IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)
e-ISSN: 2319 – 4200, p-ISSN No. : 2319 – 4197
PP 13-23
www.iosrjournals.org

Assessing and Comparing Vulnerability Detection In Web Services

Gowri.V.Nair[1], R.Sahila Devi[2]
1(Computer Science Engineering) in Rohini College of Engineering and Technology, Anna University.
[2] Computer Science Engineering Department, Rohini College of Engineering and Technology,Anna University.

Abstract: *Selecting a vulnerability detection tool is a key problem that is frequently faced by developers of security-critical web services. Research and practice shows that state-of-the-art tools present low effectiveness both in terms of vulnerability coverage and false positive rates. The main problem is that such tools are typically limited in the detection approaches implemented, and are designed for being applied in very concrete scenarios. Thus, using the wrong tool may lead to the deployment of services with undetected vulnerabilities. This paper proposes a benchmarking approach to assess and compare the effectiveness of vulnerability detection tools in web services environments using WSDL. This approach was used to define two concrete benchmarks for SQL Injection vulnerability detection tools. The first is based on a predefined set of web services, and the second allows the benchmark user to specify the workload that best portrays the specific characteristics of his environment. The two benchmarks are used to assess and compare several widely used tools, including four penetration testers, three static code analyzers, and one anomaly detector. Results show that the benchmarks accurately portray the effectiveness of vulnerability detection tools (in a relative manner) and suggest that the proposed benchmarking approach can be applied in the field.*

Index Terms: *Benchmarking, vulnerability detection, penetration testing, static analysis, and runtime anomaly detection,WSDL*

I. INTRODUCTION

Web services (WS) are nowadays widely used to sup-port many enterprise systems, linking suppliers and clients in sectors such as banking, transportation, and manufacturing, just to name a few [1]. Web services are a key element in service oriented architectures (SOA) and consist of standard-based self-describing components that can be used by other software across the web in a platform-independent manner. This makes web services the lingua franca for systems integration.

The security of web applications is, in general, quite poor [2]. Web services are no exception and are frequently deployed with code vulnerabilities (as shown in [3], [4]). This is confirmed by the field study presented in [5], which describes an experimental evaluation of the security vulner-abilities in 300 publicly available web services. Four well-known vulnerability scanners have been used to identify security flaws in the services implementations and a large number of vulnerabilities has been observed (25 of the tested services presented some type of security vulnerability that could be exploited), confirming that many services (more than 8 percent) are deployed without proper security testing. A key observation was that injection vulnerabilities are particularly frequent [2]. These consist of improperly coded applications that allow the attacker to inject and execute commands in the vulnerable service, enabling, for instance, access to critical data. Vulnerabilities allowing SQL Injection and XPath Injection are especially relevant, as web services frequently use a data persistence solution sup-ported by a relational database [6] or a XML solution [7].

Web services are so widely exposed that any security vulnerability will most probably be uncovered and exploited by hackers. This way, to prevent vulnerabilities, developers should apply coding best practices, perform security reviews of the code, use static code analyzers, exe-cute penetration tests, etc. [8]. However, most times, devel-opers focus on the implementation of functionalities to satisfy the user's requirements and the time-to-market con-straints, thus disregarding security aspects. In this context, vulnerability detection tools provide an easy and low cost way to test web services for vulnerabilities.

Vulnerability detection tools are widely used by web services' developers to support automated security check-ing and comprise some of the best examples of critical tools for secure software development. Different techniques for vulnerabilities detection have been proposed in the past [8], including penetration testing and static code analysis, which are the two most used ones. Due to time constraints or resource limitations, developers frequently have to select a specific tool from the large set of tools available (usually without really knowing how good each tool is) and strongly rely on that tool to detect potential security problems in the code being developed.

Previous work shows that the effectiveness of many of these tools is quite low [5], [9], [10]. In fact, the low coverage and the high number of false positives frequently observed highlight the limitations of many vulnerability detection tools. Furthermore, it is clear that the performance of a given tool strongly depends on the specificities of the application scenario (i.e., the class of target web services (e.g., SOAP, REST), the types of vulnerabilities to detect, etc.), and that the same tool may have different performance levels in different scenarios. Although some studies focused on the evaluation bug detection tools [11], [12], [13], none has provided a systematic way to evaluate and compare vulnerability detection tools. This way, developers urge the definition of a practical approach that helps them assessing and comparing alternative tools concerning their ability to detect vulnerabilities.

This paper proposes an approach for benchmarking vul-nerability detection tools for web services. This approach specifies all the components and steps needed to define benchmarks to assess and compare alternative tools, with particular focus on two metrics: precision (ratio of correctly detected vulnerabilities to the number of all detected vul-nerabilities) and recall (ratio of correctly identified vulner-abilities to the number of all known vulnerabilities). These are proven and well known metrics that are widely used in several domains, although originally proposed for informa-tion retrieval systems [14]. Additionally, it defines the other required components, which include a workload (work that the vulnerability detectors under testing have to do, in the form of a set of web services that should be searched for vul-nerabilities) and a well-defined benchmarking procedure (set of steps that have to be followed for conducting a benchmarking campaign, ranging from the preparation of the experiments to the ranking of the tools). A key aspect is that the proposed approach is generic and can be used to specify different benchmarks for different application domains and types of vulnerabilities.

The benchmarking approach has been used to define two concrete benchmarks. The first targets tools capable of detecting SQL Injection vulnerabilities in SOAP web serv-ices, including detection approaches based on penetration testing, static code analysis, and runtime anomaly detection. This benchmark uses a well defined and large set of web services adapted from standard performance benchmarks, and includes both vulnerable and non-vulnerable versions of the services. The main limitation is that, although based on a well-defined set of rules, it is not protected against "gaming" (i.e., adaptations/tuning that allow producing optimistic or biased results). In fact, as the workload is well known, providers can easily tune their tools to maximum effectiveness in the context of the benchmark, while failing in different scenarios.

To demonstrate an alternative approach, we propose a second benchmark for penetration testing tools capable of detecting SQL Injection vulnerabilities in SOAP web serv-ices. This benchmark circumvents the "gaming" problem by allowing the benchmark user to specify the workload (i.e., the workload is not predefined and is unknown to the tools' providers) that best represents his specific development conditions, thus providing more realistic (and specific to the development environment) results. To support the user in the task of defining the workload, the benchmark includes a procedure and a tool to identify vulnerabilities in the target web services, thus avoiding the need for conducting such analysis manually.

When compared with related benchmarking works, our approach inovattes in the following. Contrarily to benchmarks for bug detection tools, the services used as workload should work correctly from a functional point of view, although containing vulnerabilities that may be exploited by a security attack, which raises difficult chal-lenges when defining workloads. Also, the focus on the web services environment brings the need for the benchmarks to be useful for providers and consumers. This way, the metrics should be easy to understand for both parties and must allow comparing different types of vulnerability detection tools. Due to the diversity of potential users, the procedure should be the most automated possible. Finally, considering that the efficiency of vulnerability detection tools depend on the con-text where they are applied, it is very important to support benchmarks that allow users to define their own workloads, making the results of their campaigns much more useful.

To demonstrate the benchmarking approach and the two concrete benchmarks, several widely used commer-cial and open-source vulnerability detection tools have been benchmarked, including four penetration testers, three static code analyzers, and one anomaly detector. The results allowed us to successfully rank the tools according to several criteria, while fulfilling key proper-ties such as repeatability, portability, representativeness, non-intrusiveness, and simplicity of use. This suggests that the proposed approach can be applied in the field.

In summary, the contributions of this paper are:

- A generic benchmarking approach for vulnerability detection tools for web services. The approach is based on the clear definition of the benchmarking domain (i.e., of the characteristics of the target tools) and defines the components and metrics needed to specify concrete benchmarks. This approach is based on a preliminary proposal presented in [15], which has been detailed and extended to support the defi-nition of benchmarks based on user-specific work-loads (the original version considered only the use of predefined workloads).

- A concrete benchmark for penetration testing, static code analysis, and runtime anomaly detection tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This benchmark is based on a set of predefined web services and has been used to assess and rank a set of tools, including four penetra-tion testers, three static code analyzers, and one anomaly detector. This benchmark is based on the preliminary proposal in [15].
- A new benchmark (not present before) for penetra-tion testing tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This bench-mark allows the user to define the workload (thus preventing "gaming") and includes a tool for identi-fying the existing vulnerabilities in such workload. The benchmark was used to compare four penetra-tion testers.

The outline of this paper is as follows. Section 2 presents background and related work. Section 3 dis-cusses the benchmarking approach. Section 4 introduces the tools used to demonstrate the concrete benchmarks. Section 5 presents the benchmark based on the predefined workload and Section 6 discusses the benchmark based on the user-defined workload. Both include the experi-mental evaluation and the benchmark properties discus-sion. Section 7 concludes the paper.

II. BACKGROUND AND RELATED WORK

Published studies show that, in general, web applications present dangerous security flaws. For example, the NTA's Annual Security Report 2008 [16] states that 25 percent of the companies tested presented one or more high-risk vulner-abilities. This number is lower than the 32 percent reported in 2007 [17]. Nevertheless, the overall vulnerabilities found increased in some critical sectors (finance, government, legal, retail and utilities). The NTA's Annual Web Application Security Report 2011 [18], focused on web applications, states that 8 percent of the applications tested contained at least one high-risk vulnerability and that 26 percent of them contained medium risk vulnerabilities. These results cannot be general-ized to web services, but show a high number of software applications being deployed without proper security cau-tions, including web applications.

In a web services environment, the vulnerability distri-bution might be slightly different from typical web sites, but web services are also frequently deployed containing security vulnerabilities [3], [4]. The study presented in [5] shows that numerous public web services have some type of vulnerability that can be exploited, confirming that many are deployed without proper security testing. To mitigate this, developers should use appropriate tools to detect vulnerabilities. The problem is that even state-of-the-art detectors frequently present low effectiveness both in terms of vulnerability detection coverage (ratio between the number of vulnerabilities detected and the total num-ber of existing vulnerabilities) and false positives (ratio between the number of true vulnerabilities detected and the total number of vulnerabilities reported) [5], [9], [10]. This way, benchmarking approaches that allow develop-ers to select the most effective tools for each particular sce-nario are of utmost importance.

VULNERABILITY DETECTION

Penetration testing and static code analysis are two well-known techniques frequently used by developers to identify security vulnerabilities in web services [8]. Although pene-tration testing is based on the effective execution of the code, vulnerabilities detection consists in the analysis of the responses, which limits the visibility on the internal service behavior. On the other hand, static code analysis is based on the analysis of the source code (or the bytecode in more advanced analyzers), which allows identifying specific code patterns prone to security vulnerabilities. However, it lacks a dynamic view of the service behavior in the presence of a realistic workload.

Penetration testing tools provide an automatic way to test an application for vulnerabilities, based on specifically tampered input values. Previous research shows that the effectiveness of these tools in web services is very poor. For example, the work presented in [5] shows several limita-tions, namely: large differences in the vulnerabilities detected by each tool, low coverage (less than 20 percent for two scanners), and high number of false positives (35 and 40 percent in two cases). These limitations are also con-firmed by the studies presented in [9], [10].

Static code analyzers provide an automatic manner for highlighting possible coding errors without actually executing the software [19]. In [20] authors evaluated three tools and compared their effectiveness with the effectiveness of code reviews. The tools achieved higher efficiency than the reviews in detecting software bugs (the study did not consider security issues in particular) in five Java-based applications, but all the tools presented false positive rates higher than 30 percent. This is also confirmed in [9].

Runtime anomaly detection consists in the search for deviations from an historical profile of valid commands and is an alternative approach for vulnerability detection. For example, in [21] an approach is

proposed that combines penetration testing with anomaly detection for uncovering SQL Injection vulnerabilities. Another example is Analysis and Monitoring for NEutralizing SQL-Injection Attacks (AMNESIA) [22], a tool that combines static analysis and runtime monitoring to detect and avoid SQL injection attacks. The problem is that these approaches are typically based on a learning phase whose completeness is difficult to guarantee, thus the model representing the valid/ expected behavior may be incomplete (guaranteeing com-plete learning is extremely difficult), leading to false posi-tives and undetected vulnerabilities [21], [22].

BENCHMARKING

Computer benchmarks are standard tools that allow eval-uating and comparing different systems or components according to specific characteristics (e.g., performance, dependability, etc.) [23]. The work on performance bench-marking has started long ago [23]. Ranging from simple benchmarks that target very specific hardware systems or components to very complex benchmarks focusing com-plex systems (e.g., database management systems, operat-ing systems), performance benchmarks have contributed to improve successive generations of systems. Research on dependability benchmarking boosted in the beginning of this century [24]. Several works have been done by dif-ferent groups and following different approaches (e.g., experimental, modeling, fault injection) [24]. Finally, work on security benchmarking is a new topic with many open questions [25], [26].

Several studies show the importance of evaluating test-ing techniques using controlled experiments. In fact, in [27] the authors present the difficulties behind creating controlled environments, and introduce an infrastructure for supporting controlled experimentation with software testing and regression testing. An attempt to create a benchmark containing multithreaded bugs is presented [11]. The benchmark uses a data set of applications that was built using students assigned to write buggy multi-threaded Java programs and document those bugs. Undocumented bugs, i.e., introduced unintentionally, were also considered in the data set. Although this study has obvious problems in terms of representativeness, the problems found in the detection tools showed the utility of this kind of benchmarks. Regarding the evaluation of bug detecting approaches, in [12] the authors present "BugBench," a data set consisting of real-life applications with varying sizes and containing bugs. The bugs were manually collected and most of them are related to mem-ory. Conversely, iBUGS [13] is an approach to semiauto-matically extract data sets of real bugs and corresponding tests from the history of a project. The authors demon-strated the approach extracting 369 bugs from the history of AspectJ Project and then used these bugs to evaluate one bug localization tool. However, these studies always focused on classical bugs, which differ from the problem-atic raised by security vulnerabilities.

Although several works have tried to assess the effec-tiveness of vulnerability detection tools (e.g., [5], [9], [10], [20], [28]) none has proposed a generic and standard approach that allows the comparison of results. In fact, existing works compare different tools under very specific conditions, which cannot be generalized or easily repli-cated, thus results are of limited use. A first attempt to define a benchmarking approach for vulnerability detec-tion tools was presented at [15], which included a concrete benchmark for tools able to detect SQL Injection vulner-abilities. The current work extends that approach and pro-poses a new benchmark that addresses the problem of "gaming" of results. Another relevant work is presented in [10]. It proposes a method to evaluate web vulnerability scanners using software fault injection techniques. Soft-ware faults are injected in the application code and the tool under evaluation is executed, showing its strengths and weaknesses concerning coverage of vulnerability detection and false positives. However, this study was focused on a specific family of applications, namely data-base centric web-based applications written in PHP, and the benchmarking approach cannot be generalized and applied to other domains.

2.3 Benchmarking Properties

Computer benchmarking is primarily an experimental approach. As an experiment, its acceptability is largely based on two salient facets of the experimental method: 1) the ability to reproduce the observations and the meas-urements, either on a deterministic or on a statistical basis, and 2) the capability of generalizing the results through some form of inductive reasoning. The first aspect gives confidence in the results and the second makes the bench-mark results meaningful and useful beyond the specific setup used in the benchmarking process. In practice, bench-marking results are normally reproducible in a statistical basis. On the other hand, the necessary generalization of the results is inherently related to the representativeness of the benchmark experiments. The notion of representativeness is manifold and touches almost all the aspects of benchmark-ing, as it really means that the conditions used to obtain the measures are representative of what can be found in the real world.

To achieve acceptance by the computer industry or by the user community a benchmark should fulfill a set of key properties [23]: representativeness, portability, repeatabil-ity, non-intrusiveness, and simplicity of use. These proper-ties must be taken into account from the beginning of the definition of the components and must be validated after the benchmark has been completely defined.

To be credible, a benchmark for vulnerability detection tools must report similar results when run more than once over the same tool. However, repeatability has to be under-stood in statistical terms, as it might be impossible to repro-duce exactly the same conditions concerning the tool and the web services state during the benchmark run. In prac-tice, small deviations in the measurements in successive runs are normal and just reflect the non-deterministic nature of web applications.

Another important property is portability, as a benchmark must allow the comparison of different tools in a given domain. In practice, the workload is the component that has more influence on portability, as it must be able to exercise the vulnerability detection capabilities of a large set of tools in the domain.

In order to report relevant results, a benchmark must represent real world scenarios in a realistic way. In our work, representativeness is mainly influenced by the work-load, which must be based on realistic code and must include a realistic set of vulnerabilities. This can more easily be taken into account in the case of benchmarks based on a predefined workload, as it is possible to address representa-tiveness issues during the benchmark specification. How-ever, this may be a issue in the case of user-defined workloads, as the benchmark user may not be aware of the representativeness issues of the services considered and, consequently, of the results obtained.

A benchmark must require minimum changes (or no changes at all) in the target tools. If the implementation or execution of the benchmark requires changes in the tools (either in the structure or in the behavior) then the bench-mark is intrusive and the results might not be valid.

Finally, to be accepted, a benchmark must be as easy to implement and run as possible. Ideally, the benchmark should be provided in a form ready to be used or, if that is not possible, as a document specifying in detail how the benchmark should be implemented and executed. In addi-tion, the benchmark execution should take the smallest time possible (preferably not more than a few hours per tool). This is obviously easier to achieve in benchmarks based on a predefined workload, as in the case of user-defined work-loads the benchmark user has the added work of defining and characterizing the workload.

III. APPROACH FOR BENCHMARKING VULNERABILITY DETECTION TOOLS

Our proposal to benchmark vulnerability detection tools is inspired on measurement-based techniques. The basic idea is to exercise the tools under benchmarking using web serv-ices code with and without vulnerabilities and, based on the detected vulnerabilities, calculate a small set of measures that portray the tools' detection capabilities.

Due to the high diversity of web services, types of vulner-abilities, and vulnerability detection approaches, the definition of a benchmark for all vulnerability detection tools is an unattainable goal. This way, as recommended in [23], a benchmark must be specifically targeted to a particular domain. In fact, the division of the spectrum into A tool that achieves a precision of 0.7 is able to detected

Table 1 Tool under Benchmarking

Code	Provider	Tool	Technique
VS1	HP	WebInspect	Penetration testing
VS2	IBM	Rational AppScan	
VS3	Acunetix	Web Vuln. Scanner	
VS4	Univ. Coimbra	IPT-WS	
SA1	Univ. Maryland	FindBugs	Static code analysis
SA2	SourceForge	Yasca	
SA3	JetBrains	IntelliJ IDEA	
RAD	Univ. Coimbra	CIVS-WS	Anomaly detection

b. Ranking and selection. Rank the tools using F-Measure, precision, and recall. Select the most effective tool (or tools) using the preferred ranking.

In the case of benchmarks based on a predefined work-load Step 1.a is not required, as the target web services are characterized in the benchmark specification (including the number of existing vulnerabilities). On the other hand, for benchmarks based on a user-defined workload Step 1.a is extremely relevant, as it greatly influences the benchmark results (e.g., if the workload does not contain representative vulnerabilities then the measures will not be representative of the tools effectiveness).

The benchmark execution is a straightforward process and consists of using each tool to detect vulnerabilities in the workload code. Depending on the tool under bench-marking this may require some configuration of the tool parameters. After executing the benchmark it is necessary to compare the vulnerabilities detected by the tool with the ones that effectively exist in the workload code. Vulnerabil-ities

correctly detected are counted as true positives and vulnerabilities detected but that do not exist in the code are counted as false positives. This is the information needed to calculate the precision and recall of the tool, and conse-quently the F-measure.

IV. CASE STUDY

To demonstrate the proposed approach we designed two different benchmarks:

- VDBenchWS-pd. Benchmark based on a predefined workload, targeting vulnerability detectors based on penetration testing, static analysis, and runtime anomaly detection, able to detect SQL Injection in SOAP web services. This benchmark is presented in Section 5.

- PTBenchWS-ud. Benchmark based on a user-defined workload, targeting penetration testing tools, able to detect SQL Injection vulnerabilities in SOAP web services. This benchmark is presented in Section 6. These benchmarks have been used to assess and compare a set of vulnerability detection tools, which are summarized in Table 1 (the code is used to refer the tools during experi-mental evaluation and results description). As shown, four penetration testing tools have been benchmarked, including three well-known commercial tools. The last penetration tester considered implements the approach proposed in [29]. An important aspect is that, when allowed by the test-ing tool, information about the domain of each input param-eter was provided. If the tool requires an exemplar invocation per operation, the exemplar respected the input domains. All the tools in this situation used the same exem-plar to guarantee a fairness.

Three vastly used static code analyzers able to detect vul-nerabilities in Java applications' source or bytecode have also been considered in this study, namely: FindBugs, Yasca, and IntelliJ IDEA. During the experiments the static analyzers were configured to fully analyze the services code. For the analyzers that use binary code, the deploy-ment ready version was used.

The last tool, named Command Injection Vulnerability Scanner for Web Services (CIVS-WS) [21], combines runtime anomaly detection with penetration testing for uncovering SQL Injection vulnerabilities in web services.

V. EXAMPLE 1: BENCHMARK BASED ON A PREDEFINED WORKLOAD (VDBENCHWS-PD)

In this section we present a benchmark (VDBenchWS-pd) targeting the following domain:

- Class of web services. SOAP web services implemented in Java, which are nowadays widely used for data exchange and systems integration [30].

- Type of vulnerabilities. SQL Injection. Vulnerabilities allowing SQL Injection are particularly relevant in web services [2], as these frequently use a data per-sistence solution based in a relational database.

- Detection approaches. penetration testing, static code analysis, and runtime anomaly detection [31], [32], [33].

WORKLOAD DEFINITION

As mentioned before, the workload is the component most influenced by the benchmarking domain and strongly determines the benchmark results. In order to define a rep-resentative workload we have decided to adapt code from three standard benchmarks developed by the Transactions processing Performance Council, namely: TPC-App, TPC-C, and TPC-W (see details http://www.tpc.org). TPC-App is a performance benchmark for web services infrastruc-tures and specifies a set of web services accepted as repre-sentative of real environments. TPC-C is a performance benchmark for transactional systems and specifies a set of transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at warehouses. Finally, TPC-W is a benchmark for web-based transactional systems. The business is a retail store over the Internet where several clients access the website to browse, search, and process orders.[1]

As an adaptation of real applications the proposed work-load follows the realistic workloads approach, and thus needs to include realistic SQL Injection vulnerabilities. Although feasible, artificial vulnerabilities injection [10] would 1. Although TPC-C and TPC-W do not define the transactions in the form of services, they can easily be implemented as such. tools effectiveness from the point-of-view of the service they provide (i.e., vulnerabilities reported) and not based on the internal behavior.

The use of the proposed benchmark is quite simple to use (in part, because most steps are automatic). In fact, we have been able to run the benchmark for all the tools in about six man-days, which correspond to an

average of 0.75 man-days per benchmarking experiment. Running the bench-mark only requires executing the tools and comparing the reported vulnerabilities with the ones that effectively exist. As different tools report vulnerabilities in different formats (e.g., XML file, text file, GUI), to automate the vulnerability comparison step, we need to convert the output of the tools to a common format. Although possible, we decided not to do it in this work (it is just a technical issue with no scientific relevance).

VI. EXAMPLE 2: BENCHMARK BASED ON A USER-DEFINED WORKLOAD (PTBENCHWS-UD)

In this section we present a benchmark targeting the follow-ing domain:

- Class of web services. SOAP web services [30].
- Type of vulnerabilities. SQL Injection [2].
- Detection approaches. penetration testing [31].

6.1 Workload Definition and Characterization

The set of web services that will compose the benchmark workload is to be defined by the benchmark user. This should include a number of SOAP web services with and without SQL Injection vulnerabilities. As defined in Section 3.2, this workload can be real, realistic, or synthetic. What is important is to understand that the workload defi-nition determines the benchmark results and properties, thus the user should be aware of the impact of the decisions regarding the web services being considered.

A key aspect is the characterization of the existing vulnerabilities. As the target of the benchmark are pene-tration testing tools, the number of vulnerable inputs is needed to later calculate the metrics. Such characteriza-tion can be based on an extensive manual analysis of the selected web services in order to identify the existing vulnerabilities (in a similar way to what we did in Sec-tion 5). The problem is that such process can become extremely expensive if the set of services is large and complex. Thus, as an alternative, we propose an auto-matic approach for identifying the base set of vulnerabilities, grounded on the use of a tool that combines attack sig-natures and interface monitoring to detect SQL Injection vulnerabilities in web services [37]. Although the pro-posed approach does not guarantee the detection of all existing vulnerabilities, it assures that no false positives are reported. The vulnerabilities detected will serve as reference to estimate the number of true and false posi-tives of the tools under benchmarking, as discussed in the next sections.

A key aspect is that the proposed benchmark can be easily extended to other types of injection vulnerabilities. The only constraint is that the benchmark user has to define a workload containing other types of vulnerabilities

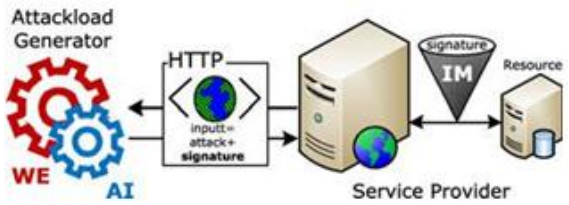


Fig. 3 Simplified representation of the Sign-WS detection tool.

and then manually characterize those vulnerabilities. In fact, although the technique presented next can be easily extended to other Injection vulnerabilities (see [37]), in this work we are targeting only SQL Injection.

6.1.1 Vulnerabilities Identification

For the vulnerability identification, the benchmark includes the Sign-WS tool, which implements the technique pro-posed in [37]. This technique addresses the limitations of penetration testing by using attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. The goal is to improve the detection process by providing enhanced visibility, yet without needing to access or modify the code of the target service. The key assumption is that most injection attacks manifest, in some way, in the interfaces between the attacked web service and other resources (e.g., database, operating system) and serv-ices. For example, a successful SQL Injection attack leads the service to send malicious SQL queries to the database. Thus, it can be observed in the SQL interface between the service and the database server.

Comparing to traditional penetration testing, this approach allows achieving higher effectiveness, as it pro-vides the information needed to increase the number of vul-nerabilities detected and completely eliminate the false positives. Still, the application is tested as a black-box as the only requirement is to monitor is the interface between the application and the used resources as allowed in this spe-cific scenario. A workload emulator module analyzes the web service description and generates a set of valid requests, which are

afterwards modified by the attack injec-tor module. During this process, the interfaces are moni-tored to detect the signatures that represent vulnerabilities. Fig. 3 portrays the approach.

BENCHMARK METRICS ESTIMATION

The signatures and monitoring approach provides informa-tion that is not available to the penetration testing tools under benchmarking, thus it is expected to detect more vul-nerabilities and present less false positives. In fact, and based on the precise detection of signatures, no false posi-tives are expected (see [37]). Thus, the vulnerabilities identi-fied using interface monitoring can be used as a baseline for evaluating other tools, as explained next.

The detection coverage is the percentage of real vulner-abilities that are detected by a tool. Assuming that the number of vulnerabilities reported by the signatures and monitoring approach is a valid estimation of the total num-ber of existing vulnerabilities, then the percentage of those vulnerabilities that are reported by a given penetration testing tool is also a valid estimation for its vulnerability

Table 8 Coverage and False Positives for the Example

Tool	Estimated Coverage Rate	Estimated False Positive Rate
PTA	8 / 10 = 80%	6 / (8 + 6) ~= 43%
PTB	4 / 10 = 40%	1 / (4 + 1) = 20%

detection coverage. Similarly, we can estimate the false positives rate, which represents the percentage of vulner-abilities reported by the tool that in fact do not exist. Con-sidering that the set of vulnerabilities detected using our approach does not include false positives (guaranteed by the adequate signatures), we can estimate the false posi-tives rate of a penetration tester by calculating the differ-ence between the vulnerabilities reported by such tool and the ones identified via interface monitoring.

To better understand our proposal, let's consider a simple scenario. Consider that the signatures system is able to detect 10 SQL Injection vulnerabilities in a given web service and that a penetration testing tool A detects eight of those and six more, and that a penetration testing tool B detects four of those and one more. As shown in Table 8 we can use these values to estimate the coverage and false positives of both tools. Note that, the considered total number of vulnerabilities is only an estimated value, as there is no guarantee of perfect detection coverage from the signatures system. This way, the total number of vulnerabilities will be always equal or superior to the esti-mated number of vulnerabilities and this fact can dimin-ish the importance of the evaluation in two ways.

First, the coverage rates calculated for the evaluated tools may be overestimated. Although this seems a key problem, it is important to stress that the evaluation of the different tools is done for benchmarking purposes (e.g., to select one) and not for assessing actual effectiveness (as this depends on several factors, including the target application, pro-graming language, type of vulnerability, etc.). Thus, taking a relative perspective of the results (rather than an absolute perspective), the overestimation should be equivalent for all the evaluated tools, affecting them similarly, while main-taining a fair comparison.

Second, the false positive rates for the evaluated penetra-tion testers may also be overestimated. Again, although this seems a major issue, in practice the impact will be minor, as the Sign-WS tool uses extra information on the internal behavior of the web services, provided by the signatures and the interface monitoring, to achieve much higher detec-tion coverage than the penetration testing tools, which are the target of this benchmark. This way, it is highly probable that a vulnerability detected by a tester will also be detected by Sign-WS. In fact, during our experiments only one vul-nerability was detected by a penetration testing tool and not detected by Sign-WS, representing less than 1 percent of the total vulnerabilities considered. Thus, the estimation for the false positives should be close to the real values, which again is adequate for a relative view of the results. Finally, it is important to remember that the dependence on the Sign-WS tool can be avoided if the benchmark user has the resources to perform a manual characterization of the workload.

Table 9 PTBenchWS-ud Benchmarking Results

Tool	F-Measure	Precision	Recall
VS1	0.437	0.446	0.427
VS2	0.353	0.388	0.325
VS3	0.050	1.000	0.026
VS4	0.413	0.567	0.325

EXPERIMENTAL EVALUATION

To demonstrate the benchmark we considered the set of web services included in the benchmark proposed in Section 5 (this will allow to compare the results of both benchmarking campaigns). However, we assume no knowledge about the existing vulnerabilities. This way, to characterize the workload (Phase 1. Preparation) we used the attack signatures and interface monitoring approach. The penetration testing tools under benchmarking (pre-sented in Table 1) were run over the workload code (Phase 2. Execution). The vulnerabilities reported were manually confirmed and compared with the ones identi-fied in the preparation phase to calculate the benchmark metrics and rank the tools (Phase 3. Comparison).

CHARACTERIZATION OF THE WORKLOAD

The vulnerabilities detected by the Sign-WS tool have been manually confirmed to guarantee the absence of false posi-tives (as claimed in [37]). The tool indeed reported 0 false positives, but the coverage was only of 74.05 percent (117 true positives out of 158 true vulnerabilities). As we will show later, although not all the true vulnerabilities are con-sidered in the calculation of the metrics, the ones reported by Sign-WS are enough for a good estimation of the tools effectiveness.

BENCHMARKING RESULTS

Table 9 presents the benchmark metrics for each tool, con-sidering as base set the 117 vulnerabilities reported by the Sign-WS tool. As we can see, VS1 is the tool with the highest F-Measure, closely followed by VS4. VS2 presents very poor F-Measure results. Regarding precision, VS3 is the best as it reported no false positives, and VS4 presents the best results. Finally, in terms of recall, VS1 has the best results, while VS2 and VS4 perform equally. The recall of VS3 is very low as it detected only three vulnerabilities.

The results presented in Table 9 were used to rank the tools according to the different criteria: F-Measure, Preci-sion, and recall. Table 10 shows the proposed rank.

Fig. 4 shows the vulnerabilities reported by the pene-tration testing tools (the last bar in the graph presents the number of vulnerabilities detected by the Sign-WS tool). A key observation is that all the tools detected less than 43 percent of the vulnerabilities reported by Sign-WS, which makes the base set of vulnerabilities a good refer-ence. A key aspect is that VS1 reported a true vulnerabil-ity that was not reported by Sign-WS, but this was the only case. We will discuss later the impact of this in the metrics calculation when compared to the benchmark based on a predefined workload.

Table 10 PTBenchWS-ud Tools Ranking

Criteria	1st	2nd	3rd	4th
F-Measure	VS1	VS4	VS2	VS3
Precision	VS3	VS4	VS1	VS2
Recall	VS1	VS2/VS4		VS3

Table 11 Results for Both Benchmarks

	Tool	F-Measure	Precision	Recall
VDBenchWS-pd	VS1	0.378	0.455	0.323
	VS2	0.297	0.388	0.241
	VS3	0.037	1	0.019
	VS4	0.338	0.567	0.241
PTBenchWS-ud	VS1	0.437	0.446	0.427
	VS2	0.353	0.388	0.325
	VS3	0.050	1.000	0.026
	VS4	0.413	0.567	0.325

COMPARISON WITH THE VDBENCHWS-PD BENCHMARK

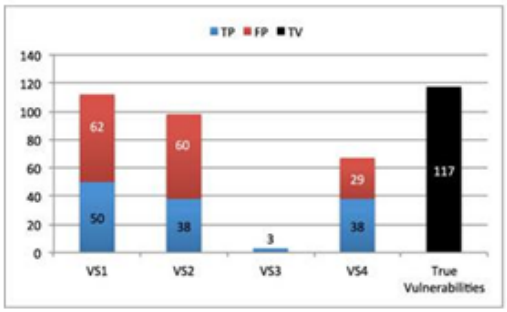
A key aspect is to compare the results of the present bench-mark with the ones of the benchmark based on a predefined workload. Note that, although we are considering the same set of web services, in the benchmark based on a user-defined workload we consider only a subset of the existing vulnerabilities (has reported by the Sign-WS tool). This is obviously also a way for validating the workload characteri-zation and metrics estimation approaches proposed to sup-port the benchmark.

Table 11 summarizes the metrics obtained for both benchmarks (it is a marge of Tables 3 and 9 for the case of the penetration testing tools). As expected, the metrics differ slightly because the base set of true vulnerabilities is differ-ent in the two cases. The F-Measure values are consistently lower in VDBenchWS-pd. This is due to the higher values for recall in PTBenchWS-ud, which are related to the lower number of true

vulnerabilities considered as reference. Finally, precision is the same in both benchmarks, except for the case of VS1. This is due to the fact that VS1 detected a vulnerability that was not reported by the Sign-WS tool, and thus was not included in the base set of true vulnerabil-ities. This obviously harms the reported tool precision, but as the coverage of the Sign-WS is very high, the impact is minimum. In fact, it does not affect the relative results and the tools' ranking is precisely the same for both benchmarks (see Tables 6 and 10).

PROPERTIES DISCUSSION

The representativeness of the benchmark depends on work-load defined by the user. In fact, although leaving to the user the responsibility for defining the workload allows obtaining environment-specific results and prevents "gamming," it may also affect the validity of the results if the web services and vulnerabilities in the workload are not representative of real scenarios. Obviously, in the case of the experimental evaluation presented in the previous section, the representativeness issues are as discussed in Section 5.3. The ranking obtained (equal to the benchmark



Tool	% TP
VS1	42.74%
VS2	32.48%
VS3	2.56%
VS4	32.48%

Tool	% FP
VS1	55.36%
VS2	61.22%
VS3	0.00%
VS4	43.28%

Fig. 4 PTBenchWS-ud results for the penetration testing.

presented in Section 5) suggests that the procedure and the approaches for characterizing the workload and estimating the metrics are quite adequate for characterizing the tools under assessment even when there is no previous knowl-edge about the existing vulnerabilities.

Regarding portability, the benchmark seems to be quite portable in the specified domain. In fact, we were able to benchmark four penetration testers, from different vendors and having diverse functional characteristics.

In terms of repeatability we executed the benchmark for VS1 (penetration tester with the highest F-Measure) two more times. Small variations where observed, but they were always under 0.01, which suggests that the benchmark is quite repeatable. In fact, the repeatability results are similar to the ones discussed in Section 4.3, thus they are not further discussed due to space reasons. The non-intrusiveness property is guaranteed, as the benchmark does not require any changes to the tools.

Although the proposed benchmark is quite simple to use (most steps are automatic), the fact that the user has to pro-vide the workload and characterize the existing vulnerabil-ities, may increase its complexity. Obviously, the approach proposed for the metrics estimation based on the Sign-WS approach makes the work easier. We have been able to run the benchmark for all the tools in about four man-days, which correspond to an average of 1 man-day per experi-ment. In practice, running the benchmark only requires exe-cuting the tools and comparing the reported vulnerabilities with the ones reported by the Sign-WS tool. The problem of different formats in the reports (e.g., XML file, text file, GUI) also applies, and automating the vulnerability comparison step would require converting the output of the tools to a common format.

VII. CONCLUSION

This paper proposed an approach to define benchmarks for vulnerability detection tools in web services. This approach has been used to define two concrete benchmarks targeting tools able to detect SQL Injection vulnerabilities using WSDL. The first benchmark is based on a predefined workload, while the second leaves to the user the responsibility for defining that workload (thus avoiding "gamming" problems). Several tools have been benchmarked, including commercial and open-source tools.

The results show that the proposed benchmarks can be easily used to assess and compare penetration testers, static code analyzers, and anomaly detectors. In fact, the benchmark metrics provided an easy way to rank the tools under benchmarking, leading to similar rankings in in both cases. The properties of the benchmarks were validated and discussed in detail and suggest that the proposed benchmark-ing approach can be applied in the field to specify benchmarks for vulnerability detection tools targeting different domains.

Future work includes extending the benchmarks to other types of vulnerabilities and applying the

benchmarking approach to define benchmarks for other types of web serv-ices. The approach can be extended for other domains as long as it is possible to provide a representative set of code with vulnerabilities (workload). The task of gathering and characterizing the workload may be costly, but in some sce-narios it certainly may be worth the work as it allows us to understand the effectiveness of the different tools available to detect vulnerabilities. Also, we plan to study of vulnera-bility injection techniques for the automatic generation of syntactic workloads is an interesting challenge. Finally, we also plan to provide support for the validation of the results and the maintenance of a benchmark. An idea is to define a core branch of the benchmark and variations to more spe-cific scenarios, and devise new rules to avoid gaming tactics, and to integrate new applications provided by external con-tributors and that we believe may be useful and representa-tive, originating new versions of the benchmark.

REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, Web Services: Concepts, Architectures and Applications. first ed., Springer, 2010.

[2] S. Christey and R.A. Martin, "Vulnerability Type Distributions in CVE," The MITRE Corporation. V1, 1 2007.

[3] L. Lowis and R. Accorsi, "On a Classification Approach for SOA Vulnerabilities," Proc. 33rd Ann. IEEE Int'l Computer Software and Applications Conf. (COMPSAC '09), vol. 2, pp. 439-444, 2009.

[4] M. Jensen, N. Gruschka, R. Herkenhoner, and N. Luttenberger, "SOA and Web Services: New Technologies, New Standards— New Attacks," Proc. Presented at the Fifth European Conf. Web Serv-ices (ECOWS '07), pp. 35-44, 2007.

[5] M. Vieira, N. Antunes, and H. Madeira, "Using Web Security Scanners to Detect Vulnerabilities in Web Services," Proc. IEEE /IFIP Int'l Conf. Dependable Systems Networks (DSN '09), pp. 566-571, 2009.

[6] TPC, "TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.3," http://www.tpc.org/tpc_app/, 2014.

[7] W. Meier, "eXist: An Open Source Native XML Database," Proc. Revised Papers from the NODe 2002 Web and Database- Related Workshops Web, Web-Services, and Database Systems, pp. 169-183, 2003.

[8] J. Fonseca, M. Vieira, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," Proc. Presented at the 13th Pacific Rim Int'l Symp. Depend-able Computing (PRDC '07), pp. 365-372, 2007.

[9] Y. Eytani and S. Ur, "Compiling a Benchmark of Documented Multi-Threaded Bugs," Proc. 18th Int'l Parallel and Distributed Proc-essing Symp., p. 266, 2004.

[10] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for Evaluating Bug Detection Tools," Proc. Workshop the Evaluation of Software Defect Detection Tools, pp. 1-5, 2005.

[11] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localiza-tion Benchmarks from History," Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng., pp. 433-436, 2007.

[12] C.J. Van Rijsbergen, Information Retrieval. Buttersworth, 1979.

[13] N. Antunes and M. Vieira, "Benchmarking Vulnerability Detec-tion Tools for Web Services," Proc. IEEE Int'l Conf. Web Services (ICWS), pp. 203-210, 2010.