Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study

Nuno Seixas, José Fonseca, Marco Vieira, Henrique Madeira
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
naseixas@dei.uc.pt, josefonseca@ipg.pt, mvieira@dei.uc.pt, henrique@dei.uc.pt

Abstract—**This paper presents a field study on web security vulnerabilities from the programming language type system perspective. Security patches reported for a set of 11 widely used web applications written in strongly typed languages (Java, C#, VB.NET) were analyzed in order to understand the fault types that are responsible for the vulnerabilities observed (SQL injection and XSS). The results are analyzed and compared with a similar work on web applications written using a weakly typed language (PHP). This comparison points out that some of the types of defects that lead to vulnerabilities are programming language independent, while others are strongly related to the language used. Strongly typed languages do reduce the frequency of vulnerabilities, as expected, but there still is a considerable number of vulnerabilities observed in the field. The characterization of those vulnerabilities shows that they are caused by a small number of fault types. This result is relevant to train programmers and code inspectors in the manual detection of such faults, and to improve static code analyzers to automatically detect the most frequent vulnerable program structures found in the field.**

Keywords-Security vulnerabilities; software faults; programming languages; field study

I. INTRODUCTION

Web applications are frequently deployed with critical software bugs that can be maliciously exploited. These applications are so widely exposed that existing vulnerabilities will most probably be uncovered and exploited by hackers. To prevent vulnerabilities, developers should apply best coding practices, perform security reviews of the code, execute penetration tests, use code vulnerability analyzers, etc. However, many times developers focus on the implementation of functionalities and on satisfying the user's requirements and disregard security aspects. Additionally, numerous developers are not specialized on security and the common time-to-market constraints limit an in depth test for security vulnerabilities. Knowing the preponderant role of web applications in most organizations, one can realize the importance of finding ways to reduce the probability of deploying applications with security vulnerabilities.

Software bugs that are responsible for security vulnerabilities may have a devastating cost if exploited by hackers. Although configuration and human issues are also potential causes for vulnerabilities, the root cause of most security attacks are vulnerabilities created by software faults. Knowing the preponderant role of web applications in most organizations, one can realize the importance of finding ways to reduce the probability of deploying applications with security vulnerabilities.

Although there are many publications and periodic organization reports (e.g., Open Web Application Security Project Foundation) showing that web application vulnerabilities are a major concern, very few scientific studies have been focused on the detailed analysis of the fault types behind such vulnerabilities.

A recent field study [1] analyzed 655 security patches of six widely used web applications developed in PHP. The types of faults that are most likely to lead to software vulnerabilities were characterized, in order to better understand the potential relation between software defects and security vulnerabilities. Results show that a small subset of generic software faults is responsible for almost all the security problems studied (essentially, Cross Site Scripting (XSS) and SQL injection) and that there is a single fault type (Missing Function Call Extended[1]) that is responsible for 73% of all the security problems analyzed.

The problem is that the study presented in [1] is limited to web applications written in the PHP language. PHP is a dynamically typed language (weak typing) widely used in the development of web applications. Although we believe that the results from [1] are also valid for other weak typed programming languages (e.g., PERL, CGI), we consider that they cannot be applied to languages that use a different type system such as Java, C#, and VB.NET.

In the present work we analyzed 24 widely used open source web applications written in statically-typed languages (strong typing) in order to understand the most frequent types of software faults that lead to web security vulnerabilities such as XSS and SQL injection. It is worth mentioning that SQL injection and XSS are two of the most critical vulnerabilities in web applications [2]. The popularity of attacks exploiting these types of vulnerabilities is typically related to the easiness in finding and exploiting such vulnerabilities, the importance of the assets they can disclose, and the level of damage they may inflict.

All the vulnerabilities reported for the selected applications were carefully analyzed in order to understand and classify the software fault that made the code vulnerable.

[1] The Missing Function Call Extended fault type refers to a vulnerability caused by missing the use of a function to clean the values stored in the target variable.

The web applications used in the study are widely used open source applications, and some of them are actually used to support real business. All applications already have several released versions in which software defects were fixed and are not newcomers in the field.

Great care was taken to allow a fair comparison of the results between our current study and the previous study for PHP applications [1]. The goal is to help understanding security vulnerabilities from the point-of-view of the language used for the development of the applications.

The paper is structured as follows: section II presents some background in security vulnerabilities in programming languages focusing the language type system. Section III presents the methodology used in our field study, including the description of the applications used and the process followed to analyze and classify each vulnerability patch. Section IV presents the results and discusses lessons learned, and Section V concludes the paper.

II. Security Vulnerabilities in Web Programming Languages

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

Programming languages can be classified using several different taxonomies (e.g., programming paradigm, type system, execution mode, generation, etc). 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 those types, and how those types interrelate. Based on the type system, programming languages can be classified in the following way [3]:

- **Typed *vs* untyped**. A typed language defines for each operation the applicable data types (e.g., only numbers can be divided, only strings can be concatenated, dividing a number by a string is not possible). On the other hand, an untyped language (e.g., assembler) allows any data type to be used in any operation. In this case, all data types are understood as bits sequences that can be manipulated by any operation.
- **Static *vs* dynamic typed**. Static typing implies that all expressions must have their types defined before execution, typically during compilation (e.g., the sum of two integers cannot be stored in a date variable; an integer number cannot be passed as a parameter to a function that is expecting a string). In dynamic typing, operations are analyzed at runtime to determine and enforce their type-safety (i.e., types are associated based on actual values at runtime rather than based on the source code expression itself).
- **Weak *vs* strong typed**. In weak typed languages a value of one type can be treated as another type (e.g., a string can be treated as a number). Strong typed languages prevent this situation and an attempt to use a wrong type value in a given operation raises an exception.

In this work we are particularly interested in understanding the impact of the type system in terms of security vulnerabilities. This is of particular interest, as many critical security vulnerabilities like XSS and SQL Injection (see [2] for details on these vulnerabilities) are strongly related to the way the language manages data types. For example, SQL Injection attacks take advantage of improperly validated inputs to change the SQL commands that are sent to the database.

In dynamic typed languages it is sometimes possible to 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 static 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 will raise an error. However, this does not mean that SQL injection is not possible in static typed languages. In fact, it is indeed possible but only by taking advantage of variables of string-type, which reduces the number of variables through which a hacker can try to inject SQL code.

As mentioned before, the field study presented in this paper consists in identifying and classifying real security vulnerabilities detected and corrected in open source web applications developed using static strong typed languages. The results were analyzed and compared to the ones presented in [1] for PHP (a dynamically-weak typed language). The goal was to try to understand if the types of defects that lead to vulnerabilities are programming language independent. The rest of this paper presents the field study approach and the results obtained.

III. Field Study Methodology

The first step in our study consisted in selecting a set of web applications developed using strong typed languages and identifying the security vulnerabilities discovered and fixed in the different versions of those applications.

The identified vulnerabilities were analyzed and classified using the classification approach proposed in [1]. The results were then analyzed and compared to the ones presented in that study, in order to identify the impact of the language type system in the number and in the type of vulnerabilities found in the field.

A. Applications Studied

Our target applications were open source and with reported security vulnerabilities. The goal was to be sure that it was possible to have access to the source code (including the code from older versions) in order to be able to analyze and understand the security vulnerability and how it was fixed. Actually, the way a given vulnerability is fixed is a key aspect in the classification of the type of vulnerability. This is essential to assure that our classification is orthogonal and guarantees a valid comparison with previous studies [1]. Typically, a vulnerability can be fixed in more than one way.

However, it is the way the programmer actually used to fix the vulnerability that is considered for its classification. This allows us to classify each vulnerability as being of a single type.

As one of our goals was to compare the results from this study with the ones presented in [1], we have considered primarily web applications from the same domains used in that study. This way, we focused our search in the following types of applications: Bloggers, Content Management Systems (CMS), Forum Software, Issue Tracking, Portals, Webmail and Wiki Engines.

The site "Open Source Software in Java" [4] was particularly useful as entry point for the process of identifying the web applications for this study, as the most representative open source applications written in Java are registered and described in this web site.

After identifying a large set of applications in the domains mentioned before, we started searching for security vulnerabilities using three well know repositories: Security Focus [5], OSVDB [6], and Secunia [7]. Initially, we were expecting to identify a large number of vulnerabilities in applications developed using this language. However, unlike in [1] where six web applications accounted for the 655 vulnerabilities studied, we found a very low number of reported vulnerabilities per application.

Thus we decided to extend our study to incorporate also applications developed using C# and VB.NET (found in the "Open Source Software in C#" web site [8]), which have the same data type system as Java. Although the total number of vulnerabilities identified increased, it still remained quite low, at least when comparing to the results presented in [1] (see Section IV for details).

Although we analyzed a total of 24 web applications, we were able to find vulnerabilities descriptions for only 11 applications. This way, our study focused on the following set of applications with reported vulnerabilities (see more details on this on section IV.C): JForum [9], OpenCMS [10], BlojSom [11], Roller WebLogger [12], JSPWiki [13], SubText [14], DotNetNuke [15], YetAnotherForum [16], BugTracker .NET [17], Deki Wiki [18], ScrewTurn Wiki [19].

It is important to reemphasize that many of these applications are widely used, including in the support of real businesses. All applications already have several released versions and are not newcomers to the field.

B. Patch Analysis and Vulnerability Classification

For all the applications analyzed, we collected the source code of both the vulnerable version and the patched version. By comparing these two versions we could understand the vulnerability and classify what code has been changed to correct it.

To gather information on the security patches (including source code) we used mirror websites, sites with source code, online reviews, news sites, sites related to security, changelog files of the application, the version control system repository, etc. Finding the source code of old versions is usually a very difficult task that requires searching in different sources. However, for the purpose of this study, we only needed the original piece of code and the piece of the code that corrected the vulnerability (i.e., the source code of the entire application is not required). The two main types of sources used were:

- **The version control system repository**: most of the applications analyzed have their source code completely available through SVN or CVS servers that are publicly accessible.
- **The web site of the application**: some of the applications had all the versions available in their web sites, ready for download.

Once the source code was obtained, a differential analysis was performed to identify the locations in the code where the faults were fixed. This operation was done through the use of diff tools and manual analysis of the code.

The software faults that generated the detected vulnerabilities were classified using the classification approach proposed in [1]. Table I summarizes the fault types considered. It is important to emphasize that the fault types in Table I are used in this paper as reference for the fault classification. These types of faults are simply the most common types of faults observed in [1] and are not discussed in this paper in the context of strong/weak typing language. They are simply used as starting point for fault classification.

To allow meaningful comparison between both studies, the guidelines for classifying the security vulnerabilities were exactly the same ones used in [1], namely:

1. When the patch can fix both XSS and SQL Injection the corresponding fault type is accounted for both security vulnerabilities.

TABLE I. FAULT TYPES CLASSIFICATION

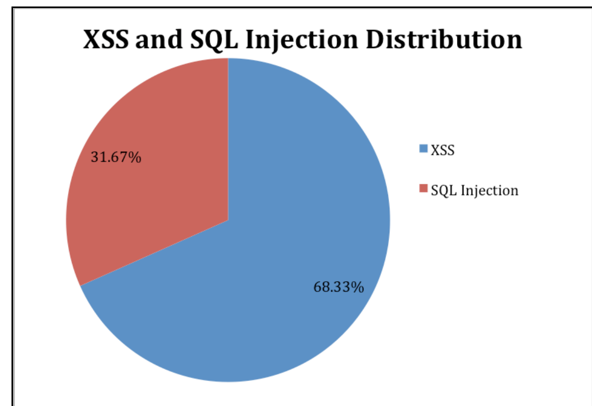
Fault type	Description
MFC	Missing function call
MFC extended	Missing function call returning the same data type as the argument
MVIV	Missing variable initialization using a value
MIA	Missing if construct around statements
MIFS	Missing if construct plus statements
MIEB	Missing if construct plus statements plus else before statements
MLPA	Missing small and localized part of the algorithm
WPFV	Wrong variable used in parameter of function call
WLEC	Wrong logical expression used as branch condition
EFC	Extraneous function call

2. It is assumed that the information publicly disclosed in specialized sites is accurate and that the fix made by the programmer of the patch and made available by the provider that develops the web application solves the stated problem.
3. To correct a single vulnerability several code changes may be necessary and they are counted as several vulnerabilities. All the changes will be considered as a series of singular fault type fixes. For example, suppose that two functions are needed to properly sanitize a variable. Missing any of these functions makes the application vulnerable, so both of them must be taken into account.
4. When a particular code change corrects immediately several vulnerabilities, each one is considered as a singular fix.
5. A security vulnerability may affect several versions of the application but the fix is accounted only for one.

IV. RESULTS AND DISCUSSION

The goal of this practical experience report is to report a field study focusing on SQL Injection and XSS vulnerabilities in web applications developed using strong typed languages. We analyzed and classified the faults the lead to these two types of vulnerabilities, using the methodology presented above. The field study was agnostic concerning the types of faults; that is, we simple classified all faults (that originated SQL Injection and XSS vulnerabilities) reported for the open source applications presented in Section III.A and we did not focus on selecting a specific subtype of faults (e.g., faults related to the strong/weak typing aspects). In other words, the goal is not to analyze only faults strictly related to the strong/weak typing features, but to provide a field study on faults reported in applications written in strong typed languages and compare the results (i.e., all faults observed) with the observations of [1].

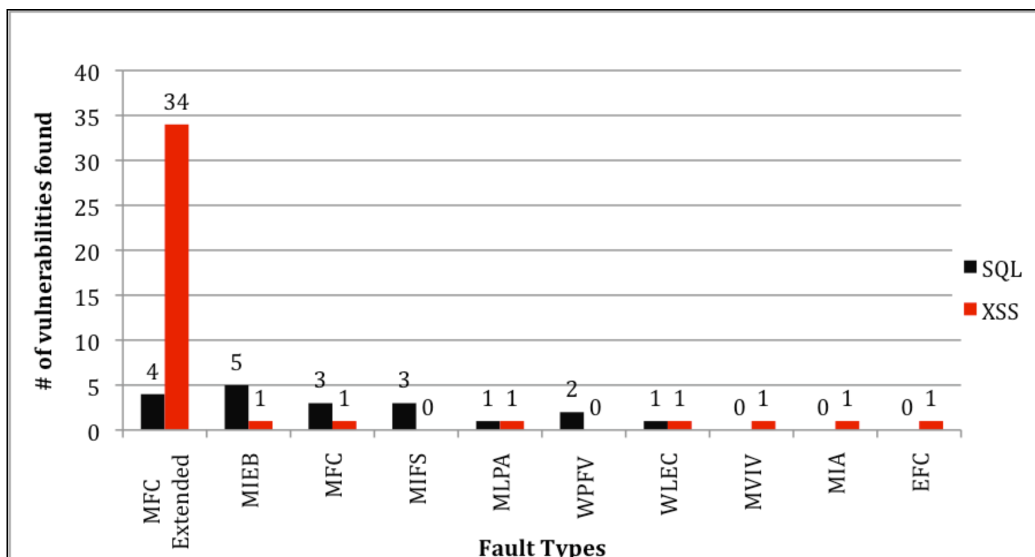
Figure 1. XSS vs SQL Injection Vulnerabilities



We have collected and classified 60 XSS and SQL Injection security vulnerabilities, distributed over 11 different web applications. XSS is the most frequent type of vulnerability observed in our sample, accounting for 68.33% of the vulnerabilities analyzed, while SQL Injection corresponds to 31.67% (see Fig. 1). Comparing this result with the distribution observed in [1] for web applications written using a weakly-typed language, we can conclude that the distribution is similar. In fact, the distribution observed in [1] was 70% for XSS against 29.47% for SQL Injection. This result also confirms different CVE reports [2][20] that point out XSS as the most frequent type of web security vulnerability.

The detailed distribution of vulnerabilities by fault types is presented in Fig. 2. As shown, the most frequent fault type was MFC Extended, which corresponds to a missing function call returning the same data type as the argument.

Figure 2. Vulnerabilities fault type summary



This fault type is normally associated to the use of a function responsible for the sanitization of an input. For example, in ScrewTurn Wiki, the code:

```
user = user.Replace("\r","").Replace("\n","");
```

was replaced by:

```
user = Sanitize(user);
```

This fault type is the most frequent one because the majority of vulnerabilities are due to inputs not validated or not properly sanitized, and so, the obvious fix is to implement functions that simply clean and validate the input received from the user. Another interesting fact is that most of these fixes are related to XSS vulnerabilities, which by definition are prone to this kind of error.

A. Strong Typing vs Weak Typing

Table II shows the fault types responsible for the vulnerabilities observed in our current field study on web applications written in statically typed languages (strong typing) and the results of the previous study on dynamically typed languages (weak typing).

The MFC extended (Missing Function Call extended) type of software fault is clearly the main cause of web application vulnerabilities, no mater the type system of the programming language used (63.33% of the faults in our current study and 75.87% in the previous study [1]). In other words, the occurrence of MFC extended software faults seems independent from the type system of the programming language used. However, nothing can be concluded regarding language independence, as we do not have enough data to confirm that.

The MFC extended type of fault is also responsible for both SQL injection and XSS vulnerabilities, although its prevalence in XSS is even more evident (82.93% in strong typed languages and 77.27% in weak typed).

Looking at the results presented in Table II, the big differences between weak typing and strong typing concern the faults classified in second and third place, MIEB (Missing if construct plus statements plus else before statements) and MFC (Missing function call). While in weak typed applications, the second and third positions were WPFV (Wrong variable used in parameter of function call) and MIFS (Missing if construct plus statements), in our study we have found for these positions the MIEB and MFC fault types.

MIEB shows that the fixes were not only based on the introduction of an "if" statement, which would correspond to the MIFS found in the weak typed. This kind of fix can be interpreted in two ways: (i) there were some more complex algorithm steps to be performed or (ii) the team was more conservative and followed the best practices recommendations that state the need for having complete "if" and "else" statements [21][22]. This is currently accepted as good practice when working with strong typed languages and explains why the MIEB appears in second place, against the third place for MIFS in the weak typed applications.

Another major difference lays in the fact that WPFV (Wrong variable used in parameter of function call) represents only 3.33% in the current study, while in the weak

TABLE II. DISTRIBUTION OF FAULT TYPES PER VULNERABILITIES

Fault type	Strong type (Java, C#, VB.Net)				Weak dynamic type (PHP) [1]			
	# Faults	SQL Inj. (%)	XSS (%)	SQL + XSS (%)	# Faults	SQL Inj. (%)	XSS (%)	SQL + XSS (%)
MFC Ext	38	21.05%	82.93%	63.33%	497	72.54%	77.27%	75.88%
MIEB	6	26.32%	2.44%	10.00%	0	0.00%	0.00%	0.00%
MFC	4	15.79%	2.44%	6.67%	4	0.52%	0.65%	0.61%
MIFS	3	15.79%	0.00%	5.00%	34	6.22%	4.76%	5.19%
MLPA	2	5.26%	2.44%	3.33%	0	0.00%	0.00%	0.00%
WPFV	2	10.53%	0.00%	3.33%	46	17.10%	2.81%	7.02%
WLEC	2	5.26%	2.44%	3.33%	0	0.00%	0.00%	0.00%
MVIV	1	0.00%	2.44%	1.67%	9	0.52%	1.73%	1.37%
MIA	1	0.00%	2.44%	1.67%	2	0.00%	0.43%	0.31%
EFC	1	0.00%	2.44%	1.67%	6	0.52%	1.08%	0.92%
MVAE	0	0.00%	0.00%	0.00%	0	0.00%	0.00%	0.00%
MLAC	0	0.00%	0.00%	0.00%	9	1.04%	1.52%	1.37%
MVAV	0	0.00%	0.00%	0.00%	0	0.00%	0.00%	0.00%
WVAV	0	0.00%	0.00%	0.00%	28	1.04%	5.63%	4.27%
WFCS	0	0.00%	0.00%	0.00%	18	0.52%	3.68%	2.75%
MLOC	0	0.00%	0.00%	0.00%	1	0.00%	0.22%	0.15%
WAEP	0	0.00%	0.00%	0.00%	0	0.00%	0.00%	0.00%
ELOC	0	0.00%	0.00%	0.00%	1	0.00%	0.22%	0.15%
Total	**60**	100%	100%	100%	**655**	100%	100%	100%

typed applications [1] it represents 7.02%, being the second most frequent fault type. This shows that the number of programming faults resulting in the use of wrong variables in strong typed languages (our current study) is lower than the one observed in weak typed, suggesting that the main problems are due to inputs not correctly/completely sanitized and not caused by the use of wrong variables 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 weakly typed one variable can handle different types of data, especially in dynamic typed languages like PHP (the one used in [1]). This means that even when the programmer uses a variable of a wrong type, the mistake will not be detected at compile time in weak typed languages. However, in strong typed languages, the programmer needs to use a variable declared for that specific type of data and so, the probability of using wrong variables is much lower.

B. Vulnerability Analysis: XSS & SQL Injection

Looking at XSS column in Table II we can see that the most frequent fault type is MFC extended, with a percentage of 82.93%. This can be explained by the fact that the most important way of exploiting this vulnerability is through the input of special data into the application. So, if the application has a way to sanitize and validate this input, the vulnerability will not be present anymore. Also, in the majority of cases, this kind of fault type was fixed by the introduction of a sanitizing function, to clean up the input received from the user.

Concerning the SQL Injection vulnerability, we can see in Table II that vulnerabilities have been caused by different types of faults, as we have not observed the preponderance of a single type of software fault (as was the case for XSS).

The most frequent fault type that causes SQL injection vulnerabilities is MIEB with 26.32%, followed by MFC Extended with 21.05%. This shows that the lack of input sanitization is not the most frequent problem. The fix of this kind of security vulnerabilities must be done not only by sanitizing the user input but also by the verification of other application states, through the introduction of "if… else…" statements.

C. Lessons Learned

From the results discussed in the previous two sections, we can summarize the main differences observed in vulnerabilities found in the field in applications written with strong typed and weak typed languages.

The first difference comes right from the number of detected security vulnerabilities. While in [1], the authors identified 655 security vulnerabilities in only 6 applications; in this study we identified 60 security vulnerabilities in 11 applications, taken from an initial set of 24 web applications. That is, we could not find any recorded vulnerability for 13 of the web applications analyzed. Obviously, this does not mean that these 13 applications have not had vulnerabilities fixed: we just could not find any vulnerability description recorded for these applications.

The general reading of these results is that strong typed languages do contribute to decrease the frequency of SQL injection and XSS vulnerabilities in application code. In fact, the strong typed languages, which, by definition have variables defined for a specific type, make the use of wrong variables less probable. But strong typed languages, by themselves, do not eliminate the need to validate and sanitize all the inputs from the user, in order to avoid the introduction of executable instructions (XSS) or SQL queries (SQL Injection). This is clearly corroborated by our field results, as we still can find vulnerabilities in web applications written in strong typed languages.

Another interesting analysis can be made based on the age of the applications. The majority of the vulnerabilities that we analyzed were identified after 2005. By this time, the concern about XSS and SQL Injection vulnerabilities was already disseminated throughout the world, which can be confirmed by the publication of several studies in this area [23][24][25][26][27]. So, when the analyzed applications were implemented, the technical community was already concerned about these problems, and could start addressing it earlier in the application development roadmap. The fact that we can still find a significant number of vulnerabilities reported in the field shows that producing secure code is far from being trivial.

The results presented in this paper also show an important feature: the vulnerabilities found in the field are not caused by a large diversity of software fault types. On the contrary, there is a small set of fault types that is responsible for most of the vulnerabilities, as already observed in section IV. This fact can be used to train programmers, focusing their attention on the correct treatment of the program structures related to the most frequent types of faults. Additionally, the knowledge of the most common types of software faults that lead to SQL injection and XSS vulnerabilities can also be useful to improve effectiveness of code inspections, as the inspection team will be aware of the fact that most vulnerabilities are caused by a small number of faulty program structures.

Static code analyzes tools can also benefit from the identification of the most common types of faults observed in our field study, as the tools can be optimized to improve detection of such vulnerabilities.

V. CONCLUSION

In this study, we analyzed open source web applications, written with strong typed languages (Java, C# and VB.NET) focusing on security vulnerabilities. For this analysis, we used the methodology presented in [1]. Comparing the results from the two studies, we can conclude that the use of strong typed languages does influence security vulnerabilities. Applications written with strong typed languages seem to have a smaller number of reported vulnerabilities.

In this study we found 60 vulnerabilities in 11 applications, while in [1] the authors found 655 in 6 applications. We can also state that the fault types identified in both studies belong to a narrow list, which points a path to

improve web applications, namely in the context of code inspections and the use of tools for static analysis [28].

This study also showed that the way the programmers fixed the reported vulnerabilities has some differences, depending on the language used.

REFERENCES

[1] Fonseca, J., Vieira, M.: "Mapping Software Faults with Web Security Vulnerabilities", IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, USA, June, 2008

[2] Stock, A., Williams, J., Wichers, D., "OWASP top 10", OWASP Foundation, July, 2007

[3] Tomatis, N., Brega, R., Rivera, G., Siegwart, R., "May you have a strong (-typed) foundation, Why strong-typed programming languages do matter", IEEE International Conference on Robotics and Automation, New Orleans, LA, USA, April 26-May 1, 2004

[4] Open Source Software in Java, December, 2008, *http://java-source.net/*

[5] Security Focus, December, 2008, *http://www.securityfocus.com/*

[6] OSVDB: The Open Source Vulnerability Database, December, 2008, *http://osvdb.org*

[7] Secunia, December, 2008, *http://secunia.com*

[8] Open Source Software in C#, December, 2008, *http://csharp-source.net/*

[9] JForum, December, 2008, *http://www.jforum.net/*

[10] OpenCMS from Alkacon Software, December 10th, 2008, *http://www.opencms.org/*

[11] BlojSom, December, 2008, *http://wiki.blojsom.com/wiki/display/blojsom3/About+blojsom*

[12] The Roller WebLogger, December, 2008, *http://rollerweblogger.org/project/*

[13] JSPWiki, December, 2008, *http://www.jspwiki.org/*

[14] SubText, December, 2008, *http://subtextproject.com/*

[15] DotNetNuke, December, 2008, *http://www.dotnetnuke.com/*

[16] YetAnotherForum, December, 2008, *http://www.yetanotherforum.net/*

[17] BugTracker .NET, December, 2008, *http://www.ifdefined.com/bugtrackernet.html*

[18] Deki Wiki, December, 2008, *http://wiki.developer.mindtouch.com/*

[19] ScrewTurn Wiki, December, 2008, *http://www.screwturn.eu/Default.aspx?AspxAutoDetectCookieSupport=1*

[20] Steve, C., Martin, R., " Vulnerability Type Distributions in CVE", Mitre report, May, 2007

[21] C# Coding Standards and Best Practices, December 2008, *http://www.codeproject.com/KB/cs/c__coding_standards.aspx*

[22] C# Coding Standards and Best Programming Practices, December 2008, *http://www.dotnetspider.com/tutorials/BestPractices.aspx*

[23] Valeur, F., Mutz, D., Vigna, G.: "A Learning-Based Approach to the Detection of SQL Attacks", IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005), Vienna, Austria, July, 2005

[24] Christey, S., "Unforgivable Vulnerabilities", Black Hat Briefings, 2007

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

[26] David, P., Stroud, R., "Conceptual Model and Architecture of MAFTIA", LAAS-CNRS, 2003

[27] Jovanovic, N., Kruegel, C., Kirda, E., "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities", IEEE Symposium on Security and Privacy, 2006

[28] Nagy, C., Mancoridis, S., "Static security analysis based on input-related software faults", 13th European Conference on Software Maintenance and Reengineering (CSMR'09), Kaiserslautern, Germany, March, 2009