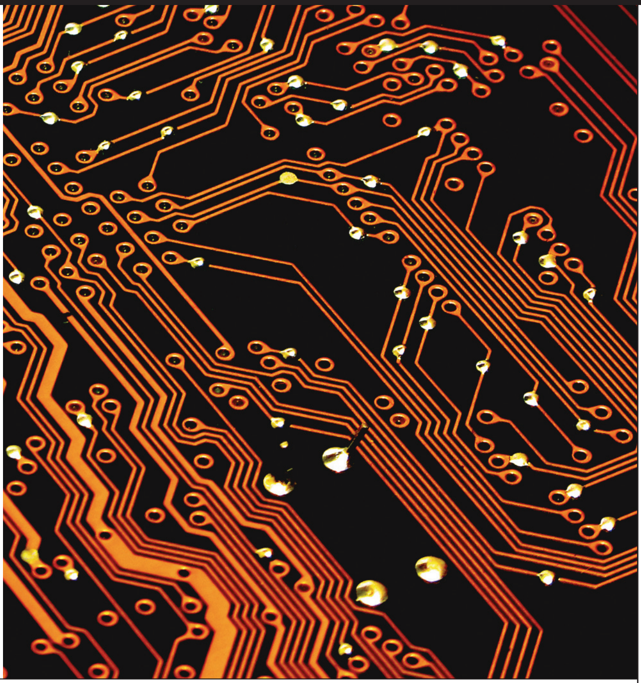
Defending against Web Application Vulnerabilities

Nuno Antunes and Marco Vieira

University of Coimbra, Portugal



Although no single tool or technique can guard against the host of possible attacks, a defense-in-depth approach, with overlapping protections, can help secure Web applications.

Today's Web applications can contain dangerous security flaws. The global distribution of these applications makes them prone to attacks that uncover and maliciously exploit a variety of security vulnerabilities.

The two most common risks in the Web environment, *injection*—namely SQL injection, which lets attackers alter SQL queries sent to a database—and *cross-site scripting* (XSS), are also two of the most dangerous (www.owasp.org/index.php/Category:OWASP_Top_Ten_Project). Injection attacks take advantage of improperly coded applications to insert and execute attacker-specified commands, enabling access to critical data and resources. XSS vulnerabilities exist when an application sends user-supplied data to a Web browser without first validating or encoding that content.

Although a 2009 report from the Open Web Application Security Project (OWASP) indicated that investment in security was increasing (www.owasp.org/index.php/Category:OWASP_Security_Spending_Benchmarks), NTA Monitor's 2010 Web Application Security Report demonstrated that Web security had actually decreased compared to the previous year (www.nta-monitor.com). In fact, Web application vulnerabilities represent huge problems for companies and organizations. According to WhiteHat Security's most recent Website Security Statistics Report, 63 percent of assessed websites are vulnerable, each having an average of six unsolved flaws (www.whitehatsec.com/home/resource/stats.html). These vulnerabilities create and feed an underground economy based on attacking and stealing data and resources.

Web applications need a *defense-in-depth approach* to avoid and mitigate security vulnerabilities.[1] This approach assumes that every security precaution can fail, so security depends on having several layers of mechanisms that cover the failures of each other. To minimize the probability of successful attacks, software engineering teams must apply the effort necessary to introduce adequate security precautions. Achieving this goal is only possible by using various techniques and tools to ensure security in all phases of the software product's development life cycle.

SECURITY IN THE SOFTWARE DEVELOPMENT LIFE CYCLE

Although the software development life cycle can be divided in different ways, as Figure 1 shows, it usually includes the following phases, which application developers can repeat iteratively: initialization, specification and design, implementation (coding), testing, deployment, and decommissioning.[2]

Although developers should address code security concerns during the entire software product development life cycle,[3] they should specifically focus on three key phases:[1]

- *Implementation*. During coding, developers must use best practices that avoid the most critical vulnerabilities in the specific application domain. Example practices include input and output validation, the identification of malicious characters, and the use of parameterized commands.[4] Although these techniques are usually effective in avoiding most Web security vulnerabilities, developers do not always apply them or they apply them incorrectly because they lack security-related knowledge. The "Why Don't Developers Use Secure Coding Practices?" sidebar addresses this issue in more detail.
- *Testing*. Many techniques are available for identifying security vulnerabilities during testing, including penetration testing (by far the most popular technique), static analysis, dynamic analysis, and runtime anomaly detection.[4] The problem is that developers often focus on testing functional requirements and disregard security aspects. Furthermore, existing automated tools usually provide poor results—either low vulnerability detection coverage or too many false positives.
- *Deployment*. At runtime, it is possible to include different attack detection mechanisms in the environment. These mechanisms can operate at different levels and use various detection approaches. Obstacles to their use relate to performance overhead and to the false positives that disrupt normal system behavior.

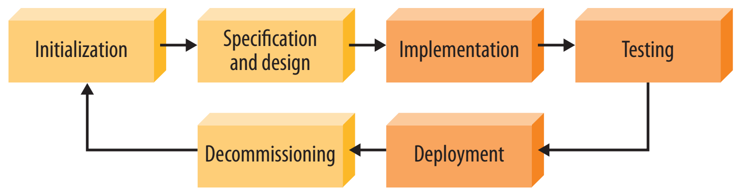


Figure 1. Simplified version of a software development life cycle.

DEVELOPING SECURE CODE

To produce code without vulnerabilities,[4] developers of business-critical Web-based infrastructures should follow coding practices that include a defense-in-depth approach, assuming that each security precaution can fail. Using an approach that depends on several layers of defense mechanisms is extremely important during implementation, as a unique precaution or protection can be insufficient to avoid security vulnerabilities.

The characteristics of Web applications require the use of three distinct lines of defense: input validation, hotspot protection, and output validation.

Input validation

Most security exploits are only possible because the target application incorrectly verifies the input data.[1] Therefore, applications must consider all inputs malicious until proven otherwise, including any data that comes from untrusted environments.

Input validation is a first line of defense that consists of reducing an application's input domain as a whole, acting directly upon the user-provided values. This type of defense relies on forcing the input parameters to be within a given valid domain, or on stopping execution when a user provides a value outside the domain. In Web applications, this should start with normalization of the inputs to a baseline character set and encoding. Then, the application must use filtering strategies on the normalized inputs, rejecting those that contain values outside the valid domain. This practice can avoid many problems in Web applications, which use positive pattern matching or positive validation to perform input validation. In these cases, developers establish input validation routines that identify acceptable inputs, rather than unacceptable ones. Although developers cannot predict every type of attack, they should be able to specify all forms of legal input.

A key issue is that input validation is frequently inadequate because an input parameter's data domain can allow for the existence of malicious data, independently of the validation performed. For example, in the case of SQL injection vulnerabilities, most SQL statements use a quotation mark as a string delimiter, but that means

WHY DON'T DEVELOPERS USE SECURE CODING PRACTICES?

Experience shows that the failure to use secure coding practices relates to training and education: developers cannot prevent security flaws in code if they do not know how. Unfortunately, most computer science programs lack courses about secure design and coding or security testing.

Some developers give less importance to security than to functional requirements because they consider security boring and uninteresting: it does not directly contribute to developing new and exciting applications. In fact, many developers think that someone else should take care of security, like network management staff. But what developers do not know is that their code is the main target of attackers.

Another reason for neglecting security is that developers often see security practices as limiting application functionality. However, while building an application that allows anyone to read data without prior authentication and authorization might enhance usability, it also lets attackers access data. Including more features in an application introduces additional potential security vulnerabilities. In practice, more features usually lead to more hotspots that attackers can exploit with some imagination and patience. Thus, applications that are easy for users to access are also easy for attackers to penetrate.

attackers can use it to perform a SQL injection attack.[4] However, in some cases, a string input's domain must allow the presence of quotation marks, so applications cannot exclude all the values that contain quotation marks.

Hotspot protection

A second line of defense is necessary to mitigate the limitations of input validation.

Each type of attack targets a *hotspot*, a given set of statements in the application's code that is prone to specific types of vulnerabilities. Contrary to generic input validation, in which the application validates or changes inputs in the context of the entire Web application, this secondary defense focuses on protecting only key hotspots, for instance by guaranteeing that the values actually used in these lines lie within their input domain.

Security inspections are a crucial procedure when developing software for critical systems.

A specific example is SQL injection attacks, the majority of which use single and double quotation marks. Some programming languages provide mechanisms for escaping these characters so that they can be used within an SQL expression, but only for delimiting values in the statement.[4] These techniques, however, have two main problems. First, more elaborate injection techniques, such as combining quoting with escaping characters, can circumvent these mechanisms. Second, introducing characters for escaping increases the length of the string and thus can cause data truncation when the resulting string is longer than what the database allows.

Correctly using parameterized commands is the most effective way to avoid injection vulnerabilities.[1] In this case, the developer defines the structure of the commands using placeholders to represent the command's variable values. Later, when the application attaches the respective values to the command, the command interpreter can use them correctly, without interfering with the command's structure.

The most well-known use of this technique is database prepared statements, also called parameterized queries.[4] When an application creates a prepared statement, the statement's structure goes to the database. The application uses placeholders—typically question marks or labels—to mark the query's variable parts. Afterward, each time the query executes, the application must bind values to the corresponding variable part. No matter what the data's content, the application will always use the expression as a value and not SQL code. Consequently, it is impossible to modify the query's structure.

To help ensure the correct use of data, many languages allow typed bindings. However, prepared statements by themselves cannot fix insecure statements—developers must use them correctly. For example, using prepared statements in the same way as regular statements—that is, building the SQL queries using string concatenation—instead of using placeholders for the variable part of the query will result in similar vulnerabilities.

Output validation

Validating the output of a process before it is sent out prevents users from receiving information they should not have, such as details about exceptions inside the application that can help in conducting other attacks. In another example of output validation, the protection system searches an application's output for critical information, such as a credit card number, and replaces it with asterisks before sending it to the recipient. Encoding is a type of output validation that avoids XSS vulnerabilities.[4] If the data sent to the browser goes into a webpage, it should be encoded using either HTML or percent encoding, depending on its destination in the page. This way, even malicious characters used in XSS attacks become innocuous, but the encoding preserves the data's meaning.

DETECTING VULNERABILITIES

Identifying security issues requires not only focusing on testing the application's functionalities but also on finding dangerous hidden flaws in the code that attackers can exploit.[5] The two main approaches for detecting vulnerabilities are white-box analysis and black-box testing.

White-box analysis

White-box analysis consists of examining the code without executing it. Developers can do this in one of two ways: manually, during code inspections and reviews; or automatically, using automated analysis tools.

Code inspection is the process in which a programmer's peers systematically examine the delivered code, searching for programming mistakes.[6] Security inspections are the most effective way to minimize vulnerabilities in an application; they are a crucial procedure when developing software for critical systems. Nevertheless, such inspections usually take a long time, are expensive, and require deep knowledge of Web security.

A less expensive alternative is code review,[6] a simplified version of inspections that is useful for analyzing less critical code. Reviews are also done manually, but they do not include a formal inspection meeting. Several experts perform the review individually, and a moderator filters and merges the outcomes. Although also an effective approach, code review is still quite expensive.

To reduce the cost of white-box analysis, developers sometimes rely on automated tools, such as static code analyzers. Static code analysis tools vet software code, either in source or binary form, in an attempt to identify common implementation-level bugs.[4] The analysis performed using existing tools varies depending on their sophistication, ranging from those that consider only individual statements and declarations to others that consider dependencies between lines of code. Among their other uses, such as for model checking and data flow analysis, these tools automatically highlight possible coding errors. The main problem is that exhaustive analysis is difficult and cannot find many security flaws because of the source code's complexity and the lack of a dynamic (runtime) view.

Although of great importance, the use of static code analysis tools sometimes reduces programmer productivity, predominantly because of the false positives reported, which leads to useless additional work.[7] To avoid this situation, in addition to having adequate time to learn how to use these tools, developers need policies to ensure correct tool use. For example, it is necessary to specify rules for classifying and selecting the warnings that developers should address. Also, developers must configure the analysis tools to report only the warnings that are relevant to the current development context. Developers who do not adopt effective practices for using static analysis usually end up underestimating its real benefits and consequently do not take advantage of all its functionalities.

Black-box testing

Black-box testing refers to the analysis of program execution from an external point of view. In short, it consists of comparing the software execution outcome with the expected result.[5] Testing is probably the most used technique for software verification and validation.

There are several levels for applying black-box testing, ranging from unit to integration to system testing. The testing approach also can be formal (based on models and well-defined test specifications) or less formal (referred to as "smoke testing," a type of rough testing intended to quickly reveal simple bugs).

The goal of robustness testing, a specific form of black-box testing, is to characterize the system's behavior in the presence of erroneous input conditions. Penetration testing is a special type of robustness testing that analyzes program execution in the presence of malicious inputs, searching for potential vulnerabilities. In this approach, testers apply fuzzing techniques, which consist of submitting unexpected or invalid items of data, to a Web application and review its responses, using HTTP requests.[4] Testers do not need to know the implementation details—they test the application inputs from the user's point of view. The number of tests can reach hundreds or even thousands for each vulnerability type.

Penetration testing tools automatically search for vulnerabilities, avoiding the repetitive and tedious task of conducting hundreds or even thousands of tests by hand for each vulnerability type. The most common automated security testing tools for Web applications are generally called either Web security or Web vulnerability scanners. These scanners allow easy testing of applications against vulnerabilities. Because they have a predefined set of test cases adapted to the target application, users only need to configure the scanner and let it test the application. Once the scanner completes the test, it reports any detected vulnerabilities. While most scanners are commercial tools, there are also some free application scanners, but these often have limited use because they lack most of the functionalities of their proprietary counterparts.

The use of static code analysis tools sometimes reduces programmer productivity, predominantly because of the false positives reported, which leads to useless additional work.

Limitations of vulnerability detection

Penetration testing and static code analysis can be manual or automatic. Because manual tests or inspections require specialized security resources and are time-consuming, automated tools are the typical choice of Web application developers. An important fact when considering the limitations of vulnerability detection tools is that testing for security is difficult. Indeed, measuring an application's security is challenging: although finding some vulnerabilities can be easy, guaranteeing that the application has no vulnerabilities is difficult.[1]

Both penetration testing and static code analysis tools have intrinsic limitations. Penetration testing relies on effective code execution; however, in practice, vulnerability identification only examines the Web application's output. Thus, the lack of visibility into the application's internal behavior limits penetration testing's effectiveness.

On the other hand, exhaustive source code analysis can be difficult. Code complexity and the lack of a dynamic (runtime) view might prevent finding many security flaws. Of course, penetration testing does not require access to the source code, while static code analysis does.

Using the wrong detection tool can lead to the deployment of applications with undetected vulnerabilities. Figure 2 compares the effectiveness of well-known and widely used penetration testing and static analysis tools in the detection of SQL injection vulnerabilities in Web services.[8] Results show that the coverage of static code analysis tools—including FindBugs, Fortify 360, and IntelliJ IDEA (anonymized as SA1 through SA3 in the figure)—is typically much higher than that of penetration testing tools, including HP WebInspect, IBM Rational AppScan, Acunetix Web

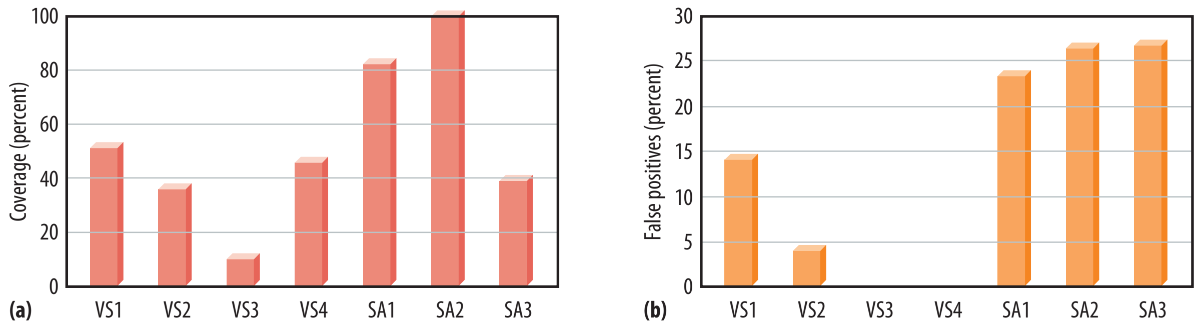


Figure 2. (a) Penetration testing versus (b) static code analysis.

Vulnerability Scanner, and a prototype tool developed at the University of Coimbra (anonymized as VS1 through VS4 in the figure). False positives are a problem for both approaches, but have more impact in static analysis. A key observation is that different tools implementing the same approach frequently report different vulnerabilities in the same code.

The results of studies highlighting the limitations of vulnerability tools suggest that it is necessary to improve the state of the art in vulnerability detection by, for instance, combining different approaches. Also, developers need to define mechanisms for evaluating and comparing different tools so they can select the tools that best fit each development scenario.

DETECTING ATTACKS

To prevent attacks against Web applications, software engineering teams must put in place attack detection mechanisms, usually called intrusion detection systems (IDSs) or Web application firewalls (WAFs). Different tools can function at the application or network level or even at one of the application's resources, such as the database, and they can use diverse approaches such as anomaly detection or signature matching to detect attacks.

Attack detection approaches

Attack detection consists of identifying deviations from learned behavior. Attack detection tools use approaches based on either anomaly detection or signatures.[8]

Anomaly detection tools usually require a training phase. Training exposes the system to nonmalicious requests, and the tool observes its behavior at a given architectural level and learns the regular operation. These tools consider the specifics of each Web application, but can produce many false warnings when the application's correct behavior changes or learning is incomplete.

In contrast, signature-based tools look for patterns of a predefined set of rules or signatures indicating an attack. Because these signatures are usually application independent, the tools' success is unrelated to the application's operational profile or any training process.

Tools that operate at the network level usually monitor and analyze network traffic to detect attacks before they reach the Web application. Attack detection tools that work at the application level analyze requests sent to the application and try to take advantage of the application-specific correlations between server-side programs and parameters inside the requests. Tools that work at the resource level protect resources related to each vulnerability type. These tools operate below the application level and close to the protected resource. A common example is IDSs that detect SQL injection attacks by monitoring accesses to the database server.

Tools use various strategies to gather the information about application requests and possible attacks. Some tools use sniffing strategies to monitor and analyze the data transferred over the network to observe HTTP traffic, but encryption, encoding, or encapsulation can limit their effectiveness. Also, networks can carry large amounts of data unrelated to the protected application. Other tools analyze the logs produced by the application or even the logs produced by the server that is running the application. Although this strategy does not directly delay requests to the application, it is limited by the information available in logs.

Yet another strategy is to introduce a proxy between the source of requests and the protected application or resource. This can easily stop attacks because it provides useful information about the target application or resource, however it impacts an application's normal behavior by introducing undesirable delays.

Limitations of attack detection

Because the specifics of each Web application affect the performance of attack detection tools, as does the architectural level at which the tools operate, their real effectiveness is frequently unknown.[9] Research shows that well-known tools have low effectiveness and only perform well in specific scenarios.[10] Most tools achieve low detection coverage (less than 20 percent in many cases), and they can report many false alarms (as high as 50 percent of the alarms generated). In addition, some tools present interesting results in particular scenarios, but provide poor results in others.

Database-level tools generally perform better than application-level tools,[10] which generate many false positives from requests that could never successfully attack the database. Anomaly-detection-based tools perform better for simpler applications, while signature-based tools are better for more complex applications. Tools can learn and better characterize behavior in simpler applications, consequently detecting deviations from a pattern more accurately. In fact, anomaly detection's success depends on the training phase. If training is incomplete or if the application's regular operational profile changes after training, the attack detection tool's effectiveness decreases.

Developers who use these tools often lack the training required to create adequate configurations. This reduces the tool's effectiveness, highlighting the importance of benchmarking methodologies to evaluate and compare different tools and configurations.[10]

NEW TRENDS AND DIRECTIONS

Achieving better results and improved effectiveness requires new techniques to overcome the intrinsic limitations of vulnerability-detection tools. However, overcoming these limitations is not easy because it requires shifting from traditional approaches to disruptive methods. The key is to relax some constraints and combine different methods to overcome individual limitations.

Acunetix AcuSensor is an example of a commercial technology that combines black-box scanning with feedback obtained during test execution (www.acunetix.com/websitesecurity/rightwvs.htm). This feedback comes from sensors previously implanted in the target application's code. Acunetix states that this technique finds more vulnerabilities and indicates exactly where they are in the code, while reporting fewer false positives.

A recently proposed technique that tries to achieve similar effects in a less intrusive way uses attack signatures together with interface monitoring to overcome the limitations of penetration testing for injection vulnerabilities.[11] This is a black-box testing technique because it only monitors the interface between the application and the resources related to the vulnerabilities (the database interface).

The Analysis and Monitoring for Neutralizing SQL-Injection Attacks (Amnesia) tool combines static analysis with runtime monitoring to detect SQL injection attacks.[12] It performs static analysis of a Web application's source code, building a model of legitimate queries that such applications generate. At runtime, it monitors dynamically generated queries, checking for compliance with the statically generated model. This tool classifies a query that violates the model as an attack and prevents that query from accessing the database.

Development processes also must evolve to react to new threats to Web application security. For example, the Microsoft Security Development Lifecycle complements the company's development process and is particularly aimed at addressing security issues, including activities such as specific security training for developing teams.[13] According to Microsoft, adopting this process has reduced the number of security flaws in its software. Although this is just an example, it shows that work on security processes for software development is a key issue in the industry.

Developers must consider security during the entire software product development and deployment life cycle. They must apply best practices in secure coding, perform adequate security testing, and use attack-detection systems to protect applications at runtime. Developers need help in this task both with training to acquire the required skills and the tools to increase their productivity.

Researchers should find ways to propose innovative tools that can be easily adopted in the development process and that are efficient and productive to meet deployment constraints. In the center of this evolution are the security testing tools that will continue to be crucial for verifying and validating applications to detect security vulnerabilities. Nevertheless, new hypotheses must be explored. A foreseeable possibility is the development of compilers that not only enforce the use of best coding practices, but also can automatically fix existing security vulnerabilities. C

References

1. M. Howard and D.E. Leblanc, *Writing Secure Code*, Microsoft Press, 2002.
2. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2002.
3. G. McGraw, *Software Security: Building Security In*, Addison-Wesley, 2006.
4. D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*, John Wiley & Sons, 2007.
5. B. Arkin, S. Stender, and G. McGraw, "Software Penetration Testing," *IEEE Security & Privacy*, Jan.-Feb. 2005, pp. 84-87.
6. D.P. Freedman and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, Dorset House, 2000.
7. N. Ayewah and W. Pugh, "A Report on a Survey and Study of Static Analysis Users," *Proc. Workshop Defects in Large Software Systems* (DEFECTS 08) ACM, 2008, pp. 1-5.
8. N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the

Detection of SQL Injection Vulnerabilities in Web Services," *Proc. 15th IEEE Pacific Rim Int'l Symp. Dependable Computing* (PRDC 09), IEEE CS, 2009, pp. 301-306.

9. E. Biermann, E. Cloete, and L.M. Venter, "A Comparison of Intrusion Detection Systems," *Computers & Security*, Dec. 2001, pp. 676-683.
10. I.A. Elia, J. Fonseca, and M. Vieira, "Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study," *Proc. 21st IEEE Int'l Symp. Software Reliability Eng.* (ISSRE 10), IEEE CS, 2010, pp. 289-298.
11. N. Antunes and M. Vieira, "Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services," *Proc. IEEE Int'l Conf. Services Computing* (SCC 11), IEEE CS, 2011, pp. 104-111.
12. W.G.J. Halfond and A. Orso, "Preventing SQL Injection Attacks Using AMNESIA," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), IEEE CS, 2006, p. 798.
13. M. Howard and S. Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.

Nuno Antunes *is a PhD student in the Department of Information Science and Technology 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 Information Science and Technology 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 and ACM. Contact him at mvieira@dei.uc.pt.*