Impact of Plugins on the Security of Web Applications

James Walden, Maureen Doyle, Rob Lenhof, John Murray, Andrew Plunkett
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099
{waldenj,doylem3,lenhofr1,murrayj5,plunketta}@nku.edu

Abstract

Many web applications have evolved into complex software ecosystems, consisting of a core maintained by a set of long term developers and a range of plugins developed by third parties. The security of such applications depends as much on vulnerabilities found in plugins as it does in vulnerabilities in the application core. In this paper, we present a study of vulnerabilities in twelve open source web applications and 13,535 plugins for those applications. We used automated static analysis tools to count vulnerabilities.

Plugins made up 91% of the aggregate code base of 11.7 MLOC and contained 92% of the 135,907 vulnerabilities found. Six projects had lower vulnerability density (vulnerabilities per thousand lines of code) in their code code than plugin code. Plugin vulnerability density was significantly correlated with plugin code size ($\rho = 0.91$), but core vulnerability density was not correlated with code size. We also analyzed the density of individual vulnerability categories, finding plugins to have many more cross-site vulnerabilities and fewer injection vulnerabilities than core code.

1. Introduction

Web applications are frequently the target of attackers, with 82% of reported vulnerabilities affecting web technologies [2]. Even deployments of web applications such as blogs, which offer no direct source of income to attackers, are used by attackers to distribute malware. In May 2010, there were three outbreaks of malware that target WordPress [15] deployments alone. These attacks can target either the core code or plugins of the web application.

Plugins, known variously as add-ons, modules, and snippets, are an essential part of widely deployed complex web applications. These add-ons accomplish such tasks as adding forms to a content management system or connecting a blog with social networking systems. Some plugins help secure the application, such as the Exploit Scanner plugin for WordPress. Plugins are so widely used by some applications that developers or third parties package and distribute the applications together with different sets of popular plugins.

While previous studies examined the security of the core code of web applications [12, 13], this study focuses on the impact of plugins on the security of web applications. Since many web applications are typically deployed in conjunction with a set of plugins, examining the application without the plugins provides an incomplete picture of the security of the application. Plugins can increase both the attack surface and the number of vulnerabilities present in a system.

We focus on vulnerabilities and address the following questions: do plugins account for most security flaws? How are vulnerabilities distributed across core and plugin source code? How does the source code of a web application and its plugins differ in number and types of vulnerabilities?

To address these questions, we conducted an empirical study of twelve of the most popular open source web applications written in PHP, including some of the most widely used ones, such as Drupal and WordPress. We selected applications written in a single language in order to compare vulnerability and code metrics objectively. We choose PHP since most widely used open source web applications are written in that language.

We based our vulnerability analysis on static analysis tools instead of reported vulnerabilities. There are two primary advantages of using automatic analysis tools to count vulnerabilities. First, it allows us to study the many plugins that do not have publicly reported vulnerabilities. Second, static analysis tools perform the same error checks evenly across all of the source code, while manual reports tend to overrepresent flaws that were exploited or which were in parts of code examined by vulnerability researchers.

The primary contributions of this work are:

1. The first study of the security of web application plugins and one of the largest studies of web application security, considering 11.7 MLOC.

2. An analysis of the effect of web application plugins on vulnerability, including a comparison of the secure of core web application code with plugin code.

3. A study of indicators of plugin security, such as code size.

After discussing related work in section 2, we describe the study methodology in section 3. Study results are analyzed in section 4, while section 5 describes threats to validity. Section 6 completes the paper, summarizing conclusions and describing future work.

2. Related Work

Several studies have used static analysis tools to analyze the security of open source projects [3, 6, 12, 13]. The last two of those papers were empirical studies of web application security. One was a study of changes in vulnerabilities of PHP web applications over a two year period [12], while the second study compared the security of PHP and Java web applications [13].

No other studies have examined the security of web application plugins. However, the analogous topic of operating system driver security has been examined in two previous works. Chou et. al. found that driver code had error rates three to seven times higher for different types of errors than other Linux kernel code [5], while Coverity found that drivers account for 50-65% of Linux security defects [4].

3. Methodology

We examine the following hypotheses in this study:

1. The average SAVD of the plugin code base will be higher than the average SAVD of core code.

2. SAVD for some vulnerability categories will also be higher for plugins than core code.

3. Plugin SAVD will follow a decreasing curve, with a few plugins having a very high SAVD and many plugins having zero or low SAVD.

We expected the quality of plugin code to be lower than the quality of core web application code, since plugin developers have less experience working with the code base and plugins are less well tested since plugins are only deployed on a subset of the sites the web application runs on. These expectations parallel our expectations for drivers, which previous studies [4, 5] have shown to contain more vulnerabilities or bugs than operating system kernel code.

We used three measures of security: vulnerability count, percentage of vulnerable components, and static analysis vulnerability density (SAVD), which is the number of vulnerabilities detected by a static analysis tool per KLOC (thousand lines of code). We also computed these metrics for individual vulnerability categories, such as cross-site scripting or SQL injection. We used the OWASP Top 10 for 2010 [9] vulnerability categories, and created a map to match the categories reported by our static analysis tool to the OWASP Top 10 categories.

3.1 Application Selection

We selected the twelve open source PHP web applications from the top 50 most popular PHP projects at `freshmeat.net` that met the plugin criteria described below. These applications are listed with the version numbers that we analyzed in table 1. We selected open source applications so that we would have access to their source code for analysis, and we analyzed applications written in the same language so that software and security metrics could be compared objectively between them. We chose PHP since the vast majority of popular open source web applications are written in PHP [7]. PHP is used by 30% of the top million sites, compared to 7.7% for J2EE; WordPress is used by 80% of blogs, and Drupal is the most widely used content management system, with 41% of sites [11].

We chose the most popular applications since popular applications are more likely to have a wide variety of plugins. Each of the applications had to have a primary plugin repository with at least five open source plugins to be selected. Applications whose plugins could not be automatically downloaded were not selected. There were two common reasons for this problem: some applications, such as phpBB, have plugins scattered across a wide variety of third party sites, while other applications, typically wikis, embedded their plugins in HTML in a wide variety of ways.

achievo 1.4.3	dotproject 2.1.3	drupal 6.16
gallery 2.3.1	knowledgetree 3.7.0.2	mantis 1.2.1
modx 1.0.3	phpwebsite 1.6.3	roundcube 0.3.1
smarty 2.6.26	squirrelmail 1.4.20	wordpress 2.9.2

Table 1. PHP Open Source Web Applications

While we downloaded the most recent version of all plugins for each of the twelve applications as of April 2010, some plugins are language packs or themes that do not contain source code. Such plugins are not included in our analysis, since they cannot contain any PHP vulnerabilities.

3.2 Data Collection

Our data collection infrastructure was almost completely automated. Custom scripts were written to download the

plugins for each application, since the layout of plugin repositories differs considerably between applications. A single back-end program extracted plugin source code from archive files, called an external static analysis program, and parsed the resulting XML vulnerability report files to produce a CSV file containing the number of vulnerabilities in each category reported by the static analysis tool. Statistical analysis of the resulting CSV files, along with production of graphs, was performed using a combination of Excel and R.

Our static analysis tool was Fortify Source Code Analyzer version 5.8, which identifies 73 categories of PHP vulnerabilities, including common vulnerabilities like cross-site scripting and PHP specific misconfigurations such as use of `register_globals`. While open source static analysis tools for PHP exist, such as Pixy [8], PHP-Front, and PHP-SAT [10], none of them satisfy our criteria of supporting PHP versions 4 and 5, being able to detect more than two types of vulnerabilities, and not being a beta release. Code size was measured using the source lines of code (SLOC) metric by sloccount [14].

4. Results

We analyzed vulnerabilities in twelve open source web application projects and their plugins. The complete data set consisted of the 13,555 open source plugins that could be downloaded from the central repositories of those applications. Table 2 includes the following data for each project: number of plugins, number of vulnerabilities identified in those plugins, the SAVD of the aggregate plugin code base, and the highest SAVD value for an individual plugin of that project.

4.1 Plugin Size and Vulnerabilities

Plugin size ranged from one line of code to 168,134 lines of code, with the number of plugins following the gradually declining curve shown in figure 1. The long, sparse tail is cut off at 1000 lines of code, so the shape of the curve for the large majority (87%) of plugins smaller than that size can be seen. Only 156 (1.2%) plugins are larger than 10,000 source lines, and only four plugins are longer than 100,000 lines.

Plugins also varied widely in number of vulnerabilities, ranging from zero to 528, following a sharply declining curve shown in figure 2. Note that the scale for the vertical axis, which describes the number of plugins, is logarithmic, so the horizontal axis, which describes vulnerability counts, starts at one, with points on the axis representing the fact that no plugin had that particular number of vulnerabilities. The tail is cut off at 125 vulnerabilities, as most numbers of vulnerabilities beyond that point are represented by zero plugins with the occasional single plugin matching a particular vulnerability count.

Table 2. Plugin Summary

Name	Number Plugins	Plugin Vulns	Plugin SAVD	Max SAVD
achievo	17	130	1.19	74.61
dotproject	38	413	10.61	53.75
drupal	3630	3977	1.67	666.67
gallery	10	44	1.64	4.24
knowledgetree	69	556	3.54	61.85
mantisbt	36	117	3.97	27.77
modx	380	3127	10.72	368.42
phpwebsite	34	99	1.67	19.42
roundcube	105	322	5.63	384.42
smarty	23	27	16.49	250.00
squirrelmail	210	1706	11.21	478.26
wordpress	8989	115032	15.42	2466.67
Total	13535	125550		
Average	507	1169	7.68	

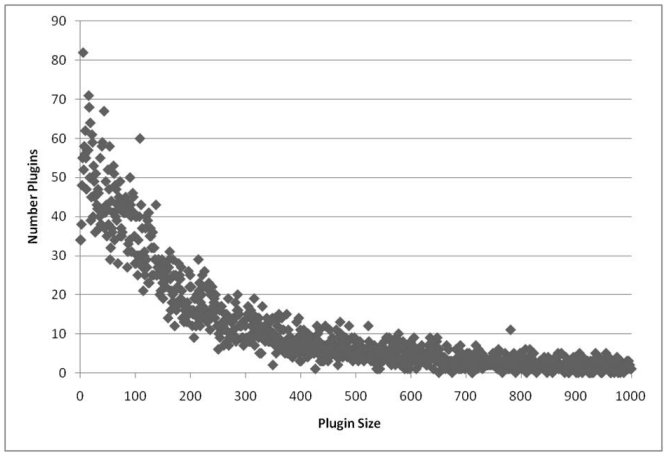

Figure 1. Plugin Size Distribution

Previous work [1, 5] found that larger functions had higher error rates, so we examined whether plugin size could be used as a rough indicator of security. We divided plugins into six bins, whose widths were defined to distribute the plugins somewhat evenly. The bins are defined in table 3.

The percentage of plugins that contain security vulnerabilities increases with plugin size as shown in figure 3. The larger the plugin the more likely it is to have a vulnerability. The odds of selecting a plugin without vulnerabilities at random are less than even for any plugin with more than 50 lines of code. Of the 156 (1.2%) plugins that are larger than

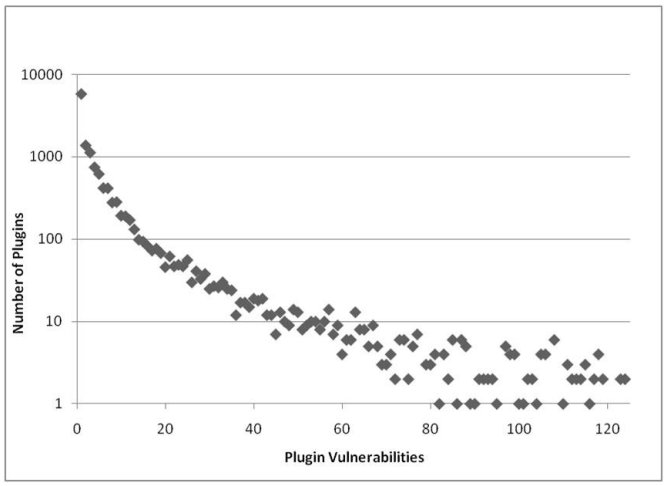

Figure 2. Plugin Vulnerability Distribution

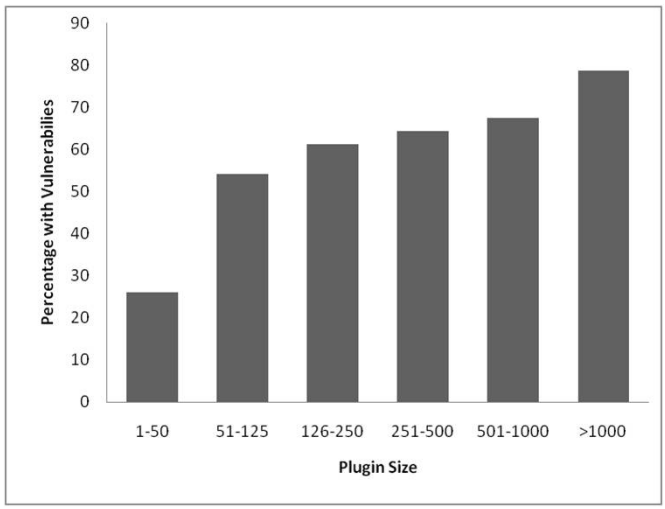

Figure 3. Percentage of Vulnerable Plugins

Table 3. Plugin Bins

SLOC Range	Number Plugins	Vuln Plugins
1-50	2466	643
51-125	2861	1551
126-250	2647	1619
251-500	2233	1439
501-1000	1532	1035
>1000	1796	1417
Total	13535	7704

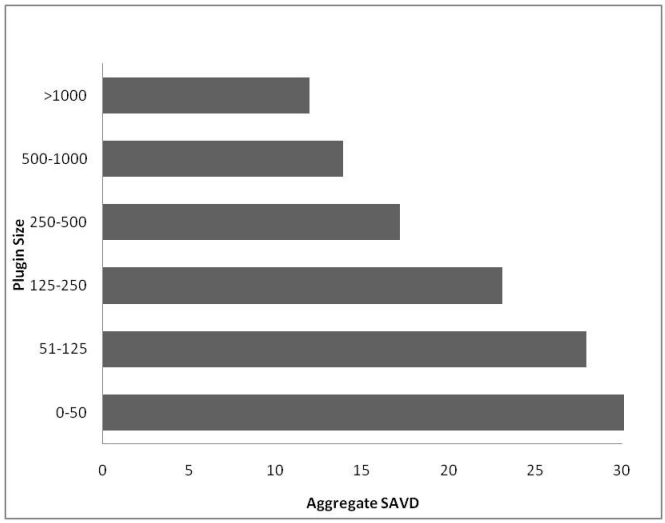

Figure 4. Aggregate SAVD by Code Size

10,000 source lines, 90.1% contain vulnerabilities, and all of the four plugins longer than 100,000 lines have vulnerabilities.

Figure 4 shows the average SAVD for each plugin size bin. SAVD decreases as plugin size increases. This is not completely unexpected, since the SAVD is a function of size.

The large number of plugins per bin allowed the assumption of normality, and a one-way Analysis of Variance was performed for the average SAVD per bin. The null hypothesis was that all means were the same and this was rejected at $p < 0.0001$. The Tukey 95% confidence interval was computed for all combinations and signficant differences, are indicated in table 4 with a $>$. For all but the smallest group of plugins, it is more likely that a larger plugin will have a vulnerability; however, its SAVD will not be signficantly worse than a plugin half its size. For plugins that are smaller than 50 lines of code, it is less likely it will have a vulnerability; however, due to the small size of the plugin, SAVD will be quite large if it does contain a vulnerability.

4.2 Plugin Vulnerability Density

We computed two per-project vulnerability density metrics for plugins. The first, aggregate plugin SAVD, is the vulnerability density for the aggregate code base of the plugins, i.e. it is the total number of plugin vulnerabilities for a particular project divided by the total lines of code for plugins of that project. The second, average plugin SAVD, is the average of the individual plugin vulnerability densities for a project.

Both are useful metrics when evaluating the security risk of installing a plugin for an application. When the average SAVD is lower than the project aggregate SAVD, it in-

Table 4. SAVD Difference Plugins Size

Bin	125	250	500	1000	>1000
50		>	>	>	>
125	-		>	>	>
250		-		>	>
500	<		-		
1000	<	<		-	

dicates that the lower SLOC-count plugins have more of the vulnerabilities. This is the case for all but two projects, KnowledgeTree and dotProject. KnowledgeTree's aggregate and average SAVD values are not too far apart; however, dotProject's lower SAVD average indicates that plugins with more SLOC have a proportionally more of the vulnerabilities. Figure 5 compares the two metrics for plugins from all twelve projects.

Figure 6 graphs the number of plugins and maximum SAVD for all of the projects, showing that projects with larger numbers of plugins tend to have plugins with a higher maximum vulnerability density. The Spearman rank correlation coefficient of these two factors is 0.8111 (p=0.0007). While this relationship is expected given that more plugins is likely to result in a wider spread of code quality, it is interesting to note a small number of exceptions, such as achievo and smarty, that have higher maximum vulnerability densities than would be expected for their small numbers of plugins.

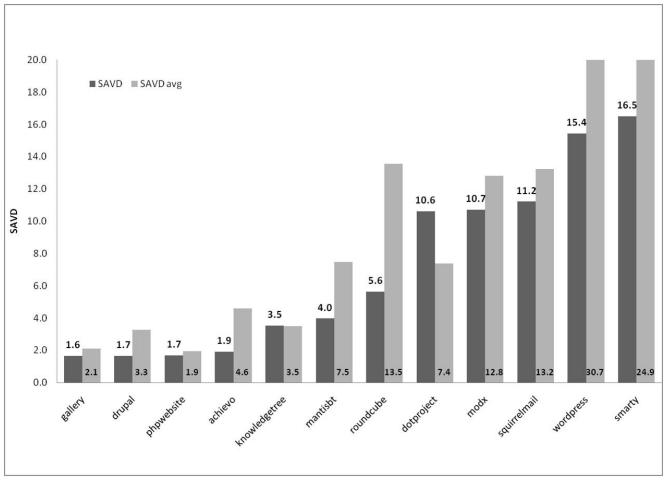

Figure 5. Average v. Aggregate SAVD

4.3 Core and Plugin Code Comparison

The aggregate code base of the core applications consisted of 996,698 lines of PHP source code, while the aggregate plugin code base consisted of 10,722,571 lines. Similar to the results of Chou et. al. [5], which found that drivers accounted for 70% of the code in the Linux kernel, plugins accounted for 91% of the total code base for our twelve applications.

However, there is a high amount of variability between projects. Two projects (KnowledgeTree and Roundcube) have approximately the same amount of plugin code as core code, and four projects have more plugin code than core code, two of them (Drupal and WordPress) having an order of magnitude more plugin than core code. If we ignore the two largest projects, plugins only account for 50% of the total code base. Table 5 shows how widely the ratio of plugin source code size to core size varies.

Figure 7 compares vulnerability densities of core and aggregate plugin code for each web application. Six of the projects (MODx, dotProject, MantisBT, phpWebSite,

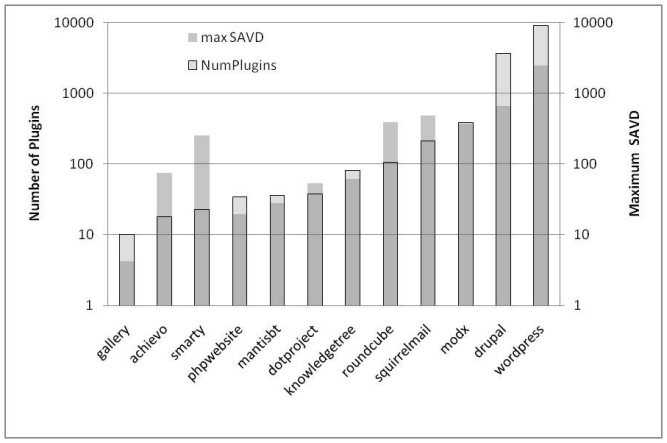

Figure 6. Plugin Counts and Maximum SAVD

Table 5. SLOC Comparisons

Project Name	Core SLOC	Plugin SLOC	Plugin/Core Size Ratio
achievo	117048	67815	58%
dotproject	93395	38894	42%
drupal	24255	910821	3755%
gallery	37717	26743	71%
knowledgetree	168920	157066	93%
mantisbt	108007	29456	27%
modx	78059	291538	374%
phpwebsite	185819	59023	32%
roundcube	62446	57194	92%
smarty	4432	1637	37%
squirrelmail	35136	152089	433%
wordpress	81464	7458748	9157%

Gallery, and Drupal) have higher core vulnerability densities than aggregate or average plugin vulnerability densities, while the other six projects have lower core vulnerability rates. To our surprise, this result is contrary to our first hypothesis and the plugin results are not similar to operating system driver results [4, 5], since correlation tests showed no significant relationship between plugin code and vulnerability rates ($\rho = 0.2, p = 0.5$).

However, both of those studies only examined bug and vulnerability distribution in the context of a single operating system kernel. Six of our 12 projects shared the same relationship between add-on and core code. The correlation of code size to vulnerabilities for all applications was $\rho = 0.38$; however, when considering only the 6 projects with lower core vulnerability rates, the correlation was $\rho = 0.48$. Neither of these correlations are significant, and this factor is an area of future work.

While web application plugins and Linux kernel drivers have similarities in that both are add-on chunks of code designed to work with another program that is typically designed by a different group of developers, there are also differences. While kernel drivers, with the exception of proprietary drivers, are distributed as part of the Linux kernel, plugins are typically distributed separately and downloaded individually by end users. The two types of add-ons are also written in different languages, with drivers being written in C or assembly language and web application plugins typically being written in PHP.

On the other side of the comparison, the core code of web applications and the Linux kernel differ considerably as well. The core Linux code base is much larger than the code base of any of the twelve web applications and is developed by a correspondingly larger number of developers with more formal software development and testing procedures. The kernel has also been developed for a much longer period, starting in 1991, while the oldest of the twelve applications, squirrelmail, began development in 1999 with most starting several years after that.

However, core vulnerability density is always less than the maximum plugin SAVD per project, shown in table 2. There is no significant correlation between core SAVD and number of plugins, core SAVD and plugin SAVD, nor core SAVD and maximum SAVD. This supports the premise that plugin development is done independently of the project, and more secure projects do not necessarily have any more or fewer plugins than insecure projects.

We also examined the correlation between code size and number of vulnerabilities. Intuition suggests that when security is not considered in development, more code will result in more vulnerabilities. Core code does not have a significant correlation between SLOC and vulnerabilities ($\rho = 0.38, p = 0.19$); however, plugin source code displays a significant correlation ($\rho = 0.91, p = 0.0001$). This

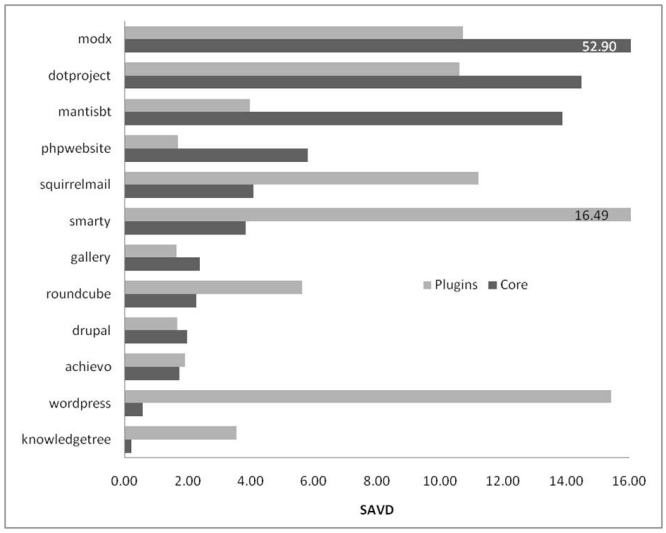

Figure 7. Aggregate SAVD: Core v. Plugin

result may indicate that there are core projects do take security seriously, as the presence of coding standards including secure coding guidelines indicates for six of the projects, but this care does not extend to the plugins.

Finally, we examined the effect of adding plugins to the vulnerability density of an application. Figure 8 shows this effect for adding one up to 100 plugins. The lowest line in the figure shows effect of adding plugins to the the aggregate code base, assuming all projects and plugins have equal SAVD. The other two lines show the effect of adding plugins on WordPress, for which over half the plugins in our study were written, using both aggregate and average SAVD.

4.4 Vulnerability Categories

While Fortify Source Code Analyzer 5.8 documents 73 possible categories of PHP vulnerabilities, it only detected 25 different types of PHP vulnerabilities in the code base we studied. No vulnerabilities were reported from any of the other 48 categories. Many of the unrepresented types are subcategories of categories like cross-site scripting, which were reported. Other missing categories are specific to software features and frameworks, such as the Cake PHP MVC framework, that were not used in any of the twelve applications we analyzed.

Since this classification of vulnerabilities is specific to Fortify Source Code Analyzer, we mapped 18 of the non-zero categories to the well-known OWASP Top 10 for 2010 [9] classification. The other seven non-zero categories reported by Source Code Analyzer were not represented by any of the OWASP Top 10 entries. We mapped the

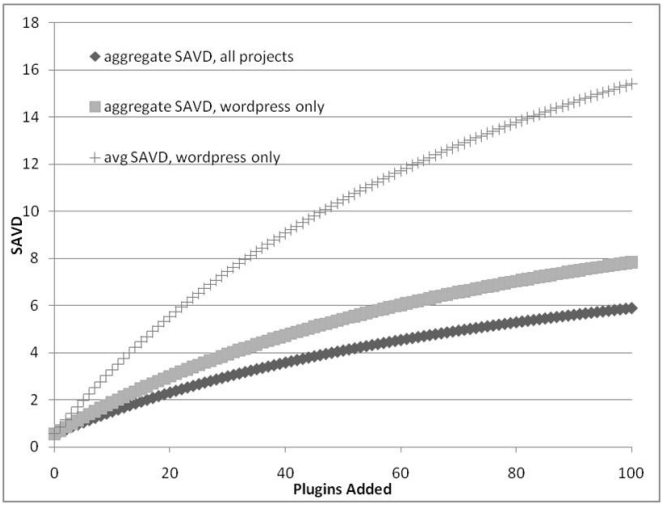

Figure 8. Effect of adding plugins on SAVD

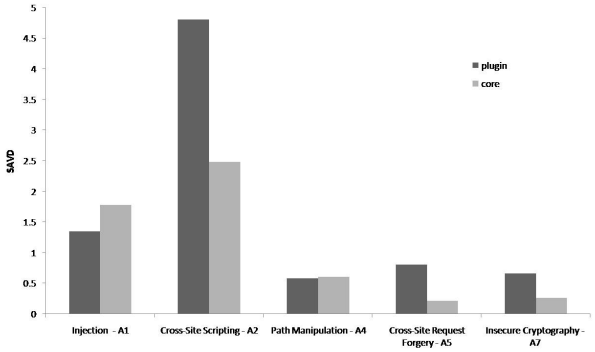

Figure 9. SAVD by OWASP Category

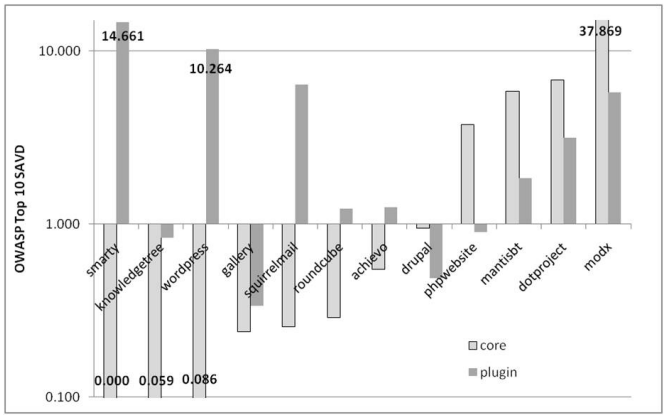

Figure 10. OWASP Top 10 Density Comparison

18 categories to the following five entries in the OWASP Top 10: injection, cross-site scripting, insecure direct object references, cross-site request forgery, and insecure cryptographic storage. No category reported by Source Code Analyzer matched any of the other entries in the OWASP Top 10.

Figure 9 shows the density of each OWASP vulnerability category in both core and plugin aggregate code bases. While core code has noticeably higher rates of injection vulnerabilities and slightly higher rates of path manipulation, plugin code has higher rates of the other three OWASP categories. The substantially greater incidence of cross-site scripting and cross-site request forgery in plugin code may be a result of more plugin code being client facing than core code, while the greater incidence of injection code in core code can be attributed to the same fact, since injection vulnerabilities typically occur in backend SQL, XML, or shell code.

By summing vulnerabilities matching the five OWASP categories, we can produce an OWASP vulnerability density metric and use that to compare the core and plugin code. This comparison is graphed in figure 10. Seven of the twelve projects have a lower OWASP vulnerability density for core core than aggregate plugin code. This metric indicates that core code is slightly better than the overall SAVD metric indicates.

5. Threats to Validity

We attempted to limit internal validity threats by automating as much of the data collection as possible and by validating our data collection tools and processes. The data collection software libraries and tools were tested with a unit test suite to verify that the results produced were correct and that code modifications did not alter the results. We also manually inspected a subset of the data.

While automation ensures a high degree of consistency, automated code analysis tools report false positives, where vulnerabilities are reported that are not actually present in the code. Manually evaluating the vulnerability reports for two open source PHP web applications yielded a false positive rate of 18.1%. This number is consistent with the false positive rate of less than 14% reported by Coverity [3].

Static analysis tools also fail to report vulnerabilities that they do not have code to check for. Earlier versions of Source Code Analyzer only reported thirteen categories of vulnerabilities for PHP code, while the current version reports 73 categories of vulnerabilities. Later versions may add additional checkers that identify further categories of vulnerabilities. Other static analysis tools have different sets of vulnerabilities, so the vulnerability counts reported in this paper would be different if another static analysis

tool had been used.

The twelve open source PHP projects were the only projects found in the top 50 most popular PHP projects at `freshmeat.net` that had at least five plugins. The correlations described in this paper may be specific to this set of applications. The choice of language impacts both code size and possible vulnerabilities, so these results may not generalize to applications or plugins written in languages other than PHP.

This work is empirical, computing and reporting the correlations between two phenomena, but we cannot prove that these are cause-effect relationships. Normality could not be assumed for core project analyses, so nonparametric analyses to compensate for this. Plugin size analyses used binning, and different bin widths may alter the results. A parametric analysis was used to determine if there was a difference in mean SAVD across the six bins. This analysis assumes that the SAVD values are normally distributed across the bins with equal variability. The assumption of normality can be relaxed in this case because of the large sample sizes, but there is no evidence in the sample to suggest that equal variablity was satisfied.

6. Conclusion

We conducted an empirical study of twelve open source PHP web applications and their plugins, measuring code size, vulnerabilities, and vulnerability density. We also applied those metrics to individual types of vulnerabilities categorized by the OWASP Top 10 list. While most of the vulnerabilities (92%) were found in plugin code, most of the code (91%) was also found in the plugins, and we did not find any indicator that plugin code was of lower security than core code in general. Six web applications had higher plugin vulnerability densities when compared to core code, and six applications had lower plugin vulnerability densities.

We found that plugin size is a good indicator of plugin security. Plugin code size is strongly correlated with number of vulnerabilities ($\rho = 0.91$). We also found that more than half of plugins longer than 50 lines of code contained vulnerabilities, while less than 30% of plugins smaller than 50 lines of code had vulnerabilities.

We found that plugin code differed strongly from core code in the types of vulnerabilities present. Vulnerabilities characteristic of client-facing code, such as cross-site scripting and cross-site request forgery, were more prevalent in plugin code, while vulnerabilities characteristic of back-end code, such as injection vulnerabilities, were more prevalent in core code. This result is to be expected since many plugins are devoted to altering the user interface of applications, while a smaller percentage of plugins are focused on altering back-end data.

Future work will include determining the most appropriate empirical distribution for use in modeling the number of plugins per vulnerability. Possible discrete distributions that can be used to model this count are the Geometric distribution, the Poisson distribution, and the Log-Series distribution. Maximum likelihood estimates of the parameter associated with these distributions will be found, and the Chi-Square goodness-of-fit test will be used to assess the adequacy of the various distributions. We plan to analyze a wider range of web applications and fully automate our statistical analysis and graph generation using R.

7. Acknowledgments

We would like to acknowledge Brooke Buckley and Dhanuja Kasturiratna for assistance with statistics. We also acknowledge the help of our student Justin Brown in gathering data.

References

[1] Basili, V., Perricone, B. Software errors and complexity: an empirical investigation. Communications of the ACM, v.27 n.1, p.42-52, Jan. 1984.

[2] Cenzic. Web Application Security Trends Report, Q3-Q4 2009. `http://www.cenzic.com/downloads/Cenzic_AppSecTrends_Q3-Q4-2009.pdf`, accessed May 28, 2010.

[3] Coverity. Open Source Report 2009. `http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2009.pdf`, accessed May 28, 2010.

[4] Chou, A., Fulton, B., and Shallem, S. Linux Kernel Security Report, `http://www.coverity.com/library/pdf/linux_report.pdf`, 2005.

[5] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. 2001. An empirical study of operating systems errors. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM, New York, NY, 73-88.

[6] Fortify Security Research Group and Larry Suto. Open Source Security Study. `http://www.fortify.com/landing/oss/oss_report.jsp`, July 2008.

[7] `http://freshmeat.net/`, accessed May 28, 2010.

[8] Jovanovic, N., Kruegel, C., Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. *Proceedings of the 2006 IEEE Symposium on Security and Privacy* IEEE, 2006, pp. 258 - 263.

[9] Open Web Application Security Project, OWASP Top Ten for 2010, http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, accessed May 28, 2010.

[10] http://strategoxt.org/PHP/PhpSat, accessed May 28, 2010.

[11] http://trends.builtwith.com/, accessed May 28, 2010.

[12] Walden, J., Doyle, M., Welch, G. A., and Whelan, M. 2009. Security of open source web applications. In Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement (October 15 - 16, 2009). Empirical Software Engineering and Measurement. IEEE Computer Society, Washington, DC, 545-553.

[13] Walden, J. and Doyle, M. and Lenhof, R. and Murray, J. 2010. Idea: Java vs. PHP: Security Implications of Language Choice for Web Applications. In Proceedings of the 2010 2nd Engineering Secure Software and Systems (February 3 - 4, 2010). Springer, 61-69.

[14] D.A. Wheeler, http://www.dwheeler.com/sloccount/ accessed May 28, 2010.

[15] http://www.wpsecuritylock.com/blog, accessed May 28, 2010.