2009 15th IEEE Pacific Rim International Symposium on Dependable Computing

Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services

Nuno Antunes, Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
nmsa@dei.uc.pt, mvieira@dei.uc.pt

***Abstract*—Web services are becoming business-critical components that must provide a non-vulnerable interface to the client applications. However, previous research and practice show that many web services are deployed with critical vulnerabilities. SQL Injection vulnerabilities are particularly relevant, as web services frequently access a relational database using SQL commands. Penetration testing and static code analysis are two well-know techniques often used for the detection of security vulnerabilities. In this work we compare how effective these two techniques are on the detection of SQL Injection vulnerabilities in web services code. To understand the strengths and limitations of these techniques, we used several commercial and open source tools to detect vulnerabilities in a set of vulnerable services. Results suggest that, in general, static code analyzers are able to detect more SQL Injection vulnerabilities than penetration testing tools. Another key observation is that tools implementing the same detection approach frequently detect different vulnerabilities. Finally, many tools provide a low coverage and a high false positives rate, making them a bad option for programmers.**

Keywords-Security; Vulnerabilities; SQL Injection; Penetration Testing; Static Code Analysis; Web Services

I. INTRODUCTION

The security of web applications is, in general, quite poor [5][20]. To prevent vulnerabilities, developers should apply coding best practices, perform security reviews of the code, execute penetration tests, use code vulnerability analyzers, etc. However, many times, developers focus on the implementation of functionalities and on satisfying the user's requirements and disregard security aspects. Additionally, numerous developers are not specialized on security and the common time-to-market constraints limit an in depth test for security vulnerabilities.

Web services are nowadays a strategic mean for data exchange and systems integration as they provide a simple interface between a provider and a consumer [4]. The Simple Object Access Protocol (SOAP) [6] is used for exchanging XML-based messages between the consumer and the provider over the network (using for example http or https protocols). In each interaction the consumer (client) sends a request SOAP message to the provider (server). After processing the request, the server sends a response message to the client with the results. A web service may include several operations (in practice, each operation is a method with several input parameters) and is described using WSDL (Web Services Definition Language) [6], which is a XML format used to generate server and client code, and for configuration. A broker is used to enable applications to find web services.

Web services are typically so widely exposed that any existing security vulnerability will most probably be uncovered and exploited by hackers. Security vulnerabilities like SQL Injection are particularly relevant in web services as these typically use a relational database for persistent storage. Additionally, SQL Injection vulnerabilities are directly related to the way the web service code is structured [16][20] as, in practice, SQL Injection attacks take advantage of improperly validated input parameters to change SQL commands that are sent to the database.

Penetration testing and static code analysis are two well-know techniques frequently used by web service developers to identify security vulnerabilities in their code. Penetration testing consists in stressing the application from the point of view of an attacker ("black-box" approach) using specific malicious inputs. On the other hand, static code analysis is a "white-box" approach that consists in analyzing the source code of the application (without execution it) looking for potential vulnerabilities (among other types of software defects). Both penetration testing and static code analysis can be performed manually or automatically. However, automated tools are the most frequent choice as, comparing to manual tests and inspection, execution time and cost is quite low. A key difference between penetration testing and static code analysis is that the first does not require access to the source code (or bytecode) while the second does. In the context of this paper we assume the use of these tools by the web services developers, which always have access to the source code.

Penetration testing tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type (see Section II.B for examples on existing vulnerability scanners). Previous research shows that the effectiveness of these tools in web services is very poor. In [18] authors used four commercial tools (including two different versions of a given brand) to identify security flaws in 300 publicly available web services. The differences in the vulnerabilities detected by each tool, the low coverage (less than 20% for two of the scanners), and the high number

978-0-7695-3849-5/09 $26.00 © 2009 IEEE
DOI 10.1109/PRDC.2009.54

of false positives (35% and 40% in two cases) observed, highlight the limitations of these tools.

Static code analyzers analyze the code without actually executing it [12]. The analysis performed by existing tools varies depending on their sophistication, ranging from tools that consider only individual statements and declarations to others that consider the complete code. Among other usages (e.g., model checking and data flow analysis), these tools provide an automatic way for highlighting possible coding errors (see Section II.C for examples of existing static analyzers). In [21] authors evaluated three bug finding tools and compared their effectiveness with a review team inspection. The tools achieved higher efficiency than the review team in detecting software bugs (the study did not consider security issues) in five industrial Java-based applications, but all the tools presented false positive rates higher than 30%.

Due to time constraints or resource limitations developers frequently have to choose between performing penetration testing or static code analysis. Additionally, they typically have to select a specific tool from the large set of tools available (usually, without really knowing how effective each tool is) and strongly trust on that tool to detect potential security issues in the code being developed. In this work we conducted an experimental evaluation of several automatic penetration testing tools and static analysis tools, focusing on two key measures of interest: **coverage** and **false positives** rate. The first portrays the percentage of existing vulnerabilities that are detected by a given tool, while the second represents the number of reported vulnerabilities that in fact do not exist. The three key observations from our experiments are: 1) the coverage of static code analysis is typically higher than of penetration testing; 2) false positives are a problem for both approaches, but have more impact in the case of static analysis; and 3) different tools implementing the same approach frequently report different vulnerabilities in the same piece of code. Note that, although the results of this study cannot be generalized (we did not test all the tools available in the field and the web services used are limited in scope), they highlight several differences between penetration testing and static code analysis and the strengths and limitations of several existing tools.

The structure of the paper is as follows. Section II presents the experimental study, including the tools considered. Section III presents the results obtained. Section IV concludes the paper and suggests future work.

II. THE EXPERIMENTAL STUDY

Our experimental study consisted of four steps:

1. **Preparation**: select the penetration testers and static code analyzers to be experimented and the web services to be considered.
2. **Execution**: use the tools to identify potential vulnerabilities in the web services.
3. **Verification**: perform manual testing to confirm that the vulnerabilities identified by the tools do exist (i.e., are not false Ωpositives).
4. **Analysis**: analyze the results obtained and systematize the lessons learned.

A. Web Services Tested

Eight web services providing a total of 25 operations have been used in the study (see Table I). Four of these services implement a subset of the web services specified by the standard TPC-App performance benchmark [17]. Four other services have been adapted from code publicly available on the Internet [14]. These eight services are implemented in Java and use a relational database to store data and SQL commands to manage it.

Table I characterizes the web services (the source code can be found at [2]), including the number of operations per service (#Op), the total lines of code (LoC) per service, the average number of lines of code per operation (LoC/Op), and the average cyclomatic complexity [13] of the operations (Avg. C.). These indicators were calculated using SourceMonitor [15] and due to space constraints we do not discuss this characterization (the information provided is quite intuitive).

B. Penetrating Testing Tools Studied

In this work we have considered three well known commercial penetration-testing tools (representative of the state-of-the-art) and a tester proposed in a previous research work [3].

HP WebInspect "*performs web application security testing and assessment for today's complex web applications, built on emerging Web 2.0 technologies. HP WebInspect delivers fast scanning capabilities, broad security assessment coverage and accurate web application security scanning results*" [8]. This tool includes pioneering assessment technology, including simultaneous crawl and audit (SCA) and concurrent application scanning. It is a broad application that can be applied for penetration testing in web-based applications.

TABLE I. WEB SERVICES CHARACTERIZATION.

	Service	Short Description	#Op	LoC	LoC/Op	Avg. C.
TPC-App	ProductDetail	Get details about a product	1	105	105,0	6,0
	NewProducts	Add new product to the database	1	136	136,0	6,0
	NewCustomer	Add new customer to the database	1	184	184,0	9,0
	ChangePaymentMethod	Change customer's payment method	1	97	97,0	11,0
Public-Code	JamesSmith	Manages personal data about students	5	270	54,0	6,0
	PhoneDir	Phone book	5	132	26,4	2,8
	Bank	Manages bank operations	5	175	35,0	3,4
	Bank3	Manages bank operations (different from the Bank service)	6	377	62,8	9,0

IBM Rational AppScan "*is a leading suite of auto-mated Web application security and compliance assessment tools that scan for common application vulnerabilities*" [9]. This tool is suitable for users ranging from non-security experts to advanced users that can develop extensions for customized scanning environments. IBM AppScan can be used for security testing in web applications, including web services.

Acunetix Web Vulnerability Scanner "*is an automated web application security testing tool that audits your web applications by checking for exploitable hacking vulnerabilities*" [1]. Besides web services, Acunetix Web Vulnerability Scanner can be applied for security testing in web applications in general.

The last penetration-testing tool considered implements the approach proposed at [3]. This tool has shown that, in many cases, it is able to achieve better results than commercial scanners, providing higher coverage and lower rate of false positives. It uses a representative workload to exercise the services, implements a large set of attacks (which are a compilation of all the attacks performed by a large set of existing scanners plus many attack methods that can be found in the literature), and apply well defined rules to analyze the web services responses in order to improve coverage and remove false positives.

For the results presentation we have decided not to mention the brand of the commercial scanners to assure neutrality and because licenses do not allow, in general, the publication of evaluation results. This way, the penetration-testing tools are referred in the rest of this paper as VS1, VS2, VS3, and VS4 (without any order in particular).

An important aspect is that, before running a penetration-testing tool over a given service, the underlying database was restored to a predefined state. This avoids the cumulative effect of previous tests and guarantees that all the tools started the service testing in a consistent state. If allowed by the testing tool, information about the domain of each parameter was provided. If the tool requires the user to set an exemplar invocation per operation, the exemplar respected the input domains of operation. All the tools in this situation used the same exemplar.

C. Static Code Analyzers Studied

Three vastly used static code analyzers that provide the capability of detecting vulnerabilities in Java applications' source or bytecode have been considered in this study. Note that, we have selected analyzers that focus on the Java language as this language is nowadays largely used for the development of web applications. Additionally, the web services used in the experiments are implemented in Java. In the rest of this paper we referred to these tools as SA1, SA2 and SA3 (with no particular order).

FindBugs [7] is "*a program which uses static analysis to look for bugs in Java code*" that is able to scan the bytecode of Java applications detecting, among other problems, security issues (including SQL injection). It is one of the most used tools for static code analysis.

Yasca (Yet Another Source Code Analyzer) [19] is "*a framework for conducting source code analyses*" in a wide range of programming languages, including java. Yasca includes two components. The first is a framework for conducting source code analyses and the second is an implementation of that framework that allows integration with other static code analyzers (e.g., FindBugs, PMD, Jlint).

IntelliJ IDEA [10] is a commercial tool that provides a powerful IDE for Java development and includes "inspection gadgets" plug-ins with automated code inspection functionalities. IntelliJ IDEA is able to detect security issues, such as SQL problems, in java source code.

An important aspect is that, during the experiments the static analyzers were configured to fully analyze the services code. For the analyzers that use binary code, the deployment-ready version was used.

III. Results and Discussion

This section presents the experimental results. As first step, a team of security experts performed a manual code inspection to identify the existing vulnerabilities. Note that, for the results analysis we assume that the services have no more vulnerabilities than the union of the vulnerabilities detected by the security experts and by the tools used (obviously, excluding the false positives).

A. Web Services Manual Inspection

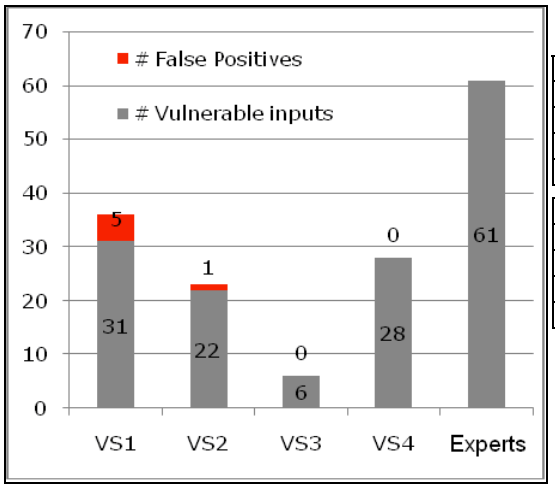
To perform a correct evaluation of our approach it is essential to correctly identify the vulnerabilities that exist in the services code. This way, a team of developers with experience in security of database centric applications was invited to review the source code looking for vulnerabilities (false positives were eliminated by cross-checking the vulnerabilities identified by different people).

A key difficulty of this study is that different tools report (and count) vulnerabilities in different ways. In fact, for penetration-testing tools (that identify vulnerabilities based on the web service responses) a vulnerability is counted for each vulnerable parameter that allows SQL injection. On the other hand, for static analysis tools (that vet services code looking for possible security issues) a vulnerability is counted for each vulnerable line in the service code. Due to this dichotomy, we asked the security experts to identify both the parameters and source code lines prone to SQL Injection attacks.

Table II presents the summary of the vulnerabilities detected by the security experts. The results show a total number of 61 vulnerable inputs and of 28 vulnerable lines of code in the set of services considered. Due to space reasons we do not detail these results. Interested readers can found those at [2].

TABLE II. Vulnerabilities found in the services' code.

Service	#Vuln. Inputs	#Vuln. Lines
ProductDetail	0	0
NewProducts	1	1
NewCustomer	15	2
ChangePaymentMethod	2	1
JamesSmith	20	5
PhoneDir	6	4
Bank	4	3
Bank3	13	12
Total	**61**	**28**



Tool	% F. P.
VS1	14.0%
VS2	4.0%
VS3	0.0%
VS4	0.0%

Tool	Coverage
VS1	50.8%
VS2	36.1%
VS3	9.8%
VS4	45.9%

Figure 1. Coverage and false positives for penetration testing.

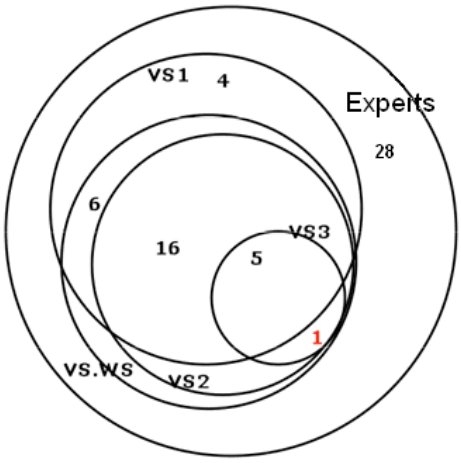


Figure 2. Vulnerabilities intersections for penetration testing.

B. *Penetration Testing Results*

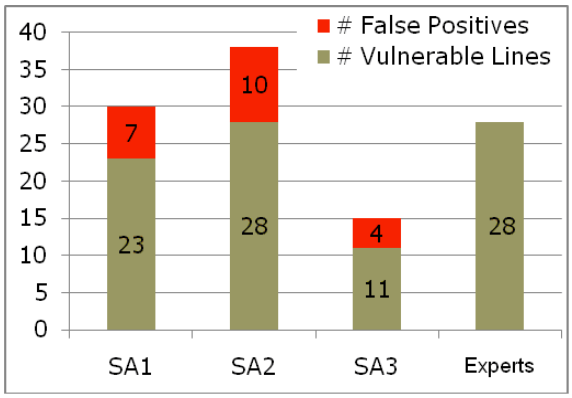
Fig. 1 shows the results for the execution of the penetration testing tools (the number of vulnerabilities detected by the security experts is shown in the last column of the graphic). As we can see, the different tools reported different numbers of vulnerabilities. An important observation is that none of the penetration-testing tools detected more than 51% of the vulnerabilities detected by the security experts (and, excluding the false-positives, penetration testing tools did not detect any vulnerability that was not previously detected by the security team). VS1 identified the higher number of vulnerabilities (50.8% of the total vulnerabilities). However, it was also the scanner with the higher number of false positives (it detected 5 vulnerabilities that, in fact, do not exist). VS4 as a coverage slightly lower than VS1, but reports no false positives. The very low number of vulnerabilities detected by VS3 can be partially explained by the fact that this tool does not allow the user to set any information about input domains, nor it accepts any exemplar request. This means that the tool generates a completely random workload that, most probably, is not able to test the parts of the code that require specific input values in order to be executed.

The intersection of the vulnerable inputs detected by the different tools and by the security experts is illustrated in Fig. 2. Here, the areas of the circles are roughly proportional to the number of vulnerabilities detected by the respective tool. The same does not happen with the intersection areas, as it would be impossible to represent it graphically. As we can see, different penetration testing tools detected different vulnerabilities. From the set of vulnerabilities detected by at least one tester, we observe that VS1 misses only one. This is interesting, considering that all the other scanners detected it, although they presented lower coverage than VS1. Additionally, only 5 vulnerabilities were detected by all the penetration testing tools, but this number is, obviously, limited by the low coverage of VS3.

C. *Static Code Analysis Results*

Fig. 3 shows the number of vulnerable lines identified by the static code analyzers (the number of vulnerabilities detected by the security experts is shown in the last column of the graphic). As we can see, SA2 detected the higher number of vulnerabilities, with 100% of coverage, but identified 10 false positives, which represents 26.3% of the vulnerabilities pointed. The coverage of SA3 (39.3) is very low when comparing to the other two scanners. The high rate of false positive is, in fact, a problem shared by all the static analyzers used, as all reported more than 23% of false positives. These tools detect certain code patterns that usually indicate vulnerabilities, but the problem is that many times they detect vulnerabilities that do not exist (like in the Example 2 presented in Table III).

Fig. 4 illustrates the intersection of vulnerable lines detected by the different tools. Here it is visible that different analyzers detect different vulnerabilities. SA2 detected exactly the same 28 vulnerabilities pointed by the security experts. A key observation is that only 9 out of the 28 existing vulnerabilities were detected by all the static



Tool	% F. P.
SA1	23.3%
SA2	26.3%
SA3	26.7%

Tool	Coverage
SA1	82.1%
SA2	100.0%
SA3	39.3%

Figure 3. Coverage and false positives for static analysis.

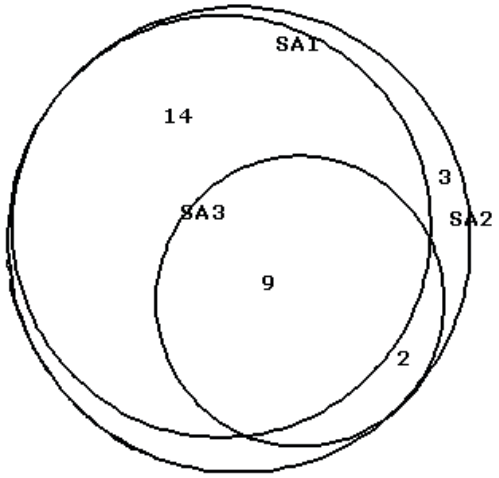


Figure 4. Vulnerabilities intersections for static analysis.

analyzers, but this is obviously limited by the coverage of SA3.

D. Comparing Penetration Testing with Static Analysis

As mentioned before different tools report (and count) vulnerabilities in different ways (i.e., vulnerable parameters or vulnerable lines), which means that it is not possible to compare penetration testing tools and static code analyses tools in terms of the absolute number of vulnerabilities detected. This way, in this section we compare the two detection approaches based on the coverage and false-positives percentages. It is important to emphasize that this comparison cannot be generalized as we tested a limited set of tools and used a limited set of web services. In other words, the coverage and false-positive results presented are only valid in the context of the experiments performed in the present work. Nevertheless, these results provide an interesting indication on the usefulness of the two approaches and on the effectiveness of some existing tools.

Fig. 5 compares the coverage and the false-positives for the tools tested in the present work. As we can see, in general, the static code analyzers present better coverage results than the penetration testing tools. The only exception is SA3 that has a detection coverage lower than VS1. The other two static analyzers achieved a coverage much higher than any of the penetration testers. However, all the static code analyzers reported many more false positives than any of the penetration testing tools. The difference is in fact high (more than 10%), even if we compare the analyzers with VS1, which is the penetration-testing tool with higher rate of false positives (but it is also the one with higher coverage).

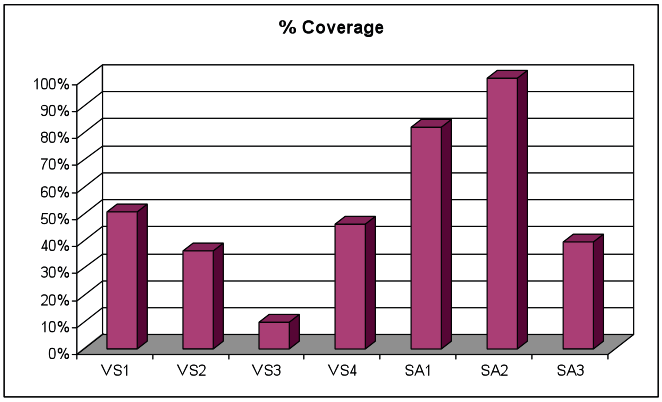
To better understand some of the differences between penetration testing and static code analysis, Table III presents two examples of situations where one of these techniques typically fails on detecting vulnerabilities, while the other typically succeeds (unfortunately, due to space reasons we cannot present more examples). It is important to emphasize that these are simple but real code examples (from the web services used in the present study) that are often present in real applications.

Example 1 represents a typical case where it is impossible for a penetration testing tool to detect the existing SQL Injection vulnerability as the operation has no return value and exceptions related with SQL mal-formation do not leak out to the invocator (i.e., the testing tool). On the other hand, using static code analysis this vulnerability is quite easy to detect.

Example 2 presents a typical situation where a static code analyzer would detect a vulnerability that does not exist. In this case, analyzers identify the vulnerability because the SQL query is a non-constant string, built by concatenating string values that include, at least, one of the input parameters. However, in this example, the input is previously verified by executing the parseInt instruction that raises a RuntimeException if a non-integer value is used, which prevents code injection attacks. In the case of penetration testing, this false positive would not be reported as the exception would be reported to the invocator during runtime.

IV. CONCLUSION AND FUTURE WORK

This paper presented an experimental study on the comparison of several web vulnerability detection tools implementing either penetration-testing or static code



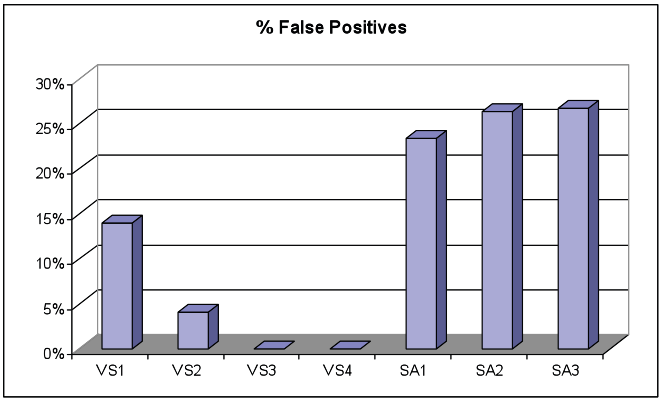


Figure 5. Penetration testing *vs* Static code analysis.

TABLE III. EXAMPLES OF PROBLEMATIC VULNERABILITIES.

Example 1

```
public void operation(String str) {
 try {
  String sql = "DELETE FROM table WHERE id='" + str + "'";
  statement.executeUpdate(sql);
 }
 catch (SQLException se) {}
}
```

Example 2

```
public void operation(String str) {
 int i = Integer.parseInt(str);
 try {
  String sql = "DELETE FROM table WHERE id='" + str + "'";
  statement.executeUpdate(sql);
 }
 catch (SQLException se) {}
}
```

analysis. Several commercial and open source tools were used to detect SQL Injection vulnerabilities in a set of vulnerable services. The results for penetration testing tools and static code analysis tool were analyzed separately and then compared to better understand the strengths and weaknesses of each approach.

Results showed that the coverage of static code analysis tools is typically much higher than of penetration testing tools. False positives are a problem for both approaches, but have more impact in the case of static analysis. A key observation is that different tools implementing the same approach frequently report different vulnerabilities in the same piece of code. Although the results of this study cannot be generalized, they highlight the strengths and limitations of both approaches and suggest that future research on this topic is of utmost important. Finally, results show that web services programmers should be very careful when selecting a vulnerability detection approach and when interpreting the results provided by automated tools.

Future research includes improving the current state-of-the-art on testing and static code analysis for vulnerabilities detection. Future approaches should take the best from penetration testing and static code analysis in order to build a better vulnerability detection method that allows developers to analyze their services in a more confident manner.

REFERENCES

[1] Acunetix Web Vulnerability Scanner, 2008, http://www.acunetix.com/vulnerability-scanner/

[2] Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H., "Penetration Testing vs Static Code Analysis", 2009, http://eden.dei.uc.pt/~mvieira

[3] Antunes, N., Vieira, M., "Detecting SQL Injection Vulnerabilities in Web Services", Fourth Latin-American Symposium on Dependable Computing (LADC 2009), João Pessoa, Paraíba, Brazil, September 2009.

[4] Chappel, D. A., Jewell, T., "Java Web Services: Using Java in Service-Oriented Architectures", O'Reilly, 2002.

[5] Christey, S., Martin, R., "Vulnerability Type Distributions in CVE", Mitre report, May, 2007.

[6] Curbera, F. et al., "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI", Internet Computing, IEEE, vol. 6, pp. 86-93, 2002.

[7] FindBugs 1.3.8 - http://findbugs.sourceforge.net/

[8] HP WebInspect, 2008, http://www.hp.com

[9] IBM Rational AppScan, 2008, http://www-01.ibm.com/software/awdtools/appscan/

[10] IntelliJ IDEA 8.1 - http://www.jetbrains.com/idea/

[11] Laranjeiro, N., Vieira, M., Madeira, H., "Protecting Database Centric Web Services against SQL/XPath Injection Attacks", 20th International Conference on Database and Expert Systems Applications (DEXA 2009), Linz, Austria, August 2009.

[12] Livshits, V., Lam, M., "Finding security vulnerabilities in java applications with static analysis", 14th USENIX Security Symposium, Baltimore, MD, USA, 2005.

[13] Lyu, M., "Handbook of Software Reliability Engineering". IEEE Computer Society Press, McGraw-Hill, 1996.

[14] Planet Source Code - http://www.planet-source-code.com/

[15] SourceMonitor 2.5 - http://www.campwoodsw.com/sourcemonitor.html

[16] Stuttard, D., Pinto, M., "The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws", Wiley, ISBN-10: 0470170778, 2007.

[17] Transaction Processing Performance Council, "TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.1", 2005.

[18] Vieira, M., Antunes, N., Madeira, H., "Using Web Security Scanners to Detect Vulnerabilities in Web Services", Intl. Conf. on Dependable Systems and Networks, Lisbon, 2009.

[19] Yet Another Source Code Analyzer 1.3 - http://www.yasca.org/

[20] Zanero, S., Carettoni, L., Zanchetta, M., "Automatic Detection of Web Application Security Flaws", Black Hat Briefings, 2005.

[21] Wagner, S., Jurjens, J., Koller, C., Trischberger, P., "Comparing bug finding tools with reviews and tests", Lecture Notes in Computer Science, vol. 3502, 2005, pp. 40-55.