Analysis of Field Data on Web Security Vulnerabilities

José Fonseca, Nuno Seixas, Marco Vieira, and Henrique Madeira

Abstract—Most web applications have critical bugs (faults) affecting their security, which makes them vulnerable to attacks by hackers and organized crime. To prevent these security problems from occurring it is of utmost importance to understand the typical software faults. This paper contributes to this body of knowledge by presenting a field study on two of the most widely spread and critical web application vulnerabilities: SQL Injection and XSS. It analyzes the source code of security patches of widely used web applications written in weak and strong typed languages. Results show that only a small subset of software fault types, affecting a restricted collection of statements, is related to security. To understand how these vulnerabilities are really exploited by hackers, this paper also presents an analysis of the source code of the scripts used to attack them. The outcomes of this study can be used to train software developers and code inspectors in the detection of such faults and are also the foundation for the research of realistic vulnerability and attack injectors that can be used to assess security mechanisms, such as intrusion detection systems, vulnerability scanners, and static code analyzers.

Index Terms—Security, Internet applications, languages, review and evaluation

1 Introduction

Most information systems and business applications built nowadays have a web front end and they need to be universally available to clients, employees, and partners around the world, as the digital economy is becoming more and more prevalent in the global economy. These web applications, which can be accessed from anywhere, become so widely exposed that any existing security vulnerability will most probably be uncovered and exploited by hackers.

In the context of the present work, we use the terminology introduced by Avizienis et al. [4] that considers an error as a "*deviation of an external state of the system from the correct service state*," a fault as "*the adjudged or hypothesized cause of an error*," a vulnerability an "*internal fault that enables an external fault to harm the system*," and an attack as a "*malicious external fault*."

The security of web applications becomes a major concern and it is receiving more and more attention from governments, corporations, and the research community [7], [13], [25], [52], [56]. Attackers also followed the move to the web and as such more than half of current computer security threats and vulnerabilities affect web applications [24]. Not surprisingly, the number of reported attacks that exploit web application vulnerabilities is increasing [19]. In fact, numerous data breach attacks are frequently reported due to web application security problems [31], [36], [43], [45], [53].[1]

Given the preponderant role of web applications in many organizations, one can realize the importance of finding ways to reduce the number of vulnerabilities. This can be helped with a deeper knowledge on software faults behind such vulnerabilities [16], [21], [23], [49]; however, this is a vast field and there is still a lot of work to be done, like the one presented by Scholte et al. [46].

This paper contributes to fill this gap by presenting a study on characteristics of source code defects generating major web application vulnerabilities. The main research goal is to understand the typical software faults that are behind the majority of web application vulnerabilities, taking into account different programming languages. To understand the relevance of these kinds of vulnerabilities for the attackers, the paper also analyzes the code used to exploit them.

Regarding the programming language perspective, we considered some of the most relevant in the context of web applications. First, we focused on the most widely used weak typed language, PHP. Then, we analyzed strong typed languages, namely Java, C#, and VB. Recall that our goal is not to analyze each programming language in what concerns their ability to prevent security vulnerabilities, but to analyze the vulnerabilities and their relation with some language characteristics, like the type system.

The vulnerabilities analyzed in our field study were cross-site scripting (XSS) and SQL injection (SQLi), as these are two of the most common and critical vulnerabilities found in web applications [31], [36]. To characterize the types of software faults that are most likely to create these vulnerabilities, we classified the pieces of code used to fix

- *J. Fonseca is with the Research Unit for Inland Development, Institute Polytechnic of Guarda and the Centre for Informatics and Systems, University of Coimbra. E-mail: josefonseca@ipg.pt.*
- *N. Seixas, M. Vieira, and H. Madeira are with the Centre of Informatics and Systems, University of Coimbra. E-mail: {naseixas, mvieira, henrique}@dei.uc.pt.*

Manuscript received 3 June 2013; revised 3 June 2013; accepted 22 Aug. 2013; published online 5 Sept. 2013.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2013-01-0014. Digital Object Identifier no. 10.1109/TDSC.2013.37.

1. The most relevant vulnerabilities are known for many years; however, they are still proliferating, in spite of the development of tools that help automate their mitigation (e.g., Java Pathfinder [27]). In fact, the knowledge about them and how they are created is still insufficient.

them in a set of real web applications. Each patch was inspected in depth to gather the precise characteristics of the code that was responsible for the security problem and classified them according to an adaptation of the orthogonal defect classification (ODC) [6], [8].

The proposed methodology allows gathering the information on common mistakes that developers should avoid. This knowledge is helpful for training [17], and it is crucial for the specification of guidelines for security code reviewers, for the evaluation of penetration testing tools, as well as for the creation of safer internal policies for programming practices, among others. It can also be used to build a realistic attack injector [18]. In our study, we observed that not every vulnerability is equally important for an attacker, and when not all vulnerabilities can be fixed in due time, these data may be used to select those that should be addressed first, for example.

The structure of the paper is as follows. Section 2 discusses related work. Section 3 presents some background on security vulnerabilities and web programming languages. Section 4 details the classification of software faults, the data sources (for web applications and patches), and the process followed to analyze and classify the patch of each vulnerability. Section 5 discusses the results of the field study, and Section 6 presents a detailed vulnerability analysis. Section 7 presents another study, but on the exploits developed to attack web applications. Section 8 discusses the threats to validity. Finally, Section 9 concludes the paper and suggests future work.

2 Related Work

It is unfeasible to produce complex applications without defects, and even when this occurs, it is impossible to know it, prove it, and repeat it systematically [22]. Software developers cannot assure code scalability and sustainability with quality and security, even when security is defined from the ground up [20].

One of the aspects that contribute to security problems seems to be related to how bad different programming languages are in terms of propensity for mistakes. Clowes [9] discussed common security problems related to the easiness in programming with PHP and its features, but this affects many other programming languages. The choice of the type system (strong or weak) and the type checking (static or dynamic) of the programming language also affects the robustness of the software. For example, a strong typed language with a static type checking can help deliver a safer application without affecting its performance [51]. Scholte et al. [46] presented an empirical study on a large set of input validation vulnerabilities developed in six programming languages. However, that work focused on the relationship between the specific programming language used and the vulnerabilities that are commonly reported, not going into details in what concerns the typical software faults that originate vulnerabilities, like we do in the present work.

One of the best practices to find software faults is to perform a static analysis to the code [27]. This is a labor-intensive job, usually done with automated tools, although they lack the precision of the manual counterpart. To improve them and to help predict software failures, a new defect classification scheme was proposed [32]. Another research work proposed a security resources indicator that seems to be strongly correlated with change in vulnerability density over time [54].

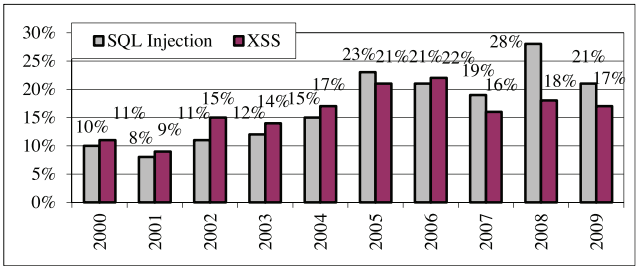


Fig. 1. Evolution of disclosed reports of SQLi and XSS causes of vulnerabilities [33].

Web application vulnerabilities have been addressed by recent studies from several points of view, but without any code analysis [3], [7], [31], [36], [54]. To overcome the low level of detail of existing vulnerability databases, some researchers proposed approaches based on the market, instead of on software engineering [37].

The attacker's perspective has also been of some focus in the literature [9], [16], [23], [30], but mainly through empirical data gathered by the authors highlighting social networking and what could be obtained from attacking specific vulnerabilities. Some studies analyzed the attacks from the victim's perspective, including the proposal of a taxonomy to classify attacks based on their similarities [2] and the analysis of attack traces from HoneyPots to separate the attack types [12]. There is, however, a lack of knowledge about existing exploits and their correlation with the vulnerabilities.

To improve software quality, developers need a deeper knowledge about the software faults that must be mitigated. The underlying idea is that knowing the root cause of software defects helps removing their source, therefore contributing to the quality improvement [29]. Researchers at IBM developed a classification scheme of software faults, intended to improve the software design process and, consequently, reduce the number of faults [6], [8]. It is the ODC and it is typically used to classify software faults or defects after they have been fixed and it is also broadly used by the industry and researchers outside IBM [32].

3 Vulnerabilities and Programming Languages

The Open Web Application Security Project Report listed the 10 most critical web application security risks, having SQLi at the top, followed by XSS [36]. Other studies also found XSS and SQLi as the most prevalent vulnerabilities [1], [24], [34], [54]. Fig. 1 depicts the yearly percentage of disclosed XSS and SQLi among all the causes of web application vulnerabilities showing that they are increasing over time [33].

SQLi attacks take advantage of unchecked input fields in the web application interface to maliciously tweak the SQL query sent to the back-end database. By exploiting an

XSS vulnerability, the attacker is able to inject into web pages unintended client-side script code, usually HTML and Javascript. SQLi and XSS allow attackers to access unauthorized data (read, insert, change, or delete), gain access to privileged database accounts, impersonate other users (such as the administrator), mimic web applications, deface web pages, view, and manipulate remote files on the server, inject and execute server side programs that allow the creation of botnets controlled by the attacker, and so on. Details on the most common vulnerabilities, including SQLi and XSS, along with the reasons of their existence, attacks, best practices to avoid, detect, and mitigate them can be found in many referenced works, such as [23], [36], [49].

Many programming languages are currently used to develop web applications. Ranging from proprietary languages (e.g., C#, VB) to open source languages (e.g., PHP, CGI, Perl, Java), the spectrum of languages available for web development is immense.

Programming languages can be classified using taxonomies, such as the programming paradigm, the type system, the execution mode, and so on. The type system, particularly important in the context of the present work, specifies how data types and data structures are managed and constructed by the language, namely how the language maps values and expressions into types, how it manipulates these types, and how these types correlate. Regarding the type system, they can be typed versus untyped, static versus dynamic typed, and weak versus strong typed [51]. In particular, strong typed languages provide the means to produce more robust software, since a value of one type cannot be treated as another type (e.g., a string cannot be treated as a number), as in weak typed languages.

One of the contributions of this work is to help understanding the impact of the type system in the security of web applications. This is of particular significance, as critical security vulnerabilities like XSS and SQLi are strongly related to the way the language manages data types [36]. For example, it is common to find attacks that inject SQL code by taking advantage of variables that supposedly should not be strings (e.g., numbers, dates) as the type of the variable is determined based on the assigned value. On the other hand, in strong typed languages, this is not possible because the type of variables is determined before runtime and the attempt to store a string in a variable of another type raises an error. However, this does not prevent the occurrence of vulnerabilities in strong typed languages, but only by taking advantage of string variables. In fact, although Java is intrinsically a safe programming language [5] and it is a strong typed language, vulnerabilities can be found in Java programs due to implementation faults [28].

4 Security Patch Field Study Methodology

This section presents the methodology to obtain and classify the source code and the security patches of the web applications of our field study.

4.1 Web Applications Analyzed

One mandatory condition of our field study is the ability to analyze the source code of current and previous versions of the target web applications, together with the associated security patches. Therefore, we restricted our search to open source web applications. We also decided to choose only web applications with a large number of downloads or installations, and we also preferred award-winning web applications.

TABLE 1
Versions of Weak Typed (PHP) Apps

Web app.	Versions analyzed
PHP-Nuke	6.0, 6.5, 6.9, 7.0, 7.2, 7.6, 7.7, 7.8, 7.9
Drupal	4.5.5, 4.5.6, 4.6.5, 4.6.6, 4.6.7, 4.6.8, 4.6.9, 4.6.10, 4.6.11, 4.7.6, 5.1
PHP-Fusion	6.00.106, 6.00.108, 6.00.110, 6.00.204, 6.00.206, 6.00.207, 6.00.303, 6.00.304, 6.01.4, 6.01.5, 6.01.6, 6.01.7, 6.01.8, 6.01.9, 6.01.10, 6.01.11, 6.01.12
WordPress	1.2.1, 1.2.2, 1.5.2-1, 2.0, 2.0.10-RC2,2.0.4, 2.0.5, 2.0.6, 2.1.2, 2.1.3 2.1.3-RC2, 2.2, 2.2.1, 2.3
phpMyAdmin	2.1.10, 2.4.0, 2.5.2, 2.5.6, 2.5.7PL1, 2.6.3PL1, 2.6.4, 2.6.4PL4, 2.7.0PL2, 2.8.2.4, 2.9.0, 2.9.1.1, 2.10.0.2, 2.10.1, 2.11.1.1, 2.11.1.2 and SVN revisions
phpBB	.0.3, 2.0.5, 2.0.6, 2.0.6c, 2.0.7, 2.0.8, 2.0.9, 2.0.10, 2.0.16, 2.0.17

These restrictions are aimed to allow only web applications relevant to a wide range of users and, wherever possible, with a higher quality granted by the awards. Although this does not guarantee generalization of results, it still shows how mainstream web applications are in relation to security. Under these conditions, we have chosen the most relevant applications we could manually analyze, given the resources at our disposal.

Because PHP is the most widely used language present in web applications, we used it for the weak typed programming language study. Due to time constraints, other programming languages like PERL could not be considered. Given the high number of security problems found, we only used six web applications (see Table 1): PHP-Nuke (phpnuke.org), Drupal (drupal.org), PHP-Fusion (php-fusion.co.uk), WordPress (wordpress.org), phpMyAdmin (phpmyadmin.net), and phpBB (phpbb.com).

Drupal, PHP-Fusion, and phpBB are web Content Management Systems (CMS). Drupal won first place at the 2007 Open Source CMS Award [39]. PHP-Fusion was one of the five overall award winner finalists at the 2007 Open Source CMS Award [39]. Finally, phpBB is the most widely used Open Source forum solution and it was the winner of the 2007 Sourceforge Community Choice Awards for Best Project for Communications [48]. PHP-Nuke is a well-known web-based news automation system built as a community portal and it has been downloaded over 8 million times from the official site [41]. WordPress is reportedly the most widely used personal blog publishing platform with millions of users. phpMyAdmin is a web-based MySQL administration tool. It is included in many Linux distributions, it is available in 47 languages, and it was the winner of the 2007 Sourceforge Community Choice Awards for Best Tool or Utility for SysAdmins [48].

For the strong typed programming languages, for which we found less security problems, we used 11 web applications developed in Java, C#, and VB (see Table 2): JForum (jforum.net), OpenCMS (opencms.org), BlojSom (sourceforge.net/projects/blojsom), Roller WebLogger (rollerweblogger.org), JSPWiki (jspwiki.org), SubText

TABLE 2
Versions of Strong Typed Apps

Web app	Versions analyzed	Lang.
JForum	2.1.8, 3	Java
OpenCMS	6.0.3, 6.2.2, 6.2.3, 7.0.3, 7.0.4	Java
JSPWiki	2.0.45, 2.2.13_beta, 2.4.103, 2.4.104, 2.5.79_alpha, 2.5.139_beta, 2.6.1	Java
BlojSom	2.31, 2.32	Java
Roller Web-Logger	2.3, 2.3.1_RC2, 2.3, 3.0	Java
SubText	1.0.0.2, 1.5.1	C#
Dot-NetNuke	2.1.2, 3.0.13, 3.3.5, 4.08.02, 4.08.03, 4.08.04	VB
YetAnother-Forum	0.9.9, 1.0.0	C#
BugTrack-er.NET	0.91, 2.2.7, 2.7.1, 2.7.2	C#
Deki Wiki	8.05, 8.05.1, 8.08	C#
ScrewTurn Wiki	2.0.30, 2.0.31	C#

(subtextproject.com), Dot-NetNuke (dotnetnuke.com), YetAnotherForum (yetanotherforum.net), BugTracker.NET (ifdefined.com/bugtrackernet.html), Deki Wiki (developer.mindtouch.com), and ScrewTurn Wiki (screwturn.eu).

JForum and YetAnotherForum are discussion board system forums. OpenCMS is a web CMS with a large community of users and has six books published about it. BlojSom, Roller WebLogger, and SubText are blog software packages. Roller WebLogger drives important blogs such as blogs.sun.com, blog.usa.gov, IBM Lotus Connections, and IBM Developer Works blogs. JSPWiki, Deki Wiki, and ScrewTurn Wiki are Wiki engines used by many communities and organizations. JSPWiki is well known and used by enterprises such as Recursa, IBM, Shopping.com, and Oxford University SPIE Project. Dot-NetNuke is a web application framework for creating interactive web sites and has a community of over 440,000 users. BugTracker.NET is a web-based customer support issue tracker.

4.2 Classification of Software Faults from the Security Vulnerability Point of View

After choosing a web application, we searched the web for all reported SQLi and XSS patches that were classified based on the work presented in [15]. This classification is derived from the code defect types (assignment, checking, interface, and algorithm) of the ODC software fault types [6], [8]. As ODC fault types are still too broad [15], we detailed them according to the nature of the defect: missing construct, wrong construct, and extraneous construct.

All the security vulnerabilities collected could be classified using only 15 of the fault types already identified in [15] and one extra fault type, the missing function call extended (MFCE); however, not all were found in both weak typed and strong typed web applications (see Table 3).

The missing function call extended (marked with an * in Table 3) is a new addition and it is based on a missing function in situations where the return value is used in the code (as opposed to the MFC where the return value is not used).

4.3 Obtaining the Patch Code

For our field study, we need to obtain the web application code, as well as the source code of the patches. By comparing them, we analyze the cause of the vulnerabilities and classify the changes made in the code to fix them.

To gather the source code of security patches, we used several sources of data, such as developer sites, online magazines, news sites, sites related to security, hacker sites, change log files of the application, the version control system (VCS) repository, and so on.[2] Next are the main sources of information:

1. *Security patched files.* These files are applied to the application by replacing the vulnerable files. To extract only the code change that these files provide, we used the UNIX diff command applied to both the patch and the original (vulnerable) file.
2. *Updated versions of the web application.* This represents completely new releases of the application containing new features and fault fixes (including security ones). Although this was our primary source of data, it was also the most labor-intensive one. It is necessary to compare all the files of the vulnerable and updated versions of the application looking for security fixes. This process can be eased when there is a change log file. This file consists of the summary of changes made to the new version of the application, including faults and security issues fixed. After identifying the vulnerable source file and the fix, the UNIX diff command was used.
3. *Security diff files.* These are files containing only the code changes needed to fix a referenced vulnerability. The contents are ready to be applied to the target application using the UNIX patch command. This single file has all the information needed; however, it was not very common.
4. *Version control system repositories.* Many applications are developed using a VCS to manage the contributions of the community of developers from around the world. We were granted permissions to query some VCS repositories, so we had access to all the revisions (similar to versions) of the application and their change log files. Through the change log file, we can identify the revisions of the application where vulnerabilities were fixed. A differential analysis using the UNIX diff command obtained the code changes that fixed the vulnerabilities.

4.4 Patch Code Analysis Guidelines

The patch code was analyzed according to the extension of the ODC classification, emphasizing the nature of the fix as missing, wrong, or extraneous code. When there was no information about it, the decision whether it was an XSS or

2. The complete list of web resources queried was the following: blogsecurity.net, buayacorp.com, bugs.debian.org, dev.wp-plugins.org, digitrustgroup.com, downloads.securityfocus.com, downloads.sourceforge.net, drupal.org, michaeldaw.org, milw0rm.com, notsosecure.com, nuked gallery.net, nukeresources.com, nvd.nist.gov, osvdb.org, php-fusion.co.uk, phpbb.com, phpmyadmin.net, phpmyadmin.svn.sourceforge.net, phpnuke.org, phpsecure.info, seclists.org, secunia.com, securiteam.com, secunia.com, securitydot.net, securityfocus.com, sourceforge.net, trac.wordpress.org, trapkit.de, waraxe.us, wordpress.org, www.virtuax.be.

TABLE 3
Fault Types Observed in the Field and Corresponding ODC Fault Type

Fault type	Description	To correct the fault we need to	ODC type
EFC	Extraneous function call	Delete the function call	Algorithm
ELOC	Extraneous "OR EXPR" in expression used as branch condition	Delete the OR expression in the branch condition	Checking
MFC	Missing function call	Add a function whose return value is not used in the code	Algorithm
MFCE *	Missing function call extended	Add a function whose return value is used elsewhere in the code	Algorithm
MIA	Missing if construct around statements	Add an if construct surrounding a set of statements	Checking
MIEB	Missing if construct plus statements plus else before statements	Add an if construct with an else condition and a set of statements surrounded by them	Algorithm
MIFS	Missing if construct plus statements	Add an if construct and a set of statements surrounded by it	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	Add an AND expression in the branch condition	Checking
MLOC	Missing "OR EXPR" in expression used as branch condition	Add an OR expression in the branch condition	Checking
MLPA	Missing small and localized part of the algorithm	Add a small set of statements	Algorithm
MVIV	Missing variable initialization using a value	Initialize a variable with a constant value	Assignment
WFCS	Wrong function called with same parameters	Replace the function call by the right function	Algorithm
WLEC	Wrong logical expression used as branch condition	Correct the logical expression	Checking
WPFV	Wrong variable used in parameter of function call	Correct the variable used in the parameter of the function	Interface
WVAV	Wrong value assigned to variable	Correct the value assigned to the variable	Assignment

* *Fault type added to the extension of the ODC, based on the MFC fault type, when the return value is used in the code.*

an SQLi vulnerability was done by looking at the source code. SQLi involves changing an SQL query string and XSS displaying an unchecked variable. The classification process followed these guidelines:

1. We assumed that the information publicly disclosed in specialized sites is accurate and that the fix available by the developer of the web application solves the stated problem.
2. When the patch can fix both XSS and SQLi, the corresponding fault type is counted for both vulnerabilities.
3. To correct a single vulnerability several code changes may be necessary. We consider all the changes as a series of individual fault type fixes, because missing any of them makes the application vulnerable.
4. When a particular code change corrects several vulnerabilities, each vulnerability corrected is counted.
5. When a single vulnerability affects several versions of the application and the patch is the same for all, then it accounts for a single fix.

The methodology to classify the web application security patches was the following:

1. Manual analysis of the vulnerable source code of the application and of the code after being patched.
2. Classification of each singular code fix found in the patch from the perspective of what was done wrong when developing the application.
3. Loop through the previous steps until all patches of the web application are analyzed.

By following the above guidelines, it was possible to classify 95 percent of all the code fixes gathered. The discarded 5 percent, for which we could not identify the code responsible for the security fix, account for patches too complex or the merge of security problems with other faults.

5 Results and Discussion of the Vulnerability Field Study

This section presents and discusses the results of the field study. We used the Pearson product-moment correlation (statistically significant when $P < 0.05$) to see the strength and direction of the relationship of two variables. A positive correlation (positive r) indicates that when one variable increases so does the other and a negative correlation (negative r) indicates that when one variable increases the other decreases. Strong correlation is when r is between 1 and 0.5; medium correlation when r is between 0.5 and 0.3; weak correlation when r is lower than 0.3 [11]. The number of samples is n.

5.1 Results for PHP Weak Typed Applications

We classified 655 XSS and SQLi security fixes found in six web applications developed using PHP. The distribution of the occurrences throughout the 12 classification fault types is shown in Fig. 2. Comparing with Table 3, we see that we did not found any sample for three fault types (MIEB, MLPA, and WLEC).

The most representative is the MFCE, accounting for around 3/4 of all the types found. The high value observed may be related to the common use of specific functions to validate or clean input data.

The next three most common fault types are the WPFV, MIFS, and WVAV (see Table 3 for details on these types). These vulnerabilities were mainly found in the following situations:

1. Missing quotes around a variable in SQL queries allowing an attacker to inject strings that are treated as part of the structure of the query.
2. Missing IF around a statement. When a variable should not be null, it needs to be initialized to a specific value; otherwise, a malicious code may be

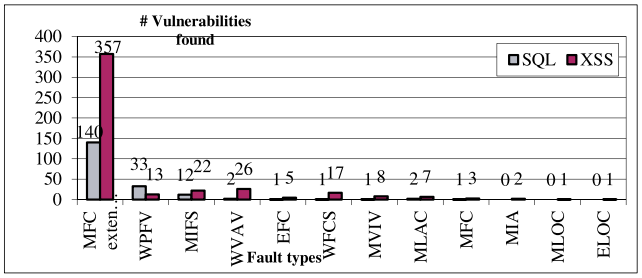


Fig. 2. Weak typed (PHP) web applications vulnerability fault types summary.

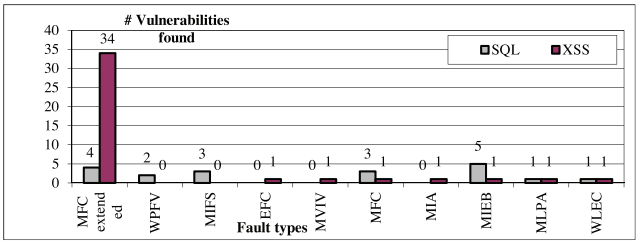


Fig. 3. Strong typed (Java, C#, VB) web applications vulnerability fault types summary.

injected.[3] This situation is an exploit of the PHP directive "`register_globals = on`" [42], allowing assigning values to variables, based on input from GET, POST, or COOKIE data. If the developer relies on the default value and does not assign a value to the variable, an attacker may exploit it by tweaking the HTTP GET request, for example.

3. A poor regular expression (regex)[4] used to filter the input. We frequently found several past versions of the same application, with the same regex string being slightly updated as new attacks were discovered.

Excluding the fault types already discussed, the remaining types represent only 7.63 percent of the vulnerabilities.

Our results also show that all fault types contribute to XSS and only eight to SQLi. The four fault types that do not contribute to SQLi (MFC, MIA, MLOC, and ELOC) are residual (1.22 percent). A Pearson correlation showed a strong, positive correlation to the number of SQLi and XSS vulnerabilities, which was statistically significant ($r = 0.975$, $n = 12$, $P < 0.0005$).

A common belief is that vulnerabilities related to input validation are mainly due to missing IF constructs or even missing conditions in the IF construct. However, our results show that the typical PHP approach is to clean the input data with a function and let the program run normally, instead of stopping it and raise an exception. In fact, missing IF fault types (MIFS and MIA) account for 5.5 percent and missing condition fault types (MLAC and MLOC) account only for 1.52 percent.

5.2 Results for Strong Typed Applications

For the strong typed language, we collected and classified 60 XSS and SQLi vulnerabilities, distributed over 11 web applications presented in Fig. 3. Comparing with Table 3, five fault types (WVAV, WFCS, MLAC, MLOC, ELOC) were not found in this study.

Our data show that MFCE is the most frequent as the majority of vulnerabilities are sanitized using functions that clean and validate the input. Another interesting fact is that most of these fixes (82.93 percent) are related to XSS vulnerabilities, which by definition is prone to this kind of error. A Pearson correlation was run to determine the relationship between the number of SQLi and XSS vulnerabilities; however, it was not statistically significant ($r = 0.402$, $n = 10$, $P < 0.250$). We need more samples to make strong assumptions about these results.

Concerning SQLi, we have not observed the preponderance of a single fault type, as was the case for XSS. The most frequent is MIEB with 26.32 percent, followed by MFCE with 21.05 percent. This shows that the way this kind of vulnerabilities are fixed in strong typed languages is not only by using simple verification of other application states (through the introduction of IF...ELSE statements), but also by using a function to sanitize user input. This simplification of the sanitation process using IF statements, comparing with the PHP results, may be a direct contribution of the type of language used having more robust structures to manipulate the variables.

5.3 Lessons Learned

From the results discussed previously, we can summarize the main differences and correlations observed in the vulnerabilities found in the field for weak and strong typed web applications.

The major divergence comes from the number of vulnerabilities detected. While for PHP web applications, we could identify 655 vulnerabilities in only six applications, for strong typed languages, we needed 11 applications to identify only 60 vulnerabilities. It is, indeed, possible to write vulnerable code using strong typed languages, but our observation suggests that they are safer than those developed with weak typed languages, which is also the trend shown in other reports [46], [55]. The fact that, in a strong typed language, a well-defined integer variable cannot be used to store characters is, by itself, a huge benefit to prevent many vulnerabilities. Some authors go even further and they agree that a strong type system is, indeed, necessary to improve the application security by presenting additions to the type system [44]. Nonetheless, strong typed languages, by themselves, do not eliminate the need to validate and sanitize all the inputs from the user, as we can see by our field results.

The distribution of XSS and SQLi vulnerabilities found is similar for both weak and strong typed applications (see Fig. 4). With two-third of all vulnerabilities, XSS is the most frequent. Usually, XSS is easier to find although SQLi is more interesting for the attacker [31], [50].

Table 4 summarizes the distribution of the 15 fault types per vulnerability type for both weak and strong typed web applications. As we can see highlighted with shade background, the MFCE is clearly the main cause of vulnerabilities, regardless the type system used (63.33 percent for

3. PHP, as other scripting languages like Perl, does not require variable initialization (a NULL value is automatically assigned).

4. A regex string describes a search pattern, according to specific syntax rules, that is used to search inside another string.

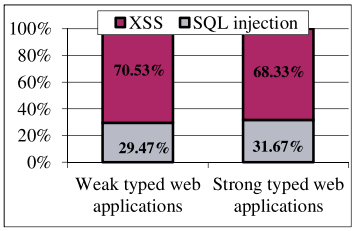


Fig. 4. XSS and SQLi distribution.

strong typed and 75.87 percent for weak typed). More details and examples on MFCE are shown in Section 6. A Pearson correlation was run to determine the relationship between the number of all the strong and weak typed vulnerabilities. There was a strong, positive correlation, which was statistically significant ($r = 0.976$, $n = 15$, $P < 0.0005$). This shows that both weak type and strong type vulnerabilities follow a similar pattern, but looking at Table 4, we see that the MFCE is responsible for most of both XSS and SQLi vulnerabilities, although its prevalence in XSS is more evident (82.93 percent in strong typed languages and 77.27 percent in weak typed).

The big difference between weak and strong typed applications concerns the faults classified in second and third places. For the weak typed, WPFV and MIFS ranked second and third, respectively, while for the strong typed, we found MIEB and MFC in these positions.

MIEB shows that fixes were not only based on the introduction of simple IF statements, which would correspond to the MIFS found in the weak typed. This kind of fix can be interpreted in two ways: 1) there were some more complex algorithm steps to be performed or 2) the developer was more conservative and followed the best practice recommendations that state the need for having complete IF...ELSE statements [10], [14].

Another major difference lies in the fact that WPFV represents only 3.33 percent of the strong typed applications, while it represents 7.02 percent of the weak typed, being the second most frequent fault type. This suggests that the use of wrong variables (or wrong values of the variables) in strong typed applications is lower than the number observed in weak typed applications, as expected. The fact that WPFV is only in the fifth place also suggests that the main security problem in applications built with strong typed languages is due to inputs not correctly sanitized and not by the use of wrong variables (or variables with wrong values) in the algorithm. This can be linked to the fact that, by definition, in strong typed languages, a variable has a predefined type of data, while in weak typed, a variable can handle many types, especially in dynamic typed languages, like PHP.

Another analysis can be made based on the age of the applications. The majority of the strong typed web application vulnerabilities analyzed were only well identified and disclosed after 2005. By that time, the concern about XSS and SQLi was already disseminated throughout the world (see Fig. 1), which can be confirmed by several studies [7], [13], [25], [31], [52], [56]. Therefore, when most of the strong typed applications analyzed were implemented, the technical community was already concerned about these problems. However, the weak typed web applications analyzed were released between 1998 and 2003, many years before most strong typed languages web applications (Java JDK was released in 1996, C#, and VB around 2002), when these vulnerabilities were not yet widely known. This may explain the larger number of vulnerabilities found in PHP web applications (PHP was created in 1995).

TABLE 4
Distribution of Fault Types per Vulnerabilities

Fault type	Weak type (PHP)						Strong type (Java, C#, VB)					
	SQLi		XSS		Total		SQLi		XSS		Total	
	#	(%)	#	(%)	#	(%)	#	(%)	#	(%)	#	(%)
MFCE	140	72.54	357	77.27	497	75.88	4	21.05	34	82.93	38	63.33
WPFV	33	17.1	13	2.81	46	7.02	2	10.53	0	0	2	3.33
MIFS	12	6.22	22	4.76	34	5.19	3	15.79	0	0	3	5
WVAV	2	1.04	26	5.63	28	4.27	0	0	0	0	0	0
WFCS	1	0.52	5	3.68	18	2.75	0	0	1	0	0	0
MVIV	1	0.52	17	1.73	9	1.37	0	0	0	2.44	1	1.67
MLAC	1	1.04	8	1.52	9	1.37	0	0	1	0	0	0
EFC	2	0.52	7	1.08	6	0.92	0	0	0	2.44	1	1.67
MFC	1	0.52	3	0.65	4	0.61	3	15.79	1	2.44	4	6.67
MIA	0	0	2	0.43	2	0.31	0	0	1	2.44	1	1.67
MLOC	0	0	1	0.22	1	0.15	0	0	0	0	0	0
ELOC	0	0	1	0.22	1	0.15	0	0	0	0	0	0
MIEB	0	0	0	0	0	0	5	26.32	1	2.44	6	10
MLPA	0	0	0	0	0	0	1	5.26	1	2.44	2	3.33
WLEC	0	0	0	0	0	0	1	5.26	1	2.44	2	3.33
Total	**193**	**100**	**462**	**100**	**655**	**100**	**19**	**100**	**41**	**100**	**60**	**100**

The results presented in this paper also unveil an important trend, also confirmed by other researchers [23]: a small set of fault types is responsible for most of the vulnerabilities. This observation can be used to train software developers, focusing their attention on the correct treatment of the software structures related to the most frequent types of faults. Additionally, this knowledge can also be useful to improve the effectiveness of code inspections, as the team will be more focused on a few important code structures that can cause most vulnerabilities.

6 Detailed Vulnerability Analysis

During the classification of web application vulnerabilities, we saw repeating patterns in the code. We observed that the instructions that fixed the vulnerabilities belonged to a restricted subset of all the possible code structures of each fault type. This kind of deep understanding can be explored to build security tools, like an attack simulator that injects realistic vulnerabilities to validate intrusion detection systems and other security mechanisms, like automated program repair [26].

To make use of these data, which are more granular, and accommodate the precise situations found, we defined subtypes for the four most common fault types (MFCE, WPFV, MIFS, and WVAV). They are shown in Table 5, along with their occurrences for PHP applications, from which we have more samples. Once again, we can observe that a few set of subtypes, MFCE subtypes A and B, are responsible for most of the vulnerabilities (63.66 percent). There are also important differences between XSS and SQLi: MFCE-A is more important in SQLi, but MFCE-B is the opposite; WPFV-A has a huge importance in SQLi and none was found in XSS.

TABLE 5
Fault Subtypes in PHP

Fault sub-types		Description	SQL (%)	XSS (%)	Total (%)
MFCE	A	Missing cast to numeric of a variable	64.25	37.45	45.34
	B	Missing assignment of a variable to a custom made function	4.15	24.24	18.32
	C	Missing assignment of a variable to a predefined function	4.15	15.58	12.21
WPFV	A	Missing quotes in variables inside a string argument of a SQL query	16.06	0.00	4.73
	B	Wrong regex string of a function argument	1.04	1.08	1.07
	C	Wrong sub-string of a function argument	0.00	1.08	0.76
	D	Wrong variable when it is an argument of a function	0.00	0.65	0.46
MIFS	A	Missing traditional "if…then…else" condition	5.18	4.55	4.73
	B	Missing "if…then…else" condition in compact form	1.04	0.65	0.76
WVAV	A	Missing pattern in a regex string assigned to a variable	0.00	3.03	2.14
	B	Wrong value in an array or a concatenation of a new substring inside a string	0.00	0.87	0.61
	C	Wrong variable assigned to a variable	0.00	0.87	0.61
	D	Missing quotes in variables inside a string in a SQL query assignment	1.04	0.00	0.31
	E	Missing destruction of the variable	0.00	0.65	0.46
	F	Extraneous concatenation operator "." in an assignment	0.00	0.22	0.15

The following paragraphs present a detailed analysis of these fault types and subtypes, discussing the conditions/locations where they were observed in our field study, with examples when needed to clarify them.

Missing function call extended (MFCE). This fault type was observed in situations where there is a missing function returning a value that is used elsewhere in the code. This function is always related to the filtering of one of the arguments where the other arguments are the configuration of the filtering process. Next are the constraints of the subtypes A, B, and C:

A. *Missing casting to numeric of a variable* using a language specific function or type cast. Furthermore, we analyzed the situations where this subtype occurred in PHP and we found the use of the "`intval()`"function in 83 percent of the cases and the use of the "`(int)`" type cast in 17 percent of the cases (see Table 6). The function can also act as an argument of other functions. This situation was found when the patch added an entire assignment line, for example: `$var = int)$_GET[$var];` or when there was a replacement of one variable in a string concatenation. For example, replace: ..."'`str1'.$var. 'str2'`"; with ..."'`str1' .intval($var)`. 'str2'"; or in the case of a function: `$var1 = func(intval($var1))`;
B. *Missing assignment of a variable to a custom made function.* This subtype is similar to MFCE-A and was found in the same situations of MFCE-A, except that the filtering function was not a language specific predefined function and was instead custom made.

TABLE 6
MFCEA Situations Found

Situations found	Total #	Total (%)
intval()	243	81.54
intval() concatenated inside a string	3	1.01
(int) cast	47	15.77
(int) cast concatenated inside a string	5	1.68

C. *Missing assignment of a variable to a predefined function.* This subtype is similar to the MFCE-A and was found in the same situations of the MFCE-A, except that the filtering function is not one of those already present in the MFCE-A (casting to numeric).

Wrong variable used in parameter of function call (WPFV). This was typically found when the following changes occurred in the argument of a function:

A. *Missing quotes in variables inside a string argument of a SQL query.* For example, replace `func("SELECT... FROM...WHEREid = $var")` with `func("SELECT... FROM...WHERE id = '$var'")`
B. *Wrong regex string of a function argument.* When the patch code is the change in the regex string of a function argument. In the code analyzed, the regex string was used to check a variable closely related to an input value, looking for known suspicious values that can be part of an attack. For example, replace the vulnerable regex string REGEXP (`"^\.$id$|\.$id$"`) with `REGEXP` (`"^\\\.$id$|\\\.$id$"`)
C. *Wrong substring of a function argument.* When the argument of the function is the result of the concatenation of several strings and variables and the patch code removed or changed one of them. When the programming language converts the type automatically, the variable replaced may be of a different type than the new variable, for example, from a numeric type to a character type.
D. *Wrong variable when it is an argument of a function.* For example, replace: `func($var1)` with `func($var2)`

Missing if construct plus statements (MIFS). This type was found only when an IF condition and just one or two surrounding statements were missing:

A. *Missing traditional "if...then...else" condition.* When it is a traditional IF..THEN...ELSE condition, an ELSIF or an ELSE.
B. *Missing "if...then...else" condition in compact form.* This fault type was also found when the condition is in the compact form, for example: `(($var != '') ? "true" : "false")`

Wrong value assigned to variable (WVAV). This was typically found when the following situations changed the variable assignment:

A. *Missing pattern in a regex string assigned to a variable.* In the code analyzed, the regex string was used to check a variable closely derived from an input value, looking for known XSS attacks.

B. *Wrong value in an array or a concatenation of a new substring inside a string.* The patch changed one of the concatenation strings or removed one of the items of the array.
C. *Wrong variable assigned to a variable.* For example, replace `$var1 = $var2;` with `$var1 = $var3;`
D. *Missing quotes in variables inside a string in a SQL query assignment.* For example, replace `SELECT... FROM... WHERE id = $var` with `SELECT... FROM...WHERE id = '$var'`
E. *Missing destruction of the variable.* For example, unset ($var);
F. *Extraneous concatenation operator "." in an assignment.* For example, replace `$var. = ...` with `$var = ...`

From this analysis on PHP vulnerabilities, we observe that MFCE-A, which accounts for about half of all the vulnerabilities, can be mitigated directly by a strong type system. In fact, the MFCE-A represents the missing cast to numeric of a variable, which is unnecessary in strong typed languages, because these variables would be declared as numeric.

7 Analysis of XSS and SQLi Exploit Data

To characterize the relevance of the vulnerabilities analyzed, including the scope and criticality of possible attacks, we conducted another study, but this time focusing on their known exploits. The exploit is the code developed by hackers to attack a specific vulnerability (or a set of vulnerabilities) of the target system or application. The goal is to understand the correlation between the number of vulnerabilities and exploits, and the level of the exploit damage.

This analysis was done with data collected from the database of exploits found in the Milw0rm web site (milw0rm.com). The Milw0rm is a hacker-related site devoted to share exploits of vulnerabilities developed by several hacking groups and individuals. Its database contains around 10,000 exploits. The collection of exploits available contains attacks to vulnerabilities already fixed, as well as 0 day vulnerabilities, for which no solution is available yet. It is one of the most popular exploit databases and it is the largest that we are aware of. Indeed, many of the exploits available at the Milw0rm site are also distributed by other hacker- and security-related sites, like RedOracle (redoracle.com), osvdb (osvdb.org), SecurityReason (securityreason.com), and SecurityFocus (securityfocus.com). The well-known Metasploit framework (metasploit.com), widely used by hackers and developers for penetration testing and vulnerability detection, also has some modules based on exploits from the Milw0rm database.

The Milw0rm database contained 121 XSS and SQLi exploits for the same web applications we used in the vulnerability analysis already presented. These exploits were distributed in the following way: 118 for the six PHP web applications and three for the 11 strong typed web applications. This is shown in Table 7. A Pearson correlation was run to determine the relationship between XSS and SQLi exploits in each application. It showed a strong, positive correlation, which was statistically significant ($r = 0.588$, $n = 17$, $P < 0.013$).

TABLE 7
Vulnerabilities and Exploits Analyzed

Web application	Language	Type system	# Vulnerability		# Exploit	
			XSS	SQLi	XSS	SQLi
PHP-Nuke	PHP	Weak	137	158	1	29
Drupal	PHP	Weak	55	4	4	0
PHP-Fusion	PHP	Weak	33	21	1	23
WordPress	PHP	Weak	109	6	5	30
phpMyAdmin	PHP	Weak	73	1	0	2
phpBB	PHP	Weak	55	3	5	18
JForum	Java	Strong	1	0	0	1
OpenCMS	Java	Strong	4	0	0	1
JSPWiki	Java	Strong	17	0	1	0
Total by vulnerability type			**503**	**212**	**17**	**104**
Total			**715**		**121**	

Obviously, the number of vulnerabilities and exploits is not constant among web applications because the quality of the code, the hacker interest, and the number of vulnerabilities disclosed varies. Table 7 clearly shows that there is a huge difference in the number of exploits developed for the various web applications analyzed. For example, PHP web applications are clearly preferred by hackers to exploit. Moreover, we only found exploits for PHP and Java-based web applications. For the applications developed in VB and C#, we could not find any exploit, although applications developed with these programming languages are (obviously) also exploitable [55]. This does not mean that their vulnerabilities are neither exploitable nor exploited, only that they were not present in the source data analyzed. In fact, we were able to find other exploits targeting some of these web applications, attacking local file inclusion vulnerabilities, which we did not consider because they were outside the scope of our work.

To compare vulnerabilities and exploits for the same web applications, we run two Pearson correlations: one for XSS and another for SQLi. Both had a statistically significant strong correlation: ($r = 0.594$, $n = 17$, $P < 0.012$) for XSS and ($r = 0.596$, $n = 17$, $P < 0.012$) for SQLi. In fact, we cannot assert if the popularity of the exploits is due to the type of language or the popularity of the application, since PHP applications are, by far, more common.

Table 6 also shows that we have found some SQLi exploits in applications where we do not have SQLi vulnerabilities to analyze (JForum and OpenCMS). This occurs because both studies (the one where we collected vulnerabilities and the other of the exploits) were executed independently using different sources of information. This is the result of a lack of a unified repository that collects both vulnerabilities and exploits in a systematic and standardized fashion.

The exploits analyzed were developed between 2003 and 2009 (see Fig. 5). For the SQLi case, the peak was in 2008, but for XSS, we cannot see a clear trend. Comparing Fig. 5 with Fig. 1, we see that the growing number of XSS vulnerabilities does not have a correspondence with the exploits developed, whereas for SQLi, the exploit growth even overcomes the vulnerability trend, reinforcing the interest in exploiting

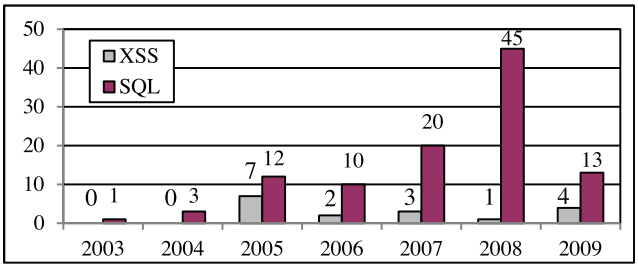


Fig. 5. Release date of the exploits.

this kind of vulnerability. We also see this from Table 7. This may be due to the fact that with SQLi, the attacker can access one of the most valuable assets of the enterprise: the database data. The database data may contain credit card numbers, account numbers, social security numbers, user names, passwords, e-mail accounts, and so on. These goods have a huge demand in the underground economy, which indicate that they have a higher cost/benefit ratio compared to other types of attacks [50].

To understand the importance of each exploit, we classified the criticality of the effect (in fact it represents the criticality of the vulnerability, from a management point of view) according to the Payment Card Industry Data Security Standard (PCI-DSS), widely used in e-commerce, e-banking, and other financial applications [40]. This standard uses five severity levels to classify the danger that the vulnerabilities pose to the enterprise, where levels 5 and 1 are the most and the least critical, respectively. To be compliant with the PCI-DSS, an application cannot have high-level vulnerabilities (levels 5, 4, or 3).

These data were obtained from the analysis of the source code of the exploits. According to the damage the exploit was able to inflict, we classified 96 percent of the vulnerabilities exploited as level 5. The remaining 4 percent were classified with level 3 that still belongs to the designated high-level vulnerabilities. From the source code of the exploits, we saw that the hacker always wants to target the most valuable asset. The vast majority of exploits allowed the attacker to either obtain the user name and password stored in the application back-end database, or to access the web server as the root or administrator of the machine.

Given that we could find a considerable number of exploits and that 96 percent of them are among the most critical, we can say that these results reinforce the importance of addressing XSS and SQLi vulnerabilities and the need to increase awareness about them.

8 Validity of the Results

The web applications analyzed are just a small sample of the whole population so, although most of the results have statistical significance, they may lack practical significance (they cannot be considered as representative) [38]. Our observations may not apply to other applications, even for those written with the same programming languages. There are many ways and tools to develop an application and they may influence the outcome. This can also be seen from our data, if we take into consideration the high standard deviation values that represents the data dispersion related to the number of vulnerabilities and exploits per application (see Table 8). Naturally, our results will fit better to applications developed with the same languages analyzed, but as improvements are being introduced to those languages results may also change. For example, the decline of malicious file execution attacks may be due to improvements in the fifth release of PHP [36].

TABLE 8
Number of Vulnerabilities and Exploits per Application

	Vulnerabilities		Exploits	
	XSS	SQLi	XSS	SQLi
Mean	29.59	12.47	1.00	6.12
Std	42.29	37.89	1.80	11.08

However, results from other studies [23], [31], [50], [55] are in line with ours, so we are confident that most of the trends shown will apply to similar applications, although the particular values may be quite different: weak typed with more vulnerabilities than strong typed, XSS easier to find, but SQLi with more valuable attacks, newer applications still with old school vulnerabilities like XSS and SQLi, few types of mistakes responsible for most of security problems with MFCE at the top, with most of vulnerabilities due to unchecked numeric fields.

9 Conclusion

This paper analyzes 715 vulnerabilities and 121 exploits of 17 web applications using field data on past security fixes. Some web applications were written in a weak typed language and others in strong typed languages. Results suggest that applications written with strong typed languages have a smaller number of reported vulnerabilities and exploits. We had to consider more strong typed applications to obtain a fair amount of vulnerabilities when compared to the weak typed.

According to our findings, weak typed are the preferred targets for the development of exploits. We also observed that a single fault type (MFCE) was responsible for most (76 percent) of the security problems analyzed. We saw that the fault types responsible for XSS and SQLi belong to a narrow list, which points a path to the improvement of web applications, namely in the context of code inspections and the use of tools for static analysis. This study showed that the way programmers fix vulnerabilities seems to have a degree of dependence with the type of language used. However, the number of vulnerabilities analyzed in our and other studies show that the use of a specific language is not guarantee of success in preventing vulnerabilities. It is just one of the many factors that contribute to building a safer application.

The most relevant fault types analyzed were thoroughly detailed providing enough information for the definition of vulnerability fault models that can be used by researchers interested, for example, in realistic vulnerability and attack injectors.

This work can be extended by comparing more vulnerabilities of web applications written in different languages and developed by independent programmers. Another

follow-up work may focus on the importance of the attack surface in the distribution of vulnerabilities and exploits. This may compare different results of vulnerabilities and exploits of both web applications and their add-ons, regarding their size, for example.

ACKNOWLEDGMENTS

This work was partially supported by the project "ICIS—Intelligent Computing in the Inter-net of Services" (CENTRO-07-ST24-FEDER-002003), cofinanced by QREN, in the scope of the Mais Centro Program and European Union's FEDER, and by the PEst-OE/EGE/UI4056/2011, financed by the Science and Technology Foundation.

REFERENCES

[1] Acunetix Ltd., "Is Your Website Hackable? Do a Web Security Audit with Acunetix Web Vulnerability Scanner," http://www.acunetix.com/security-audit/index/, May 2013.

[2] G. Álvarez and S. Petrovic, "A New Taxonomy of Web Attacks Suitable for Efficient Encoding," *Computers and Security,* vol. 22, no. 5, pp. 435-449, July 2003.

[3] P. Anbalagan and M. Vouk, "Towards a Unifying Approach in Understanding Security Problems," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 136-145, 2009.

[4] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.

[5] US-CERT Vulnerability Notes Database, "Homepage," http://www.kb.cert.org/vuls/, May 2013.

[6] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurement," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 943-956, Nov. 1992.

[7] S. Christey, "Unforgivable Vulnerabilities," *Proc. Black Hat Briefings,* 2007.

[8] J. Christmansson and R. Chillarege, "Generation of an Error Set That Emulates Software Faults," *Proc. IEEE Fault Tolerant Computing Symp.,* pp. 304-313, 1996.

[9] S. Clowes, "A Study in Scarlet, Exploiting Common Vulnerabilities in PHP Applications," http://www.securereality.com.au/studyinscarlet.txt, 2013.

[10] T. Manjaly, "C# Coding Standards and Best Practices," http://www.codeproject.com/KB/cs/c__coding_standards.aspx, May 2013.

[11] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences,* second ed., Lawrence Erlbaum, 1988.

[12] M. Cukier, R. Berthier, S. Panjwani, and S. Tan, "A Statistical Analysis of Attack Data to Separate Attacks," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 383-392, 2006.

[13] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J.C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner, and A. Wespi, "Conceptual Model and Architecture of MAFTIA," Project IST-1999-11583, https://docs.di.fc.ul.pt/jspui/bitstream/10455/2978/1/03-1.pdf, 2003.

[14] Dotnet Spider, "C# Coding Standards and Best Programming Practices," http://www.dotnetspider.com/tutorials/BestPractices.aspx, May 2013.

[15] J. Durães and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *Trans. Software Eng.,* vol. 32, pp. 849-867, 2006.

[16] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Pektov, *XSS Attacks: Cross Site Scripting Exploits and Defense.* Syngress, 2007.

[17] J. Fonseca, M. Vieira, and H. Madeira, "Training Security Assurance Teams Using Vulnerability Injection," *Proc. Pacific Rim Dependable Computing Conf.,* pp. 297-304, 2008.

[18] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 93-102, 2009.

[19] M. Fossi et al., "Symantec Internet Security Threat Report: Trends for 2010," Symantec Enterprise Security, 2011.

[20] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, "Modeling Security Requirements through Ownership, Permission and Delegation," *Proc. IEEE Int'l Conf. Requirements Eng.,* pp. 167-176, 2005.

[21] W. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *Proc. Black Hat Briefings,* 2005.

[22] L. Hatton, "The Chimera of Software Quality," *IEEE Software,* vol. 40, no. 8, pp. 104-103, Aug. 2007.

[23] M. Howard, D. LeBlanc, and J. Viega, "19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them," McGraw-Hill, 2005.

[24] IBM Global Technology Services "IBM Internet Security Systems X-Force® 2010 Trend & Risk Report," technical report, IBM Corp., 2011.

[25] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," *Proc. IEEE Symp. Security and Privacy,* pp. 27-36, 2006.

[26] C. Le Gues et al., "A Systematic Study of Automated Program Repair: Fixing 55 Out Of 105 Bugs for $8 Each," *Proc. Int'l Conf. Software Eng.,* pp. 3-13, 2012.

[27] B. Livshits and S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proc. USENIX Security Symp.,* pp. 18-18, 2005.

[28] F. Long, "Software Vulnerabilities in Java," Cert. technical note, Software Eng. Inst., Carnegie Mellon Univ., 2005.

[29] R. Mays, C. Jones, G. Holloway, and D. Strudinsky, "Experiences with Defect Prevention," *IBM Systems J.,* vol. 29, pp. 4-32, 1990.

[30] K. Mitnick and W. Simon, *The Art of Deception: Controlling the Human Element of Security,* first ed., Wiley, 2002.

[31] S. Christey and R. Martin, "Vulnerability Type Distributions in CVE," http://cwe.mitre.org/documents/vuln-trends/index.html, May 2007.

[32] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, M. Vouk, "Preliminary Results on Using Static Analysis Tools for Software Inspection." *Proc. Int'l Symp. Software Reliability Eng.,* pp. 429-439, 2004.

[33] S. Neuhaus and T. Zimmermann, "Security Trend Analysis with CVE Topic Models" *Proc. Int'l Symp. Software Reliability Eng.,* pp. 111-120, 2010.

[34] NTA, "Tests Show Rise in Number of Vulnerabilities Affecting Web Applications with SQL Injection and XSS Most Common Flaws," http://www.nta-monitor.com/posts/2011/03/01-tests_show_rise_in_number_of_vulnerabilities_affecting_web_applications_with_sql_injection_and_xss_most_common_flaws.html, May 2013.

[35] OSVDB, "Open Sourced Vulnerability Database," http://osvdb.org, May 2013.

[36] OWASP Foundation, "OWASP Top 10," https://www.owasp.org/index.php/Top_10_2010-Main, July 2010.

[37] A. Ozment, "Vulnerability Discovery & Software Security," PhD thesis, Computer Laboratory Computer Security Group, Univ. of Cambridge, 2007.

[38] J. Pallant, *SPSS Survival Manual,* fourth ed., Open Univ. Press, 2011.

[39] Packt Publishing Ltd., "Homepage," http://www.packtpub.com, May 2013.

[40] PCI Security Standards Council, "Payment Card Industry (PCI) Data Security Standard, Requirements and Security Assessment Procedures, version 1.2," www.pcidss.ru/files/pub/pdf/padss_v1.2_english.pdf, 2008.

[41] PHP-Nuke, "Homepage," http://phpnuke.org, Dec. 2007.

[42] The PHP Group, "Description of Core php.ini Directives," http://pt.php.net/register_globals, May 2013.

[43] The Privacy Rights Clearinghouse, "Chronology of Data Breaches: Security Breaches 2005-Present," http://www.privacyrights.org/data-breach, May 2013.

[44] W. Robertson and G. Vigna, "Static Enforcement of Web Application Integrity through Strong Typing," *Proc. 18th Conf. USENIX Security Symp. (USENIX '09),* pp. 283-298, 2009.

[45] SANS Inst., "Top 25 Most Dangerous Programming Errors," http://www.sans.org/top25errors/, May 2013.

[46] T. Scholte et al., "An Empirical Analysis of Input Validation Mechanisms," *Proc. ACM Symp. Applied Computing,* pp. 1419-1426, 2012.

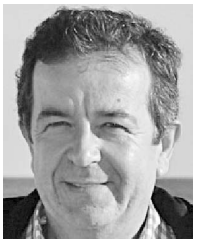
[47] Secunia, "Homepage," http://secunia.com, May 2013.

[48] Sourceforge, "2007 Community Choice Awards," http://sourceforge.net/blog/cca07, May 2013.
[49] D. Stuttard and M. Pinto, *The Web Application Hackers Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.
[50] Symantec, "Symantec Report on the Underground Economy," http://www.symantec.com/threatreport/topic.jsp?id=fraud_activity_trends&aid=underground_economy_servers. 2008.
[51] N. Tomatis, R. Brega, G. Rivera, and R. Siegwart, "'May You Have a Strong (-Typed) Foundation' Why Strong Typed Programming Languages Do Matter," *Proc. IEEE Int'l Conf. Robotics and Automation*, 2004.
[52] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," *Proc. Second Int'l Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '05)*, pp. 123-140, 2005.
[53] Verizon, "2011 Data Breach Investigations Report," http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2011_en_xg.pdf, 2011.
[54] J. Walden, M. Doyle, G. Welch, and M. Whelan, "Security of Open Source Web Applications," *Proc. Int'l Symp. Empirical Software Eng. and Measurement*, 2009.
[55] "WhiteHat Website Security Statistics Report, ninth ed.," https://www.whitehatsec.com/seekinfo/statsSpring10.html, WhiteHat Security Inc., Spring 2010.
[56] S. Zanero, L. Carettoni, and M. Zanchetta, "Automatic Detection of Web Application Security Flaws," *Proc. IEEE Int'l Symp. Secure Software Eng.*, 2005.

José Fonseca received the PhD degree in informatics engineering from the University of Coimbra, Portugal, in 2011. Since 2005, he has been with CISUC as a researcher. He has been teaching computer-related courses at the Polytechnic Institute of Guarda since 1993. He is the author or coauthor of more than a dozen papers in refereed conferences. His research on vulnerability and attack injection was granted with the DSN's William Carter Award of 2009, sponsored by the IEEE Technical Committee on Fault-Tolerant Computing and the IFIP Working Group on Dependable Computing and Fault Tolerance (WG 10.4).

Nuno Seixas received the master's degree in software engineering from Carnegie Mellon University and the University of Coimbra, Portugal, in 2008 and the MSc degree in informatics engineering from the University of Coimbra in 2007. From 2004 to 2006, he was with CISUC as a researcher. He has been with the Portugal Telecom Inovação company since 2005, where he is now part of the technological coordination group.

Marco Vieira is currently an assistant professor at the University of Coimbra, Portugal. He is an expert on dependability benchmarking and his research interests also include experimental dependability evaluation, fault injection, security benchmarking, software development processes, and software quality assurance, subjects in which he has authored or coauthored tens of papers in refereed conferences and journals. He has participated in many research projects, both at the national and European level. He has served on program committees of the major conferences of the dependability area and acted as a referee for many international conferences and journals in the dependability and databases areas.



Henrique Madeira is currently an associate professor at the University of Coimbra, Portugal, where he has been involved in the research on dependable computing since 1987. He has authored or coauthored more than 100 papers in refereed conferences and journals and has coordinated or participated in tens of projects funded by the Portuguese government and by the European Union. He was the program cochair of the 2004 International Performance and Dependability Symposium track of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-PDS 2004) and was appointed conference coordinator of IEEE/IFIP DSN 2008.