Java vs. PHP: Security Implications of Language Choice for Web Applications

James Walden, Maureen Doyle, Robert Lenhof, and John Murray

Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099

Abstract. While Java and PHP are two of the most popular languages for open source web applications found at `freshmeat.net`, Java has had a much better security reputation than PHP. In this paper, we examine whether that reputation is deserved. We studied whether the variation in vulnerability density is greater between languages or between different applications written in a single language by comparing eleven open source web applications written in Java with fourteen such applications written in PHP. To compare the languages, we created a Common Vulnerability Metric (CVM), which is the count of four vulnerability types common to both languages. Common Vulnerability Density (CVD) is CVM normalized by code size. We measured CVD for two revisions of each project, one from 2006 and the other from 2008. CVD values were higher for the aggregate PHP code base than the Java code base, but PHP had a better rate of improvement, with a decline from 6.25 to 2.36 vulnerabilities/KLOC compared to 1.15 to 0.63 in Java. These changes arose from an increase in code size in both languages and a decrease in vulnerabilities in PHP. The variation between projects was greater than the variation between languages, ranging from 0.52 to 14.39 for Java and 0.03 to 121.36 in PHP for 2006. We used security and software metrics to examine the sources of difference between projects.

Key words: web application security, security metrics, open source

1 Introduction

While Java and PHP are two of the most popular languages for open source web applications found at Freshmeat [**?**], they have quite different security reputations. In this paper, we examine whether the variation in vulnerability density is greater between languages or between different applications written in a single language. We compare eleven open source web applications written in Java with fourteen such applications written in PHP. We also analyzed the source code of two different revisions of each of these applications to track the evolution of vulnerability density over time. PHP applications included both PHP 4 and PHP 5 code. The Java applications were compiled with Sun Java SE 6, but the 2006 versions of some applications had to be compiled with Sun Java SE 5.

Despite differences in security reputations, more than twice as many open source web applications are written in PHP than Java, and twelve of the fourteen PHP applications studied are more popular than any of the Java applications [?]. In part, PHP's poor security reputation [?] arises from default language features enabled in earlier versions of the language. However, these features have gradually been turned off as defaults or removed from the language. For example, the *register globals* feature which automatically created program variables from HTTP parameters was turned off as default in PHP 4.2 and removed in PHP 6.

We measured security through the number of vulnerabilities of types common to both languages as reported by a static analysis tool. Static analysis tools find common secure programming errors by evaluating source code without executing it. Static analysis has the advantage of being repeatable and checking all parts of the code equally, unlike human code reviewers or vulnerability researchers. The objective nature of static analysis makes it suitable for comparing different code bases, though, like human reviewers, static analysis tools make mistakes at times. We computed code size and complexity metrics and also a security resources indicator metric [?] to examine the source of differences between projects.

We discuss related work in section 2 and study design in section 3. Overall results are described in section 4, with section 5 analyzing results by vulnerability type. Sections 6 and 7 examine software and security metrics to determine the causes of differences between applications. Limitations of our analysis are discussed in section 8. Section 9 finishes the paper, giving conclusions and describing future work.

2 Related Work

Coverity used their Prevent static analysis tool to analyze a large number of open source projects written in C and C++ [?], using the static analysis vulnerability density metric. Fortify analyzed a small number of Java projects [?] with their static analysis tool, using the same metric. Nagappan used static analysis tools to measure defect density [?] to predict post-release defects. Note that defect density may not correlate with vulnerability density, as security flaws differ from reliability flaws.

Ozment and Schechter [?] and Li et. al. [?] studied how the number of security issues evolves over time. Ozment found a decrease in OpenBSD, while Li found an increase in both Mozilla and Apache.

Shin [?] and Nagappan et. al. [?] analyzed correlations of cyclomatic complexity with vulnerabilities. They had mixed results, with Shin finding a weak correlation for Mozilla and Nagappan finding three projects out of five having strong correlations. Shin also analyzed nesting complexity, finding significant but weak correlations with vulnerabilities for Mozilla.

Neuhaus and Zimmerman [?] studied the effect of dependencies on vulnerabilities in several thousand Red Hat Linux packages. Zimmerman et. al. [?] analyzed the problem of predicting defects based on information from other projects,

finding that only 3.4% of cross-project predictions were both significant and had strong correlation coefficients.

3 Study Design

We examined the project history of 25 open source web applications, eleven of which were written in Java, fourteen of which were written in PHP. The applications are listed in table 1.

Table 1. Open Source Web Applications

Java			PHP		
alfresco	contelligent	daisywiki	achievo	obm	roundcube
dspace	jackrabbit	jamwiki	dotproject	phpbb	smarty
lenya	ofbiz	velocity	gallery2	phpmyadmin	squirrelmail
vexi	xwiki		mantisbt	phpwebsite	wordpress
			mediawiki	po	

To be selected, an application had to have a source code repository with revisions ranging from July 2006 to July 2008. The selected applications were the only applications that had revisions from those periods that could be built from source code in their repositories. While most third-party PHP libraries can be found in the PEAR or PECL repositories, third-party Java libraries are scattered among a variety of sites. Java developers often use tools like Maven to retrieve third-party software and manage builds.

Eight Java applications were not included in the study because they could not be built due to missing third-party software. Some older revisions used repositories of third-party tools that no longer existed, in which case we modified the Maven configuration to point to current repositories. This approach succeeded in some cases, but failed in others, as current Maven repositories do not contain every software version needed by older revisions. Some projects used other techniques to fetch dependencies, including ivy and custom build scripts.

Only five of the PHP projects and none of the Java projects maintained a public vulnerability database or had a security category in its bug tracker. While there were 494 Common Vulnerabilities and Exposures (CVE) listings for the PHP projects, there were only six such listings for the Java projects. The number of CVE entries does not necessarily indicate that a project is more or less secure. Due to the sparse and uneven nature of this data, documented vulnerabilities could not be used to measure the security of these applications. Instead, we used static analysis to find vulnerabilities in the source code of these applications.

We used Code Analyzer to compute SLOC, cyclomatic complexity, and nesting complexity for Java, and SLOCCount and Code Sniffer for PHP. We used Fortify Source Code Analyzer version 5.6 for static analysis. While there is no release quality free PHP static analysis tool, two of the Java web applications used

the free FindBugs [?] static analysis tool. No web application showed evidence of use of a commercial static analysis tool in the form of files in the repository (which is how we identified use of FindBugs) or web site documentation.

Vulnerability density can be measured using the static analysis vulnerability density (SAVD) metric [?], which normalizes vulnerability counts by KSLOC (thousand source lines of code.) However, Fortify finds 30 types of vulnerabilities for Java and only 13 types for PHP in our set of applications, which prevents SAVD from being compared directly between the two languages. Since only four vulnerability types are shared between the two groups of applications we studied, we created a common vulnerability metric (CVM), which is the sum of those four vulnerability types, to more accurately compare results between Java and PHP. Common vulnerability density (CVD) is CVM normalized by KSLOC.

The four common vulnerability types were cross-site scripting, SQL injection, path manipulation, and code injection. Three of the four types are in the top five application vulnerabilities reported by MITRE in 2007 [?]. The two missing types from MITRE's top five are PHP remote file inclusion, which is found only in PHP, and buffer overflows, which are found in neither language.

4 Results

Examining the aggregate code base of the fourteen PHP applications, we found that common vulnerability density declined from 6.25 vulnerabilities/KSLOC in 2006 to 2.36 in 2008, a decrease of 62.24%. Over the same period, CVD declined from 1.15 in to 0.63 in the eleven Java applications, a decrease of 45.2%. Common vulnerabilities in PHP declined from 5425 to 3318, while common vulnerabilities increased from 5801 to 7045 in Java. The decrease in density for Java is the result of a tremendous increase in code size, from 5 million to 11 million SLOC. The expansion of the PHP code base was much smaller, from 870,000 to 1.4 million SLOC.

Java projects were larger on average than PHP projects. While one Java project, xwiki, had over a million of lines of code, the other ten Java projects ranged from 30,000 to 500,000 lines. The largest PHP project had 388,000 lines, and the smallest had under 6,000 lines, with the other twelve ranging from 25,000 to 150,000 lines. This difference tends to support the contention that PHP requires fewer lines of code to implement functionality than Java, especially as projects implementing the same type of software, such as wikis, were smaller in PHP than Java.

If we compare all vulnerability types, including all 30 categories of Java vulnerabilities and 13 categories of PHP vulnerabilities, we find that the vulnerability density of the Java code base decreased from 5.87 to 3.85, and PHP decreased from 8.86 to 6.02 from 2006 to 2008. The total number of PHP vulnerabilities increased from 7730 to 8459, while the total number of Java vulnerabilities increased from 29,473 to 42,581.

CVD varied much more between projects than between languages. In 2006, PHP projects ranged from 0.03 to 121.4 vulnerabilities/KLOC while Java projects

had a much smaller range from 0.52 to 14.39. In 2008, both ranges shrank, with PHP projects varying from 0.03 to 60.41 and Java projects ranging from 0.04 to 5.96. Photo Organizer (po) had the highest CVD for both years. Figures 1 and 2 show change in vulnerability density between the initial and final revision for each project. In sections 6 and 7, we examine some possible sources of these differences between projects.

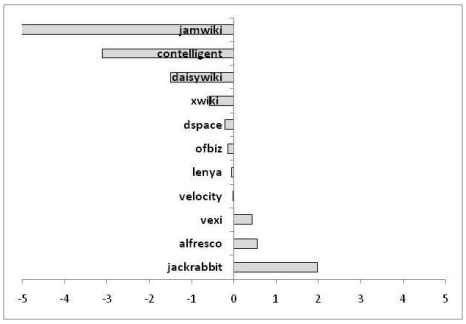

Fig. 1. Change in CVD for Java

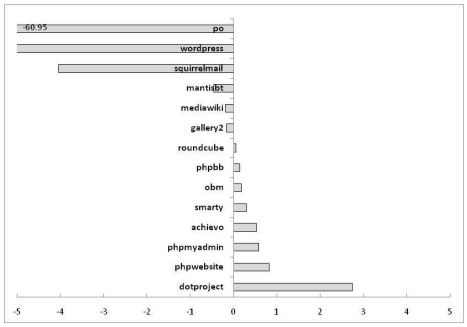

Fig. 2. Change in CVD for PHP

5 Vulnerability Type Analysis

In this section, we examine the four vulnerability types that make up the CVM: cross-site scripting, SQL injection, path manipulation, and command injection. Figure 3 shows the changes in each vulnerability type between 2006 and 2008 for the aggregate Java and PHP code bases. The number of vulnerabilities in all four categories increased for Java, while they decreased for PHP.

Individual projects did not follow these overall patterns; two Java projects, contelligent and jamwiki, had reductions in three of the four categories. No Java project reduced the number of command injections. Two projects, alfresco and jackrabbit, did not reduce the number of vulnerabilities in any category.

Despite the overall decrease for PHP, nine of the fourteen PHP applications increased CVD. Two projects showed small decreases, while the remaining three contributed the bulk of the vulnerability reductions: photo organizer, squirrelmail, and wordpress. Photo organizer is the only PHP project that saw a reduction in all four error types. Eight of the remaining PHP projects increased cross-site scripting errors, and nine increased path manipulation errors.

We also examined the contribution of each vulnerability type to the overall CVM and how that changed over the two years. Figure 4 compares the percentage contribution of each of the four vulnerabilities to the total CVM for Java and PHP projects in 2006 and 2008.

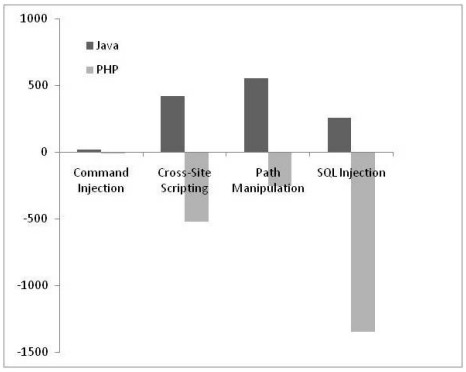

Fig. 3. Type Contribution to CVM

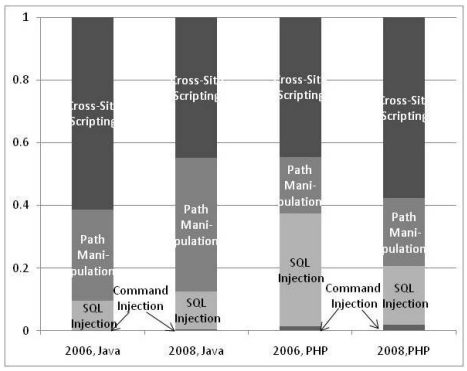

Fig. 4. Type Changes: 2006-2008

The 2008 ranking of the contributions of each error type for both languages and both years are the same: cross-site scripting, followed by path manipulation, SQL injection, and Command Injection. The total number of command injections is tiny compared to the other three types, which are found in MITRE's top five. The majority of the PHP change resulted from removing SQL injection flaws. Cross-site scripting vulnerabilities showed the largest decrease in Java, though the change was not as dramatic as the SQL injection reduction in PHP.

6 Software Metric Analysis

Based on prior work and research [?,?,?,?,?,?], we selected software metrics which had demonstrated correlations to vulnerability or defect density: cyclomatic complexity (CC) and nesting complexity. We used the same metric definitions as in [?], including three variants of each complexity metric: average, total, and maximum. Average is computed per-function for PHP and per-class for Java. While PHP 5 supports classes, these applications organized their code primarily with functions.

Figure **??** displays the correlations of metrics to CVD for both revisions. Correlation was computed using the Spearman rank correlation coefficient (ρ) since no assumptions can be made about the underlying distributions.

Significant correlations were found for maximum cyclomatic complexity and nesting complexity with change in CVD over the two year period ($p = 0.02$) for Java projects, but no correlations are signficant for the remaining metrics. While total code complexity is an indicator of changes in vulnerability density for Java projects, there are no significant correlations between software metrics and CVD for PHP projects.

We also compared change in metric values over the time period with change in CVD. We found only one signficant correlation; CVD is negatively correlated with SLOC for PHP projects. Since CVD decreased with time for this group of projects while SLOC increased, this result is not unexpected.

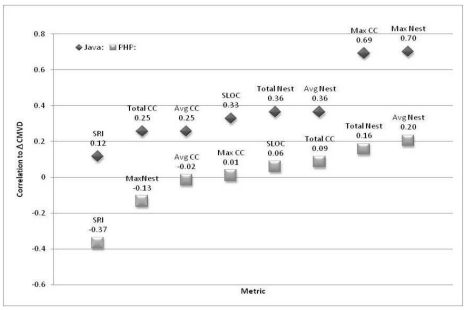

Fig. 5. ΔMetric correlations to ΔCVD

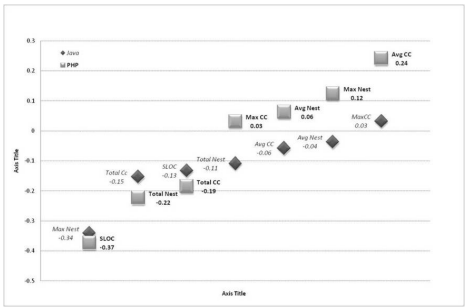

Fig. 6. Metric correlations to CVD

7 Security Resource Indicator

We measured the importance of security to a project by counting the public security resources made available on the project web site. We used the security resource indicator metric (SRI) [**?**], which is based on four items: security documentation for application installation and configuration, a dedicated e-mail alias to report security problems, a list of vulnerabilities specific to the application, and documentation of secure development practices, such as techniques to avoid common secure programming errors. The SRI metric is the sum of the four indicator items, ranging from zero to four.

Six of the eleven Java projects had security documentation, but none of the projects had any of the other three indicators. These results are similar to the results of Fortify's survey [**?**], in which only one of the eleven projects they examined had a security e-mail alias and two had links to security information. Their survey did not include the other components of the SRI metric.

PHP results were substantially different. While the percentage of projects with security documentation was lower, with only five of the fourteen projects having such documentation, six PHP projects had security e-mail contacts, five had vulnerability databases, and four had secure coding documentation. While there is no significant correlation of SRI with change in CVD, there is a significant correlation ($p < 0.05$) with a strong Spearman rank correlation coefficient, ρ, of 0.67, of SRI with change in SAVD, counting all PHP vulnerability categories.

The difference in SRI may result from the differences in application popularity. Open source PHP web applications are much more widely used than open source Java web applications. Popular projects are more likely to have vulnerabilities listed in the National Vulnerability Database [**?**], and therefore have a stronger incentive to track vulnerabilities and provide security contacts.

In addition to the greater number and higher Freshmeat popularity of PHP applications, language popularity is also revealed in what languages are supported by web hosting providers. Sixteen of the top 25 web hosting providers from `webhosting.info` listed supported languages: 87.5% supported PHP while only 25% supported Java. Several of the top hosting providers offered hosting

for popular PHP applications, including Drupal, Joomla, Mambo, phpBB, and WordPress. None provided hosting for specific Java web applications.

8 Analysis Limitations

The 25 open source web applications were the only projects found on `freshmeat.net` that met our analysis criteria. Our analysis may not apply to other projects that were not analyzed in this work. Different static analysis tools look for different types of vulnerability and use different analysis techniques, so the vulnerability density from one tool cannot be compared directly to another. Static analysis tools also search for different vulnerabilities in different languages.

Static analysis tools report false positives, where a program mistakenly identifies a line of code as containing a vulnerability that it does not. Walden et. al. [**?**] found that the Fortify static analysis tool had a false positive rate of 18.1% when examining web applications written in PHP. Coverity [**?**] found a false positive rate of less than 14% for millions of lines of C and C++ code.

9 Conclusion

We found that Java web applications had a substantially lower CVD than similar applications written in PHP, with 2008 values of 2.36 vulnerabilities/KSLOC for PHP and 0.63 for Java. Both sets of applications improved from 2006 to 2008, with PHP improving faster due to a decrease in vulnerability count while Java's improvement was due to a lower rate of vulnerabilities being inserted as code size grew. A large part of PHP's decrease was from a decline in SQL injection vulnerabilities, which could arise from higher usage of parameterized query interfaces as hosting providers offered newer versions of PHP database libraries.

The variation between projects was much greater than the variation between languages, ranging from 0.52 to 14.39 vulnerabilities/KSLOC for Java and 0.03 to 121.36 in PHP for 2006. Eight of the PHP projects had higher vulnerability densities in 2008 than 2006, while only three Java projects did. SRI was a useful predictor of how vulnerabilities evolved in PHP projects, but not for Java since none of the Java projects had security contacts or vulnerability listings. Complexity metrics were useful predictors for Java but not PHP vulnerability evolution.

In summary, programming language is not an important consideration in developing secure open source web applications. The correlation coefficient, $\rho = -0.07$, between language and CVD, was quite low, but it was not statistically significant. However, neither language had a clear advantage over the other in CVD over time and the variation between applications was much larger than the variation between languages.

References

1. Ayewah, N., Pugh, W.J., Morgenthaler, D., Penix J., Zhou. Y.: Evaluating Static Analysis Defect Warnings On Production Software. In: The 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 2007.
2. Christey, S.M. and Martin, R.A.: `http://www.cve.mitre.org/docs/vuln-trends/index.html`, published May 22, 2007.
3. Coverity, Coverity Scan Open Source Report 2009, `http://www.coverity.com/scan/`, September 23, 2009.
4. Fenton, N.E. and Pfleeger, S.L.: *Software Metrics: A Rigorous and Practical Approach*, Brooks/Cole, Massachusetts, 1998.
5. Fortify Security Research Group and Larry Suto: Open Source Security Study. `http://www.fortify.com/landing/oss/oss_report.jsp`, July 2008.
6. `http://freshmeat.net/`, accessed September 27, 2009.
7. Li, Z., Tan, L., Wang, Xuanhui and Lu, Shan and Zhou, Yuanyuan and Zhai, Chengxiang: Have things changed now?: an empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, Association of Computing Machinery, New York, 2006, pp. 25-33.
8. Nagappan, N. and Ball, T: Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th International Conference on Software Engineering, Association of Computing Machinery, New York, 2005, pp. 580 - 586.
9. Shiflett, C.: PHP Security Consortium Redux. `http://shiflett.org/blog/2005/feb/php-security-consortium-redux`.
10. Nagappan, N., Ball, T., and Zeller, A.: Mining Metrics to Predict Component Failures. In: Proceedings of the 28th International Conference on Software Engineering, Association of Computing Machinery, New York, 2006, pp. 452 - 461.
11. Neuhaus, S., and Zimmerman, T.: The Beauty and the Beast: Vulnerabilities in Red Hat's Packages. In: Proceedings of the 2009 USENIX Annual Technical Conference (USENIX 2009), San Diego, CA, USA, June 2009.
12. Ozment, A. and Schechter, S.E.: Milk or Wine: Does Software Security Improve with Age?. In: Proceedings of the 15th USENIX Security Symposium, USENIX Association, California, 2006, pp. 93-104.
13. Shin, Y. and Williams, L.: An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, Association for Computing Machinery, New York, 2008, pp. 315-317.
14. Shin, Y. and Williams, L: Is Complexity Really the Enemy of Software Security?. In: Quality of Protection Workshop at the ACM Conference on Computers and Communications Security (CCS) 2008, Association for Computing Machinery, New York, 2008, pp. 47-50.
15. Walden, J., Doyle, M., Welch, G., Whelan, M.: Security of Open Source Web Applications. In: Proceedings of the International Workshop on Security Measurements and Metrics, IEEE, 2009.
16. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B.: Cross-project Defect Prediction. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009), Amsterdam, The Netherlands, August 2009.