Security of Open Source Web Applications

James Walden, Maureen Doyle, Grant A. Welch, Michael Whelan
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099
waldenj@nku.edu, doylem3@nku.edu, welchg1@nku.edu, whelanm1@nku.edu

Abstract

In an empirical study of fourteen widely used open source PHP web applications, we found that the vulnerability density of the aggregate code base decreased from 8.88 vulnerabilities/KLOC to 3.30 from Summer 2006 to Summer 2008. Individual web applications varied widely, with vulnerability densities ranging from 0 to 121.4 at the beginning of the study. While the total number of security problems decreased, vulnerability density increased in eight of the fourteen applications over the analysis period.

We developed a security resources indicator metric, which we found to be strongly correlated ($\rho = 0.67, p < 0.05$) with change in vulnerability density over time. Traditional software metrics, such as code size, cyclomatic complexity, nesting complexity, and churn, had significant ($p < 0.05$) but much smaller correlations ($\rho = 0.31$ at best) with vulnerability density. Vulnerability density was measured using the Fortify Source Code Analyzer static analysis tool.

1. Introduction

As people bank, communicate, and shop online, they become dependent on the security of the software used for these purposes. Identity theft, malware infections, and other forms of computer crime cost consumers and companies billions of dollars [8] and erode the trust that is necessary for people to be willing to do business online.

Web applications have become the primary source of security vulnerabilities. From 2006 through the first half of 2008, web application vulnerabilities represented 51% of all vulnerability disclosures reported in IBM's X-Force Trend Statistics Report [12]. MITRE found that the most common two vulnerability types since 2005 were cross-site scripting and SQL injection [4], which are primarily found in web applications. In 2007, the CSI Computer Crime and Security Survey reported that 44% of corporations had experienced a security incident involving their web site [21].

In this paper, we survey security trends in open source web applications. We studied fourteen web applications written in PHP, including some of the most widely used ones, such as WordPress and Mediawiki, the software on which Wikipedia runs. We mined the source code repositories of these applications to measure vulnerability density and to collect a variety of software metrics, including our security resources indicator metric as well as traditional metrics such as code size, churn, and complexity. We believe this is the largest survey of web application security, both in terms of code size and number of application users.

The primary contributions of this work are:

1. It reports the current state of web application security vulnerabilities in open source PHP applications and how vulnerabilities changed from Summer 2006 to Summer 2008.

2. It develops a security resources metric and shows that that metric can be used to predict changes in vulnerability density over time.

3. It analyzes whether code size and code complexity metrics can be used to predict vulnerability densities in web applications at the project level.

4. It analyzes whether code changes, measured by churn and lines deleted, can be used to predict vulnerability densities in web applications.

We measured security problems in open source web applications by counting the number of vulnerabilities found in source code by a static analysis tool. Static analysis tools find common secure programming errors by evaluating source code without executing it. We used version 5.1 of the Fortify Source Code Analyzer static analysis tool. We discuss alternative techniques for measuring vulnerabilities and why we chose static analysis below.

978-1-4244-4841-8/09/$25.00 ©2009 IEEE

We used the static analysis vulnerability density (SAVD) metric, which is the number of vulnerabilities detected by a static analysis tool per KLOC (thousand lines of code). The tool categorized vulnerabilities into thirteen categories, such as cross-site scripting and SQL injection, which were analyzed separately.

Our results show that overall vulnerability density is improving over time, from 8.88 in the initial set of revisions to 3.30 in the final set, though individual projects vary considerably, with eight of the fourteen projects raising their vulnerability densities over time. The reductions largely result from decreases in the number of the three most common vulnerability types: SQL injection, cross-site scripting, and path manipulation. However, the incidence of the next two most common vulnerability types–dangerous file inclusion and dangerous function use–increased over time.

We suspected that projects that advertise that they are working on security through public online resources tend to improve security over time. To measure these resources, we developed a security resources indicator metric, which we found to be strongly correlated ($\rho = 0.67, p < 0.05$) with change in vulnerability density over time.

Three software metrics showed significant but small correlations with vulnerability density for the aggregate code base of the fourteen projects. These metrics were maximum cyclomatic complexity, average cyclomatic complexity, and average nesting complexity. However, no single software metric was a predictor for every individual project.

After discussing related work in section 2, we describe the design of this study in section 3. Vulnerability density results are analyzed in section 4, while section 5 describes software metric results. Section 6 analyzes vulnerabilities by type, while section 7 describes the security resource indicator metric. Limitations of our analysis are discussed in section 8. Section 9 finishes the paper, giving conclusions and describing future work.

2. Related Work

Software security researchers have measured vulnerabilities using both databases of reported vulnerabilities such as the National Vulnerability Database (NVD) and static analysis results. Some of these researchers have studied software metrics to determine if these metrics can predict vulnerability density. There has also been research using static analysis results and software metrics to predict defect density, rather than vulnerability density. None of these studies analyzed web applications or code written in PHP.

Coverity reported on their analysis of a large number of open source projects written in C and C++ [6], finding a strong, linear relationship between code size and number of vulnerabilities discovered via static analysis. Fortify analyzed a small number of Java projects [10] with their static analysis tool and also evaluated the access to security expertise that each project provided, observing whether each project provided a security contact email, a security URL, and easy access to security experts.

Ozment and Schechter analyzed the OpenBSD operating system to measure vulnerability lifetimes and the change of vulnerability density with each release [20]. They found that the rate of vulnerability reports in foundational OpenBSD code was slowly decreasing, with a median lifetime of 2.6 years.

McCabe's cyclomatic complexity (CC) is a popular metric for estimating the number of defects. Shin [23] found a weak correlation of CC with vulnerabilities for the Mozilla Javascript Engine. Nagappan et. al. [18] had mixed results, with three projects out of five showing strong correlations between defect density and CC. They also used static analysis tools to measure defect density [17]. Note that defect density may not correlate with vulnerability density, as security flaws differ from reliability flaws.

Shin [23] analyzed an additional complexity metric: nesting complexity, which measures the depth of the nesting of loops or conditionals. This study showed that multiple releases of the Mozilla Javascript Engine had weakly significant correlations between vulnerabilities and nesting complexity.

Nagappan and Ball [16] predicted system defect density based on the size of code changes. They examined two releases of Windows Server 2003. Their hypothesis was that projects whose code changed often pre-release would have more post-release errors than projects with fewer pre-release changes. They measured code changes using relative churn, defining relative churn as absolute churn divided by the number of lines of code. Their definition of absolute churn was the sum of lines of code that were added and changed. They found that relative code churn was a better measure of system defect density than absolute code churn.

It is unknown whether the results of the papers described above can be generalized to web applications, as web applications handle input and output in a different manner than the traditional desktop or server applications that were analyzed in these studies.

3. Study Design

We examined the project history of fourteen open source web applications selected from the most popular PHP web applications on `freshmeat.net`. The applications are listed in table 1. We used a single language for our study so that metrics could be compared objectively, as programs written to perform the same task in different languages will differ in both lines of code and in code complexity due to differing styles and control structures. We chose PHP since it is the language in which the most widely used open source web applications are written.

achievo	obm	roundcube
dotproject	phpbb	smarty
gallery2	phpmyadmin	squirrelmail
mantisbt	phpwebsite	wordpress
mediawiki	po	

Table 1. PHP Open Source Web Applications

To be selected, an application had to have a Subversion repository containing at least 100 weeks of revisions. The fourteen applications studied were the only applications listed on `freshmeat.net` that matched our criteria. While one project was small, with under 6000 lines of code, and another one was particularly large, with nearly 400,000 lines of code, the size of the remaining projects ranged from 25,000 to 150,000 lines.

We selected a single revision from each week to analyze, choosing the first change to be made during that week. The reasons for this choice were pragmatic. We wanted to observe the projects using identical time intervals, which neither individual revisions nor official releases would have permitted. Additionally, a single revision typically involves the alteration of only two to three lines of code, rarely introducing or removing a vulnerability, though a rare few revisions modified thousands of lines of code. Public releases were too few and irregular in schedule to use for our analysis. It is also important to note that since these projects have public source repositories, users who need fixes immediately or who want features quickly frequently download and use source code from the repository without waiting for an official release.

We analyzed the number of vulnerabilities in each application by counting the number of vulnerabilities found in an application. While user input could reach a single SQL injection vulnerability through multiple code paths, the vulnerability is only counted once. There are multiple approaches to identify vulnerabilities to count, including reported vulnerabilities from a database such as the National Vulnerability Database (NVD) [19], static analysis tool results, and dynamic analysis tool results.

Reported vulnerabilities from any single source are an undercount of the actual number of vulnerabilities. There are several causes for the undercount: latent vulnerabilities that have not been discovered or reported, reported vulnerabilities that are not cross-listed with the NVD, and multiple vulnerabilities that are reported as a single database entry. The Common Vulnerabilities and Exposures (CVE) guidelines, which are the source of most NVD vulnerabilities, require merging vulnerabilities of the same type in the same version into a single entry [7]. Several latent vulnerabilities appeared in the NVD for our web applications after the completion of our study. A more complete discussion of the issues in interpreting reported vulnerability statistics can be found in [3].

Dynamic analysis requires installation of the software, and results depend on the configuration and environment in which the application is deployed. Static analysis has the advantage that it can be used as soon as code is available without requiring software installation. However, static and dynamic analysis tools may find different vulnerabilities, and both static and dynamic analysis results may include false positives, which are unlikely to occur in public vulnerability databases.

We chose static analysis as the technique to measure vulnerabilities, since it enabled us to measure vulnerabilities without installing 100 versions of each project and the undercount problem is not as severe with static analysis tools as it with reported vulnerabilities. We found an order of magnitude more vulnerabilities with our static analysis tool than are reported for the set of applications. While some of these vulnerabilities are false positives, as discussed in section 8, the false positive rate is not high enough to reduce the number of vulnerabilities found to the number of reported vulnerabilities. We measured vulnerability density using the static analysis vulnerability density (SAVD) metric.

Coverity [6] and Fortify [10] measured SAVD in their reports. Researchers at Microsoft found static analysis defect density to be an accurate predictor of pre-release defect density [17]. We used the Fortify Source Code Analyzer version 5.1 to compute SAVD. Lines of code were measured using the source lines of code (SLOC) metric, which excludes blank lines and comments. The SLOCCount [24] tool was used to measure SLOC.

While commercial static analysis tools have become more widely used in recent years, these tools are rarely used in open source development due to their high cost. Open source static analysis tools are free, but we found only three open source tools for PHP: Pixy [13], PHP-Front, and PHP-SAT. None of these tools can serve as a replacement for a commercial static analysis tool. Neither PHP-Front nor PHP-SAT has produced a stable release yet, and Pixy is limited to PHP 4 code and can only detect two types of vulnerabilities.

In addition to static analysis vulnerability density, we collected and analyzed the following software metrics: SLOC, McCabe's cyclomatic complexity [14], nesting complexity, churn, and number of lines deleted between revisions. Churn is a measure of the size of changes between versions, being the sum of the number of lines of code added and changed. Nesting complexity counts the depth of nested conditionals and loops. Cyclomatic Complexity and Nesting Complexity were computed using PHP

CodeSniffer [5]. Churn and number of deleted lines were computed from Subversion diffs by a custom Ruby script.

4. Results

Security problems identified by static analysis in the selected open source web applications decreased from Summer 2006 to Summer 2008. Examining the aggregate code base of all fourteen web applications, we found 7750 vulnerabilities in the initial set of revisions and 4628 vulnerabilities in the final set of revisions. At the same time, code size grew from 872,319 lines to 1,404,178 lines. The combined reduction in vulnerabilities and growth in code size produced a change in static analysis vulnerability density from 8.88 in initial revisions to 3.30 for the final set of revisions.

These average vulnerability densities are much higher than the initial and final values for average density reported for C and C++ applications by Coverity [6], which were 0.30 and 0.25 respectively. However, they are smaller than the average vulnerability density of 17.72 reported for a sample of Java applications by Fortify [10]. The use of different languages and classes of applications is likely the cause of some of the difference between these results and ours. The Coverity study used a different static analysis tool, which is another source of difference.

Vulnerability density varied widely between projects, with initial revisions ranging from 0 to 121.4 vulnerabilities/KLOC. In the final set of revisions, variation in SAVD was smaller, ranging from 0.20 to 60.86 vulnerabilities/KLOC. Vulnerability density decreased in six projects and increased in the other eight projects over the period studied. The Photo Organizer (po) project had the highest SAVD but the largest improvement, with SAVD decreasing from 121.4 to 60.86. All of the projects except for two increased in size. Figure 1 shows the change in vulnerability density between the initial and final revision for each project.

Static analysis vulnerability density of projects did not correlate with the number of vulnerabilities reported in the National Vulnerability Database. The reasons for vulnerability reports undercounting the number of vulnerabilities were discussed above. However, the number of NVD vulnerabilities for a project was correlated with the project popularity on `freshmeat.net` [11] with a Spearman's rank correlation coefficient of 0.53 at a 95% level of confidence. This correlation may result from attackers devoting more effort to constructing exploits for widely deployed systems or vulnerability researchers preferring to analyze higher profile software.

The reasons for large changes in code size or vulnerability count were sometimes discernable from Subversion log entries or diffs between revisions. For example, two projects merged libraries into their code bases, increasing code size dramatically with a single revision. One of those two projects, Achievo, merged the ATK project into the repository in January 2008, increasing both code size and vulnerability density.

Squirrelmail is a project where we can briefly explain each change in the number of vulnerabilities, since the revisions are well commented and there are few security changes. Application code size and vulnerability count over time are shown in figure 2. The application dramatically decreased the number of vulnerabilities in a single revision early in the study. Examination of Subversion comments revealed that the entire process for handling input was changed, including the addition of new data sanitization code in a single large revision. A later revision fixed

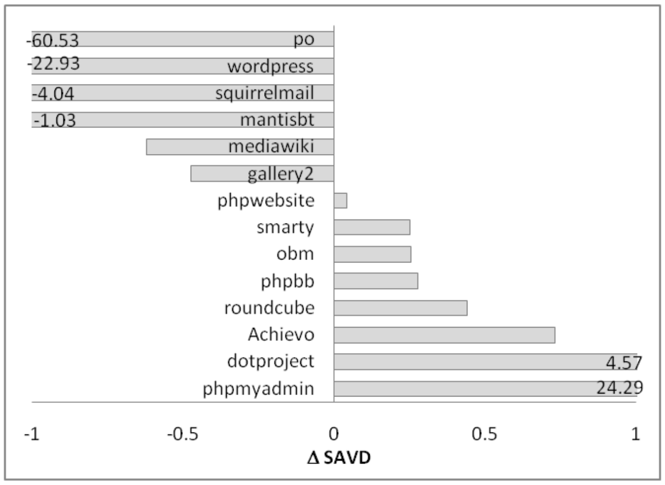

Figure 1. Change in Vulnerability Density

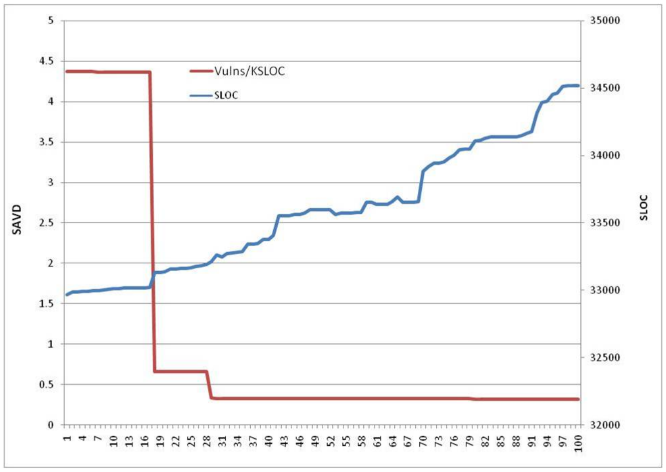

Figure 2. Squirrelmail SLOC and SAVD vs. Time

vulnerabilities reported in CVE-2006-3174 according to the comment. After that fix, the vulnerability count remained constant despite many modifications and continual growth in application size.

5. Software Metric Analysis

Based on prior research [6, 16, 17, 18, 20, 22], we selected software metrics which had demonstrated correlations to vulnerability or defect density. These metrics are churn, cyclomatic complexity, and nesting complexity. We also examined revision number and SLOC, but these results are not presented here. No previous study found a single complexity measure that predicted either defect or vulnerability density for all applications.

We computed three variations of the CC metric: average CC, total CC, and max CC. Total CC is the sum of CC for all PHP functions in a project. Average CC is total CC divided by the number of functions in a project. Max CC is the largest CC value for all functions in a project.

Figure 3 displays the correlation coefficients for the SAVD of all revisions and the metrics selected. The Spearman's rank correlation coefficient (ρ) was used for evaluating correlations between vulnerability density and metrics because no assumptions can be made about the underlying distributions. For simplicity, the Cohen Scale [2] scale is used to define strongly correlated as $|\rho| \geq 0.5$, medium correlation as $0.3 \leq |\rho| < 0.5$, and weakly correlated $|\rho| \leq 0.3$.

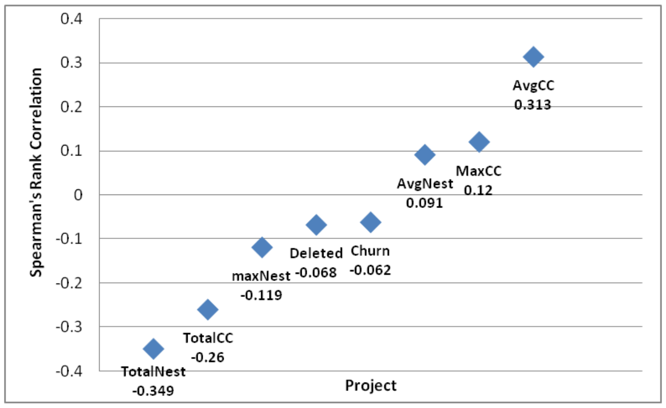

Figure 3. Metric correlations with SAVD

There are two metrics with significant negative correlations: TotalNest and TotalCC. These results indicate that as the metric value increases, the vulnerability density decreases, or that as the value decreases, the vulnerability density increases. Total complexity metrics, which sum the complexity metric values for all functions, may not be good indicators of overall code complexity since a program that is organized into a number of low complexity functions can have the same complexity as a program consisting of a single high complexity function.

As Shin [22] found for Mozilla, we observed weak correlations with complexity metrics. Though weak, the correlations are statistically signficant, and the most promising metrics appear to be average CC, maximum CC, and average nesting complexity. Our results agree with previous studies except for Nagappan and Ball [16], as we observed no significant correlation between churn and vulnerability density. A possible reason for the difference is because Nagappan examined two official releases while we studied weekly revisions.

We also computed correlations on a per project basis. There were some very strong correlations as identified in table 2. For this analysis, Spearman's coefficient is significant at the 99% confidence level when $|\rho| > 0.25$. For ease in reading, all significant positive correlations are in **bold** and all significant negative correlations are in *italics*.

Project	Max CC	Avg CC	Avg Nesting
achievo	**0.74**	*-0.48*	0.15
dotproject	0.17	*-0.46*	**0.63**
gallery2	*-0.74*	*-0.60*	*-0.41*
mantisbt	*-0.82*	*-0.96*	*-0.91*
mediawiki	*-0.86*	*-0.60*	-0.21
obm	*-0.65*	**0.80**	**0.91**
phpbb	**0.35**	*-0.31*	*-0.37*
phpmyadmin	**0.77**	*-0.93*	*-0.88*
phpwebsite	*-0.77*	*-0.73*	*-0.67*
po	**0.70**	**0.54**	**0.58**
roundcube	**0.83**	**0.88**	**0.80**
smarty	**0.29**	**0.78**	**0.57**
squirrelmail	*-0.32*	*-0.46*	*-0.63*
wordpress	-0.16	-0.19	*-0.71*

Table 2. Project SAVD and metric correlations

This table illustrates the diversity of projects with respect to the individual metrics and indicates that caution should be used when using metrics to predict vulnerability densities. There are a large number of negatively correlated results for all three metrics indicating they are a poor measure for the individual project. Maximum or average CC could be used as a weak indicator of security vulnerabilities.

6. Vulnerability Type Analysis

We examined the vulnerabilities by category. The Fortify Source Code Analyzer categorized the vulnerabilities in these applications into thirteen types, which are listed in table 3. The tool can detect other types of vulnerabilities, such as buffer overflows, that were not found in any of the applications studied.

Command Injection	Hardcoded Domain in HTML
Cross-Site Request Forgery	Javascript Hijacking
Cross-Site Scripting	Often Misused
Dangerous File Inclusion	Open Redirect
Dangerous Function	Path Manipulation
Dynamic Code Evaluation	SQL Injection
File Permission Manipulation	

Table 3. Security Vulnerabilities Found

In the initial revision of the applications, the five most common vulnerability types contain 92% of vulnerabilities. In order, these five types are cross-site scripting, SQL injection, path manipulation, dangerous function, and dangerous file inclusion. Figure 4 shows the types. Four of these five are found in the top five vulnerability types in MITRE's vulnerability type distributions in CVE report [4]. MITRE does not use the dangerous function category reported by Fortify and has buffer overflow as their fourth most common vulnerability type, which is not a flaw found in PHP applications though it can occur in the PHP interpreter itself.

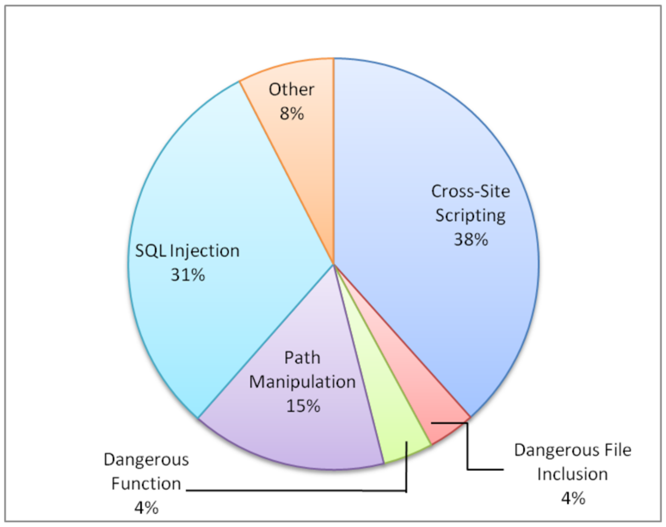

Figure 4. Initial Revision Types

The distribution of types of flaws changed between the initial and final revisions. While the five most common types remain the same, their ranks have changed and they only make up 87% of all vulnerabilities. Cross-site scripting remains the most common category of flaw, but SQL injection has dropped dramaticaly from 31% of the initial flaws to only 13% of the final vulnerabilities. Path manipulation flaws have increased their proportion slightly and are the second most common vulnerability in the final revision. Figure 5 displays the vulnerability type distribution in the final revision.

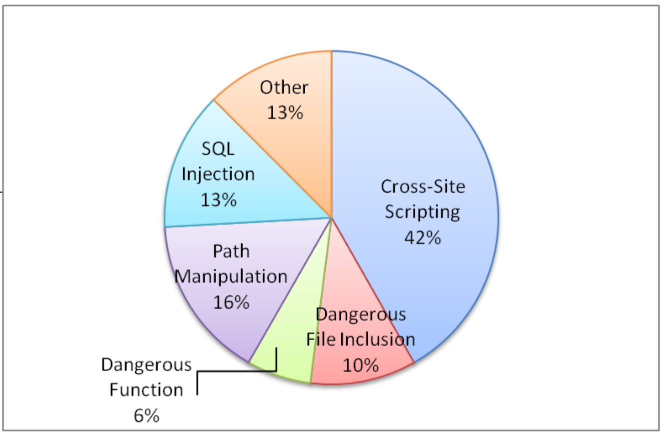

Figure 5. Final Revision Types

While the overall count of vulnerabilities decreased from the initial to the final revision, only six of the thirteen categories decreased in absolute numbers and there were more vulnerabilities in seven of the categories in the final revision. The six vulnerability types that decreased included the three most common types: cross-site scripting, SQL injection, and path manipulation. Figure 6 shows the changes in vulnerability count for each category.

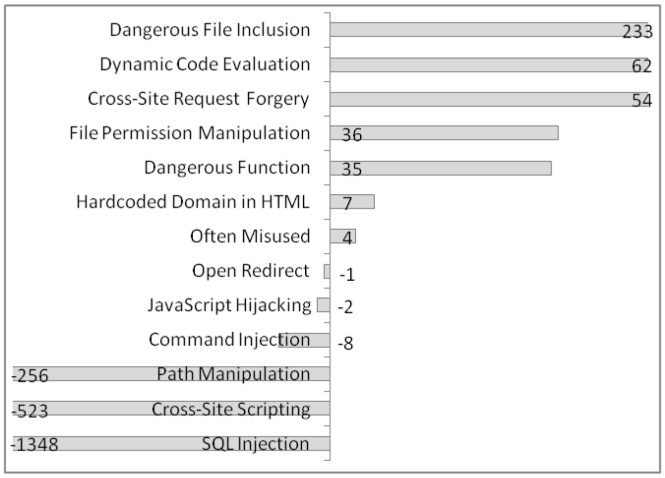

Figure 6. Vulnerability Type Changes over Time

6.1 Per-Project Vulnerability Type Analysis

The distribution of the types of flaws varied between projects, with no project having all types of flaws. Figure 7 presents the change in SAVD for the fourteen projects divided among the four dominant error types (SQL injection, path manipulation, dangerous file inclusion, and cross-site scripting). Two additional categories (all other ¿=0, all other ¡0) combine the vulnerability types with positive and negative negative changes for the project, respectively.

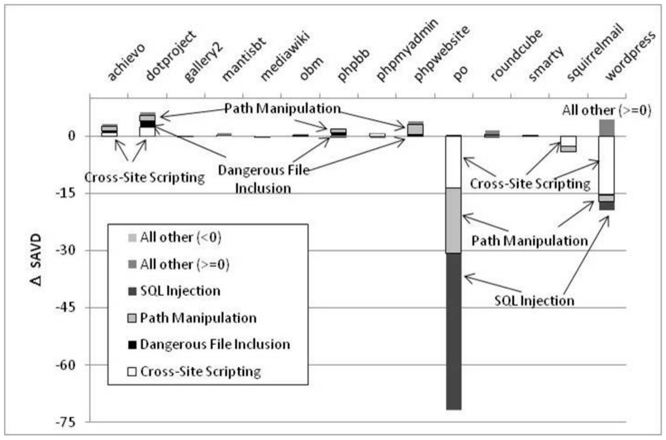

Figure 7. △SAVD vs. Vulnerability Types

Large improvements can be seen in figure 7 for po, squirrelmail, and WordPress. These improvements were largely made through the elimination of SQL injection, path manipulation, and cross-site scripting errors in these projects. Most other projects show small decreases in SQL injection vulnerabilities over the two years; only three projects saw a small increase of less than ten vulnerabilities for this category.

The large decrease in cross-site scripting vulnerabilities is due to improvements in three projects: WordPress, po and squirrelmail. Surprisingly, seven projects had more cross-site scripting vulnerabilities at the end of the study, and two of those (dotproject and phpmyadmin) added more than 100 vulnerabilities of this type. Path manipulation has a similar profile, though only a single project (dotproject) added more than 100 vulnerabilities. The same three projects showed the greatest improvement in path manipulation as in SQL injection. The vulnerability type with the largest increase, dangerous file inclusion, is only clearly visible on the graph for dotproject. No project saw a substantial decrease in this error.

The Spearman's rank correlation coefficient, ρ, was computed for all metrics and vulnerability types for all releases. Results are significant for $\rho > 0.1$ (p = 0.001). Using the Cohen Scale, only one correlation was found to be large (> 0.5), which was between maximum nesting complexity and dangerous file inclusion ($\rho = 0.526$). The metric with the greatest number of medium correlations ($0.3 \leq \rho < 0.5$) is maximum cyclomatic complexity, indicating that the larger the value of this metric, the more security flaws existed for the vulnerabilities listed in table 4.

Vulnerability	ρ
Command Injection	0.146
Cross-Site Scripting	0.310
Dangerous File Inclusion	0.445
Dangerous Function	0.458
Dynamic Code Evaluation	0.245
File Permission Manipulation	0.415
Hardcoded Domain in HTML	0.255
Open Redirect	0.325
Path Manipulation	0.403

Table 4. MaxCC and Vulnerability Correlations

Seven security vulnerabilities: command injection, cross-site request forgery, dynamic code evaluation, hardcoded domain in HTML, Javascript hijacking, often misused, and SQL injection have no correlations above 0.3, indicating that at best there are weak correlations for the metrics evaluated here.

7. Security Resource Indicator

To measure the importance of security to a project, we searched for public security resources made available on the web site for the project. The security resource indicator is based on four items: documentation of the security implications of configuring and installing the application, a dedicated e-mail alias to report security problems, a list or database of security vulnerabilities specific to the application, and documentation of secure development practices, such as coding standards or techniques to avoid common secure programming errors.

These indicators differ from those used by Fortify in their study of Java applications [10] in that we eliminated their indicator about easy access to security experts, which we found ambiguous, and we added the last two indicators described above, which are focused more on developers than users of the application. Table 5 shows the indicators for each project.

We developed a combined security resource indicator metric (SRI), which is the sum of the four indicator items, ranging from 0 to 4. Only one project, squirrelmail, had an SRI value of 4, while three projects had an SRI value of zero. SRI correlates strongly with change in vulnerability

Project	Security URL	Security Email	Vuln List	Secure Coding
achievo	no	no	no	no
dotproject	no	no	no	yes
gallery2	yes	yes	no	no
mantisbt	no	no	yes	no
mediawiki	yes	yes	no	yes
obm	no	no	no	no
phpbb	no	no	yes	no
phpmyadmin	yes	yes	yes	no
phpwebsite	no	yes	no	no
po	no	no	no	yes
roundcube	no	no	no	no
smarty	no	no	no	no
squirrelmail	yes	yes	yes	yes
wordpress	yes	yes	yes	no

Table 5. Security Resource Indicators

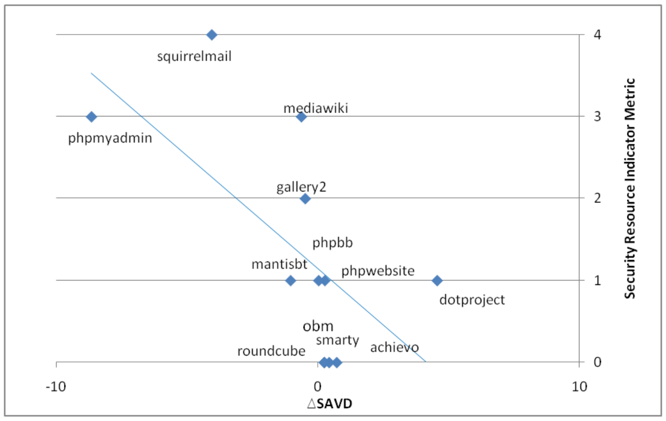

Figure 8. Security Resource Indicator vs. △SAVD

density from the initial to the final revision. The correlation had a significant (p ¡ 0.05) Spearman's rank correlation coefficient, ρ, of 0.67. Figure 8 shows SRI with change in SAVD. The Photo Organizer and WordPress applications are off the graph to the left, as both projects made large improvements in vulnerability density. A linear trendline for the displayed data points is included as a visual indicator of SAVD improvements in projects which focused on security.

8. Analysis Limitations

The fourteen open source PHP projects were the only projects found on `freshmeat.net` that met our analysis criteria. As such, this analysis describes these projects but extrapolation to other projects should be done with caution. The choice of language impacts both code size and types of possible vulnerabilities, so these results may not generalize to web applications written in other languages. An additional consideration when reviewing these data is that different software teams developed each web application and we do not know the details of the software processes used in their development.

All static analysis tools produce false positive results, where vulnerabilities are reported that are not actually present in the code. Our complete code base contained thousands of vulnerabilities in each of the 100 weekly samples, making it impractical to manually verify whether each vulnerability was a false positive or not. Instead, we examined the first and last revisions of two projects. These two projects contained 298 vulnerabilities, 54 of which were false positives. Our estimated false positive rate for Fortify Source Code Analyzer on this set of web applications is therefore 18.1%. The studies mentioned above that used static analysis tools to estimate vulnerability density or defect density did not estimate false positive rates.

9. Conclusion

We studied fourteen open source web applications written in PHP from Summer 2006 to Summer 2008, measuring static analysis vulnerability density, SLOC, cyclomatic complexity, nesting complexity, and churn for one revision each week. We found that the overall state of security of these applications was improving, as SAVD decreased from 8.88 vulnerabilities/KLOC to 3.30. While the number of NVD vulnerabilities was not correlated with SAVD, it was correlated with the popularity of the project.

Web applications vary widely in both their current number of vulnerabilities and in the evolution of vulnerability density over time. Vulnerability densities ranged from 0 to 121.4 at the beginning of the study and from 0.20 to 60.86 at the end of the study. Six of the applications showed a decreased vulnerability density over the course of the study, but the other eight increased their vulnerability densities.

We developed a security resources indicator metric to predict trends in vulnerability density and found that it was significantly correlated with the change in vulnerability density from the initial to the final revision. While three software metrics–maximum CC, average CC, and average nesting complexity–showed significant but small correlations with the total code base, no complexity or churn metric applied to every individual project.

In order to help determine whether the results of this paper can be generalized to a broader class of web applications, we plan to examine open source Java web application projects and compare them to the PHP web applications described in this paper. A wider variety of software measurement tools exist for Java than for PHP, which would enable us to study additional software metrics. Finally, we are going to examine the possibility of using our data to develop models for predicting the vulnerability density of software from software metrics.

10. Acknowledgments

We would like to acknowledge Dhanuja Kasturiratna for assistance with statistics, and we would like to acknowledge Fred Cohen for helpful criticism.

References

[1] B. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," Computer 34, 1 (Jan. 2001), 135-137.

[2] J. Cohen, *Statistical power analysis for the behavioral sciences (2nd ed.)* New Jersey: Lawrence Erlbaum 1988.

[3] S.M. Christey (CVE Editor), "Open Letter on the Interpretation of Vulnerability Statistics," http://seclists.org/bugtraq/2006/Jan/0060.html, January 4, 2006.

[4] S.M. Christey and R. A. Martin, `http://www.cve.mitre.org/docs/vuln-trends/index.html`, published May 22, 2007.

[5] `http://pear.php.net/package/PHP_CodeSniffer/` accessed January 4, 2009.

[6] Coverity, "Open Source Report 2008", `http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2008.pdf`, May 20, 2008.

[7] `http://cve.mitre.org/cve/editorial_policies/cd_abstraction.html`, accessed May 31, 2009.

[8] J. Evers, "Computer crime costs $67 billion, FBI says," `http://news.cnet.com/2100-7349_3-6028946.html`, January 19, 2006.

[9] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Brooks/Cole, Massachusetts, 1998.

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

[11] `http://freshmeat.net/`, accessed January 4, 2009.

[12] IBM Global Technology Services, "IBM Internet Security Systems X-Force 2008 Mid-Year Trend Statistics", `http://www-935.ibm.com/services/us/iss/xforce/midyearreport/`, published July 2008.

[13] N. Jovanovic, C. Kruegel, E. Kirda, "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.

[14] T.J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2(4), IEEE Press, New York, 1976, pp. 308-320.

[15] G. McGraw, B. Chess, and S. Migues, "Software [In]security: Software Security Top 10 Surprises", *informIT*, `http://www.informit.com/articles/article.aspx?p=1315431&seqNum=2`.

[16] N. Nagappan and T Ball, "Use of Relative Code Churn Measures to Predict System Defect Density", *Proceedings of the 27th International Conference on Software Engineering*, Association of Computing Machinery, New York, 2005, pp. 284 - 292.

[17] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density", *Proceedings of the 27th International Conference on Software Engineering*, Association of Computing Machinery, New York, 2005, pp. 580 - 586.

[18] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures", *Proceedings of the 28th International Conference on Software Engineering*, Association of Computing Machinery, New York, 2006, pp. 452 - 461.

[19] NVD, `http://nvd.nist.gov/`, accessed January 4, 2009.

[20] A. Ozment and S. E. Schechter, "Milk or Wine: Does Software Security Improve with Age?", *Proceedings of the 15th USENIX Security Symposium*, USENIX Association, California, 2006, pp. 93-104.

[21] R. Richardson, 12th Annual CSI Computer Crime and Security Survey, `http://www.gocsi.com/forms/csi_survey.jhtml`, 2007.

[22] Y. Shin and L. Williams, "An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics", *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*, Association for Computing Machinery, New York, 2008, pp. 315-317.

[23] Y. Shin and L. Williams, "Is Complexity Really the Enemy of Software Security?", *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.

[24] D.A. Wheeler, `http://www.dwheeler.com/sloccount/` accessed January 3, 2009.