

Using SQL Hotspots in a Prioritization Heuristic for Detecting All Types of Web Application Vulnerabilities

Ben Smith and Laurie Williams
Department of Computer Science
North Carolina State University
[ben_smith, laurie_williams]@ncsu.edu

Abstract

Development organizations often do not have time to perform security fortification on every file in a product before release. One way of prioritizing security efforts is to use metrics to identify core business logic that could contain vulnerabilities, such as database interaction code. Database code is a source of SQL injection vulnerabilities, but importantly may be home to unrelated vulnerabilities. The goal of this research is to improve the prioritization of security fortification efforts by investigating the ability of SQL hotspots to be used as the basis for a heuristic for prediction of all vulnerability types. We performed empirical case studies of 15 releases of two open source PHP web applications: WordPress, a blogging application, and WikkaWiki, a wiki management engine. Using statistical analysis, we show that the more SQL hotspots a file contains per line of code, the higher the probability that file will contain any type of vulnerability.

1. Introduction

We can get good designs by following good practices instead of poor ones.

-F. Brooks, Jr.

The war for a trustworthy Internet continues. The popular social networking site Twitter was recently compromised by two cross-site scripting attacks, which are common and easy-to-execute exploits of a code-level programming error [5]. Input validation vulnerabilities¹ like this are in the CWE/SANS Top 25 Most Dangerous Programming Errors for 2010² despite the plethora of proposed techniques for protecting

against code-level attacks (e.g. the context sensitive string evaluation method proposed by [11]). Additionally, the SANS list of Top Cyber Security Risks³ indicates that input validation vulnerabilities, such as SQL injection, cross-site scripting, and file inclusion continue to be the three most popular techniques used for compromising web sites.

Although techniques such as code reviews and design discussions can help developers reduce the number of vulnerabilities they introduce into the source code, the software development community currently has no single solution that will eliminate all security issues [7]. Furthermore, development organizations often do not have the time or resources to perform vulnerability detection efforts on every source file in a product before its release. Validation and verification (V&V) must be prioritized in such a way that the security fortification starts with the files that are most likely to be vulnerable first. SQL hotspots may help development organizations prioritize security fortification efforts. *SQL hotspots* (or just "hotspots" in this paper) are any point in the application source code where the system interacts with a database management system [3, 6]. Hotspots are typically associated with input validation vulnerabilities like SQL injection⁴ [3, 6], but they might also be useful for predicting *any* web application vulnerability since they protect the typical web application's most valuable asset: the database.

The goal of this research is to improve the prioritization of security fortification efforts by investigating the ability of SQL hotspots to be used as the basis for a heuristic for the prediction of all vulnerability types. We have already defined the identification of hotspots [14], and demonstrated that

¹ Input validation vulnerabilities occur when a system does not assert that input falls within an acceptable range, allowing the system to be exploited perform unintended functionality.

² <http://cwe.mitre.org/top25/>

³ <http://www.sans.org/top-cyber-security-risks/#summary>

⁴ *SQL injection vulnerabilities* occur when a lack of input validation could allow a user to force unintended system behavior by altering the logical structure of a SQL statement using SQL reserved words and special characters.

testers can target hotspots at the system level to expose error message information leakage vulnerabilities⁵ [15]. In this paper, we evaluate the ability of hotspots used in a model with number of lines of code to perform in prediction models that can help point testers to files in the source code that are likely to contain all types of web application vulnerabilities. We include lines of code in our model as a way of normalizing the number of SQL hotspots per file to make the comparison between files more accurate even as file sizes vary.

We built and analyzed a prediction model based on the security vulnerability reports of two open source PHP web applications: nine releases of WordPress⁶, a blogging application, and six releases of WikkaWiki⁷, a wiki management engine. We compared the evaluation of our model's ability to predict vulnerable files with a random guess calculated based on the distribution of vulnerabilities within each system.

The contributions of this paper are as follows:

- Empirical evidence that SQL hotspots can be used along with lines of code as the basis for a heuristic for prioritizing security V&V efforts because they are predictive of *all types of web application vulnerabilities*.
- A resultant design strategy that recommends separating the database concern of an application into a single file to produce a lower proportion of input validation vulnerabilities.

The rest of this paper is organized as follows. Section 2 presents background information related to vulnerability identification. Then, Section 3 reviews related work. Next, Section 4 presents our methodology for gathering and analyzing the vulnerability data. Section 5 presents the results of the study and Section 6 presents the limitations of this study. Finally, Section 7 concludes.

2. Background

According to the ISO, a vulnerability is "...an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy" [4]. Since no single validation or verification practice can detect every vulnerability in a system [7], we have to assume that the file may have latent, undiscovered vulnerabilities. We call files *vulnerable* that have been changed due to a vulnerability report. We call files that have not been changed due to vulnerability reports *neutral*.

A predictive model for classifying components as being either vulnerable or neutral will make either correct or incorrect classifications. As such, for a given test of the model, there are *true positives*, where the model correctly classifies a component as vulnerable, and *true negatives*, where the model correctly classifies the component as neutral. When the model is wrong, there are *false positives*, where the model classifies the component as being vulnerable, but the component was neutral, and *false negatives* where the model failed to identify a vulnerable component. The performance of a given model to classify components as being one of two binary options has often been evaluated using two measurements: precision and recall [10].

Precision is defined in Equation 1, where **tp** is the number of true positives identified by the model, and **fp** is the number of false positives identified by the model. Precision can be viewed as a measure of exactness that a model exhibits.

$$Precision = \frac{tp}{tp + fp} \quad (1)$$

Recall measures the number of vulnerable files the model retrieves, and is defined in Equation 2 where **tp** is the number of true positives, and **fn** is the number of false negatives.

$$Recall = \frac{tp}{tp + fn} \quad (2)$$

3. Related Work

Other researchers have empirically examined the vulnerability reports of open source applications to determine the best predictive models for vulnerability locations. Nehaus et al. [9] use their tool, Vulture, to predict vulnerable software components in versions of the Mozilla web browser. They demonstrate that vulnerabilities correlate with component imports and that component imports in the Mozilla web browser can be used to consistently and accurately predict vulnerable components. Specifically, Nehaus et al. found that certain imports are almost guaranteed to produce security problems with the importing component later in time.

Zimmerman et al. contend that predicting security vulnerabilities can be thought of as "searching for a needle in a haystack" since the vulnerabilities in their dataset are so small in number and produce a significant bias in the results [17]. These researchers analyzed the predictive power of classical software metrics such as complexity, churn, and code coverage by calculating the correlation coefficient of each metric

⁵ *Error message vulnerabilities* occur when the system does not correctly handle an exceptional condition, causing sensitive

⁶ <http://www.wordpress.org>

⁷ <http://www.wikkawiki.org>

with vulnerabilities discovered in the Windows Vista operating system. This analysis indicated that these classical metrics can be used in vulnerability prediction models with a high amount of precision, but low recall. Additionally, their results demonstrated that dependencies can be used in a predictive model with a high amount of recall, but low precision.

Gegick et al. [2] use code churn, lines of code to and static analysis alerts from the Fortify tool to predict vulnerable software components on a large telecommunications software system containing over one million lines of code that had been deployed to the field for two years. Gegick et al. determined that a model with churn and static analysis alerts were the most useful for predicting vulnerable files, and that models combining their chosen metrics were more effective than any metric on its own.

Meneely et al. use developer activity metrics to evaluate and predict vulnerable software components [8]. Developer activity metrics measure the amount of interaction occurs between developers by analyzing which files developers have touched within a small time period. These researchers performed an empirical case study on Red Hat Enterprise Linux 4 kernel and found that files developed by otherwise independent developer groups were more likely to contain a vulnerability. They also discovered that files with changes from nine or more developers were more likely to have a vulnerability than files changed by fewer than nine developers.

Shin and Williams [13] investigated the relationship between classical complexity metrics and vulnerabilities. Shin and Williams performed an empirical case study on the JavaScript Engine in the Mozilla application framework and discovered that nine complexity measures such as McCabe's cyclomatic complexity and nesting are weakly correlated with the number of vulnerabilities. These researchers indicate that complexity measures could be used as a predictor of security vulnerabilities in an application, but that other measures of complexity should be developed that more accurately capture the type of complexity that leads to security issues.

Shin, et al. [12] investigated whether complexity, code churn, and developer activity metrics could be used as effective discriminators of software vulnerabilities in two widely-used, open source projects. Shin et al. found that 24 of the 28 metrics they investigated were discriminative of vulnerabilities. Shin et al. found that using all three types of metrics together allowed the production of a model that predicted 80% of the known vulnerable files with less than 25% false positives for both projects.

Other authors have compared the security posture of applications by using static analysis alerts as a proxy measurement of reported vulnerabilities. Walden et al. [16] compare the security posture of web applications using PHP and web applications that use Java. Walden et al. introduce a security metric, CVD: the common vulnerability density. These researchers define CVD as the density per line of code for four different vulnerability types that are common to both Java and PHP. Walden et al. used the Fortify static analysis tool to gather the reported values of CVD for two revisions of 11 projects. They found that although PHP had a higher value for CVD on all of the projects, CVD was decreasing more quickly overall in the measured PHP projects than in the Java projects.

4. Methodology

We conducted two case studies to empirically investigate eight hypothesis related to hotspot source code locations and vulnerabilities reported in the systems' bug tracking systems. We present these hypotheses, as well their results, in Table 1. We will further explain the results in Section 5. Our hypotheses point to the research objective: to improve the prioritization of security fortification efforts by investigating the ability of SQL hotspots to be used as the basis for a heuristic for the prediction of all vulnerability types. We also include lines of code in our analysis as a way of improving the accuracy and predictive power of our heuristic along with SQL hotspots. Specifically, we look at the relationship between hotspots and files (H1-H2), the amount of code change as related to the vulnerability type (H3), the predictive ability of hotspots for any vulnerability type (H4-H5), and the effect that collocating hotspots can have on the number and types of vulnerability in a given system (H6-H8).

For these case studies, we analyzed the Trac issue reports for two open source web applications, WordPress⁸ and WikkaWiki⁹. Trac is a web-based issue management system, similar to Bugzilla¹⁰, which integrates Subversion¹¹ repository information. The details of our analysis are provided in Sections 4.2 through 4.5.

4.1. Selecting the Study Subjects

To improve the accuracy of tracing vulnerabilities to source code, we chose projects that use the Trac issue-management system. The Trac Users page¹² lists

⁸ <http://core.trac.wordpress.org/>

⁹ <http://wush.net/trac/wikka/>

¹⁰ <http://www.bugzilla.org/>

¹¹ <http://subversion.tigris.org/>

¹² <http://trac.edgewall.org/wiki/TracUsers>

Table 1. Results per Project

	WordPress	WikkaWiki
Releases Analyzed	Nine	Six
Security issue reports analyzed	97	61
Vulnerable files (over project's history)	26% (85 / 326)	29% (44 / 209)
Average number of hotspots (over project's history)	255	92
Average percent of files having at least one hotspot	14.2%	8.42%
Hypotheses[†] about files		
H1. The more hotspots a file contains per line of code, the more likely it is that the file contains any web application vulnerability.	True (Logistic Regression, $p < 0.05$)	True (Logistic Regression, $p < 0.05$)
H2. The more hotspots a file contains, the more times that file was changed due to any kind of vulnerability (not just input validation vulnerabilities).	True (Simple Linear Regression, $p < 0.0001$, Adjusted $R^2 = 0.4208$)	True (Simple Linear Regression, $p < 0.0001$, Adjusted $R^2 = 0.3802$)
Hypotheses about issue reports		
H3. Input validation vulnerabilities result in a higher number average repository revisions than any other type of vulnerability*.	True (MWW, $p < 0.05$)	True (MWW, $p < 0.05$)
Hypotheses about prediction		
H4. Hotspots can be used to predict files that will contain any type of web application vulnerability in the current release.	True (Predictive Modeling, see Table 2)	True (Predictive Modeling, see Table 3)
H5. The more hotspots a file contains, the more likely that file will be vulnerable in the next release.	True (Positive Coefficient on Predictive Models)	True (Positive Coefficient on Predictive Models)
Hypotheses comparing projects		
H6. The average number of hotspots per file is more variable in WordPress than in WikkaWiki.	True (F-test, $p < 0.000001$)	
H7. WordPress suffered a higher proportion of input validation vulnerabilities than WikkaWiki.	True (Chi-Squared, $p = 0.0692$)	
H8. In WordPress, more of the lines of code that were changed due to security issues were hotspots.	True (Chi-Square, $p < 0.00001$)	

*This finding is consistent with the report from SANS (see Section 1) that indicates that the most popular types of web application attacks are input validation vulnerabilities.

† Please note that we use the term "hypothesis" in this table with respect to *scientific* hypotheses and not *statistical* hypotheses.

the development teams who choose to report that they use the Trac issue-management system to track their defects. We selected the two projects for the case study (hereafter, our "subjects") by inspecting each of the projects on the Trac Users page for projects that had the following attributes.

- **Implemented in PHP** - We chose subjects that were written in PHP. Recent usage statistics indicate that 30% of web applications are implemented using PHP, which is more than any other framework¹³. We were also interested in controlling language-dependent factors of our analysis since we are not interested in comparing programming languages in terms of their security.

- **Database Interaction** - Since we are interested in studying the relationship between hotspots and vulnerabilities, we chose web applications that facilitated some type of database interaction.
- **Traceable Code Changes** - One of the main issues in selecting the subjects for this study is that we are interested in tracing vulnerabilities from the issue reports to the files containing the vulnerabilities by analyzing changes made in the Subversion repository. In Trac-based projects, a developer with commit access to the project would commit a set of changes to the repository that contained an approved version of a patch that users had either suggested or created themselves. The developer making the commit would make a comment on the issue report that said something similar to "Fixed by [3350]" which indicates that the issue was resolved by the repository revision number 3350.

¹³ From <http://trends.builtwith.com/framework>. ASP.NET follows a close second with 25%, and all other frameworks each comprise less than 20% of the web.

In both Wordpress and WikkaWiki, the developer communities were consistent about indicating that a given issue was fixed by a certain revision number in the repository.

- **Contained Security Issues** - Since we are interested in studying security vulnerabilities, we looked for subjects that contained more than five reported issues that were clear-cut security problems. When we examined a project's Trac web page, we browsed through the issue reports for the project (if there were any) and looked for those reports which we could classify using a CWE¹⁴ grouping. The CWE¹⁵ provides a unified list of prevalent security vulnerabilities with detailed descriptions, definitions, and a unique classification number. Particularly, we were interested in comparing the proportion of input validation vulnerabilities in each project, so we also produced a yes/no indicator for whether an issue report had a CWE classified input validation vulnerability. We had to manually determine the CWE classification by reading the issue report description and attempting to map this information to a CWE classification definition. Sometimes the issue description did not map to a CWE type, and in these cases, we determined that the issue report was not a security problem. When a project contained no security issue reports in its Trac web page, we rejected the project.

Using these criteria, and searching Trac's User page we arrived at two study subjects out of 532 possible subjects:

1) *WordPress* – advanced blog management software that requires the MySQL database management system v4.1.2 or greater. Recent usage statistics have indicated that 74% of websites that are running blogging software are using WordPress¹⁶. WordPress contains 138,967 source lines of code as determined by CLOC¹⁷. We examined issue reports on WordPress ranging from December 2004 through August 2009 and spanning nine public releases from Version 1.5 to Version 2.8. In WordPress, security issues are flagged using a user-specified indicator on Trac. We found that 88 out of the 6,647 (or 1.3%) total reported issues in WordPress were security-related. This low density of security-related reports is not uncommon [17].

2) *WikkaWiki* – a wiki management system that requires the MySQL database management system v3.23 or greater. WikkaWiki's website contains a list of 532 registered websites who have installed and are

actively using the software¹⁸. WikkaWiki contains 46,025 source lines of code. We examined issue reports in WikkaWiki from November 2005 through June 2009 and spanning six public releases from Version 1.1.6.1-1.1.6.6. WikkaWiki does not use a Trac flag to indicate security issues, so we manually examined every WikkaWiki issue to determine which of them were related to security by classifying them into a CWE category. We identified 61 out of the 884 (or 6.8%) reported issues as security-related.

4.2. Identifying Hotspots

We refer to our list of files and attributes and information about these files as the `files` dataset. We refer to our local copy of the Trac reports for each project, with attributes and information about these Tracs as the `tracs` dataset.

Both our study subjects accessed the database management system through the PHP-provided function `mysql_query()`. In WordPress, hotspots are wrapped in a class called `$wpdb`. Conversely, in WikkaWiki hotspots occur using a call to the `Query` function in the `Wakka` class.

Since manually identifying hotspots can be very time consuming, we wrote a script that parses the file structure and searches for all instances of the project-specific string that indicates the existence of a hotspot. Specifically, we manually inspected the code until we could create an appropriate regular expression. We created a matcher to use with this regular expression that would catch all the different source code forms for a hotspot in each project. We checked the internal correctness of this script on ten files (five from each project) by manually counting the hotspots present in these files and comparing the result to the number calculated by our script.

The script was 100% correct in each of the ten files. Our script stored the number of hotspots in each file from this procedure into the `files` dataset along with the absolute filename for the file being analyzed. The script also stored a "yes" or "no" indicating whether the number of hotspots found was greater than zero. Next, our script executed CLOC on the each project. CLOC produces a list of each PHP file in the project directory structure and the number of source lines of code in that file. Our script parsed the CLOC output and stored the value for source lines of code into the `files` dataset for each file in the project.

4.3. Mapping Vulnerabilities to Files

All files in the `files` dataset were initialized as neutral and then marked as vulnerable when they were determined to have any number of vulnerabilities by

¹⁴ <http://cwe.mitre.org/data/slices/2000.html>

¹⁵ <http://cwe.mitre.org/data/slices/2000.html>

¹⁶ <http://trends.builtwith.com/blog>

¹⁷ <http://cloc.sourceforge.net/>. Version 1.52.

¹⁸ <http://wikkawiki.org/WikkaSites>

the procedure in this section. We did not track the number of vulnerabilities in a given file and instead only marked the file as vulnerable or neutral since we are interested in predicting which files are vulnerable, not how many vulnerabilities are in each file. As described before, to be included in our study, projects had to include a developer-indicated repository revision for each reported issue in the Trac management system. If a file was changed due to a security issue that was reported in Trac, then the file is considered vulnerable for that release. We describe the process of determining whether an issue report is security-related (and thus indicates a vulnerability) in Section 4.4.

Since Trac integrates with the repository used for each project, we scripted the process of gathering the changed files, and what changes were made to each file due to a certain vulnerability report. If the Trac webpage that described the vulnerability also contained a link to a set of changes to the repository, then our script marked the file as vulnerable in the `files` dataset. If the webpage containing the description of the vulnerability contained no link to a set of changes, then the report was determined to not be a problem by the development team, or did not warrant a change in the current release of the system and was excluded from our analysis. Trac allows the user to view a text-only version of the changes that were conducted during a given repository revision over the web in diff format (by appending `?format=diff` to the end of the URL). Our script parsed this resultant diff webpage into data indicating the files that were changed as well as the number of lines changed for each file.

The entire process of analyzing Trac reports and tracing these vulnerabilities to files was achieved using a script that we created. The script stored the following information as extracted from the project Trac website into the `tracs` dataset: the unique Trac report identification number, the date the report was created, the number of repository revisions due to the report, the number of files changed, and the number of lines of code changed due to the report.

The script determined the revision number using the date stored in the `tracs` dataset and by following the logic described in this section, stored the release number the issue report belonged to into the `tracs` dataset. Issue reports can only refer to files that are in the repository at the time of the submission of the report. As such, before gathering either the hotspot information or the information about which files were vulnerable, we downloaded the release of the version of the repository referred to by the report in question. For example, in WordPress, we know that the file `wp-includes/script-loader.php` exists in WordPress

v2.1 but not in WordPress v2.0. We also know that `wp-includes/script-loader.php` is changed due to the cross-site scripting vulnerability reported in WordPress issue number 3937¹⁹. Thus, by downloading the repository revision associated with v2.1 (number 4782), we can include the `wp-includes/script-loader.php` file in our analysis and properly record the file as vulnerable due to issue 3937.

4.4. Classifying Vulnerabilities

To ensure that a non-issue was not mistaken for a vulnerability, we manually inspected and classified the 97 reported security issues for WordPress and the 61 reported issues for WikkaWiki according to their CWE classification. The CWE provides a straightforward summary for the vulnerability that was reported, such as “Data Leak Between Sessions” or “Forced Browsing”. We inferred the CWE classification based on the behavior that the reporting user described as being faulty as well as by the files that were changed due to the reported vulnerability.

In WikkaWiki, we found 22 reported issues where there was a reported security problem but no repository changes. For example, in issue number 293²⁰, a user requests a special configuration flag to force people surfing to their wiki to use HTTPS instead of HTTP. The developers denied this request, saying that the user need only to change the base URL for his or her website to start with `https://` instead of `http://`. Since the developers did not make any changes to the source code due to this ticket, the ticket was categorized as not having any code changes. WordPress contained 15 issue reports where a vulnerability was reported, but the repository does not show resultant changes to the current version. Therefore, our analysis proceeded with 75 WordPress vulnerabilities and 46 WikkaWiki vulnerabilities.

In WordPress, issues that were marked as security by WordPress developers were not always due to security. For example, issue number 2041²¹ reports a problem when updating to the latest version. The reporter’s instance of WordPress was throwing an error when trying to display images. The thread discussion eventually resolves the issue, and a patch is committed to the repository. However, this issue is incorrectly categorized as a security problem by the developers because the problem deals with correctly updating the WordPress instance. We encountered nine reports like number 2041 that we determined to be not security-

¹⁹ <http://core.trac.wordpress.org/ticket/3937>

²⁰ <http://wush.net/trac/wikka/ticket/293>

²¹ <http://core.trac.wordpress.org/ticket/2041>

related. We exclude these reports from our analysis in WordPress. Since we manually identified security issues in WikkaWiki, we classified issues as we identified them. As such, there were no issue reports in WikkaWiki that were not security-related. Therefore there were no reports to exclude.

We manually edited the `tracs` dataset to add the CWE classifier to each issue report that was security related. We were also interested in comparing the proportion of input validation vulnerabilities in each project as a part of our research hypotheses (H7), so we additionally added a variable to the `tracs` dataset that indicated a "yes/no" as to whether the Trac report in question was due to an input validation vulnerability. CWE classifies several vulnerability types as input validation vulnerabilities²² and we followed this classification in our analysis. We used the input validation vulnerability variable *only* for evaluating H7; we tested all other hypotheses and conducted the predictive modeling using the full dataset, irrespective of a reported vulnerabilities classification as input validation or non-input validation.

4.5. Detecting Changed Hotspots

We were also interested in comparing the amount of change due to problems with SQL hotspots in each project. To measure the amount of this change, we calculated the proportion of lines changed in each project that contained SQL hotspots. Using the procedure described in Section 4.3, we used our script to automatically examine each line of code that developers committed to fix security issues. We combined this technique with the script described in Section 4.2 to identify lines of code that developers committed to fix security issues that contained hotspots. We call the total number of lines of code that developers changed due to security issues Y , and the total number of lines of code within that subset that are also hotspots X . We calculated the proportion of lines of code changed due to security issues that were also hotspots X divided by Y .

4.6. Statistical Analysis

We used the R project for statistical computing to perform our statistical analysis of the data in this case study²³. We used the statistical tests provided by R to determine whether any differences we observed in any two samples occurred by chance or were statistically significant.

We used the **Mann-Whitney-Wilcoxon (MWW)** statistic to perform any population-based comparison

between two independent samples, such as between vulnerable and neutral files, or between files that contain hotspots and files that do not. The MWW test is a non-parametric determination of whether two independent samples of observations have equally large values. We used a non-parametric statistical test because we cannot assume that the outcomes in our data set are normally distributed. We also used the **Chi-Squared Test** to determine whether there was a statistically significance difference in the proportion of positive outcomes in two population groups. We also used the **F Test** to measure the difference in variance between two sample groups.

4.7. Predictive Modeling

We built logistic regression models to evaluate the number of hotspots as a predictor of whether or not a file contains any type of vulnerability in each project. We considered many alternatives for our modeling technique, and compared the precision and recall across all releases for each of the models using Weka²⁴. Based on these scores, Weka allowed us to see that logistic regression models consistently outperformed the other choices available in the modeling toolkit provided with Weka. Our model included a term for lines of code (LOC) because intuitively the larger a file is, the more likely a code change will occur in that file. Coincidentally, the model performed better with the LOC term than without.

Our logistic regression model included only hotspots and lines of code for the independent variables. Using Weka, we trained the model for each project using the information on vulnerable files from releases 1 to N , and then tested the model on release $N+1$. We repeated this process for each of the 15 releases of WordPress and WikkaWiki that information on vulnerabilities for use in training the model, for a total of eight comparisons in WordPress and five comparisons in WikkaWiki²⁵.

To evaluate our model, as well as the the ability of hotspots to predict whether a file will be vulnerable in the next release, we do not look at the precision and recall of the model by themselves since these measures give us no idea of how "difficult" the prediction is to make. A model that has a precision of 80% may seem imperfect, but this model would be far more useful than a model trained by the same data set with precision of 10%. The same goes for the model's recall. In light of this fact, we compared our model's precision and recall with a model that randomly

²² CWE-20, CWE-79, CWE-89, CWE-77, CWE-22, CWE-74, CWE-226, CWE-138, CWE-212, CWE-150

²³ <http://www.r-project.org/>, Version 2.9.2.

²⁴ <http://www.cs.waikato.ac.nz/ml/weka/>

²⁵ With two (N) datasets, a researcher can only make one ($N-1$) comparison.

assigned files as being vulnerable or neutral. To do better than random, our model must have better precision and recall than this random guess [1]. In many projects, the percentage of vulnerable files is far fewer than the percentages in our projects [17]. We designed our random guess model to assign files as being neutral or vulnerable according to the vulnerability distribution discovered in each project empirically. That is, we did not assign a "coin toss" guess of vulnerabilities, where the probability of being vulnerable is $p=0.50$. Instead, since the percentage of vulnerable files for WordPress is 26.1%, we created a model which gave the probability of a file being vulnerable as $p=0.261$. Similarly, for WikkaWiki, the percentage of vulnerable files is 29%, we assigned our model a probability of $p=0.29$. We ran the random guess for 10 trials. The results reported in this paper are the best precision and recall the random guess achieved in our trials.

5. Results

This section presents the results of our analysis.

5.1. Statistical Results and Predictive Modeling

For both projects, we performed the statistical tests as described in Section 4.6 to analyze the research hypothesis (H1-H8) described in the beginning of Section 4. We summarize the results in Table 1. In both projects, we found that the more hotspots a file contains the more likely that file will be vulnerable (H1), and the more changes developers will make to that file due to any type of vulnerability (H2). We found that issue reports related to input validation vulnerabilities result in a higher average number of repository revisions meaning that input validation vulnerabilities tend to require multiple fixes before the development team considers them fixed (H3).

We built logistic regression models to evaluate the number of hotspots as a predictor of whether or not a file is vulnerable (H4). In WordPress, our model had precision between 0.02 and 0.50, and the random guess had precision between 0.0 and 0.23. Our model had recall between 0.10 and 0.40 and the random guess had recall between 0 and 0.26. Our model had better precision than the random guess in five out of eight cases, and had better recall than the random guess in seven out of eight cases (see Table 2). In WikkaWiki, our model had precision between 0.04 and 1.0, and the random guess had precision between 0.0 and 0.13. Our model had recall between 0.09 and 1.0 and the random guess had recall between 0.0 and 0.11. Our model had better precision than the random guess in three out of five cases, and had better recall than the random guess in four out of five cases (see Table 3). The values for

precision and recall vary because the model's performance changed on each of the 15 versions of the projects we analyzed. As the model sees more vulnerable files, the model misses less vulnerabilities (higher recall), but also reports more false positives (lower precision) as it relaxes its criteria for choosing a vulnerable file.

The logistic regression model that Weka generated to predict whether files were vulnerable or not, based on number of hotspots and lines of code, consistently contained a positive coefficient for the term representing the number of hotspots for both projects. The positive coefficient indicates that a greater number of hotspots in a file in the current release results in a higher probability of that file containing any type of web application vulnerability (H5).

Based upon these research results, our prioritization heuristic is as follows: *More SQL and non-SQL vulnerabilities will be found in files that contain more hotspots per line of code.*

5.2. Comparing the Projects

In WordPress, the ability to interact with the database is exposed directly to whichever page is needing access through the `$wpdb -> query` function (and similar), which takes the SQL query in question as its parameters, and parses user input in a filtered manner into the query where required.

In WikkaWiki, however, the developers made a high-level design decision to separate the database concern to a specific source code location. Specifically, database access is required through a class, called `wakka`, which encapsulates the various queries into functions that perform database operations on behalf of the client page. The SQL hotspots found in WordPress are more spread out among the files in WordPress than in the files in WikkaWiki (H6). The developers of WikkaWiki have chosen to centralize the functionality related to database interaction into a single set of classes that contain all the SQL hotspots.

This set of classes abstracts interactions with the database into a set of semantic methods that hide the direct interaction with the database from the programmer. These methods provide the necessary implementation required for input validation techniques, such as sanitizing a URL to protect the system from URL manipulation attacks, and stripping any HTML returned to the user for any potentially dangerous tags. Centralizing database interaction also provides the convenience of not having to make changes throughout the code due to a single security problem. When there is a problem with database interaction in WikkaWiki, developers usually know immediately where to look. In WordPress, some time

Table 2. WordPress Model Performance Hotspots versus Random Guess

Release	Hotspot Model Precision	Hotspot Model Recall	Random Guess Precision	Random Guess Recall
2.0	0.50	0.10	0.14	0.10
2.1	0.38	0.13	0.20	0.17
2.2	0.43	0.32	0.23	0.26
2.3	0.28	0.21	0.11	0.17
2.5	0.19	0.18	0.04	0.05
2.6	0.12	0.40	0.00	0.00
2.7	0.31	0.40	0.09	0.07
2.8	0.02	0.17	0.00	0.00

and energy may be consumed looking around for the source of the problem.

Our data shows that this high-level design decision coincides with a key difference in the security posture of the two studied projects: WikkaWiki has less input validation vulnerabilities than WordPress. First, in the predictive models described in Section 5.1, the number of SQL queries in a given file was a better predictor of the vulnerability of that file in WordPress than in WikkaWiki. But this evidence is further supported by the fact that 29% of the total reported vulnerabilities in WordPress were input validation vulnerabilities, which is a higher proportion than the 18% of total reported vulnerabilities that were input validation vulnerabilities in WikkaWiki (H7). Finally, we examined the number of changed lines of code due to security vulnerabilities that were hotspots. In WordPress, this proportion was higher at 3% than WikkaWiki, where the proportion was only 0.7% (H8). The implications of H8 for the development team are that hotspots in a project that has utilized a design like the one in WikkaWiki are going to be changed less often because they are easier to maintain. *In short, designing a web application with all database interaction and input validation located in a single component can help decrease that applications' proportion of reported input validation vulnerabilities.*

6. Limitations

We can never find or know all vulnerabilities or faults in a given software system. As such, both WikkaWiki and WordPress will continue to have latent security vulnerabilities that are not included in this analysis. The way that hotspots were defined within the script may also have been incomplete and missed some instances of hotspots that did not resemble the chosen form. Similarly, the collection of vulnerability

Table 3. WikkaWiki Model Performance Hotspots versus Random Guess

Release	Hotspot Model Precision	Hotspot Model Recall	Random Guess Precision	Random Guess Recall
1.1.6.1	1.00	0.15	0.13	0.07
1.1.6.2	1.00	0.22	0.10	0.11
1.1.6.3	1.00	0.09	0.08	0.11
1.1.6.4	0.08	1.00	0.00	0.00
1.1.6.5	0.04	0.50	0.00	0.00

reports was analyzed by hand, but the changes due to those reports may have been incorrectly assigned due to the way the issue reports were interpreted by the collection script. The release number was chosen as a level of granularity, but some files may have only been present for part of a release, and our analysis would miss a vulnerability in these files. There may be some error in our classification of the issue reports. WordPress and WikkaWiki were chosen due to the availability of their issue reports, but there may be some unknown selection bias in our study due to the fact that both projects have such well-documented vulnerability histories and solid contributing developer communities. Also, our analysis and data gathering are limited to only the two projects we study in this paper. These results may not be repeatable in other systems with other architectures. Finally, compared to industrial products, WordPress and WikkaWiki are relatively small in terms of code size.

7. Conclusion

Hotspots appear to be key in protecting a web application against attacks because we can use prediction based upon hotspots' locations to target code inspection and testing. Developers and testers of web applications can use models based upon hotspots to predict where all types of web application vulnerabilities will be in the next release of the system. Also, testers and V&V teams can prioritize security fortification efforts to place files that these models indicate as likely vulnerable first, thus resulting in a web application with a better security posture. Our prioritization heuristic is as follows: *More SQL and non-SQL vulnerabilities will be found in files that contain more hotspots per line of code.*

Input validation vulnerabilities continue to be a prominent problem with no single solution. However, we have found empirical evidence that separating the concern of database interaction appears to improve the

security of an application with respect to the proportion of reported input validation vulnerabilities. Isolating database interaction into a single class has resulted in a lower proportion of input validation vulnerabilities reported on WikkaWiki, and fewer hotspots changed on WikkaWiki due to security issues. Future work should compare design choices like this to further investigate the effect these choices have on the security posture of web applications.

8. Acknowledgements

We would like to thank Andy Meneely for his guidance on the empirical data collection as well as the statistical analysis for this paper. We would also like to thank Yonghee Shin for introducing the notion of using SQL hotspots as an internal metric. This work is supported by the National Science Foundation under CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10. References

- [1] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861-874, 2006.
- [2] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification through code-level metrics," in *ACM Workshop on Quality of Protection (QoP2008)*, Alexandria, Virginia, 2008, pp. 31-38.
- [3] W. G. J. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *20th IEEE/ACM Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005, pp. 174-183.
- [4] ISO/IEC, "DIS 14598-1 Information technology - Software product evaluation," 1996.
- [5] J. Kirk, "Twitter Contains Second worm in a Week," in *PCWorld Business Center*, 2010, http://www.pcworld.com/businesscenter/article/206232/twitter_contains_second_worm_in_a_week.html.
- [6] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama, "Sania: syntactic and semantic analysis for automated testing against SQL injection," in *23rd Annual Computer Security Applications Conference*, Miami Beach, FL, 2007, pp. 107-117.
- [7] G. McGraw, *Software Security: Building Security In*. Reading, Massachusetts: Addison-Wesley Professional, 2006.
- [8] A. Meneely and L. Williams, "Secure open source collaboration: an empirical study of linus' law," in *ACM Conference on Computer and Communications Security (CCS2009)*, Chicago, Illinois, 2009, pp. 453-462.
- [9] S. Nehaus, T. Zimmerman, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *ACM Conference on computer and communications security*, Alexandria, Virginia, USA, 2007, pp. 529-540.
- [10] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Berlin Heidelberg: Springer, 2008.
- [11] T. Pietraszek and C. V. Berghe, "Defending Against Injection Attacks Through Context-Sensitive String Evaluation," in *Recent Advances in Intrusion Detection, Springer LNCS 3858*, Seattle, Washington, 2006, pp. 124-145.
- [12] Y. Shin, A. Meneely, L. Williams, and J. A. Osbourne, "Evaluating Complexity, Code Churn, and Developer Activity metrics as Indicators of Software Vulnerabilities," *Transactions on Software Engineering*, 2010, to appear. DOI 10.1109/TSE.2010.81.
- [13] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in *ACM workshop on Quality of protection (QoP2008)*, Alexandria, Virginia, 2008, pp. 47-50.
- [14] B. Smith, Y. Shin, and L. Williams, "Proposing SQL Statement Coverage Metrics," in *Software Engineering for Secure Systems (SESS2008), co-located with ICSE 2008.*, Leipzig, Germany, 2008, pp. 49-56.
- [15] B. Smith, L. Williams, and A. Austin, "Idea: Using system level testing for revealing SQL-injection related error message information leaks," *Lecture Notes in Computer Science*, vol. 5965, pp. 192-200, Symposium on Engineering Secure Software and Systems 2010 (ESSoS 2010), 2010.
- [16] J. Walden, M. Doyle, R. Lenhof, and J. Murray, "Idea: Java vs. PHP: Security Implications of Language Choice for Web Applications," in *Engineering Secure Software and Systems, Springer LNCS 5965*, Pisa, Italy, 2010, pp. 61-69.
- [17] T. Zimmerman, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in *International Conference on Software Testing (ICST 2010)*, Paris, France, 2010, pp. 421-428.