Penetration Testing for Web Services

Nuno Antunes and Marco Vieira, ***University of Coimbra, Portugal***

Web services are often deployed with critical software security faults that open them to malicious attack. Penetration testing using commercially available automated tools can help avoid such faults, but new analysis of several popular testing tools reveals significant failings in their performance.

Web services commonly provide the strategic vehicle for content distribution, data exchange, and other critical processes within widely adopted service-oriented architectures (SOAs).[1] As with other Web applications, Web services' exposure, in conjunction with improper software coding, can make it relatively easy for hackers to uncover and exploit any security vulnerabilities—often by entering into input fields values that have been specially tampered with to search for vulnerabilities, a particularly vicious hacking technique known as command injection (which includes SQL injection). Such attacks introduce and execute commands that allow hackers to read, modify, and even destroy important information and resources, sometimes corrupting entire databases, thus opening application providers to considerable danger.[2,3]

Application vulnerability and security issues must be kept in mind over the entire software development life cycle—for example, applying best practices during design and implementation stages to avoid potential vulnerabilities, and including environment runtime mechanisms to detect and remove potential vulnerabilities and counter possible attacks during deployment and testing.[4] As a response, various methods for identifying security vulnerabilities have evolved,[5] including static and dynamic analysis, runtime anomaly detection, and penetration testing. The latter, which simulates multiple attempted incursions of malicious values from an attacker's perspective using a black-box approach to reveal specific vulnerabilities, is particularly useful for third-party testing in a Web services environment because it doesn't require that clients and providers have access to source code, as static analysis and runtime anomaly detection do.

Penetration testing may be undertaken manually; however, using automated tools for this process can save considerable time and money, and a number of such tools are now commercially available. The question, though, is the extent to which these tools, many regarded as state of the art, are truly effective, both in terms of identifying a full range of potential vulnerabilities and of avoiding false-positive alarms. Our analysis of several widely used automated penetration testing tools, which we describe here, suggests that their performance is far from impressive for Web services security testing. Researchers and practitioners need to be aware of these limitations in order to lead the way for devising new tools and techniques to improve the effectiveness of vulnerability detection methodologies and so insure better Web services security in the future.

 Published by the IEEE Computer Society

WEB SERVICES SECURITY

Web services, including both SOAP and RESTful services,[6] are currently the most popular applications for implementing SOAs. A Web service delivers a particular business functionality in a protocol-independent distributed environment through a standardized interface; the fact that it should be reusable allows its consumption for multiple business processes. SOAP Web services are session-less and rely on XML for message formatting, using a Web Services Descriptive Language (WSDL) file to describe the interface as well as the format of the exchanged messages. RESTful Web services reuse the HTTP operations together with a simplified message format and employ a Web Application Descriptive Language (WADL) file for interface description.

Techniques for detecting software vulnerabilities in Web applications are generally classified as white box or black box; alternative techniques combining characteristics of both are called gray box. While most such detection approaches can be applied to Web services as well, it is important to keep in mind several characteristics that distinguish Web services from other Web applications. First, Web services always have a well-defined interface, which eliminates the crawling phase sometimes required to determine an interface for a Web application; this is a plus in terms of efficiency, but for purposes of vulnerability testing and detection, it serves to mask information about internal application problems extracted by testing tools based on service responses during crawling. Second, consumers and users evaluating a third-party service to choose among multiple alternatives are generally not owners and so cannot access Web service internals, a requirement for using some common vulnerability detection techniques. Third, while the interoperability and reduced dependency among services facilitate their replacement or modification without requiring changes in other parts of the system, this interoperability also reduces the time available for vulnerability detection in order to not delay deployment and requires taking into account the effect of potential interactions among services.

White-box testing analyzes software internally for vulnerabilities and can include code inspection, reviews, walkthroughs, and the like. A security inspection, during which a programmer delivers code to peers who meet formally and examine it systematically for security vulnerabilities, offers the most thorough assurance that a piece of software has minimal vulnerabilities, but it is also time consuming and often expensive. Less expensive are code reviews (almost as thorough as code inspections but not requiring formal meetings) and code walkthroughs (a more informal and less thorough process of analyzing code manually by following code paths based on predefined input conditions).[7] Automated white-box testing tools, such as static analyzers that vet software code to identify implementation-level faults, can reduce the time and costs associated with manual code inspection. However, exhaustive source code analysis may be difficult; moreover, automated white-box testing often fails to identify many security flaws due to code complexity and the absence of runtime (dynamic) view.

Black-box testing, a widely used technique for analyzing programs at runtime from an external point of view, consists of using inputs to execute the software and comparing the execution outcome with the expected result; it can range from unit testing to integration and system testing. Under best practices, test specifications should define the coverage criteria and be elaborated before development; designing tests based on specifications in advance of writing code can avoid any bias that may develop after the code is written.

Because the number of such tests can reach hundreds or even thousands for each vulnerability type, performing penetration testing manually is repetitive and quite expensive.

Gray-box testing most often takes the form of dynamic program analysis, which consists of analyzing software behavior in the process of executing it—that is, attempting to identify faults and potential vulnerabilities through examination of the internal code (similar to static analysis) as it functions in the presence of realistic inputs (runtime analysis). Obviously, the program must be executed with sufficient test inputs; code coverage analyzers help guarantee adequate coverage of the source code.

For Web applications, the black-box method of penetration testing is commonly used as a means to identify potential security vulnerabilities, analyzing program execution in the presence of malicious inputs. Penetration testing requires no knowledge of specific implementation details; fuzzing techniques provide data input from a malicious user's point of view.[5] Because the number of such tests can reach hundreds or even thousands for each vulnerability type, performing penetration testing manually is not only repetitive and tedious but can also be quite expensive. Consequently, more cost-efficient automated penetration testing tools have been developed (although even with these, it isn't feasible to test all potential input streams; test cases must be designed to provide a broadly representative testing environment once software specifications are complete).

Web security scanners, sometimes called Web vulnerability scanners, are the automated tool of choice for Web application penetration testing, but only some of these scanners support testing Web services. Among these,

Table 1. Overall number of vulnerabilities detected by an expert security team and four representative scanning tools.

Vulnerability detection method	SQL injection	XPath injection
Expert team	201	4
VS1	62	2
VS2	42	0
VS3	6	0
VS4	47	1

the few that are free, such as WSFuzzer and WSDigger, simply automate the attacking process and log pertinent responses, requiring security expertise on the user's part as well as a great deal of time in order to examine all the results and manually identify vulnerabilities in the Web service being tested; we have not considered these for the purposes of this study. The three leading commercial Web security scanners that support Web services testing are HP WebInspect, IBM Rational AppScan, and Acunetix Web Vulnerability Scanner, all of which claim to scan for, identify, and assess Web application hacking vulnerabilities.

The problem is that, in practice, for purposes of vulnerability identification, penetration testing tools must rely on analysis of the Web application output; lack of access to the application's internal behavior limits their effectiveness, as previous work evaluating penetration testing tools has confirmed. In one study, Web security scanners performed poorly when they analyzed complex websites that had large amounts of client-side navigation code and form-based wizards.[8] A later study evaluated 11 security scanners and reached similar conclusions, finding that none of the scanners was able to detect application-specific vulnerabilities.[9] It is important to point out, however, that these studies were performed in the context of Web applications, which, as noted earlier, have characteristics different from those of Web services. This highlights the need for studies aimed at understanding the effectiveness of these tools in Web service-based environments.

EXPERIMENTAL STUDY: EFFECTIVENESS OF PENETRATION TESTING

To better understand the effectiveness of penetration testing in a Web services environment, we conducted an experimental study using four Web security scanners to detect vulnerabilities in a defined set of services. Three of the scanners we used—HP WebInspect, IBM Rational AppScan, and Acunetix Web Vulnerability Scanner—are widely available and considered to provide state-of-the-art support for Web services. The fourth scanning tool we used for our study is an academic prototype that implements an approach we have proposed elsewhere.[10] For purposes of this discussion, the specific brands have been masked to assure neutrality and conform with licensing constraints, so we refer to the four tools as VS1, VS2, VS3, and VS4, with no particular assigned order. After some initial configurations, each of these automated tools is able to read the description file of a Web service, test it for vulnerabilities, and at the end of the testing process generate a file reporting the vulnerabilities found (if any).

Services used

For this study, we tested a set of 25 Web services with a total of 101 operations. The greater part of these services (20) was adapted from the implementations of three standard benchmarks developed by the Transaction Processing Performance Council (www.tpc.org), namely, TPC-App, TPC-C, and TPC-W. These performance benchmarks cover Web services infrastructures, transactional systems, and e-commerce. (Although TPC-C and TCP-W do not define transactions in the form of Web services, they can easily be implemented and deployed as such.) The other five services have been adapted from code publicly available on the Internet (www.planet-source-code.com). All of the services are implemented in Java and use a relational database to store data and SQL commands for data management—except one, which uses an XML data source. These services comprise more than 15 KLOC (thousands of lines of code), with an average cyclomatic complexity of 9, and were developed by independent programmers with no knowledge of our particular security study.

Performing a complete evaluation such as we proposed required knowing in advance the existing vulnerabilities in each of the services tested. For this purpose, we enlisted a security team comprised of four people with different backgrounds and experience in secure development practices. This team reviewed the services' source code looking for vulnerabilities, while crosschecking to eliminate false positives. Their analysis determined that the 24 services (98 operations) using a relational database contain 201 SQL injection vulnerabilities, while the other service (three operations) has four XPath injection vulnerabilities. (These results suggest that the service developers did not pay much attention to security.)

This information provided the basis for a deep analysis of the effectiveness of the scanners. First, we performed a false-positives analysis to determine which vulnerabilities

detected and reported by the scanning tools do not, in fact, exist. We then conducted a detection coverage analysis to determine which of the vulnerabilities we knew to exist in the services were, in fact, detected by the tested tools. Although our study was limited to two types of vulnerabilities, we consider the data used sufficient for showing the potential effectiveness of the four scanning tools, as we will discuss.

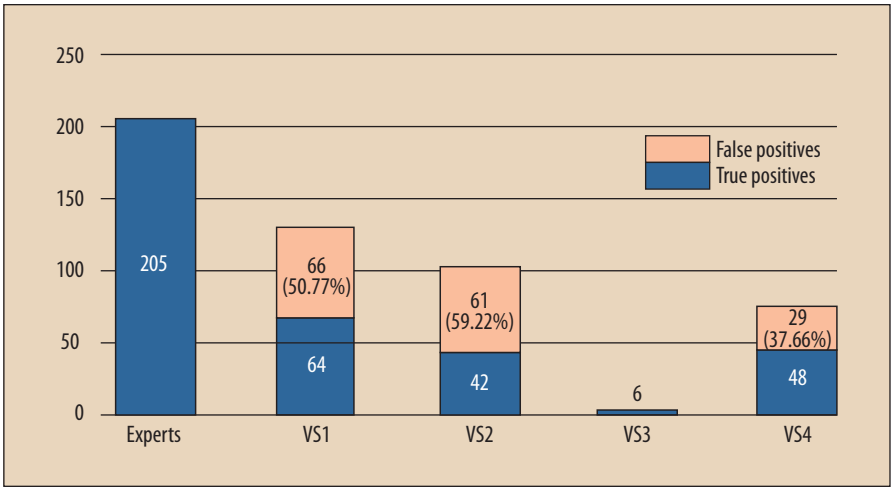

Figure 1. Number of false-positive vulnerabilities reported by Web security scanning tools compared with vulnerabilities reported by a team of security experts.

Overall results

Table 1 presents the overall results of our study, showing the total number of vulnerabilities reported by each tool. As can be observed, each of the different penetration testing tools reported a different number of vulnerabilities. This is a first indicator that tools implement different forms of penetration testing and that the outputs from different tools may be difficult to compare. Another observation worth noting is that all four tools detected SQL injection vulnerabilities, but only two (VS1 and VS4) reported XPath injection issues. Although the actual number of XPath-related vulnerabilities is quite small in comparison to SQL injection, this discrepancy is equally true in real-world scenarios because Web services more often use a traditional database instead of XML solutions for storing information (as is the case in other Web applications). This fact may also explain why some scanning tools apparently do not include features to detect XPath vulnerabilities.

A key aspect of our analysis not reported in the table, but still worth mentioning: different tools reported different vulnerabilities even in the same Web services. For example, although the number of SQL injection vulnerabilities reported by VS1 is much higher than the number of vulnerabilities detected by the other scanners, the other three actually detected some vulnerabilities not reported by VS1. We will return to this point later.

False-positives analysis

False positives (that is, occurrences when the tools detected and reported vulnerabilities that in reality do not exist) were another focus of our study. For the purposes of this evaluation, we considered the set of vulnerabilities reported by the security inspection team we enlisted as representing the complete set of vulnerabilities for each of the Web services tested. Thus, we consider a vulnerability reported by a tool as a false positive if the team of security experts did not report it and a true positive if it matched one reported by the team. Figure 1 shows the results.

The percentage of reported false positives is very high for scanners VS1 and VS2 and certainly in the high range for VS4: in the case of VS1 and VS2, more than half of the reported vulnerabilities do not exist, and in the case of VS4, false positives account for more than one-third of the vulnerabilities reported. This level of false positives suggests that in many instances, software developers may waste considerable effort "fixing" nonexistent vulnerabilities, reducing overall confidence in the tools. The number of false positives is due to the use of heuristics to detect vulnerabilities by evaluating Web services responses; these heuristics, while facilitating detection of vulnerabilities otherwise undetectable, also cause the scanners to report a large number nonexistent vulnerabilities. It is interesting to note, by the way, that the three XPath injection vulnerabilities reported by VS1 (two) and VS4 (one) all correspond to true positives.

VS3, on the other hand, reported no false-positive alarms. Obviously, one factor contributing to this low number of false positives is the relatively small number of vulnerabilities reported by VS3 overall. This scanner would seem to employ a very conservative detection approach that, although avoiding reports of false positives, also leaves many vulnerabilities undetected.

Coverage analysis

Another key factor in considering the effectiveness of security scanners is their detection coverage rate—that is, the number of vulnerabilities detected compared with the total number of actual vulnerabilities. After discarding false positives, we calculated each scanning tool's detection coverage rate, assuming the vulnerabilities reported by the expert security team as the total number of vulnerabilities. Table 2 presents these results.

In all cases, the coverage rates were quite low. In fact, none of the four scanners was able to detect even

Table 2. Detection coverage rates for penetration testing by four representative security scanners.

Scanner	Coverage rate (%)
VS1	31.22
VS2	20.49
VS3	2.93
VS4	23.41

one- third of the total number of vulnerabilities. VS1 clearly provided the best coverage, but even so detected only about 31 percent of the known vulnerabilities. VS2 and VS4 each detected fewer than a quarter of the vulnerabilities, with VS4 offering a slight edge—a difference that is actually significant considering that VS4 also reported many fewer false positives. VS3 provided an extremely low coverage rate; its conservative detection approach (which, as we have noted, avoids false positives) left 97 percent of the known vulnerabilities undetected. It is worth pointing out that although VS1 had the highest detection coverage, it is also the scanner reporting the most false positives.

Understanding the relationship among the actual vulnerabilities reported by each scanner requires a different type of analysis. Figure 2 shows the overlap among the sets of vulnerabilities reported by each scanning tool (after removing false positives), with the area of its respective circle roughly proportional to the number of vulnerabilities detected by each tool.

Clearly, there were differences among the four tools in terms of the actual vulnerabilities reported, which highlights the difficulty involved in selecting and using a single tool for penetration testing. As Figure 2 shows, for example, 17 of the vulnerabilities were detected only by VS1, and only five of the vulnerabilities were detected by all four scanners—although this low total in terms of overall detection coverage is obviously affected by the low coverage of VS3. VS4 was able to detect all the vulnerabilities detected by VS2 and VS3 plus six more. And VS1, while detecting the most vulnerabilities overall, failed to detect one vulnerability that all the other tools detected—particularly interesting given the very low number of vulnerabilities detected by VS3. (As far as we were able to determine, this specific vulnerability is located in a difficult-to-reach code path that the requests performed by VS1 were unable to exercise.)

Equally interesting, 140 vulnerabilities were undetected by any of the four tools, emphasizing even more strongly the difficulty of selecting one tool to use. Manual analysis of the services reveals that many of those undetected vulnerabilities are located in places of the code hard to reach via black-box testing, and the workloads generated by the tools are not yet complete enough to execute those code paths. There are also situations where an undetected vulnerability is preceded by another very similar one, such that the second can only be detected after the first has been fixed (that is, because the first is exploited, execution never reaches the second).

LESSONS LEARNED

Penetration testing is fundamental for deploying secure code, particularly for consumers testing in Web services environments where internals are inaccessible. But, as previous research and our own study have shown, currently available commercial Web security scanning tools are not entirely satisfactory for purposes of penetration testing of Web applications and Web services.

Essentially, users of automated penetration testing face two problems. First, the very high number of false positives reported by available penetration testing tools reduces developer confidence regarding their precision and also lowers the productivity of development teams who must analyze vulnerabilities that in fact do not exist. Second, the relatively limited vulnerability detection coverage provided by available tools inevitably means that significant numbers of vulnerabilities remain undetected, a major concern for applications with the level of exposure to external environmental factors that Web services have.

These problems result from the intrinsic limitations of penetration testing as a black-box technique: it has no access to the internal behavior of the tested services and can observe an application only from the point of view of an external user. These limitations have two primary ramifications:

- The fact that vulnerability detection is based solely on analysis of a Web service's output leads to a lack of information for decision making. Tool capability is restricted because the amount of information released to the client is insufficient to detect vulnerabilities effectively. Moreover, the fact that application output is often processed to limit (or prohibit) leakage of any system information can make detecting vulnerabilities impossible (even though vulnerabilities exist and testing tools may effectively exploit them).
- The fact that the tool cannot identify all appropriate inputs necessary to maximize the number of Web service code paths that are tested results in inadequate code coverage. Obviously, if some paths of code are not executed during the testing process, any vulnerabilities that occur in these pieces of code will not be detected.

Clearly, innovative approaches are necessary to overcome the present limitations of penetration testing. These must be designed both to improve the quality of the tests

performed and to take maximum advantage of all the interactions with the application available to users. At the same time, any new approach needs to maintain current penetration testing advantages, such as time efficiency and cost-effectiveness.

Improving testing quality will require input values representative of accurate and extensive real-world service utilization in order to achieve high code execution coverage. For Web services, the interface description should provide an accurate definition of each input's domains. While existing XML schemas (XSD) associated with Web services now allow this definition, most of the services deployed do not, in fact, provide such information—that task falls on the development team. Further, in many situations, dependencies exist among the domains of different inputs that cannot be described using only XSD. To address this problem, Nuno Laranjeiro and his colleagues recently proposed an extended domain expression language (EDEL) for Web services that allows the specification of such relations,[11] but as yet no existing tool takes advantage of such information.

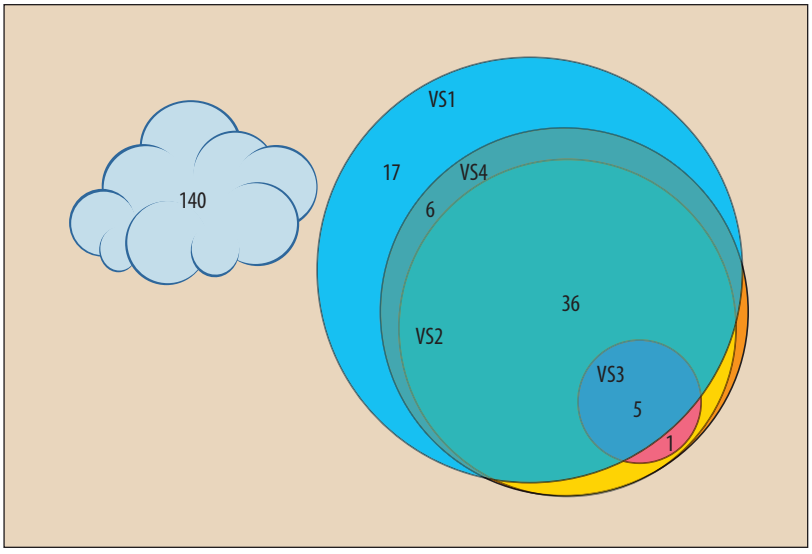

Figure 2. Overlap among the sets of vulnerabilities detected by each scanner. The areas of the circles are proportional to the number of vulnerabilities detected.

Maximizing the effectiveness of vulnerability detection based on a Web service's outputs will require correlating the results of multiple requests. Such correlation makes possible the discovery of vulnerabilities left undetected by analysis of only a single request/response. This correlation may also allow confirmation of vulnerabilities, thus avoiding false-positive alarms. Commercial tools such as Acunetix WVS and research tools such as the one we have proposed[10] already incorporate techniques for this purpose, achieving some promising results; however, as our tests show, they are still far from satisfactory.

Another improvement that can boost future penetration testing effectiveness is relaxation of black-box constraints to increase internal visibility within the tested Web service—but without requiring access to the source code. Such visibility can be achieved via two different techniques, one having a greater degree of intrusiveness than the other. The more intrusive would involve injecting the tested Web service with specially localized probes providing sensitive information about its internal behavior during the penetration testing process; using the less intrusive method, developers could monitor all interfaces between a Web service and appropriate external resources, looking for information that might help to unveil vulnerabilities. The assumption underlying this second method is that the most crucial vulnerabilities manifest in the interfaces between the attacked Web service and components of other systems—the database, OS, gateways, and so forth; for example, an SQL injection might be detected in the interface with the outside database.

Information provided by increased visibility could then be analyzed using techniques such as anomaly detection or attack signatures monitoring. Anomaly detection consists of searching for deviations from a normal, historical profile: in an initial learning phase, a set of normal requests (workload) is submitted to a service to determine its regular behavior; then, in a subsequent attack phase, a set of malicious requests (attackload) is submitted to find deviations that unveil any vulnerabilities. Attack signatures monitoring consists of introducing special tokens into the attacks and then observing the presence of those tokens in specific locations—the assumption being that, when an attack succeeds, the token will be observable somewhere in the monitored parts of the service.

Despite current limitations, penetration testing will continue to play an important role in evaluating Web services security. It is, therefore, important to keep improving these tools, using better workload and attackload generation techniques, and devising new mechanisms to detect vulnerabilities. Equally important is the ability to correlate information provided by different types of requests (regular requests, robustness-testing requests, malicious requests, and others). Furthermore, penetration testing tools should be based on standardized and consistent procedures, implementing a well-defined set of testing components to provide integrated support

for detecting a maximum number of vulnerabilities and a minimal number of false positives. A generic penetration testing tool for Web services that combines all these attributes is a goal for the future.

Finally, security concerns should be a paramount consideration throughout the entire software development process, not just during the testing phase. Applying multiple security best practices at every step to reduce potential security problems will require that developers correctly use approaches and tools already at their disposal, as well as improvements in current techniques and innovative new methods for security assessment.

References

1. D.A. Chappell and T. Jewell, *Java Web Services*, O'Reilly Media, 2002.
2. M. Vieira, N. Antunes, and H. Madeira, "Using Web Security Scanners to Detect Vulnerabilities in Web Services," *Proc. 2009 IEEE/IFIP Int'l Conf. Dependable Systems & Networks* (DSN 09), IEEE, 2009, pp. 566–571.
3. Open Web Application Security Project, *OWASP Top 10—2013,* OWASP Foundation, 2013.
4. M. Howard and D.E. Leblanc, *Writing Secure Code*, 2nd ed., Microsoft Press, 2004.
5. D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, 2nd ed., Wiley, 2011.
6. L. Richardson and S. Ruby, *RESTful Web Services: Web Services for the Real World*, O'Reilly Media, 2007.
7. D.P. Freedman and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, 3rd ed., Dorset House, 2000.
8. M. Curphey and R. Araujo, "Web Application Security Assessment Tools," *IEEE Security & Privacy*, vol. 4, no. 4, 2006, pp. 32–41.
9. A. Doupé, M. Cova, and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners," *Proc. 7th Int'l Conf. Detection of Intrusions and Malware, and Vulnerability Assessment* (DIMVA 10), 2010, Springer, pp. 111–131.
10. N. Antunes and M. Vieira, "Detecting SQL Injection Vulnerabilities in Web Services," *4th Latin-American Symp. Dependable Computing* (LADC 09), Springer, 2009, pp. 17–24.
11. N. Laranjeiro, M. Vieira, and H. Madeira, "Improving Web Services Robustness," *Proc. 2009 IEEE Int'l Conf. Web Services* (ICWS 09), 2009, IEEE, pp. 397–404.

Nuno Antunes *is a PhD student in the Department of Informatics Engineering at the University of Coimbra, Portugal, where he received an MSc in informatics engineering. His research interests include methodologies and tools for developing secure Web applications and services. Antunes is a member of the IEEE Computer Society. Contact him at nmsa@dei.uc.pt.*

Marco Vieira *is an assistant professor in the Department of Informatics Engineering at the University of Coimbra, Portugal. His research interests include dependability and security benchmarking, experimental dependability evaluation, fault injection, software development processes, and software quality assurance. Vieira received a PhD in computer engineering from the University of Coimbra. He is a member of the IEEE Computer Society. Contact him at mvieira@dei.uc.pt.*