An Empirical Study on the Effectiveness of Security Code Review

Anne Edmundson[1], Brian Holtkamp[2], Emanuel Rivera[3], Matthew Finifter[4], Adrian Mettler[4], and David Wagner[4]

[1] Cornell University, Ithaca, NY, USA
[2] University of Houston–Downtown, Houston, TX, USA
[3] Polytechnic University of Puerto Rico, San Juan, Puerto Rico
[4] University of California, Berkeley, CA, USA

Abstract. With the rise of the web as a dominant application platform, web security vulnerabilities are of increasing concern. Ideally, the web application development process would detect and correct these vulnerabilities before they are released to the public. This research aims to quantify the effectiveness of software developers at security code review as well as determine the variation in effectiveness among web developers. We hired 30 developers to conduct a manual code review of a small web application. The web application supplied to developers had seven known vulnerabilities, including three different types: Cross-Site Scripting, Cross-Site Request Forgery, and SQL Injection. Our findings include: (1) none of the subjects found all confirmed vulnerabilities, (2) more experience does not necessarily mean that the reviewer will be more accurate or effective, and (3) reports of false vulnerabilities were significantly correlated with reports of valid vulnerabilities.

1 Introduction

With the widespread adoption of and reliance on the Internet, web applications are playing an increasing role in our everyday life. With such a large user base, web applications have become a prime target for attackers who wish to hijack websites or to steal user information. Unfortunately, it is common for these applications to be susceptible to attacks. Web application vulnerabilities primarily result from bugs in application-specific code. These arise due to a widespread lack of expertise about web security among developers, and frequently involve departures from coding best practices.

Ideally, web applications would be free of vulnerabilities and therefore secure. While it is difficult to determine whether any vulnerabilities remain in an application, it is generally believed that an application is more secure when it has fewer vulnerabilities. It is therefore common for software developers and companies to make efforts to find and eliminate vulnerabilities in their software. Two common ways of accomplishing this are by manually reviewing source code and by using automated tools that are capable of identifying vulnerabilities.

In this study we focus on one of these techniques: manual code review. Specifically, we aim to measure the effectiveness of manual code review of web applications for improving their security. We used a labor outsourcing site to hire 30 web developers

J. Jürjens, B. Livshits, and R. Scandariato (Eds.): ESSoS 2013, LNCS 7781, pp. 197–212, 2013.

with varying amounts of security experience to conduct a security code review of a simple web application. These developers were asked to perform a line-by-line code review of the application and submit a report of all security vulnerabilities found. Using the data we collected:

- We quantified the effectiveness of developers at security code review.
- We estimated the optimal number of independent reviewers to hire to achieve a desired degree of confidence that all bugs will be found.
- We measured the extent to which developer demographic information and experience can be used to predict effectiveness at security code review.

These results may help hiring managers and developers in determining how to best allocate resources when securing their web applications.

2 Goals

In this work, we conduct an exploratory analysis of software developers' effectiveness in conducting security code review. We are interested in determining: (1) how effective developers are at conducting code reviews and the degree of variation among them, (2) the optimal number of independent reviewers to hire, and (3) whether we can predict in advance which developers will be most effective at performing a security review.

2.1 Effectiveness

Our research measures how well developers conduct security code review. In particular, we are interested in how effective they are at finding exploitable vulnerabilities in a PHP web application, and how much the effectiveness varies between reviewers. We are interested in answering the following questions:

1. What fraction of the vulnerabilities can we expect to be found by a single security reviewer?
2. Are some reviewers significantly more effective than others?
3. How much variation is there between reviewers?

2.2 Optimal Number of Reviewers

Depending on the distribution of reviewer effectiveness, we want to determine the best number of reviewers to hire. Intuitively, if more reviewers are hired, then a larger percentage of vulnerabilities will be found, but we want to determine the point at which an additional reviewer is unlikely to uncover any additional vulnerabilities. This would be useful for determining the best allocation of resources (money) in the development of a web application. Specifically, we will address the following questions to find the optimal number of reviewers.

4. Will multiple independent code reviewers be significantly more effective than a single reviewer?
5. If so, how much more effective?
6. How many reviewers are needed to find most or all of the bugs in a web application?

2.3 Predicting Effectiveness

We asked each reviewer about the following factors, which we hoped might be associated with reviewer effectiveness:

- Application comprehension
- Self-assessed confidence in the review
- Education level
- Experience with code reviews
- Name and number of security-related certifications
- Experience in software/web development and computer security
- Confidence as a software/web developer and as a security expert
- Most familiar programming languages

Identifying the relationship between reviewers' responses to these questions and their success at finding bugs during code review may provide insight into what criteria or factors would be most predictive of a successful security review.

3 Experimental Methodology

To assess developer effectiveness at security code review, we first reviewed a single web application for security vulnerabilities. We then hired 30 developers through an outsourcing site and asked each of them to perform a manual line-by-line security review of the code. After developers completed their reviews, we asked them to tell us about their experience and qualifications. Finally, we counted how many of the known vulnerabilities they found.

3.1 Anchor CMS

We used an existing open-source web application for the review, Anchor CMS. Anchor CMS is written in PHP and JavaScript and uses a MySQL database. We chose this application for our study due to (1) the presence of known vulnerabilities in the code, (2) its size, which was substantial enough to be nontrivial but small enough to allow security review at a reasonable cost, and (3) its relatively permissive license, which let us anonymize the code, as described below.

There are currently four release versions of Anchor. We chose to have reviewers review the third release, version 0.6, instead of the latest version. This version had more known vulnerabilities while still having comparable functionality to the latest version.

To prepare and anonymize the code for review, we modified the Anchor CMS source code in two ways. First, we removed the Anchor name and all branding. We renamed it TestCMS, a generic name that wouldn't be searchable online. We did not want developers to view Anchor CMS's bug tracker or any publicly reported vulnerabilities; we wanted to ensure they reviewed the code from scratch with no preconceptions. Our anonymization included removal of "Anchor" from page titles, all relevant images and logos, and all instances of Anchor in variable names or comments.

Once the code was anonymized, we modified the code in two ways to increase the number of vulnerabilities in it. This was done in order to decrease the role of random

noise in our measurements of reviewer effectiveness and to increase the diversity of vulnerability types. First, we took one vulnerability from a prior release of Anchor (version 0.5) and forward-ported it into our code. After this modification, the web application had three Cross-Site Scripting vulnerabilities known to us and no Cross-Site Request Forgery protection throughout the application.

Second, we carefully introduced two SQL injection vulnerabilities. To ensure these were representative of real SQL injection vulnerabilities naturally seen in the wild, we found similarly structured CMS applications on security listing websites (SecList.org), analyzed them, identified two SQL injection vulnerabilities in them, adapted the vulnerable code for Anchor, and introduced these vulnerabilities into TestCMS. The result is a web application with six known vulnerabilities. Our procedures were designed to ensure that these vulnerabilities are reasonably representative of the issues present in other web applications.

These six known vulnerabilities are exploitable by any visitor to the web application; he need not be a registered user. Additionally, the vulnerabilities are due solely to bugs in the PHP source code of the application. For example, we do not consider problems such as Denial of Service attacks or insecure password policies to be exploitable vulnerabilities in this study. Although these issues were not included in our list of six known vulnerabilities, we did not classify such reports as incorrect. Section 4.1 contains more details on how we handled such reports. Lastly, any vulnerabilities in the administrative interface were explicitly specified as out of scope for this study.

3.2 oDesk

oDesk is an outsourcing site that can be used to hire freelancers to perform many tasks, including web programming, development, and quality assurance. We chose oDesk because it is one of the most popular such sites, and because it gave us the most control over the hiring process; oDesk allows users to post jobs (with any specifications, payments, and requirements), send messages to users, interview candidates, and hire multiple people for the same job [1]. We used oDesk to publicize our study, hire developers that met our requirements, and pay our subjects for their work.

3.3 Subject Population and Selection

We recruited subjects for our experiment by posting our job on oDesk. We specified that respondents needed to be experienced in developing PHP applications in order to comprehend and work with our codebase, and they should have basic web security knowledge. We screened all applicants by asking them about how many times they have previously conducted a code review, a security code review, a code review of a web application, and a security code review of a web application. We also asked four multiple-choice quiz questions to test their knowledge of PHP and security. Each question showed a short snippet of code and asked whether the code was vulnerable, and if so, what kind of vulnerability it had; there were six answer choices to select from. We accepted all respondents who scored 25% or higher on the screening test. This threshold was chosen because it allowed us to have a larger sample size, while still ensuring some minimum level of knowledge and understanding of security issues.

3.4 Task

We gave participants directions on how to proceed with the code review, an example vulnerability report, and the TestCMS codebase. The instructions specified that no automated code review tools should be used. Also, the developers were told to spend 12 hours on this task; this number was calculated based upon a baseline of 250 lines of code per hour, as suggested by OWASP [2]. We designed a template that participants were instructed to use to report each vulnerability. The template has the following sections for the developer to fill in accordingly:

1. Vulnerability Type
2. Vulnerability Location
3. Vulnerability Description
4. Impact
5. Steps to Exploit

The type and location gave us basic information about the vulnerability. The template included "Vulnerability Description" and "Impact" sections in order to deter developers from using automated tools; it would be more challenging to successfully fill out these sections if a tool was used as opposed to a manual review. The last section, "Steps to Exploit", was intended to encourage developers to report only exploitable vulnerabilities as opposed to poor security practices in the code.

The developers were asked to review only a subset of the code given to them. In particular, we had them review everything but the administrative interface and the client-side code. They reviewed approximately 3500 lines of code in total. We specified our interest only in exploitable vulnerabilities. In return, we paid them $20/hour for a total of $240 for the completed job. This fixed rate leaves the relationship between compensation and reviewer effectiveness an area for future work.

3.5 Data Analysis Approach

Before the study, we scoured public vulnerability databases and Anchor's bug tracker to identify all known vulnerabilities in TestCMS. This allowed us to identify a "ground truth" enumeration of vulnerabilities, independent of those the reviewers were able to find. We manually analyzed each participant's report and evaluated the accuracy and correctness of all bugs they reported, which we describe in more detail in Section 4.1. After running statistical tests on the data, we were able to quantify how well developers conducted their reviews.

3.6 Threats to Validity

oDesk Population. As stated previously, we hired developers through the oDesk outsourcing website. This limits our population to registered oDesk users, as opposed to the population of all web developers or security reviewers. If the population of oDesk users differs significantly from the population of all web developers or security reviewers, then our results will not necessarily generalize to this larger population. However, given the success of oDesk, the population we study is interesting in its own right, and, at the very least, relevant to anyone hiring security reviewers using oDesk.

Artificial Vulnerabilities. As mentioned in Section 3.1, we introduced two SQL Injection vulnerabilities. Adding these artificial vulnerabilities creates an artificially flawed codebase where the application's original developer did not introduce all of the bugs. These artificial vulnerabilities could bias the results since it may make the code review easier or harder than reviewing a "naturally buggy" application. The changes made to the codebase were modeled after vulnerabilities found in other CMSs, which we hope will serve to minimize the artificiality of the codebase by ensuring that real web developers made the same mistakes before.

Static Analysis Tools. Using an outsourcing website to conduct our experiment restricted our ways of verifying how developers performed the review. This was considered when designing the experiment; we required specific and detailed information about each vulnerability reported. Requiring this information forced the developer to fully understand the vulnerability and how it affects the application, thereby minimizing the possibility that a reviewer would use static analysis tools and maximizing our ability to detect the use of static analysis tools. While there was no guarantee that developers completed their reviews manually, we assumed that all vulnerability reports completed sufficiently and accurately were a result of a manual code review and consequently we included them in our results.

Security Experts vs. Web Developers. We hired 30 reviewers for this study, some of whom specialized in security while others were purely web developers. Despite the use of a screening test, it is possible that web developers guessed correctly or that the screening questions asked about vulnerabilities that were significantly easier to detect than those found in a real application. In this case, web developers would be at a disadvantage. Security experts have a better understanding of the attacks, how they work, and how attackers can use them. It is possible that our screening process was too lenient and caused us to include web developers who are not security experts and would not get hired in the wild for security review. This could bias our results when measuring variability and effectiveness, but we anticipate that people hired to perform a security review in the wild may also include web developers who are not specialists in security.

Difficulties of Anchor Code and Time Frame. We found that the design of Anchor's source code made it particularly challenging to understand. It is neither well-documented nor well-structured. With the 12 hours that the reviewers had, results might not be the same as if the code were better-designed. The developers may not find all the vulnerabilities or they may give us many false positives. We have no way to verify that each person we hired spent the full 12 hours requested on the security review task, or that this was an accurate reflection of the amount of time they would spend on a real-world security review. If either of these conditions fails to hold, it could limit the applicability of our results.

Upshot. All empirical studies have limitations, and ours is no exception. Our hope, however, is that this study will encourage future studies with fewer or different

limitations and that ultimately, multiple different but related empirical studies will, when taken together, give us a better understanding of the questions we study.

4 Results

4.1 Vulnerability Report Classification

In order to understand the developers' code reviews and provide results, we first had to classify each vulnerability report submitted. We place each vulnerability report into one of four categories: valid vulnerability, invalid vulnerability, weakness, or out of scope. A valid vulnerability corresponds to an exploitable vulnerability in the web application. If a developer identified one of the vulnerabilities we knew about or her steps to exploit the reported vulnerability worked, then this would be classified as valid. We made no judgments on how well she described the vulnerability. Additionally, if the developer identified a known vulnerability, but her steps to exploit it were incorrect, it was still classified as valid. When tallying the correctly reported vulnerabilities, each valid vulnerability added to this tally.

An invalid vulnerability is a report that specifies some code as being vulnerable when in fact it is not. We verified that an invalid vulnerability was invalid by attempting to exploit the application using the steps the developer provided, as well as looking in the source code. When tallying the number of falsely reported vulnerabilities, each invalid vulnerability added to the tally. Both valid and invalid vulnerabilities were included in the tally of total reported vulnerabilities.

The weakness category describes reports that could potentially be security concerns, but are either not exploitable vulnerabilities or are not specific to this web application; for example, any reports of a denial of service attack or insecure password choice are considered weaknesses. Weaknesses contributed to the total number of reports each developer submitted, but were considered neither correct nor false. Reports involving vulnerabilities in the administrative interface were placed in the out of scope category, as we asked developers not to report these vulnerabilities. Reports in this category as well as duplicate reports were ignored; they were not considered when counting the developer's total reports, valid reports, or invalid reports.

As we manually checked through all reports from all developers, we encountered one valid vulnerability reported by multiple reviewers that was not known to us when preparing the experiment. Therefore, we adjusted all other totals and calculations to take into account all seven known vulnerabilities (as opposed to the six vulnerabilities initially known to us). This new vulnerability is an additional Cross-Site Scripting vulnerability, raising the number of known Cross-Site Scripting vulnerabilities in TestCMS to four.

4.2 Reviewer Effectiveness

We measured the relative effectiveness of each developer by counting the total number of valid reports from that developer and calculating the fraction of reports that were valid. We looked at how many vulnerabilities were found and which specific vulnerabilities were found most and least commonly. One out of the four Cross-Site Scripting

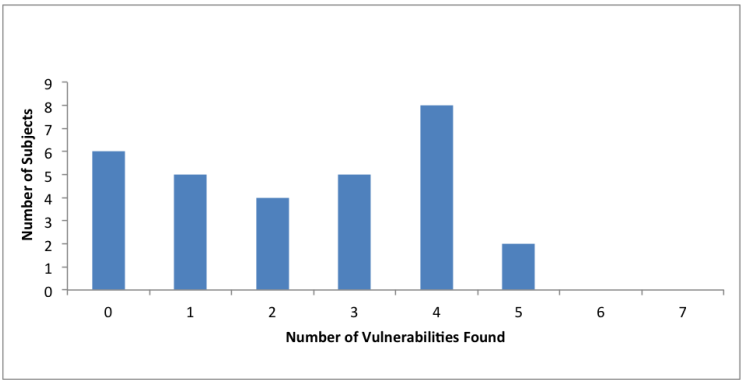

Fig. 1. The number of vulnerabilities found by individual developers

Table 1. The proportion of developers who reported each vulnerability

Cross-Site Scripting 1	37%
Cross-Site Scripting 2	73%
Cross-Site Scripting 3	20%
Cross-Site Scripting 4	30%
SQL Injection 1	37%
SQL Injection 2	20%
Cross-Site Request Forgery	17%

vulnerabilities was found by the majority of subjects, while only 17% of the reviewers found the lack of Cross-Site Request Forgery protection. Table 1 shows the percentage of developers who reported each vulnerability. The average number of correct vulnerabilities found was 2.33 with a standard deviation of 1.67. Figure 1 shows a histogram of the number of vulnerabilities found by each developer. This data shows that some reviewers are more effective than others, which addresses questions (2) and (3) in Section 2. The histogram also shows that none of the developers found more than five of the seven vulnerabilities and about 20% did not find any vulnerabilities.

4.3 Correlated Vulnerabilities

It is also interesting to note that there were cases where finding a specific vulnerability is strongly correlated to finding another specific vulnerability. Table 2 contains pairs that were significantly correlated, which is shown by their corresponding correlation coefficient (r) and p-value. The first three rows of Table 2 show vulnerabilities that were in a concentrated area in the code; this may be the reason for the correlation. The correlation between the two SQL Injection vulnerabilities may be due to the type of attack; developers may have been specifically looking for SQL Injection vulnerabilities.

Table 2. Pairs of vulnerabilities for which finding one correlates with finding the other

Pair	r	p-value
(XSS 1, XSS 2)	.4588	.0108
(XSS 1, SQLI 1)	.5694	.0010
(XSS 2, SQLI 1)	.4588	.0108
(XSS 4, SQLI 2)	.4001	.0285
(SQLI 1, SQLI 2)	.4842	.0067

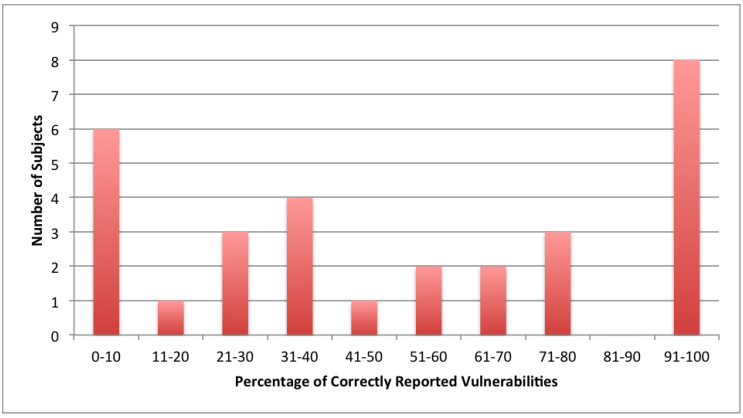

Fig. 2. The percentage of reported vulnerabilities that were correct

4.4 Reviewer Variability

While the average number of correct vulnerabilities found is relatively low, this is not indicative of the total number of vulnerabilities reported by each developer. The average number of reported vulnerabilities is 6.29 with a standard deviation of 5.87. Figure 2 shows a histogram of the fraction of vulnerabilities reported that were correct. There is a bimodal distribution, with one sizable group of reviewers having a very high false positive rate and another group with a very low false positive rate. We find significant variation in reviewer accuracy, which is relevant to questions (2) and (3) in Section 2.

Figure 3 shows the relationship between the number of correct vulnerabilities found and the number of false vulnerabilities reported. This relationship has a correlation coefficient of .3951 with a p-value of .0307. This correlation could be explained by the idea that the more closely a developer examines the code, the more possible vulnerabilities he finds, where this includes both correct vulnerabilities and false vulnerabilities. It may also reflect the level of certainty that a particular reviewer feels he must have before reporting a vulnerability. It is also somewhat similar to the trade-off in static analysis tools: when it detects fewer false alarms (false vulnerabilities), it also detects fewer true vulnerabilities; if calibrated differently, the tool may find more true vulnerabilities, at the cost of more false vulnerabilities.

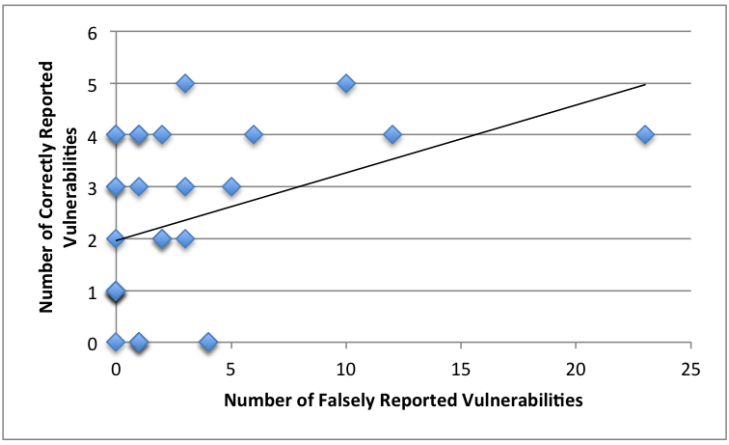

Fig. 3. The relationship between correctly reported vulnerabilities and falsely reported vulnerabilities

4.5 Optimal Number of Reviewers

In order to determine the optimal number of reviewers, we simulated hiring various numbers of reviewers. In each simulation, we randomly chose X reviewers, where $0 \leq X \leq 30$, and combined all reports from these X reviewers. This is representative of hiring X independent reviewers. For a single trial within the simulation, if this combination of reports found all seven vulnerabilities, then the trial was considered a success; if not, it was considered a failure. We conducted 1000 trials for a single simulation and counted the fraction of successes; this estimates the probability of finding all vulnerabilities with X reviewers. Figure 4 shows the probability of finding all vulnerabilities based on the number of developers hired. For example, 10 reviewers have approximately an 80% chance of finding all vulnerabilities, 15 reviewers have approximately a 95% chance of finding all vulnerabilities, and it is probably a waste of money to hire more than 20 reviewers. These results answer questions (4), (5), and (6) in Section 2.

4.6 Demographic Relationships

Our results did not indicate any correlations between self-reported demographic information and reviewer effectiveness. None of the characteristics listed in Appendix A had a statistically significant correlation with the number of correct vulnerabilities reported.

Typical hiring practices include evaluation of a candidate based on his education, experience, and certifications, but according to this data it does not have a significant impact on the effectiveness of the developer's review. We were surprised to find these criteria to be of little use in predicting developer effectiveness in our experimental data. One possible explanation for these results stems from application expectations; knowing how a system works may cause the reviewer to overlook how the system can be

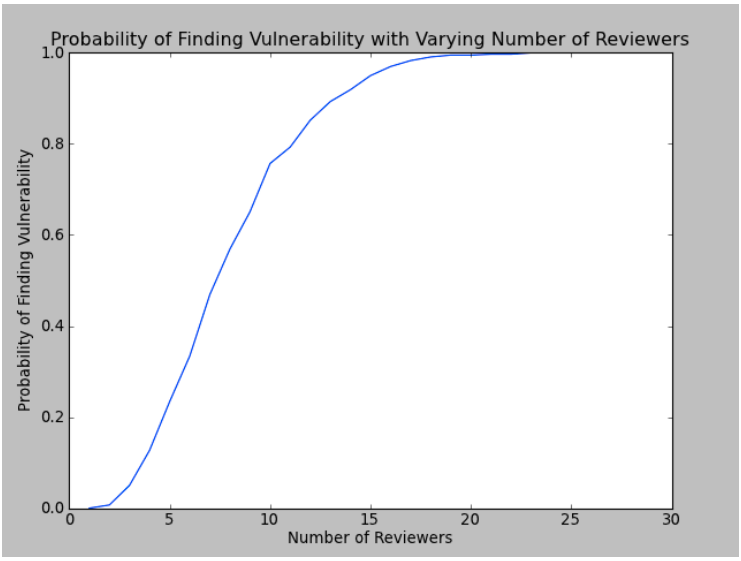

Fig. 4. A graph showing the probability of finding all vulnerabilities depending on the number of reviewers hired

used in a way that diverges from the specification [3]. Reviewers with less experience may not fully understand the system, but might be able to more readily spot deficiencies because they have no preconceived notion of what constitutes correct functionality.

Unfortunately, the design of our study did not allow us to assess the relationship between performance on the screening test and effectiveness at finding vulnerabilities, due to our anonymization procedure. We divorced the identities of participants and all information used in the hiring process from the participants' delivered results in order to eliminate possible reputation risk to subjects. We leave for future work the possibility of developing and evaluating a screening test that can predict reviewer effectiveness.

The only significant correlation found was between the number of years of experience in computer security and the accuracy of the developer's reports. We define accuracy as the fraction of correctly reported vulnerabilities out of the total reported vulnerabilities. Figure 5 shows this relationship with a correlation coefficient of -0.4141 and a p-value of .0229. While this is statistically significant in our dataset, it is not what would be expected because it indicates that the more years of experience a developer has, the lower the developer's accuracy.

We did not find significant correlations between the number of correct vulnerabilities reported and developer experience with software development, web development, or computer security. Table 3 shows the p-values for these tests.

We found a positive correlation between the number of previous web security reviews and the number of correct reports, which may be considered marginally significant. The correlation coefficient is .3117 and $p = .0936$. Figure 6 shows this relationship.

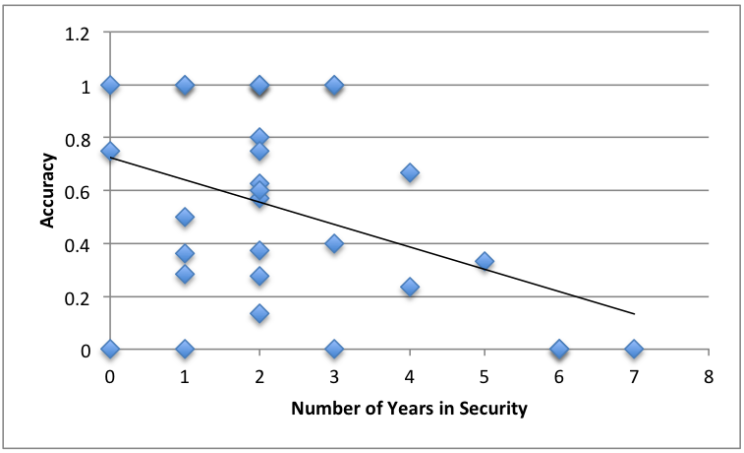

Fig. 5. The relationship between years of experience in computer security and accuracy of reports. Accuracy is defined as the fraction of correctly reported vulnerabilities out of the total number of reported vulnerabilities.

Table 3. The p-values for correlation tests between experience in different areas and the number of correct vulnerabilities reported

Area of Experience	p-value
Software development	0.6346
Web development	0.8839
Computer security	0.3612

4.7 Limitations of Statistical Analysis

A limitation of our data is that the experiment was performed with only 30 subjects. This sample size is not large enough to detect weak relationships.

5 Related Work

To our knowledge, there has been no previous research studying the effectiveness of developers at security code review. However, there have been many studies regarding the evaluation and effectiveness of code inspections. Our discussion of related work falls into three categories: code review effectiveness, methods of detecting web security vulnerabilities, and a comparison of manual code reviews to static analysis tools.

Code Review Effectiveness. One of the largest drawbacks to conducting code inspections is the time-consuming and cumbersome nature of the task. This high cost has motivated a number of studies investigating general code inspection performance and effectiveness [4–6]. Hatton [7] found a relationship between the total number of

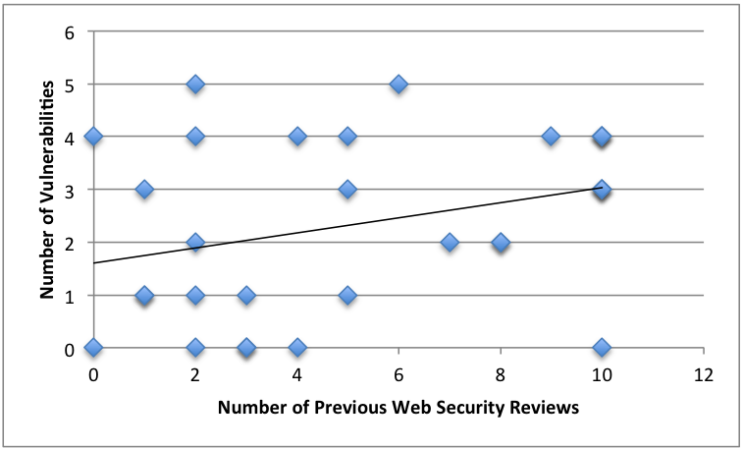

Fig. 6. The correlation between the number of previously conducted security code reviews and the number of correctly reported vulnerabilities

defects in a piece of code and the number of defects found in common by teams of inspectors. The authors gave the inspectors a program written in C with 62 lines of code; the inspectors were told to find any parts of the code that would cause the program to fail. From this research, the authors were able to predict the total number of defects. A subsequent paper by the same author [8] found that checklists had no significant effect on the effectiveness of a code inspection. Other studies have explored whether there are relationships between specific factors, such as review rate or the presence of maintainability defects, and code inspection performance [9, 10]. These experiments were carried out on general-purpose software and were not focused on security vulnerabilities, whereas our work focuses on security vulnerabilities in web applications.

Static Detection of Web Security Vulnerabilities. Most current techniques for detecting web security vulnerabilities are automated tools for static analysis. There has been work that has compared different tools and documented their differences. Bau et al. [11] showed that black-box web application vulnerability scanners do not perform well when detecting advanced and second-order forms of Cross-Site Scripting and SQL Injection. While it is more time-consuming, this may be remedied by manual code inspection. Additionally, there have been many publications evaluating and proposing new automated tools for detecting web security vulnerabilities [12–17]. Our work focuses on detecting web security vulnerabilities in a web application by manually reviewing code as opposed to using automated tools; we do not compare the reviewers' effectiveness to that of static analysis tools, though this would be a good topic for future work.

Comparing Code Review Techniques and Automated Testing. In addition to research into specific techniques for code inspection, testing, and static analysis, a number of studies have compared the effectiveness of different techniques. Basili and Selby [18]

compared the effectiveness of code reading by stepwise abstraction, functional testing, and structural testing. They found that when the experiment was performed with professional programmers, code reading detected more faults than either functional or structural testing. The experiments were performed on software written in procedural languages, but none were network-facing applications. Jones [19] showed that no single method out of formal design inspection, formal code inspection, formal quality assurance, and formal testing was highly efficient in detecting and removing defects; a combination of all four methods yielded the highest efficiency. When only one method was used, the highest efficiency for removing defects was achieved by formal design inspection followed by formal code inspection. When we conducted our experiment, we did not specify how the developers should review the code as long as they did not use any automated tools. Finifter and Wagner [20] compared the effectiveness of black-box testing and manual code review for web applications, also limiting their scope to security vulnerabilities. While manual source code review was found to be more effective than automated black-box testing, black-box testing discovered vulnerabilities not found through the manual source code review.

6 Conclusion and Future Work

We designed an empirical study to ask and answer fundamental questions about the effectiveness of and variation in manual source code review for security. We hired 30 subjects to perform security code review of a web application with known vulnerabilities. The subjects analyzed the code and prepared vulnerability reports following a provided template. A post-completion survey gave us data about their personal experience in web programming and security and their confidence in their vulnerability reports.

Our results revealed that years of experience and education were not useful in predicting how well a subject was able to complete the code review. We also found that the subject's own opinion of how well they performed showed no correlation with how effective their report was. In general, we found the overall effectiveness of our reviewers to be quite low. Twenty percent of the reviewers in our sample found no true vulnerabilities, and no developer found more than five out of the seven known vulnerabilities.

No self-reported metric proved to be useful in predicting reviewer effectiveness, leaving as an open question the best way to select freelance security reviewers that one has no previous experience with. The difficulty in predicting reviewer effectiveness in advance supports anecdotal reports that the most effective way to evaluate a freelancer is by their performance on previously assigned tasks.

It would be interesting to study whether these results apply to other populations and to evaluate whether performance on our screening test correlates with reviewer effectiveness. It would also be interesting to compare the effectiveness of manual security review to that of other techniques, such as automated penetration testing tools. We leave these as open problems for future work.

Acknowledgments. We give special thanks to Aimee Tabor and the TRUST program staff. This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology) through NSF grant CCF-0424422, by the AFOSR under MURI

award FA9550-12-1-0040, and by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the entities that provided funding.

A Demographic and Other Factors

We tested the following demographic and other factors for correlation with number of correctly reported vulnerabilities. None were statistically significant.

- Self-reported level of understanding of the web application
- Percentage of vulnerabilities the developer thought they identified
- Years of experience in software development
- Years of experience in web development
- Years of experience in computer security
- Developer's confidence in the review
- The number of security reviews previously conducted by the developer
- The number of web security reviews previously conducted by the developer
- Self-reported level of expertise on computer security
- Self-reported level of expertise on software development
- Self-reported level of expertise on web development
- Self-reported level of expertise on web security
- Education
- Number of security-related certifications

References

1. TopSite: 10 Best Outsourcing Websites, `http://www.topsite.com/best/outsourcing`
2. OWASP Foundation: Code Review Metrics (2010), `https://www.owasp.org/index.php/Code_Review_Metrics`
3. Baca, D., Petersen, K., Carlsson, B., Lundberg, L.: Static code analysis to detect software security vulnerabilities—does experience matter? In: International Conference on Availability, Reliability and Security, ARES 2009, pp. 804–810. IEEE (2009)
4. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15(3), 182–211 (1976)
5. McCarthy, P., Porter, A., Siy, H., Votta Jr., L.G.: An Experiment to Assess Cost-Benefits of Inspection Meetings and Their Alternatives: A Pilot Study. In: Proceedings of the 3rd International Software Metrics Symposium, pp. 100–111 (March 1996)
6. Biffl, S.: Analysis of the Impact of Reading Technique and Inspector Capability on Individual Inspection Performance. In: Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC), pp. 136–145 (2000)
7. Hatton, L.: Predicting the Total Number of Faults Using Parallel Code Inspections (May 2005), `http://www.leshatton.org/2005/05/total-number-of-faults-using-parallel-code-inspections/`
8. Hatton, L.: Testing the Value of Checklists in Code Inspections. IEEE Software 25(4), 82–88 (2008)

9. Albayrak, O., Davenport, D.: Impact of Maintainability Defects on Code Inspections. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 50:1–50:4 (2010)
10. Ferreira, A., Machado, R., Costa, L., Silva, J., Batista, R., Paulk, M.: An Approach to Improving Software Inspections Performance. In: 2010 IEEE International Conference on Software Maintenance (ICSM), pp. 1–8 (September 2010)
11. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: 2010 IEEE Symposium on Security and Privacy, pp. 332–345 (May 2010)
12. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing Web Application Code by Static Analysis and Runtime Protection. In: Proceedings of the 13th International Conference on the World Wide Web, pp. 40–52 (2004)
13. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: SecuBat: A Web Vulnerability Scanner. In: Proceedings of the 15th International Conference on the World Wide Web, pp. 247–256 (2006)
14. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: IEEE Symposium on Security and Privacy, pp. 263–268 (May 2006)
15. Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 32–41 (June 2007)
16. Lam, M.S., Martin, M., Livshits, B., Whaley, J.: Securing Web Applications With Static and Dynamic Information Flow Tracking. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 3–12 (2008)
17. Kieyzun, A., Guo, P., Jayaraman, K., Ernst, M.: Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In: 31st IEEE International Conference on Software Engineering, pp. 199–209 (May 2009)
18. Basili, V., Selby, R.: Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering SE-13(12), 1278–1296 (1987)
19. Jones, C.: Software Defect-Removal Efficiency. IEEE Computer 29(4), 94–95 (1996)
20. Finifter, M., Wagner, D.: Exploring the Relationship Between Web Application Development Tools and Security. In: Proceedings of the 2nd USENIX Conference on Web Application Development. USENIX (June 2011)