An Empirical Study of the Evolution of PHP Web Application Security

Maureen Doyle, James Walden
Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099
{*doylem3, waldenj*}*@nku,edu*

***Abstract*—Web applications are increasingly subject to mass attacks, with vulnerabilities found easily in both open source and commercial applications as evinced by the fact that approximately half of reported vulnerabilities are found in web applications. In this paper, we perform an empirical investigation of the evolution of vulnerabilities in fourteen of the most widely used open source PHP web applications, finding that vulnerabilities densities declined from 28.12 to 19.96 vulnerabilities per thousand lines of code from 2006 to 2010. We also investigate whether complexity metrics or a security resources indicator (SRI) metric can be used to identify vulnerable web application showing that average cyclomatic complexity is an effective predictor of vulnerability for several applications, especially for those with low SRI scores.**

***Keywords*-security metrics; software security; static analysis; code complexity**

I. Introduction

Web applications are the source of almost half of all security vulnerabilities, representing 49% of all vulnerability disclosures reported in IBM's 2010 X-Force Trend and Risk Report [12]. This is probably an undercount as many organizations develop and deploy web applications in-house whose vulnerabilities are not reported publicly. MITRE found that the most common two vulnerability types since 2005 were cross-site scripting and SQL injection [5], which are primarily found in web applications.

While worms that exploit network or operating system vulnerabilities have largely disappeared, web applications are regularly targeted by mass attacks such as the April 2011 LizaMoon mass SQL injection attacks [14]. Furthermore, popular open source web applications such as WordPress are the subject of mass attacks designed to inject malware to infect users of their sites [24].

This study is a partial replication of our previous study of vulnerabilities in fourteen open source PHP web applications from 2006-2008 [23]. While replication of experiments is important in all areas of empirical software engineering, it is particularly important in studies of software security, due to the rapidly evolving nature of the field, with new types of vulnerabilities appearing each year. Software that is thought to be secure one year is discovered to be insecure the next due to an absence of measures to prevent a type of intrusion not known before. There were no worries about cross-site scripting (XSS) before most browsers supported Javascript in the 1990s or about clickjacking until 2008.

Like the original study, this paper analyzes relationships between software metrics and vulnerabilities measured using a static analysis tool in fourteen of the most widely used open source web applications, including 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 and complexity. This is the largest survey of web application security, both in terms of code size and number of application users.

There are two important differences between this study and our previous one. The first is that this study analyzes the web applications for four years, from 2006-2010, instead of two. The longer time span allows us to verify whether the relationships we discovered between complexity and vulnerabilities were the product of the state of source code in 2006-2008 or whether they hold more generally. The second is the use of a more recent version of the Fortify Source Code Analyzer (SCA) tool, which can identify 73 types of vulnerabilities in PHP code as opposed to only 13 types of vulnerabilities in the version we used in the original study, enabling us to find latent vulnerabilities that missed in the previous study.

Replicating our study over a longer time interval with newer static analysis tools enabled us to

1) Evaluate whether trends observed in the evolution of web application security vulnerabilities in open source PHP applications from mid-2006 to mid-2008 continue through mid-2010.
2) Determine if predictions of vulnerabilities that worked in 2008 would accurately predict vulnerability evolution through 2010.
3) Determine if the ability of static analysis tools to find vulnerabilities in web applications has advanced substantially from 2008, and to determine whether the effect of these improvements would alter the trends observed in the 2008 study.

After discussing our research methodology in section 2, we describe the data collection process in section 3, aggregate findings in section 4, vulnerability type analysis in section

5, and complexity metrics in section 6. Section 7 addresses threats to validity, while section 8 discussed related work. Section 9 completes the paper, giving conclusions and describing future work.

II. Research Methodology

In this section, we specify our research hypotheses based on both complexity metrics and the security resources indicator. These metrics have been shown to be effective for vulnerability prediction in certain domains [3], [11], [18], [22], [23]. We study several aspects of vulnerabilities, including the total number of vulnerabilities, the change in vulnerabilities over time, the vulnerability density (the number of vulnerabilities per thousand lines of code), the change in vulnerability density over time, as well as vulnerability metrics on a category basis.

We also examine changes in the aggregate code base of the fourteen projects. As vulnerabilities and techniques to mitigate vulnerabilities become more widely known, then we might expect that the vulnerability density of open source projects should improve over time, especially in terms of common vulnerabilities like cross-site scripting and SQL injection. This leads to three hypotheses:

1) The vulnerability density of open source web applications should decrease with time.
2) The density of cross-site scripting vulnerabilities in open source web applications should decrease with time.
3) The density of SQL injection vulnerabilities in open source web applications should decrease with time.

A. Code Complexity

Security experts claim that complexity is the enemy of security [16]. Complex code is difficult to understand, maintain, and test, making security vulnerabilities easier to introduce and more difficult to find and mitigate. As a result, complex code should include a larger number of security vulnerabilities than simple code. Based on this reasoning, we have two hypotheses on code complexity:

1) Applications with higher code complexity have a higher number of vulnerabilities than applications with a lower code complexity.
2) Applications with higher code complexity will increase the number of vulnerabilities in the code over time at a faster rate than applications with lower code complexity.

McCabe's cyclomatic complexity [15] and nesting complexity are popular metrics measuring code complexity, and both metrics have been used to identify vulnerabilities in software [18], [22]. Nesting complexity counts the depth of nested conditionals and loops. We examine these complexity metrics in three forms: maximum complexity, which is the complexity of the function which has the highest value for the complexity metric, total complexity, which is the sum of the complexity metric for the entire application, average complexity, which is the average value of the metric per function, and average complexity per file.

B. Security Resources Indicator

In our previous study, we created a metric to measure the importance of security to a project, based on public security resources made available on the web site for the project. The Security Resource Indicator (SRI) metric is based on four items: documentation of the security implications of configuring and installing the application, a dedicated email 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. SRI is the sum of the four indicator items, ranging from 0 to 4. These indicators differ from a similar set of indicators used by Fortify in their study of Java applications [9] 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.

We expect that projects whose developers focus their attention on security will improve the security of their code over time. If a project does not focus resources on security, we expect the project to operate in a reactive mode, fixing vulnerabilties as they are reported, while a security-focused project would use active measures such as developer education and code reviews to prevent vulnerabilities from being introduced into code. As a result, we have the following hypothesis:

1) Applications with high SRI values should see a decrease in vulnerability density over time.

III. Data Collection

In 2008, we examined the project history of open source web applications written in PHP that were selected from the most popular open source web applications on freshmeat.net, all of which had a subversion code repository with at least two years of history. Only fourteen applications met these criteria. Two projects were project management applications (achievo and dotproject), two were photo organizers (gallery2 and po), and two were webmail systems (roundcube and squirrelmail). Of the remaining applications, MantisBT is a bug tracking system, mediawiki is the wiki system that runs Wikipedia, phpbb is one of the most popular webforum systems, phpmyadmin is a MySQL administration interface, phpwebsite is a content management system, smarty is a templating engine, and WordPress is one of the most popular blogging engines.

A. Source Code

When we revisited the study in 2011, we found that three of the applications (dotproject, mantisbt, and po) had

Table I
PHP OPEN SOURCE WEB APPLICATIONS

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

migrated their repositories from subversion to git, a decentralized version control system. In order to collect data from all fourteen applications in a uniform fashion, we imported the subversion repositories of the other applications into git. Decentralized source code management systems offer data that centralized systems do not, but they also present problems to source code mining, as their history is more complex and can be easily modified by developers [1]. Since we imported some of our repositories from subversion, we could not take advantage of the additional information about commits provided by git, though the fact that git checkouts are orders of magnitude faster than subversion checkouts helped us process the data more quickly.

Another issue we found with git was that two of the applications that had migrated to git, dotproject and mantisbt, had more lines of code for commits in their git repositories than they had for the original commits in their subversion repositories. We did not observe this type of discrepancy with po or with any of the projects that we imported from subversion to git, which indicates that those two projects must have made changes beyond a simple git import when converting their repositories.

We created a Ruby script to import all of the repositories, and we stored our copies of the repositories locally using gitosis to manage them. In order to observe the projects at identical time intervals, we selected a single revision from each week from June 1, 2006 through June 1, 2010 to analyze, choosing the first change made during that week. If no commit was made during a week, we replicated data from the previous week. We could not use public releases of these applications for our study, since public releases are too few and irregular in schedule. It is common enough to deploy code directory from source code repositories that several of these projects have HowTo documents written to help non-technical users do this.

While the initial revision of one project (smarty) was small, with only 5750 lines of code, the size of first revisions of the remaining projects ranged from 25,000 to 150,000 lines. However, some of the applications grew considerably over the four year period, with mediawiki being the largest at 588,763 lines of code in its final revision. Two other projects (phpwebsite and obm) had final revisions that exceeded 200,000 lines of code.

B. Static Analysis

We chose static analysis as our technique to measure vulnerabilities instead of using reported vulnerabilities for several reasons. Unlike the process of manual code review or penetration testing which produces reported vulnerabilities, static analysis is an objective, repeatable, and scalable technique for measuring vulnerabilities. Static analysis tools apply the same algorithms and rulesets each time they are used, and can scan a project in a matter of hours rather than days or weeks. Vulnerabilities can remain latent in code for years before a researcher discovers and reports them [13], which means that reported vulnerabilities are an undercount of the actual number of vulnerabilities by an unknown amount.

Using static analysis enabled us to perform a fine-grained study of the evolution of an application's vulnerabilities. Static analysis enabled us to detect changes in the number of vulnerabilities on a weekly basis, observing vulnerabilities near both the time of their introduction and their removal, whereas reported vulnerabilities from a source like the National Vulnerability Database [19] are few in number compared to static analysis findings and are not found on any regular basis. We found an order of magnitude more vulnerabilities with our static analysis tool than are reported for the set of applications. Part of this difference can be explained by the Common Vulnerabilities and Exposures (CVE) guidelines, requiring that vulnerabilities of the same type in the same version of an application be merged into a single entry [8]. A more complete discussion of the issues in interpreting reported vulnerability statistics can be found in [4]. However, some SCA vulnerabilities are false positives, which are discussed with other threats to validity below. The false positive rate is not high enough to reduce the number of vulnerabilities found to a quantity similar to the number of reported vulnerabilities.

We used version 5.10 of the Fortify Source Code Analyzer as our static analysis tool. Since we needed to scan over a hundred revisions for each of our 14 applications, consisting of over 100 million line of code, we needed a way to perform the scans in parallel. BuildBot is a widely used continuous integration tool which automates the compilation and testing of application code pulled from source code repositories. It is designed to perform builds on a distributed set of heterogenous servers. The Mozilla project has a BuildBot farm of over 700 machines [2].

We created Ruby scripts to select weekly revisions from an application's git repository and automate the creation of Python configuration files for BuildBot 0.8.3. The BuildBot configuration contained instructions to scan each weekly revision using a shell script that checked out the source code, ran metric tools including SCA, and stored the resulting data on an NFS volume. SCA was run indirectly through scan-projects, a Ruby program designed to manage the multi-

step process to run SCA, handle errors, and summarize the resulting XML vulnerability report into a short CSV output file. Data stored included metadata, such as project name, command line options, location, and timestamps, the XML report and command output from SCA, SLOCCount output, and a CSV file summarizing the information from all of the metrics tools.

Our BuildBot farm consisted of five 64-bit Ubuntu Linux 10.04.2 servers, ranging from a dual-core 2GB machine capable of running one build at a time to a 16-core 16GB machine that can run six builds simultaneously. BuildBot would dispatch new revisions to each slave as soon as old revisions were completed. Builds that failed were reported through a web interface, which also provided links to rebuild any failed builds. The number of build failures ranged from zero to half a dozen, resulting from SLOCcount failures, but all succeeded on the first rebuild. Once all selected revisions of an application were scanned, we used another Ruby script to collect and reformat the data files from each revision into a single CSV file that could be imported into a statistics package for analysis.

C. Metrics

In addition to counting vulnerabilities, our BuildBot configuration collected and analyzed the following software metrics: SLOC, cyclomatic complexity, and nesting complexity. While SCA is a commercial static analysis tool, our other metric tools were open source tools packaged with Ubuntu Linux 10.04. Cyclomatic Complexity and Nesting Complexity were computed using PHP CodeSniffer 1.10 [6] with custom classes and a Ruby script to extract only the data we needed.

In our previous study, we used SLOCCount 2.26 [25] to measure SLOC. However, this tool returned inconsistent results when run multiple times on the same source code, sometimes failing to report results at all. Therefore, we used Fortify SCA to count code in this study, which returns consistent SLOC counts for code, and as it is the same tool that we used to find vulnerabilities, we expect that it counts the same code that it finds vulnerabilities in. We still run SLOCCount on each revision as part of a check to ensure that our BuildBot-based system produced the same results as our previous study. SLOC metrics from SCA are lower than those from SLOCCount, as SCA SLOC does not include lines with only braces or declarations.

We measured vulnerability density using the static analysis vulnerability density (SAVD) metric [7] We used results from Fortify SCA version 5.10 to compute SAVD, as SCA reported both the number of vulnerabilities and SLOC. This means that both the numerator and denominator of SAVD differ from the original study, which results in much higher SAVD values due to both the larger number of vulnerabilities found and the smaller code size values. Since Fortify SCA also categorized vulnerabilities into types such as cross-site scripting and SQL injection, we could also measure vulnerability density for particular types of vulnerabilities.

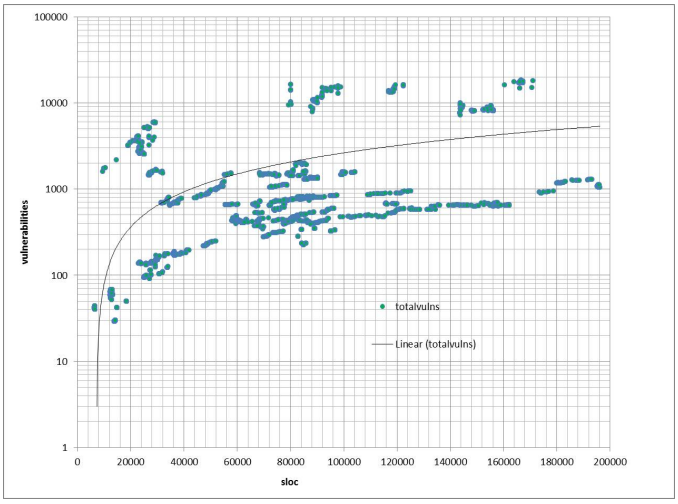

Figure 1. SLOC vs. Vulnerabilities for All Project Versions

Table II
AGGREGATE DATA

Datum	2006	2007	2008	2009	2010
SLOC	684,654	782,870	864,113	980,029	1,182,917
Vulnerabilities	19,253	17,404	24,529	24,707	23,613
OWASP Top 10	14,742	12,467	17,970	17,623	17,389
SAVD	28.12	22.23	28.38	25.21	19.96

IV. AGGREGATE FINDINGS

The size of the aggregate code base for all fourteen projects steadily grew from 684,654 sources line of code in mid-2006 to 1,182,917 lines of code in mid-2010. The number of vulnerabilities also grew throughout that time period from 19,253 to 23,613, though not steadily, as there was a large decrease in 2007 followed by a large increase in 2008. We found that the relationship between the number of vulnerabilities in a version and the code size was roughly linear and plotted in Figure 1 using a logarithmic scale. Clusters of data points are typically different versions from the same application. As the number of vulnerabilities grew slower than the the size of the code, vulnerability density decreased strongly from 28.12 vulnerabilities per thousand lines of code to 19.96. Table II summarizes the findings of this analysis.

While the overall trend was towards lower vulnerability density, individual projects evolved differently, with eight projects decreasing vulnerability density over the study and six projects increasing. In our previous study, only six projects had declining vulnerability densities. While SAVD declined for eight projects, the total number of vulnerabilities only declined for four of those projects: phpbb, po, smarty, and squirrelmail.

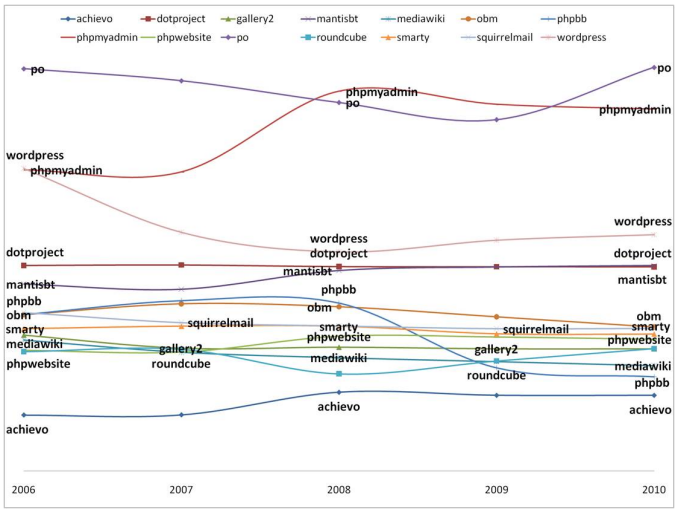

Figure 2. SAVD Evolution by Project

Vulnerability densities varied widely, from 2.1 vulnerabilities per thousand lines of code (achievo) to 202 (po) in June 2006, evolving to 2.7 (achievo) to 206 (po) in June 2010. Figure 2 shows the evolution of SAVD for each of the fourteen projects from year to year. Both of the highest SAVD projects, phpmyadmin and po, contained a set of vulnerabilities made up mostly of cross-site scripting vulnerabilities, and had hundreds of commits related to internationalization efforts, which caused swings of thousands of vulnerabilities within a week or two at times.

Figure 3 plots the SLOC and vulnerability counts for WordPress over the four-year period. In early 2007, a release of WordPress eliminated about a thousand vulnerabilities; however as new code was added, so were new vulnerabilities. WordPress contributors demonstrated that they can write more secure software; however, there are very few corrections being made after 2007. This approach seems characteristic of a project that does not have consistent security processes. However, squirrelmail, as shown in Figure 4, consistently fixed vulnerabilities over the four year period, without introducing a large number of new errors, even as the code grew in size.

V. Vulnerability Type Analysis

In this section, we examine the evolution of the frequency of vulnerability types over time. SCA 5.10 has the ability to identify 73 categories of vulnerabilities, 23 of which were detected in the fourteen projects. The Open Web Application Security Project (OWASP) identified the ten most critical web application vulnerabilities for 2010 [20]. As there is no standard categorization of vulnerabilities used by all static analysis tools, we also report on the subset of the SCA vulnerability types that match the OWASP Top 10 vulnerabilities. Note that some of the vulnerability types reported by SCA, such as Dangerous Function, do not

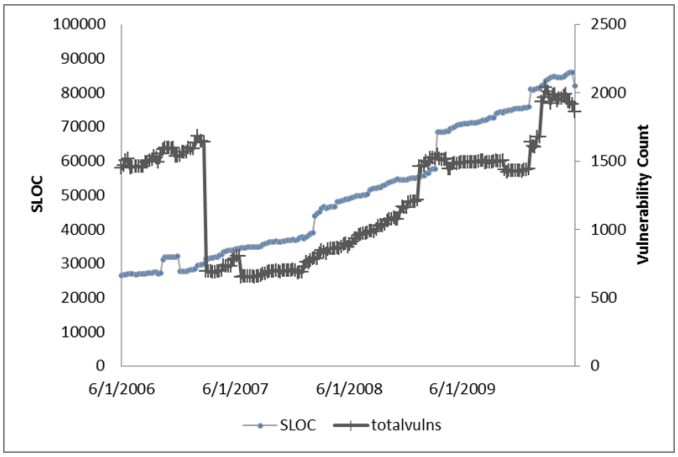

Figure 3. WordPress

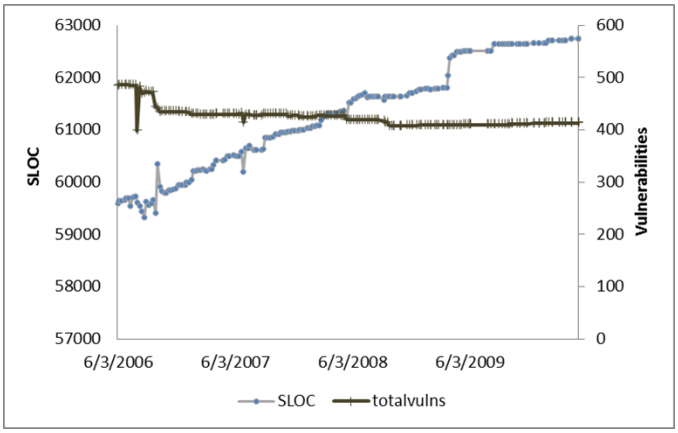

Figure 4. Squirrelmail

match any of the OWASP Top 10 vulnerabilities, while other Top 10 vulnerabilities match multiple SCA categories, such as Injection, which subsumes both the SQL Injection and Command Injection categories reported by SCA. Some Top 10 vulnerabilities, such as Unvalidated Redirects and Forwards, are not reported by SCA. The OWASP Top 10 vulnerabilities are listed in Table III.

Twelve of the 23 reported vulnerability types, representing over 70% of vulnerabilities found, are in the OWASP Top 10.

Table III
OWASP Top 10 Web Application Security Vulnerabilities

A1	Injection	A6	Security Misconfiguration
A2	Cross-Site Scripting	A7	Insecure Cryptographic Storage
A3	Broken Authentication and Session Management	A8	Failure to Restrict URL Access
A4	Insecure Direct Object References	A9	Insufficient Transport Layer Protection
A5	Cross-Site Request Forgery	A10	Unvalidated Redirects and Forwards

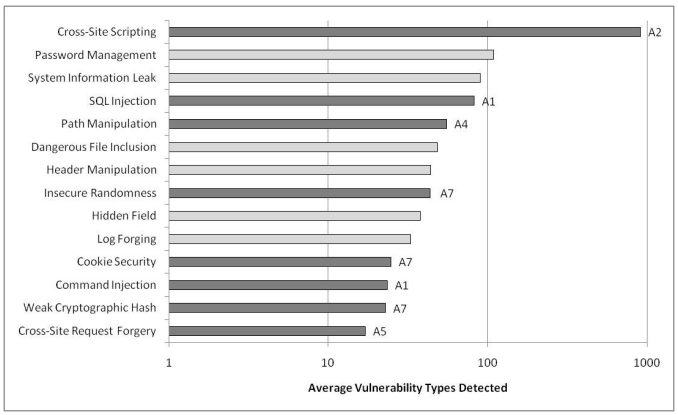

Figure 5. Aggregate Vulnerabilities by Type

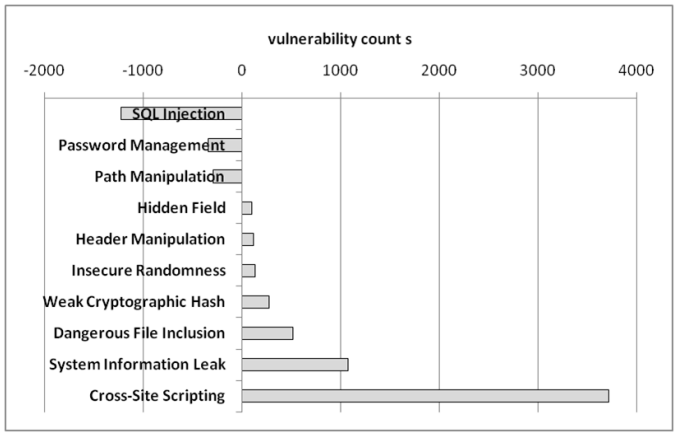

Figure 6. Change in Vulnerabilities by Type 2006-2010

The most frequently reported vulnerabilities included four of the top five OWASP web application security vulnerabilities, with cross-site scripting being the most common vulnerability in our code base as it is in the NVD. Figure 5 displays the top 14 vulnerabilities for the total code base. Vulnerabilities included in the OWASP Top 10 are emphasized by darker bars, with the OWASP vulnerability identifier beside them.

Figure 6 displays the ten largest changes in the number of vulnerabilities in each category. Two of our hypotheses concerned changes in the number of vulnerabilities in the most common categories:

1) The density of cross-site scripting vulnerabilities in open source web applications should decrease with time.
2) The density of SQL injection vulnerabilities in open source web applications should decrease with time.

While our data supports the second hypothesis, it contradicts the first one, as the number of cross-site scripting vulnerabilities increased over the four year time span during which the number of SQL injection vulnerabilities decreased.

Figure 7 shows the changes in vulnerability types for each

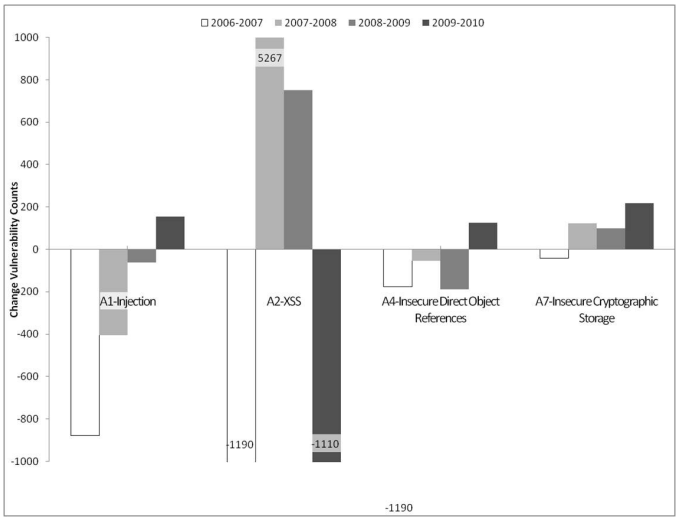

Figure 7. Annual Change in Top Vulnerabilities

year that we examined. When we look at the time evolution of the most common vulnerability types on an annual basis, we find that the change in vulnerabilities is not consistent from year to year. The alternation of adding and removing vulnerabilities implies that there is no consistent focus on software security in open source web applications.

VI. Analysis

This section is divided into three subsections. We analyze the projects individually on a weekly basis, we analyze the aggregate code base of all fourteen projects together on an annual basis, and finally we discuss analysis for the SRI metric.

The same analysis techniques are applied in each subsection. Spearman's rank correlation coefficients are computed between SAVD and cyclomatic and nesting complexity in four variants: maximum, average per function, average per file, and total. Spearman's rank correlations were selected, because Pearson's correlation assumes that the data follows a normal distribution.

A. Per-Project Analysis

Each web application project was analyzed on a weekly basis. No single metric correlated strongly with vulnerability density for all projects, as was also the case in our previous study. Only one metric, average cyclomatic complexity per function, had strong signficant correlations for more than three projects. Table IV summarizes the results. All are significant with $p < 0.0001$.

Five of the fourteen projects have strong correlations between cyclomatic complexity and SAVD, supporting our hypothesis that applications with higher complexity will have higher vulnerabilities; however, three projects, phpmyadmin (-0.53), smarty (-0.65), squirrelmail (-0.84), have strong negative correlations between average cyclomatic

Table IV
CORRELATIONS OF SAVD WITH AVGCC

SRI ≤ 2		SRI > 2	
Name	ρ	Name	ρ
achievo	-0.45	gallery2	-0.26
dotproject	0.12	mantisbt	0.62
obm	0.67	mediawiki	-0.39
phpbb	0.53	phpmyadmin	-0.53
phpwebsite	0.79	squirrelmail	-0.84
po	0.75	wordpress	-0.09
roundcube	-0.39		
smarty	-0.65		

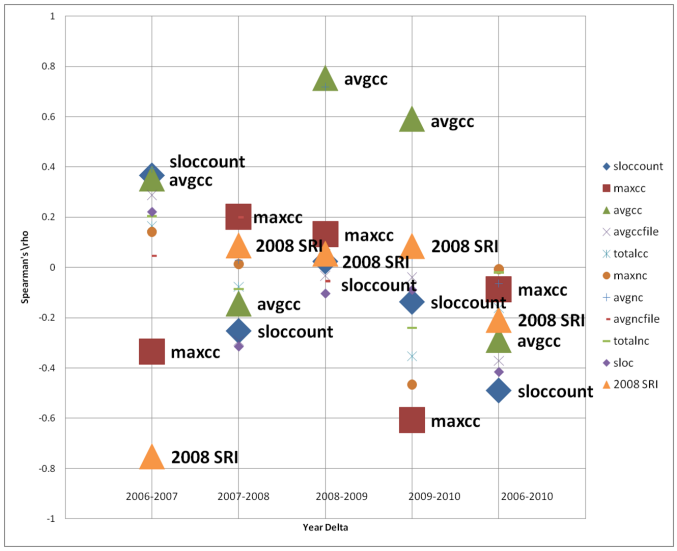

Figure 8. Correlations for Aggregate Code Base

complexity and SAVD. To explore the reasons for these differences, we examine the effect of SRI scores on these correlations.

Three of the eight projects with a SRI less than or equal to 2 negatively correlate to SAVD while only mantisbt had a high SRI and a positive correlation. The mostly negative correlations in the high SRI category show that those projects were better able to secure complex code than the positively correlated projects in the low SRI category. This result implies that the greater focus on security shown by high SRI scores indicates a capability to produce more secure code.

B. Aggregate Analysis

We examined the annual average metrics for all projects to determine if there were strong correlations for the change in metrics to the change in SAVD on a yearly basis. In our previous study, the differences were computed over the entire two year span.

The strongest correlation coefficient for the aggregate code base over multiple years is average cyclomatic complexity with correlation coefficients of 0.35, -0.14, 0.75, 0.59 for 2006-2007, 2007-2008, 2008-2009 and 2009-2010 respectively. Only two of the results: 0.75 ($p = 0.001$) and 0.59 ($p = 0.02$) are significant. Figure 8 shows the Spearman's ρ for all delta metrics measured versus delta in SAVD evaluated annually. Three metrics, average CC, maximum CC, and SRI, have larger icons indicating that they were signficant and stongly correlated for at least one time period. The average CC and 2008 SRI both appear promising, while the maximum CC does not, since it correlates negatively to SAVD. In 2009-2010, both maximum CC and SAVD increased; however, phpmyadmin had the largest increase in maximum CC while improving its SAVD slightly by reducing both its SLOC by 39K lines and vulnerabilities by 5660. The analysis was also done while removing phpmyadmin and po. In this analysis, the maximum CC correlation is not signficant, although it still negative ($\rho = -0.39$).

C. Security Resources Indicator

We found in 2009 that the SRI was a better predictor of improvements in vulnerability density than code size, complexity, or churn metrics for open source PHP web applications [23]. Table V shows SRI values measured in 2009 and 2011 for each project. The values for the four components are for the 2011 values. Only two projects, squirrelmail and wordpress, had perfect SRI scores of 4, while three projects had an SRI value of zero in both 2009 and 2011.

Table V
SECURITY RESOURCE INDICATOR

Project	SRI 2009	SRI 2011	Security URL	Security Email	Vuln List	Secure Coding
achievo	0	0	no	no	no	no
dotproject	1	2	no	yes	no	yes
gallery2	2	3	yes	yes	yes	no
mantisbt	1	3	yes	yes	yes	no
mediawiki	3	3	yes	yes	no	yes
obm	0	0	no	no	no	no
phpbb	1	1	no	no	yes	no
phpmyadmin	3	3	yes	yes	yes	no
phpwebsite	1	1	no	yes	no	no
po	1	1	no	no	no	yes
roundcube	0	0	no	no	no	no
smarty	0	0	no	no	no	no
squirrelmail	4	4	yes	yes	yes	yes
wordpress	3	4	yes	yes	yes	yes

We found that the 2009 SRI value correlated strongly with change in vulnerability density as measured by SCA 5.1 from 2006 to 2008, with a significant ($p < 0.05$) Spearman's rank correlation coefficient, ρ, of 0.67. However, the correlations of the change in vulnerability density from 2006 to 2010 as measured by SCA 5.10 with the 2009 and the 2011 SRI values were small ($\rho = 0.21$ and $\rho = 0.27$ respectively) and were not significant. The 2009 SRI had a strong correlation with SAVD for the 2006-2007 year ($\rho = -0.75$, $p = 0.001$), but there are weak correlations for

subsequent years with correlation coefficients of 0.08, 0.05, 0.09 for 2007-2008, 2008-2009 and 2009-2010 respectively. However, none of those three correlations are significant.

VII. Threats to Validity

Since our vulnerability data is based on static analysis results, our data includes false positives, where vulnerabilities are reported that are not actually present in the code. Manual code reviews to verify the tool's vulnerability reports for two open source PHP web applications yielded a false positive rate of 18%. This number is consistent with the false positive rate of less than 20% reported by Coverity [7]. Note that we did not attempt to exploit these vulnerabilities.

Our data also includes false negatives, in which the static analysis tool fails to report some vulnerabilities. Version 5.1 of Source Code Analyzer used in our 2006-2008 study reported only thirteen categories of vulnerabilities for PHP code, while version 5.10 used in the current study reports 73 categories of vulnerabilities (though only 23 of those categories were reported for the code that we examined.) While version 5.10 found more vulnerabilities, we detected the same trends in increasing or decreasing numbers of vulnerabilities for the 2006-2008 time period as we did with version 5.1. Of course, data based on reported vulnerabilities also includes latent unreported vulnerabilities. Indeed, during the course of our analysis, new vulnerabilities were reported in the NVD for our 14 projects.

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.

In order to replicate a study involving mining software repositories, researchers need the public availability of the raw data, the processed data, and of the tools and scripts used in the study [21]. To address threats to replicability, we plan to make the fourteen git repositories used publicly available, along with our public data, and the git repositories in which we store our scripts and tools. While we cannot provide copies of SCA, Fortify offers a free license to academic researchers and our data includes the XML reports produced by SCA and a Ruby gem providing an interface to extract information from those reports.

As with any empirical study, our results may not be generalizable to applications beyond the set of fourteen open source PHP web applications that we studied. To generalize the correlations described in this paper to other projects in different languages, application domains, or sizes would require additional studies.

VIII. Related Work

Shin et. al. [22] found correlations between complexity, churn, and developer metrics and vulnerable files in both Firefox and Linux. Models using all three types of metrics together predicted over 80% of the known vulnerable files. Nagappan et. al. [18] had mixed results, with three projects out of five showing strong correlations between defect density and cyclomatic complexity. Nagappan's group 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 in both nature and number.

Gegick and Williams [10] developed a model to identify vulnerable components using alert density from a static analysis tool. Gegick et. al. [11] combined static analysis alert density with code churn and SLOC to build prediction models to identify attack-prone components in a commericial telecommunications system.

Coverity has reported annually on their analysis of a large number of open source projects written in C and C++ [7] since 2008, finding roughly one static analysis defect per thousand lines of code on average. They found little change in the frequency of different vulnerability types reported since 2008, but their analysis included no web applications. Fortify analyzed a small number of Java projects [9] 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.

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 and are subject to a different set of vulnerability types.

IX. Conclusion

The goals of this study were to determine if the security of open source web applications was improving over time and to determine if complexity metrics or SRI could identify vulnerable applications. We found that the security of open source web applications improved from mid-2006 to mid-2010, with vulnerability density declining from 28.12 to 19.96 vulnerabilities per thousand lines of code. If we exclude the two most vulnerable applications, the decline is from 11.12 to 8.43. However, these numbers are still well above the approximately one vulnerability per KSLOC found in traditional desktop and server applications written in C and C++ [7], which demonstrates that web application security is not as mature as the security of traditional applications. That said, it is heartening to see a drop in SAVD, indicating that new code added to these applications introduced fewer vulnerabilities per KSLOC than the existing code base contained.

Individual projects varied considerably, with eight of the fourteen projects having declining vulnerability densities over time. Four of the projects declined in total number of

vulnerabilities too, with reductions in SQL injection vulnerabilities making up much of the decline. However, several thousand additional cross-site scripting vulnerabilities, many of them introduced in the process of internationalizing phpmyadmin and po, were added to the code base.

No single metric could distinguish high vulnerability web applications from low vulnerability ones. However, average cyclomatic complexity per function was an effective predictor for several applications, especially when SRI scores were used to classify applications into high and low security focus applications. Applications in the high SRI score did not have positive correlations of SAVD with complexity with the single exception of mantisbt, while most applications in the low SRI category had positive correlations of complexity with SAVD. The average CC and SRI combination will be explored in future work. By itself, SRI was not an effective predictor of decreasing SAVD except for the 2006-2007 annum.

ACKNOWLEDGMENT

The authors would like to thank John Murray for helping to write several tools to automate distributed static analysis of git repositories using BuildBot.

REFERENCES

[1] C. Bird et. al., "The Promises and Perils of Mining Git," pp.1-10, 2009 6th IEEE International Working Conference on Mining Software Repositories, 2009.

[2] BuildBot, http://trac.buildbot.net/wiki/SuccessStories, accessed June 4, 2011.

[3] I.Chowdhury and M.Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. 57, 3 (March 2011), 294-313.

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

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

[6] http://pear.php.net/package/PHP_CodeSniffer/ accessed June 4, 2011.

[7] Coverity, "Coverity Scan: 2010 Open Source Integrity Report," http://www.coverity.com/library/pdf/coverity-scan-2010-open-source-integrity-report.pdf, Nov 1, 2010.

[8] http://cve.mitre.org/cve/editorial_policies/cd_abstraction.html, accessed June 4, 2011.

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

[10] M. Gegick and L. Williams. 2007. "Toward the Use of Automated Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components." In Proceedings of the Second International Conference on Internet Monitoring and Protection (ICIMP '07). IEEE Computer Society, Washington, DC, USA.

[11] M. Gegick, L. Williams, J. Osborne, and M. Vouk. 2008. Prioritizing software security fortification through code-level metrics. In Proceedings of the 4th ACM workshop on Quality of protection (QoP '08). ACM, New York, NY, USA, 31-38.

[12] IBM Global Technology Services, "IBM Internet Security Systems X-Force 2010 Trend and Risk Report", http://www-935.ibm.com/services/us/iss/xforce/trendreports/, published March 2011.

[13] F. Massacci, S. Neuhaus, and V. Nguyen. 2011. "After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes." In Proceedings of the Third international conference on Engineering secure software and systems (ESSoS'11), Springer-Verlag, Berlin, Heidelberg, 195-208.

[14] F.Rashid, "LizaMoon Mass SQL Injection Attack Points to Rogue AV Site," eWeek, http://www.eweek.com/c/a/Security/LizaMoon-Mass-SQL-Injection-Attack-Points-to-Rogue-AV-Site-852537/, March 29, 2011.

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

[16] G. McGraw, *Software Security: Building Security In*, Boston, NY, Addison-Wesley, 2006.

[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 June 4, 2011.

[20] OWASP, https://www.owasp.org/index.php/Top_10_2010-Main, accessed June 4, 2011.

[21] G. Robles. "Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings." In Proceedings of the Working Conference on Mining Software Repositories, 2010.

[22] Y.Shin, A.Meneely, L.Williams, J.Osbourne, Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities, IEEE Transactions in Software Engineering, to appear, 2011.

[23] J.Walden, M.Doyle, G.Welch, M.Whelan, "Security of Open Source Web Applications," Proc. International Workshop on Security Measurements and Metrics (MetriSec'09), Lake Buena Vista, Florida, Oct. 14, 2009.

[24] Web Application Security Consortium, Web Application Hacking Incident Database, http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database, accessed June 4, 2011.

[25] D.A. Wheeler, http://www.dwheeler.com/sloccount/ accessed June 4, 2011.