensively used web applications written in feeble and strong kind of languages. In this research work, initially an overview of the various categories of SQL injection attacks is provided and presented a technique called AMNESIA, which automatically detects and prevents SQL injection threats. AMNESIA makes use of static analysis to construct a model of the genuine queries; an application can produce and subsequently, at execution, ensures that all queries produced by the application comply with this technique. In order to reveal the effectiveness of the proposed technique, assessed web applications and discovered different categories of SQL injection vulnerabilities. In case of both the high analysis speed and the low number of generated false positives, it confirms that this technique can be employed for performing efficient security audits.***

Index terms: **Security, web application vulnerability, web application data, Internet applications, languages, review and evaluation, SQL injection attacks and vulnerability description.**

I. INTRODUCTION

The rapid rise of corporate web applications offers abundant opportunities for e-businesses to flourish. Web applications have become one of the most important communication channels between various kinds of service providers and clients on the Internet. However, this also raises many security issues and exacerbates the demand for practical customer-friendly solutions [1-2]. Although there are many approaches of vulnerability analysis, web applications require a more technology-independent solution. Along with the increased importance of web applications, the negative impact of security flaws in such applications has grown as well.

Vulnerabilities that may lead to the compromise of sensitive information are being reported continuously to network [3]. Costs of the resulting damages are increasing. The main reasons for this phenomenon are time and financial constraints, limited programming skills, and lack of security awareness on part of the developers. In other words, systems can be compromised via web technologies, e.g. exploitation via a web script may start a security breach. Many web vulnerability classes have also been detected, classified and documented [4]. Berghe et al. [5] and Bazaz and Arthur [6] proposed different models of vulnerability taxonomy, on which new analytical methodologies may be designed and implemented. Others only concentrate research on potential classes of vulnerability: Jovanovic et al. contributed work on cross-site scripting (XSS) [7], Kals et al. [8] focus on SQL Injection.

The existing approaches for mitigating threats to web applications can be divided into client-side and server-side solutions. The only client-side tool known to the authors is Noxes [9], an application-level firewall offering protection in case of suspected *cross-site scripting* (XSS) attacks that attempt to steal user credentials. Server-side solutions have the advantage of being able to discover a larger range of vulnerabilities, and the benefit of a security flaw fixed by the service provider is instantly propagated to all its clients. These server-side techniques can be further classified into dynamic and static approaches. Dynamic tools (e.g., [10], and Perl's taint mode) try to detect attacks while executing the audited program, whereas static analyzers [11] scan the entire web application's source code for vulnerabilities before it is deployed.

The security of web applications becomes a major concern and it is receiving more and more attention from governments, corporations, and the research community. Given the preponderant role of web applications in many organizations, one can realize the importance of finding ways to reduce the number of vulnerabilities. *SQL Injection Attacks (SQLIAs)* have emerged as one of the most serious threats to the security of database-driven applications. In fact, the Open Web Application Security Project (OWASP), an international organization of web developers, has placed SQLIAs among the top ten vulnerabilities that a web application. This paper contributes to fill this gap by presenting a study on characteristics of source code defects generating major web application vulnerabilities. The main research goal is to understand the typical software faults that are behind the majority of web application vulnerabilities, taking into account different programming languages.

Regarding the programming language perspective, focused on the most widely used weak typed language, PHP. Then, we analyzed strong typed languages, namely Java, C#, and VB. The proposed approach not only ability to prevent security vulnerabilities, but to analyze the vulnerabilities and their relation with some language characteristics, like the type system. Like most security vulnerabilities, SQLIAs can be prevented by using defensive coding. In practice however, this solution is very difficult to implement and enforce. As developers put new checks in place, attackers continue to innovate and find new ways to circumvent these checks

Present AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks), a fully automated technique and tool for the detection and prevention of SQLIAs.' AMNESIA was developed based on two key insights:

(1) The information needed to predict the possible structure of all legitimate queries generated by a web application is contained within the application's code, and

(2) An SQLIA, by injecting additional SQL statements into a query, would violate that structure. Based on these two insights developed a technique against SQL injection that combines static analysis and runtime monitoring.

The structure of the paper is as follows. Section 2 presents some background on security vulnerabilities and web programming languages. Section 3 details the detection and prevention method for SQL injection attacks detection and classification the patch of each vulnerability. Section 4 discusses the results of the field study. Finally, Section 5 concludes the paper and suggests future work.

II. BACKGROUND KNOWLEDGE

Clowes [12] discussed common security problems related to the easiness in programming with PHP and its features, but this affects many other programming languages. The choice of the type system (strong or weak) and the type checking of the programming language also affects the robustness of the software. For example, a strong typed language with a static type checking can help deliver a safer application without affecting its performance. Scholte et al. [13] presented an empirical study on a large set of input validation vulnerabilities developed in six programming languages. However, that work focused on the relationship between the specific programming language used and the vulnerabilities that are commonly reported, not going into details in what concerns the typical software faults that originate vulnerabilities, like we do in the present work.

The attacker's perspective has also been of some focus in the literature [14-15], but mainly through empirical data gathered by the authors highlighting social networking and what could be obtained from attacking specific vulnerabilities. Some studies analyzed the attacks from the victim's perspective, including the proposal of taxonomy to classify attacks based on their similarities and the analysis of attack traces from HoneyPots to separate the attack types. There is, however, a lack of knowledge about existing exploits and their correlation with the vulnerabilities.

According to OWASP [16], the most efficient way of finding security vulnerabilities in web applications is manual code review. This technique is very time-consuming, requires expert skills, and is prone to overlooked errors. Therefore, security society actively develops automated approaches to finding security vulnerabilities. These approaches can be divided into two wide categories: black-box and white-box testing. The first approach is based on web application analysis from the user side, assuming that source code of an application is not available [17]. The idea is to submit various malicious patterns into web application forms and to analyze its output thereafter. If any application errors are observed an assumption of possible vulnerability is made. This approach does guarantee neither accuracy nor completeness of the obtained results.

The second approach is based on web application analysis from the server side, with assumption that source code of the application is available. In this case dynamic or static analysis techniques can be applied. A comprehensive survey of these techniques was made [18] by Vigna et al. Several tools and techniques have been developed to analyze vulnerabilities in web-based applications. In recent survey, Cova et al. [19] has classified vulnerability analysis of web-based applications according to detection models and analysis techniques. Analysis tools and scanners, however, are specific to web technology and practically, no single scanner provides a complete methodology to find more vulnerabilities and a technology-independent coverage of all possible SQL injection vulnerabilities in the source code with different languages. However, vulnerability description provided by tools or vulnerability databases such as Security Focus Vulnerabilities list. It does not describe the vulnerability in enough detail and a common format.

III. PROPOSED DETECTION AND PREVENTION OF SQL ATTACKS METHODOLOGY

Web Applications Analyzed One mandatory condition of our field study is the ability to analyze the source code of current and previous versions of the target web applications, together with the associated security patches. For the strong typed programming languages, for which used 11 web applications developed in Java, C#, and VB (see Table, Versions of Strong Typed Apps [23])

AMNESIA, (Analysis for Monitoring and NEutralizing SQL Injection Attacks) is a fully-automated and general technique for detecting and preventing all types of SQLIAs. The approach works by combining static analysis and runtime monitoring. Our two key insights behind the approach are that (1) the information needed to predict the possible structure of all legitimate queries generated by a web application is contained within the application's code, and (2) an SQLIA, by injecting additional SQL statements into a query, would violate that structure. In its static part, our technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, our technique monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model. Queries that violate the model represent potential SQLIAs and are reported and prevented from executing on the database. The technique consists of four main steps.

A. AMNESIA Approach

1) *Identify Hotspots:* In this step, AMNESIA performs a simple scan of the application code to identify hotspots. In the Java language, all interactions with the database are performed through a predefined API, so identifying all the

hotspots is a trivial step. Scan the application code to identify hotspots—points in the application code that issue SQL queries to the underlying database.

2) *Build SQL-query models:* In the first part, use Java String Analysis (JSA) to compute all of the possible values for each hotspot's query string. JSA computes a flow graph that abstracts away the control flow of the program and only represents string-manipulation operations performed on string variables. For each string of interest, the library analyzes the flow graph and simulates the string-manipulation operations that are performed on the string. The result is a Non-Deterministic Finite Automaton (NDFA) that expresses, at the character level, all possible values that the considered string variable can assume. Because JSA is conservative, the NDFA for a given string variable is an overestimate of all of its possible values. In the second part, we transform the NDFA computed by JSA into an SQL-query model. More precisely, we perform an analysis of the NDFA that produces another NDFA in which all of the transitions are labeled with SQL keywords, operators, or literal values. We create this model by performing a depth first traversal of the character-level NDFA and grouping characters that correspond to SQL keywords, operators, or literal values. For each hotspot, build a model that represents all the possible SQL queries that may be generated at that hotspot. A SQL-query model is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters, and placeholders for string values.

3) *Instrument application:* Instrument the application by adding calls to the monitor that checks the queries at runtime. For each hotspot, the technique inserts a call to the monitor before the call to the database. The monitor is invoked with two parameters: the query string that is about to be submitted to the database and a unique identifier for the hotspot. Using the unique identifier, the runtime monitor is able to correlate the hotspot with the specific SQL-query model that was statically generated for that point and check the query against the correct model. At each hotspot in the application, add calls to the runtime monitor.

4) *Runtime monitoring:* At runtime, the application executes normally until it reaches a hotspot. At this point, the query string is sent to the runtime monitor, which parses it into a sequence of tokens according to the specific SQL syntax considered. In our parsing of the query string, the parser identifies empty string and empty numeric literals by their syntactic position, and we denote them in the parsed query string using ε. At runtime, check the dynamically-generated queries against the SQL-query model and reject and report queries that violate the model. From detected and prevented SQL injection attacks then performed classification for detected software faults.

B. Classification of Software Faults from the Security Vulnerability Point of View

After choosing a web application, searched the web for all reported SQLi and XSS patches that were classified based on the work presented in [20]. This classification is derived from the code defect types (assignment, checking, interface, and algorithm) of the ODC software fault types [21-22]. As ODC fault types are still too broad , detailed them according to the nature of the defect: missing construct, wrong construct, and extraneous construct. All the security vulnerabilities collected could be classified using only 15 of the fault types already identified and one extra fault type, the missing function call extended (MFCE); however, not all were found in both weak typed and strong typed web applications (see Table, Fault Types Observed in the Field and Corresponding ODC Fault Type [23]). The missing function call extended (marked with an * in Table 2) is a new addition and it is based on a missing function in situations where the return value is used in the code (as opposed to the MFC where the return value is not used).

C. Obtaining the Patch Code

For our field study, need to obtain the web application code, as well as the source code of the patches. To gather the source code of security patches, used several sources of data, such as developer sites, online magazines, news sites, sites related to security, hacker sites, change log files of the application, the version control system (VCS) repository, and so on. Next are the main sources of information

1) *Security patched files:* These files are applied to the application by replacing the vulnerable files. To extract only the code change that these files provide, used the UNIX diff command applied to both the patch and the original file.
2) *Updated versions of the web application:* This represents completely new releases of the application containing new features and fault fixes. It is necessary to compare all the files of the vulnerable and updated versions of the application looking for security fixes. This process can be eased when there is a change log file. After identifying the vulnerable source file and the fix, the UNIX diff command was used.

3) *Security diff files:* These are files containing only the code changes needed to fix a referenced vulnerability. The contents are ready to be applied to the target application using the UNIX patch command. This single file has all the information needed and not common.
4) *Version control system repositories:* Many applications are developed using a VCS to manage the contributions of the community of developers from around the world. Through the change log file, can identify the revisions of the application where vulnerabilities were fixed. A differential analysis using the UNIX diff command obtained the code changes that fixed the vulnerabilities.

IV. EXPERIMENTATION RESULTS

This section presents and discusses the results of the field study. Used the Pearson product-moment correlation (statistically significant when P < 0:05) to see the strength and direction of the relationship of two variables. A positive correlation (positive r) indicates that when one variable increases so does the other and a negative correlation (negative r) indicates that when one variable increases the other decreases. Strong correlation is when r is between 1 and 0.5; medium correlation when r is between 0.5 and 0.3; weak correlation when r is lower than 0.3. The number of samples is n.

For the strong typed language, collected and classified 60 XSS and SQLi vulnerabilities, distributed over 11 web applications comparing with Table, Versions of Strong Typed Apps [23], five fault types (WVAV, WFCS, MLAC, MLOC, ELOC) were not found in this study. The data shows that MFCE is the most frequent as the majority of vulnerabilities are sanitized using functions that clean and validate the input. The high value observed may be related to the common use of specific functions to validate or clean input data. The web applications analyzed are just a small sample of the whole population so, although most of the results have statistical significance, they may lack practical significance. Observations may not apply to other applications, even for those written with the same programming languages. There are many ways and tools to develop an application and they may influence the outcome. This can also be seen from data, if take into consideration the high standard deviation values that represents the data dispersion related to the number of vulnerabilities and exploits per application (See Table, Distribution of Fault Types per Vulnerabilities [23]). Naturally, results will fit better to applications developed with the same languages analyzed, but as improvements are being introduced to those languages results may also change.

V. CONCLUSION AND FUTURE WORK

Web applications having become popular, wide spread and rapidly proliferated raises many security issues and exacerbates the demand for practical solutions. Number of reported web applications vulnerabilities is increasing dramatically. Manual security solutions targeted at these vulnerabilities are language-dependent, type-specific, labor-intensive, expensive and error-prone. Most of them result from improper or none input validation by the web application. SQLIAs have become one of the more serious and harmful attacks on database driven web applications. Analyzes 715 vulnerabilities and 121 exploits of 17 web applications using field data on past security fixes. Some web applications were written in a weak typed language and others in strong typed languages. In this article, have discussed the various types of SQLIAs known to date and presented AMNESIA, a fully automated technique and tool for detecting and preventing SQLIAs. AMNESIA uses static analysis to build a model of the legitimate queries that an application can generate and runtime monitoring to check the dynamically generated queries against this model. Also observed that a single fault type (MFCE) was responsible for most (76 percent) of the security problems analyzed. It see that the fault types responsible for XSS and SQLi belong to a narrow list, which points a path to the improvement of web applications, namely in the context of code inspections and the use of tools for static analysis. Results suggest that applications written with strong typed languages have a smaller number of reported vulnerabilities and exploits. In due course, hope to provide a commercializable tool to web site administrators and web developers to actively secure their applications. In future work will be extended into following directions: First of all, generalization of the model will be developed to support analysis of data flows through other data storage types or implemented by means of stored procedures and triggers. Second, special attention to development of automatic crawling mechanisms will be given. The present work will be extended to focus on the importance of the attack surface in the distribution of vulnerabilities and exploits. This may compare different results of vulnerabilities and exploits of both web applications and their add-ons, regarding their size.

REFERENCES

[1] K. Raina. (2004). Trends in Web Application Security [Online]. Available: http://www.securityfocus.com/print/infocus/1809
[2] J. Grossman, "WhiteHat Website Security Statistics Report," WhiteHat Security, October 2007.

[3] M. Dowd, J. McDonald, and J. Schuh, in The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Addison Wesley Professional, 2006, ch. 1, 2, 3, 4, 8, 13, 17, 18.
[4] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting." in Proc. The 20th IFIP International Information Security Conference, Makuhari-Messe, Chiba, Japan, 30 May - 1 June 2005
[5] C. V. Berghe, J. Riordan, and F. Piessens, "A Vulnerability Taxonomy Methodology applied to Web Services." in Proc. The 10th Nordic Workshop on Secure IT-systems (NORDSEC 2005), Tartu, Estonia, 20- 21 October 2005
[6] A. Bazaz, and J. D. Arthur, "Towards A Taxonomy of Vulnerabilities." in Proc. The 40th Annual Hawaii International Conference on System Sciences (HICSS-40 2007), Waikoloa, HI, USA, 3-6 January 2007, pp. 163a - 163a.
[7] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short paper)." in Proc.The 2006 IEEE Symposium on Security and Privacy (S&P'06), Berkeley/Oakland, California, USA, 21-24 May 2006, pp. 258-263.
[8] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A Web Vulnerability Scanner." in Proc. The 15th International Conference on World Wide Web (WWW 2006), Edinburgh, Scotland, 23–26 May 2006, pp. 247 – 256.
[9] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In The 21st ACM Symposium on Applied Computing (SAC 2006), 2006.
[10] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In IFIP Security 2005, 2005.
[11] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In Proceedings of the 14th Usenix Security Symposium, August 2005.
[12] S. Clowes, "A Study in Scarlet, Exploiting Common Vulnerabilities in PHP Applications," http://www. secureреality. com.au/ studyinscarlet.txt, 2013.
[13] T. Scholte et al., "An Empirical Analysis of Input Validation Mechanisms," Proc. ACM Symp. Applied Computing, pp. 1419-1426, 2012.
[14] S. Clowes, "A Study in Scarlet, Exploiting Common Vulnerabilities in PHP Applications," http://www.secureреality. com. Au / studyinscarlet.txt, 2013.
[15] M. Howard, D. LeBlanc, and J. Viega, "19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them," McGraw-Hill, 2005.
[16] Curphey, M., Wiesman, A., Van der Stock, A., Stirbei, R.: "A Guide to Building Secure Web Applications and Web Services". OWASP (2005).
[17] Andrews, M.: "The State of Web Security". IEEE Security & Privacy, vol. 4, no. 4, pp. 14-15 (2006).
[18] Cova, M., Felmetsger, V., Vigna, G.: "Testing and Analysis of Web Services". Springer (2007).
[19] M. Cova, V. Felmetsger, and G. Vigna: "Vulnerability Analysis of Web based Applications", in Test and Analysis of Web Services, Baresi, L., and Nitto, E.D. (Eds.) Springer Berlin Heidelberg, 2007, pp. 363-394, ch. IV. Reliability, Security, and Trust.
[20] J. Dura˜es and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," Trans. Software Eng., vol. 32, pp. 849-867, 2006.
[21] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurement," IEEE Trans. Software Eng., vol. 18, no. 11, pp. 943-956, Nov. 1992.
[22] J. Christmansson and R. Chillarege, "Generation of an Error Set That Emulates Software Faults," Proc. IEEE Fault Tolerant Computing Symp., pp. 304-313, 1996.
[23] Fonseca, J., Seixas, N., Vieira, M., & Madeira, H. (2014). Analysis of field data on web security vulnerabilities. Dependable and Secure Computing, IEEE Transactions on, 11(2), 89-100.
[24] Halfond, W. G., & Orso, A. (2007). Detection and prevention of sql injection attacks. In Malware Detection (pp. 85-109). Springer US.

