Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool

Tânia Basso Plínio César Simões Fernandes Mario Jino Regina Moraes
State University of Campinas, UNICAMP, Brazil
{taniabasso, pliniocsfernandes}@gmail.com {jino@dca.fee, regina@ft}.unicamp.br

Abstract

Most software systems developed nowadays are highly complex and subject to strict time constraints, and are often deployed with critical software faults. In many cases, software faults are responsible for security vulnerabilities which are exploited by hackers. Automatic web vulnerability scanners can help to locate these vulnerabilities. Trustworthiness of the results that these tools provide is important; hence, relevance of the results must be assessed. We analyze the effect on security vulnerabilities of Java software faults injected on source code of Web applications. We assess how these faults affect the behavior of the scanner vulnerability tool, to validate the results of its application. Software fault injection techniques and attack trees models were used to support the experiments. The injected software faults influenced the application behavior and, consequently, the behavior of the scanner tool. High percentage of uncovered vulnerabilities as well as false positives points out the limitations of the tool.

1. Introduction

Web applications are extremely popular nowadays. From single individuals up to large organizations, there is an increasing dependency on this technology. Information and data are stored, traded and made available on the Web. This type of application is becoming increasingly exposed as any security vulnerability can be exploited by hackers.

Automatic vulnerability scanner tools are often used by developers and system administrators to test Web applications against security vulnerabilities. Reliable results from vulnerability scanners are essential and the analysis of the scanners' effectiveness is important to guide the selection as well as the use of these tools. Effectiveness may be assessed by two main aspects: vulnerability coverage and false positive rate. The vulnerability coverage is associated to the reliability of the tool; high reliability means that the tool is able to detect correctly all security vulnerabilities in the application. (it is doubtful whether undetected vulnerabilities do not really exist in the application or the scanner was not able to detect it). It is important to minimize the rate of false positives because when a non-existent vulnerability is reported, the development team may spend a lot of time trying to correct it before realizing that the false vulnerability does not really exist.

Previous research [1][2] shows that, in general, Web vulnerability scanners present a high number of false-positives and low coverage, highlighting the limitations of this kind of tool. Although other potential causes for vulnerability do exist, the root cause of most security attacks are vulnerabilities created by software faults [3][4].

Our goal is to investigate the effect that injected Java software faults may have on security vulnerabilities. The proposal is to understand, through the analysis of the context of the source code of the applications where the faults were injected, how these faults affects the behavior of the applications with respect to security vulnerabilities. This is important to speed up the detection of security vulnerabilities, allowing that countermeasures are applied to eliminate them or to reduce the severity of their exploitation, contributing to higher levels of dependability for the application under test. Then, we want to analyze how it affects the behavior of the scanner vulnerability tool. In order to validate the scanner results it is necessary to assess its effectiveness. Based on this knowledge, we intend to extend the experiments to other scanner tools and investigate how to scale the results to more complex applications in an automatic way. Then we want to propose a methodology to analyze vulnerability scanners effectiveness based on fault injection and attack injection techniques.

The paper describes a method based on attack trees modeling to perform security tests. The approach consists of injecting software faults into small Java applications. They have to be small because the context of the source code should be analyzed to get accurate measures of the detection coverage and false positives rate, the reason why we want to have the experiments under control. Once the faults are injected, the scan is run to check if it can detect potential vulnerabilities caused by the injected fault.

Creation of vulnerabilities is confirmed through manual attacks, guided by the attack trees.

Unlike other studies, where lots of faults are injected only to validate the scanner tool results, we want to investigate, through application's source code construction and attack models, the relationship between the fault injected and the potential security vulnerabilities created. The method described in this paper will eventually lead to the development of an attack injection tool for Java applications. Later, we intend to reproduce these experiments, automatically, on larger and more complex applications, applying the knowledge acquired through these controlled experiments.

The structure of this paper is as follows: Section 2 presents the background on software faults; Section 3 describes the related work on analysis of scanner tools effectiveness; Section 4 describes the attack trees modeling approach; Section 5 shows the steps and the methodology applied to the experimental study; Section 6 presents the results and discussions on the study; and Section 7 presents our conclusions and future work.

2. Software fault injection

Few works address the relationship between software faults and security vulnerabilities. A study by Fonseca and Vieira [3] analyzed security patches of web applications developed in PHP. The types of faults that are most likely to lead to security vulnerabilities are characterized.

The work by Basso *et al* [4] presents a field data study on real Java software faults, including security faults. The field study was based on security correction patches analysis available in open source repositories. More than 550 faults were analyzed and classified, determining the representativeness of these faults. The authors also define new operators, specific to this programming language structure, guiding the definition of a Java faultload. The software fault injection technique used in this paper is the G-SWFIT [5]. This technique focuses on the emulation of just the most frequent types of faults. It is based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent the most common types of high-level software faults.

To inject the faults, a use case of the application was selected, including all classes in the source code that implements this use case. Then, the locations in this piece of the target code where the injection is performed are selected by the G-SWFIT to inject representative software faults. Each fault was injected in all possible locations of this specific use case, one at time, forming different scenarios to be analyzed.

3 Vulnerability scanner tools effectiveness

Web vulnerability scanners are regarded as an easy way to test applications against vulnerabilities. Most of these scanners are commercial tools (e.g., Acunetix [6], IBM Rational AppScan [7], N-Stalker [8] and HP WebInspect [9]); there are also free ones (e.g., Burp Suite [10] and Gamja [11]), but with limited use, not fully automatic as their commercial equivalent.

Vieira *et al* [1] present an experimental evaluation of security vulnerabilities in publicly available web services. Four well known vulnerability scanners have been used to identify security flaws in web services implementations. A large amount of differences in vulnerabilities detected and a high number of false-positives and low coverage were observed.

Fonseca *et al* [2] propose a method to evaluate and benchmark automatic Web vulnerability scanners using software fault injection techniques. Three leading commercial scanning tools were evaluated and the results also have shown that in general the coverage is low and the percentage of false positives is very high.

However, these studies were focused on a specific family of applications: web services and PHP applications, respectively. Thus, the results obtained cannot be easily generalized and our intention is to complement the results obtained previously, providing Java applications results, aiming to increase the amount of different programming languages applications to obtain the sufficient requirements to generalize the results. Furthermore, their previous study does not present a clear methodology to validate the vulnerabilities detected by scanner tools.

We investigate the behavior of scanner tool in the presence of injected Java faults, show a method using attack trees to model the possible ways to perform attacks to specific vulnerabilities, and validate the results obtained by the scanner. This is addressed in the next sections.

4. Attack trees and security vulnerabilities

Attack trees provide a structural way of describing the security of systems, based on several attacks types [12]. In attack trees, the root node represents the achievement of the ultimate goal of the attack. Each child node represents sub-goals that have to be accomplished for the parent goal to succeed. Parent nodes can have their child nodes related by an OR or an AND relationship. In an OR relationship, if any of the sub-goals are accomplished then the parent node is successful. With an AND relationship, all of the sub-goals must be accomplished for the parent node to be successful. Individual intrusion scenarios are generated by traversing the tree in a depth-first manner. The objective is to cover all actions represented in the leaves.

In our work the attack trees were used to describe the possibilities of attacking a specific type of security vulnerability. We consider three types of security

vulnerabilities: SQL Injection , Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). They were selected because of their criticality, occupying the first, second and fifth place in the OWASP Top 10 [13]. These vulnerabilities are widely spread and dangerous vulnerabilities, and may cause major damage to the victims.

XSS occurs when a web application gathers malicious data from a user. The data is usually gathered in the form of a hyperlink which contains malicious content within it. The user will most likely click on this link from another website, instant message, or simply just reading a web board or e-mail message. After the data is collected by the Web application, it creates an output page for the user, containing the malicious data that was originally sent to it, but in a manner to make it appear as valid content from the website [14]. The malicious code can also be permanently stored on the target servers, such as in a database or it can be generated dynamically through the Document Object Model of the browser. SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database [15].

CSRF works by exploiting the trust a site has for the user. Site tasks are usually linked to specific URLs (e.g: http://site/stocks?buy=100&stock=ebay) allowing specific actions to be performed when requested. If a user is logged into the site and an attacker tricks their browser into making a request to one of these task URLs, then the task is performed and logged as the logged in user [16].

For each of these three types of vulnerability an attack tree was created. Figure 1 presents the attack tree for validation of CSRF vulnerabilities. This tree was chosen because it is the vulnerability that appeared more frequently in the results. Due to space restrictions, the other trees are not presented, but they can be seen elsewhere [17]. The "OR" labels are omitted to improve the tree simplicity.

For the CSRF tree we covered the part of the CSRF attack relative to the acceptance of the requests coming from another source. The part relative to the means used to lure the user to activate the request is not covered as they are out of the defensive bounds that an application can have against CSRF.

In Figure 1, the first step to perform a CSRF attack is to have the user logged in the site because the attack will use its trust in the user authentication. If this step is not fulfilled the attack could not be realized. The next step is to analyze the request from the site that the attack will target in order to be able to reproduce it. If the site does not have CSRF countermeasures this step will lead to the next one because the request will be considered valid and will take effect on the site.

If the site uses any defensive measure it will be necessary to analyze the request and take additional actions. A known defensive method consists in appending different tokens to each request, but this approach can be bypassed if the application is vulnerable to XSS attacks. This is possible because XSS attacks permit to get valid session tokens from the application. The last path (the three remaining leaf nodes) of the tree shows how to overcome applications that use verification of the HTTP (Hypertext Transfer Protocol) Referrer attribute, although this is not a recommended defensive measure.

5. The experimental study

Two open source Web applications developed in Java were selected to carry out the experiment. The first one, which we call App1, is a Customer Relationship Manager (CRM) and Project Management Tool. It uses the MySQL database and technologies such as Hibernate, Struts

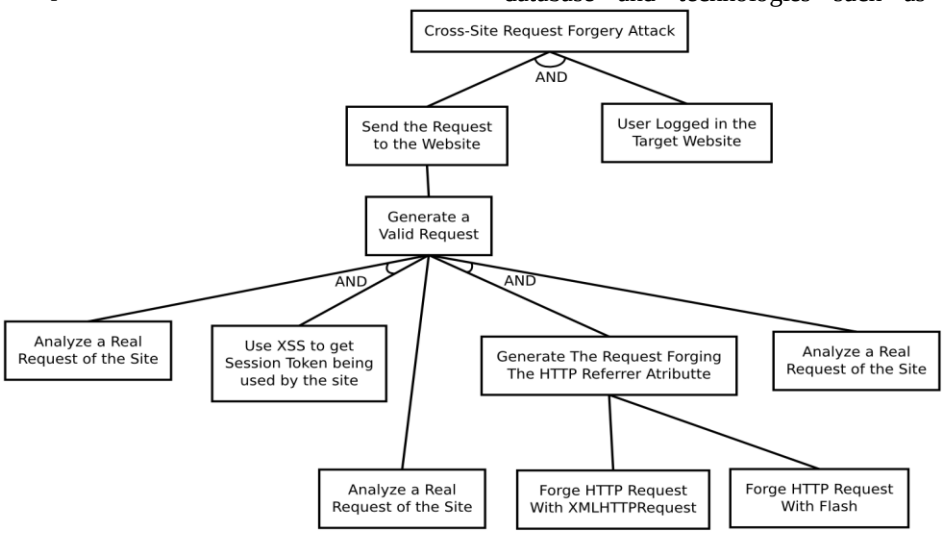

Figure 1. CSRF attack tree

framework, and Jasper Reports. The second Web application, App2, is a management system for Distance Education, developed by the Brazilian federal government. It uses the Postgres database and technologies such as Hibernate and Ajax. We have chosen similar use cases from both applications to be the target piece of code of injected faults.

The types of fault to be injected were selected from the faultload of Basso *et al* [4]. We selected the two most frequent types of faults observed: the Missing Function Call (MFC) and the Missing If construct plus the Statement (MIFS) for this first set of experiments. They represent the most common types of fault that are responsible for security vulnerabilities. The MFC is the fault type which relates to function or method call that was missing from implementation. The MIFS is a fault type that represents the omission of a block of code made of an "if" construct and its associated statements which are executed only if the "if" condition is true [5]

The security vulnerability scanner was selected because of its great market insertion and availability. We do not mention its brand because commercial licenses do not allow in general the publication of tool evaluation results. Basically, the operation of the scanner consists of two stages: explore and test. During the explore stage, requests are send to the application and the responses are analyzed, looking for indication of potential vulnerabilities. In the test stage, the tool sends thousands of custom tests to identify security problems (based on the results of the first stage) and rank their level of security risk.

The three security vulnerabilities considered for this study are discussed in Section 4

5.1. Injecting faults, executing the scans and validating the results

The tests start with a "Gold Run", where the application is tested once by the scanner tool without any fault injected. The web application may already have some vulnerabilities and this run should be able to find most of them. The results of "Gold Run" execution are collected to compare other results when the faults are injected.

After the "Gold Run", one fault is injected. The context of the code where the fault is injected is analyzed manually to understand the effect of this fault in the applications behavior. If necessary, data are inserted, removed or changed in the database of the application under test to guarantee the activation of the fault. Next, the code and database are versioned, defining a scenario to be tested.

The scanner application is run and verification for new vulnerabilities is made, i.e., we identify new vulnerabilities when comparing the vulnerabilities detected in the original application (without any fault injected) . In some cases, one fault injected can be responsible for many security vulnerabilities. If new vulnerabilities are detected, attacks are performed in the current scenario using the attack trees. To exploit the new vulnerabilities all possibilities detected through the attack tree are experimented. This aims to verify if the new vulnerability actually exists or if it is a false positive. Then, the same attacks are performed in the original application scenario (without any fault injected) in order to verify if the vulnerability existed before the fault injection and was not identified by the tool (lack of coverage).

The procedure is done for each possible location in the source code where faults can be injected in accordance with G-SWFIT technique (for the selected use case).

6. Results and discussions

For both Web applications, we analyzed, respectively, 11 and 23 different scenarios. Table 1 shows the total of scenarios that presented new security vulnerabilities detected by the scanner due to the fault injection.

Table 1. Applications scenarios and vulnerabilities

	App1	App2
Total scenarios analyzed	11	23
Scenarios with new vulnerabilities	5	7
% of faults that affected the scan	46%	30%

According to Table 1, about 40% of the injected software faults affected the scanner results. A detailed analysis of the context of application's source code where the faults were injected is important to assess the effect of the fault in the application behavior. The context analysis in conjunction with the structure of the attacks permits to assess the influence of the injected fault on potential new security vulnerabilities detected by the scanner tool. Consequently, this procedure permits to assess correctly the effectiveness of the scanner tool, through the identification of lack of coverage and false positives.

We noticed that the injected faults were not in the same locations the new vulnerabilities arose. The injected faults affected the applications behavior and, consequently, the scanner tool behavior, due to the context of the application and the procedures necessary to activate the fault. For example, many faults were

injected in locations where a null entry point is verified in the source code. Activating this fault, the application modifies its behavior by not verifying the null entry point and forcing the application to display error pages. Also, according to the attack structure in Figure 1, the verification of null entry points is not explored, i.e., it doesn't create a security vulnerability.

Even though the injected software faults are unrelated to the location of new vulnerabilities, they affected the results of the scanner tool. Table 2 shows the lack of coverage and the number of false positives obtained in the experiments. The lack of coverage is about vulnerabilities that do exist in the web applications, confirmed through successful manual attacks. The number of false positives is related to vulnerabilities indicated by the tool that were not confirmed by the manual attacks. Table 3 shows the percentage of lack of coverage and false positives according to each type of security vulnerability, where the last column resume the percentage of the total of experiments including all types of vulnerabilities.

Table 2. Applications lack of coverage and false positives

	App1	App2	Total
Vulnerabilities not detected (lack of coverage)	8	1	9
False positives	5	3	8

Table 3. Percentage of security vulnerabilities: lack of coverage and false positives

	XSS	SQL inject	CSRF	Total
Vulnerabilities	2	2	15	19
Lack of coverage (%)	0%	0%	60%	47%
False positive (%)	50%	100%	34%	43%

Based on Table 2 the App1 presented worse lack of coverage and more false positives than App2. By the analysis of the context of the source code, we believe that the App2 code is more modularized, with less coupling with other modules and fewer use cases. It is also smaller, i.e., has fewer lines of code (LOC). Thus, it is easier to activate the injected faults and easier to control the application's behavior. Similarly, it is easier for the tool to analyze the application and detect the vulnerabilities in a correct way.

All 9 undetected vulnerabilities are about CSRF and are part of the vulnerabilities presented in Table 3. They represent 60% of the lack of coverage. These lacks of coverage were identified in the original applications (without any fault injected) and in the applications with faults injected. In most of the cases, when scanning the application with faults injected, a new vulnerability detected by the tool was one already present in the original application, not identified in the "Gold Run".

Also in Table 3, the false positives come from the three types of security vulnerabilities: XSS, SQL injection and CSRF, representing, respectively, 50%, 100% and 34% of the vulnerabilities detected. The false positive associated to the XSS vulnerabilities is considered because the scanner tool integrates outdated version of internet browsers. An attack successfully executed by the tool, when executed in the later versions of internet browsers, has no effect, because these versions implement features that do not permit the execution of common XSS attacks.

The SQL injection false positives were identified through the attacks and the analysis of the source code. Both applications use the Hibernate technology, which is an object/relational persistence and query service [18]. It permits to encapsulate the queries and send objects to the database through predefined classes and methods, discarding the necessity of explicit SQL queries constructions. According to forums and some information available in technical websites [19][20], in code constructed with Hibernate it is more difficult – but not impossible – to have vulnerability to SQL injection attacks. However, the way that the application was coded, i.e., extremely encapsulated, do not open opportunities to develop successful attacks. Even the scanner tool provides no assurance about its detection result, and it informs that this detected vulnerability requires user verification.

Most of cases where CSRF false positives were identified, they happened in error pages. An attacker performing a CSRF attack to access an error page can be dangerous if the error page presents links or buttons which permit access to the application (as "back" buttons which bring back the user to the last page he/she accessed) or if the error page displays private information about the system (such as database name or table names). For both applications, the error pages do not present any way of accessing application functionalities or private information. Hence, we considered theses cases as false positives because a CSRF attack when accessing the error pages is useless.

The last column of Table 3 shows the total percentage of lack of coverage and false positives. From the 19 vulnerabilities investigated, 42% are false positives and 47% were not identified by the scanner tool. It indicates the limitations of this tool we found in this study.

7.Conclusions

In this paper we present an experimental study where we analyzed the effect that Java software faults, injected on the source code of Web applications, can have on security vulnerabilities. Also, we analyzed the influence of these faults on the security vulnerabilities detection by a well known commercial web security vulnerability scanner tool. These analyses were performed based on a method that uses attack trees modeling in order to verify the results obtained by the scanner tool.

Fault injection techniques were used to support the experiments and software faults were injected, one at time, in a controlled way, into target Java codes of two small Web applications.

The context of the application code where the fault was injected is analyzed in order to understand the relationship between the fault and potential new vulnerabilities. Manual attacks were performed guided by attack tree models to confirm the existence of vulnerability.

Results show that, according to the context of both Java target code applications and considered security vulnerabilities structure, the location of the injected faults were not where new vulnerabilities arose. However, the injected faults did affect the behavior of the application and, consequently, the behavior of the scanner tool in detecting new vulnerabilities..

Results of the scanner tool were validated through manual attacks based on attack trees. It showed high percentage of lack of coverage and many false positives, showing its limitations. Some factors that influenced this percentage are, in addition to the activation of the faults injected into the source code of the applications, the use of different development technologies (such as Hibernate) and some outdated features of the tool (as the internal internet browser).

As future work we intend to extend this experiment analyzing the effect of other types of faults and the effectiveness of other vulnerability scanner tools. We also intend to develop a tool to perform the attacks (based on attack trees) automatically.

8. References

[1] M. Vieira, N. Antunes, H. Madeira. "Using Web Security Scanners to Detect Vulnerabilities in Web Services". *IEEE/IFIP Intl Conf. on Dependable Systems and Networks, DSN 2009*, Lisboa, Portugal, June 2009.

[2] J. Fonseca, M. Vieira, H. Madeira. "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks", *13º IEEE Pacific Rim Dependable Computing Conference (PRDC 2007)*, Melbourne, Victoria, Australia, December 2007.

[3] J. Fonseca, M. Vieira. "Mapping software faults with web security vulnerability". *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2008)*, Anchorage, USA, 2008.

[4] T. Basso, R. Moraes, B. P. Sanches, M. Jino. "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults*." In: Workshop de Testes e Tolerância a Falhas - WTF2009*, João Pessoa, Brazil, 2009, pp. 1-13.

[5] J. Durães, H. Madeira. "Emulation of Software Faults: A Field Data Study and Practical Approach". *IEEE Trans. on Software Engineering*, vol. 32, n. 11, Nov. 2006, pp.849-867.

[6] Acunetix Web Appication Security. Available in http://www.acunetix.com, November/2009.

[7] IBM Rational AppScan. Available in http://www-01.ibm.com/software/awdtools/appscan/, November/2009.

[8] N-Stalker The Web Security Specialists. Available in http://www.nstalker.com/, November/2009.

[9] HP WebInspect. Available in https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__, November/2009.

[10] Burp Suite. Available in http://www.portswigger.net/suite/, November/2009.

[11] Gamja. Available in http://sourceforge.net/projects/gamja/, November/2009.

[12] B. Schneir. "Attack Trees: Modeling Security Threats", Dr. Dobb's Journal, December, 1999.

[13] OWASP Top 10 Project. Available in http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, February/2010.

[14] CGISecurity.com. "The Cross Site Scripting FAQ." Available in http://www.cgisecurity.com/xss-faq.html, November/2009.

[15] W. G. Halfond, J. Viegas, A. Orso, "A classification of SQL injection attacks and countermeasures". In Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, Virginia, March/2006.

[16] R. Auger. "The Cross-Site Request Forgery (CSRF/XSRF) FAQ". Available in http://www.cgisecurity.com/csrf-faq.html, November/2009.

[17] Research Test Group. Available in http://www.ceset.unicamp.br/docentes/regina/projeto/, December/2009.

[18] Hibernate. Available in https://www.hibernate.org/, December/2009.

[19] OWASP – Preventing SQL injection in Java. Available inhttp://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java#Hibernate, December/2009.

[20] CWE – Common Weakness Enumeration. Available in http://cwe.mitre.org/data/definitions/564.html, December/2009.