Towards Evidence-Based Assessment of Factors Contributing to the Introduction and Detection of Software Vulnerabilities

Matthew Finifter

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-49
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-49.html

May 8, 2013

Towards Evidence-Based Assessment of Factors Contributing to the Introduction and Detection of Software Vulnerabilities

by

Matthew Smith Finifter

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Chair
Professor Vern Paxson
Professor Brian Carver

Spring 2013

Towards Evidence-Based Assessment of Factors Contributing to the Introduction and Detection of Software Vulnerabilities

Abstract

Towards Evidence-Based Assessment of Factors Contributing to the Introduction and Detection of Software Vulnerabilities

by

Matthew Smith Finifter

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

There is an entire ecosystem of tools, techniques, and processes designed to improve software security by preventing, finding, mitigating, and/or eliminating software vulnerabilities. Software vendors have this entire ecosystem to choose from during each phase of the software development lifecycle, which begins when someone identifies a software need, ends when the software vendor decides to halt support for the software, and includes everything in between.

Unfortunately, guidance regarding *which* of these tools to choose is often non-existent or based solely on anecdotal evidence. In this dissertation, we present three studies to demonstrate that empirical studies can be used to enhance our understanding of the effectiveness of various tools and techniques intended to improve software security.

In our first study, we use a data set of 9 implementations of the same software specification in order to explore the relationship between web application development tools and the security of the applications developed using those tools. We found evidence that framework support for avoiding security vulnerabilities influences application security, that we can expect manual framework support to continue to be problematic, and that manual code review and black-box penetration testing are complementary techniques.

In our second study, we hire 30 code reviewers to perform manual security reviews of a content management system in an effort to better understand the effectiveness of manual security review as a technique for vulnerability discovery. We found that level of experience and education do not correlate with reviewer effectiveness at code review, that overall reviewer effectiveness is low, and that there is significant variation amongst reviewers.

Finally, in our third study, we analyze a data set of rewards paid out over the course of two exemplar vulnerability rewards programs (VRPs), that of Google Chrome and Mozilla Firefox, in an effort to better understand the costs and benefits of such programs. We found that these VRPs appear economically efficient, comparing favorably to the cost of hiring full-time security researchers, that both programs have successfully encouraged broad community participation, and that Chrome's VRP has uncovered more vulnerabilities than that of Firefox despite costing almost exactly the same amount.

Contents

List of Figures

List of Tables

Acknowledgments

My first thanks goes to my advisor, David Wagner. You gave me the freedom to explore whatever I found interesting. Your positive attitude and pleasant demeanor set a great example for me, and I'll be lucky if I can emulate even a few of your personality traits.

Thanks to the National Science Foundation for supporting me through a Graduate Research Fellowship. This award contributed greatly to my academic freedom.

Thanks to all my co-authors. You improved my research and made it more fun.

Thanks to all the friends I made during graduate school. You know who you are. Let's stay in touch.

A huge thanks to my family. You have provided the support I needed to achieve everything I have achieved.

Finally, thanks to my wife, Nancy. Nothing in particular, just everything. I love you.

Chapter 1

Introduction

Software defects are as old as software itself. They happen because software developers are human, and humans make mistakes. When a software defect enables a malicious actor to violate a security goal of the software, the defect is considered a security vulnerability.

By definition, a software vulnerability enables an attacker to circumvent intended security mechanisms, thereby giving the attacker a basis for mounting an attack. We have seen attacks that steal money [90], perform corporate espionage [76], extort companies [77], and even take actions that have been labeled by some as “cyber-warfare” [89, 43].

The exact technical details of software vulnerabilities vary widely, and are often categorized into broad vulnerability classes including memory safety, command injection, incorrect use of cryptography, and cross-site scripting, to name a few. Potential defenses vary according to the vulnerability class, and we have seen enormous amounts of research effort going into improving existing defenses and devising new ones.

The presence of software vulnerabilities has become a fact of life. In response to this reality, the software engineering community has built up a significant amount of infrastructure whose purpose is to (1) reduce the incidence of vulnerabilities introduced during software development, (2) find vulnerabilities, (2) pinpoint their root cause, (3) reproduce them, (4) patch them, and (5) deliver the patched software that fixes them. This infrastructure includes development frameworks (e.g., Django [48]), static and dynamic analysis tools (e.g., Coverity Security Advisor [44] and HP Fortify Program Trace Analyzer [74]), version control software (e.g., git [6]), bug-tracking software (e.g., Bugzilla [19]), vulnerability databases (e.g., the NVD [8]), and auto-update mechanisms (e.g., Chrome’s automatic updater [25]), among many other things, far too numerous to list here.

There is an entire ecosystem of tools, techniques, and processes designed to improve software security by preventing, finding, mitigating, and/or eliminating software vulnerabilities. Software vendors have this entire ecosystem to choose from during each phase of the software development lifecycle, which begins when someone identifies a software need, ends when the software vendor decides to halt support for the software, and includes everything in between.

Unfortunately, guidance regarding *which* of these tools to choose is often non-existent or based solely on anecdotal evidence. In this dissertation, we present three studies to demonstrate that empirical studies can be used to enhance our understanding of the effectiveness of various tools and techniques intended to improve software security.

Toward this end, this dissertation additionally takes steps toward:

- Developing methodologies for testing the security implications of various software development and vulnerability remediation techniques.
- Supplementing opinions, recommendations, anecdotes, and "best practices" with empirical evidence that some technique tends to yield more secure software than some other technique.
- Improving the state of software security by providing actionable recommendations regarding which tools or techniques to choose.

1.1 Web development languages and frameworks

The web has become one of the premier application development platforms, due to its cross-platform nature and its astonishing growth over the last several years. Web application developers have a wide variety of programming languages and development frameworks to choose from. Different programming languages offer web developers different trade-offs regarding available libraries, how quickly a prototype can be developed, and which frameworks are available. Frameworks offer the developer useful tools for common tasks in web development, such as displaying common forms, validating user input, managing a user's session, and deploying an administrative panel.

In principle, web application development frameworks occupy a valuable position for preventing the introduction of security vulnerabilities. Frameworks can, for example, automatically sanitize untrusted user input to prevent cross-site scripting attacks [5]. They can transparently and correctly manage a user's session, thereby disallowing session fixation attacks. In short, there is reason to believe that choosing one framework instead of another may help to avoid common classes of vulnerabilities and may therefore influence application security.

These development tools—programming languages and web application development frameworks—are the focus of the first empirical study we discuss, presented in Chapter 3. Using a data set consisting of 9 implementations of the same web application specification, we search for differences in security that correlate with differences in programming language and available framework security features. Our study reveals that manual security features exposed by frameworks, such as providing sanitizers the developer must manually invoke, do not appear very effective at preventing the types of vulnerabilities they are intended to prevent. On the other hand, the presence of more automatic features, like CSRF protection and secure session management, correlate almost perfectly with security against these classes of attacks. Based on the evidence from this study, we recommend choosing a framework that has fully automatic support for as many vulnerability classes as possible, and we encourage future research into developing more automated support for more vulnerability classes. We believe this study should serve as a valuable reference for software developers who are concerned about security and looking for guidance as to which web development tools to choose.

1.2 Vulnerability-finding techniques

There are a variety of techniques available to software vendors to locate vulnerabilities in their software. These include (1) manual source code review, (2) static analysis tools, (3) dynamic analysis tools, and (4) black-box penetration testing tools. In order to reduce the number of vulnerabilities in their software, software vendors would like to choose the technique or combination of techniques that permits them to locate as many vulnerabilities in their software as possible at the lowest cost possible.

Manual source code review is one such technique. Opinions vary as to how it compares to other techniques, e.g., automated black-box penetration testing. Our study on web application development tools, mentioned above, contributes empirical evidence to discussions of the relative costs and benefits of these two vulnerability-finding techniques. In our study, the two techniques found largely different sets of vulnerabilities, making the two techniques appear to be complementary.

After improving our understanding of manual source code review relative to black-box penetration testing, we devised another study that focused only on manual security review, which we present in Chapter 4. The goal of this study was to better understand how well this particular technique fares in an absolute sense. We hired 30 developers to manually review code that had several known vulnerabilities, and we analyzed the distribution of the number of vulnerabilities found by these developers. In addition, we sought potential predictors of these developers' efficacy at manual security review, including number of years of education, certifications held, and number of security reviews previously performed.

We learned from this study that the overall effectiveness of these outsourced code reviewers was rather low in an absolute sense; a reviewer found on average only 2.33 of the 7 known vulnerabilities. Furthermore, the distribution of the fraction of *valid* vulnerability reports appeared bimodal; several developers reported very few false positives, while another large group reported mostly false positives. None of our survey questions (e.g., asking the reviewer how many years of experience they had working in the field of computer security) proved predictive of the effectiveness of our reviewers, which indicates that hiring outsourced code reviewers is, unfortunately, somewhat of a gamble. We believe this study should serve as a valuable reference for software vendors considering outsourcing their code review in a manner similar to that used in the study.

1.3 Vulnerability rewards programs

A vulnerability rewards program (VRP) is a commitment by a software vendor to pay for the responsible disclosure of software vulnerabilities in one or more of its products. VRPs typically include a set of eligibility criteria, program rules, guidelines for reward amounts, and instructions for responsible vulnerability disclosure.

VRPs represent an emerging strategy for organizations to cost-effectively outsource the search for vulnerabilities to the security community. Because of their relative immaturity as a technique, the costs and benefits of VRPs are not yet clear. The study we present in Chapter 5 adds empirical evidence to the discussion by studying two successful VRPs in depth. Using available data, we uncover information about the cost-effectiveness of these VRPs, the number and types of vulnerabilities they find, and how much money individual participants can expect to earn, among other things. We believe this study should serve as a valuable reference for software vendors considering the development of a new VRP or modifications to an existing VRP.

Chapter 2

Related work

In this chapter, we survey work related to this dissertation. This includes work that aims to answer similar questions to those answered by this dissertation, as well as work that uses similar techniques to those used in this dissertation.

2.1 Efficacy of tools and techniques

This section surveys work that studies how effective various tools are at finding bugs or vulnerabilities. This work is particularly relevant to Chapter 4, which studies the efficacy of manual source code review at finding vulnerabilities, and Chapter 5, which studies how well two vulnerability rewards programs (VRPs) work in practice.

Web application vulnerability detection

Most current techniques for detecting web security vulnerabilities are automated tools for static analysis. Accordingly, there have been many publications proposing and evaluating new automated tools for detecting web application vulnerabilities [69, 73, 72, 114, 78, 75].

Code review

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 [58, 79, 35]. Hatton [66] found a relationship between the total number of 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 with surprising accuracy; the average of the predicted number of defects across all 22 2-person teams was 26.47, and the actual number of defects was 26 ± 2.

A subsequent paper by the same author [65] 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 (i.e., code considered difficult to change as requirements change), and code inspection performance [27, 60]. These experiments were carried out on general-purpose software and were not focused on security vulnerabilities, whereas our study in Chapter 4 focuses on security vulnerabilities in web applications.

Defect prediction

A large body of work addresses defect prediction using empirical techniques; we refer the reader to a survey by Catal et al. [41]. Defect prediction techniques rely on classical metrics such as code complexity and lines of code. Vulnerabilities, as opposed to the more general category of software defects, are harder to predict using empirical techniques. Shin and Williams found only a weak correlation between complexity measures and vulnerabilities in the Mozilla JavaScript engine [106]. Zimmerman et al. identify this "needle in the haystack" nature of vulnerability prediction and suggest the need for measures specific to software security [116].

Neuhaus et al. use a data set of Firefox security advisories in combination with the Firefox code base to map vulnerabilities to software components and predict which components are likely to contain vulnerabilities [92].

Meneely and Williams performed two studies to evaluate how well Linus' Law, "Given enough eyeballs, all bugs are shallow" [101], holds within real software projects. In the first study [82], they evaluate correlations between developer activity metrics and vulnerabilities in the Linux kernel. They identify several metrics that correlate with more vulnerabilities, and they find that files with changes from many developers (9 or more) are much more likely to be vulnerable. In the second study [83], they replicate the first study on two different software projects: PHP and Wireshark. They confirm their findings from the first study and strengthen their models that predict vulnerable components based on the number of developers that have committed to the code.

2.2 Comparisons between different tools

This section surveys work that explicitly compares one tool to another. This work is particularly relevant to Chapter 3, which aims to compare programming languages, web application development frameworks, and vulnerability-finding techniques. Additionally, our study on VRPs in Chapter 5 makes a comparison between the VRP of Google Chrome and that of Mozilla Firefox.

Programming languages

The 9th edition of the WhiteHat Website Security Statistic Report [115] offers insight into the relationship between programming language and application security. Their data set, which includes over 1,500 web applications and over 20,000 vulnerabilities, was gathered from the penetration-testing service WhiteHat performs for its clients. Their report found differences between languages in the prevalence of different vulnerability classes as well as the average number of "serious" vulnerabilities over the lifetime of the applications. For example, in their sample of applications, 57% of the vulnerabilities in JSP (JavaServer Pages) applications were XSS vulnerabilities, while only 52% of the vulnerabilities in Perl applications were XSS vulnerabilities. Another finding was that PHP applications were found to have an average of 26.6 vulnerabilities over their lifetime, while Perl applications had 44.8 and JSP applications had 25.8.

Walden et al. [113] measured the vulnerability density of the source code of 14 PHP and 11 Java applications, using different static analysis tools for each set. They found that the Java applications had lower vulnerability density than the PHP applications, but the result was not statistically significant.

While these analyses sample across distinct applications, our study in Chapter 3 samples across different implementations of the same application. Our data set is smaller, but its collection was more controlled. The first study focused on fixed combinations of programming language and framework (e.g., Java JSP), and the second did not include a framework comparison. Our study focuses separately on language and framework.

Dwarampudi et al. [51] compiled a fairly comprehensive list of pros and cons of the offerings of several different programming languages with respect to many language features, including security. No experiment or data analysis were performed as a part of this effort.

The Plat_Forms [100] study (from which our study in Chapter 3 acquired much of its data) performed a shallow security analysis of their data set. They ran simple black-box tests against the implementations in order to find indications of errors or vulnerabilities, and they found minor differences, which are consistent with our results. We greatly extended their study using both white- and black-box techniques to find vulnerabilities.

Black-box penetration testing tools

We are aware of three separate efforts to compare the effectiveness of different automated black-box web application security scanners. Suto [107] tested each scanner against the demonstration site of each other scanner and found differences in the effectiveness of the different tools. His report lists detailed pros and cons of using each tool based on his experience.

Bau et al. [33] tested 8 different scanners in an effort to identify ways in which the state of the art of black box scanning could be improved. They found that the scanners tended to perform well on reflected XSS and first-order SQL injection vulnerabilities (i.e., one in which malicious input flows directly into a SQL query), but poorly on second-order vulnerabilities (e.g., stored XSS and SQL injection vulnerabilities in which stored malicious content flows into a SQL query). We augment this finding with the result that manual analysis performs better for stored XSS, authentication and authorization bypass, CSRF, insecure session management, and insecure password storage, and black-box testing performs better for reflected XSS and SQL injection (Chapter 3).

Doupé et al. [50] evaluated 11 scanners against a web application custom-designed to have many different crawling challenges and types of vulnerabilities. They found that the scanners were generally poor at crawling the site, that they performed poorly against "logic" vulnerabilities (e.g., application-specific vulnerabilities, which often include authorization bypass vulnerabilities), and that they required their operators to have a lot of knowledge and training to be able to use them effectively.

While these studies compare several black-box tools to one another, we compare the effectiveness of a single black-box tool to that of manual source code analysis (Chapter 3) and we subsequently analyze manual source code analysis in detail (Chapter 4). Our choice regarding which black-box scanner to use when designing our study on web application development tools was based in part on these studies of black-box scanners.

Bug-finding techniques

Wagner et al. [112] performed a case study against 5 applications in which they analyzed the true- and false-positive rates of three static bug-finding tools and compared manual source code review to static analysis for one of the 5 applications. This study focused on defects of any type, making no specific mention of security vulnerabilities. They found that all defects the static analysis tools discovered were also found by the manual review. Our study in Chapter 3 focuses specifically on security vulnerabilities in web applications, and we use a different type of tool in our study than they use in their study.

Two short articles [45, 80] discuss differences between various tools one might consider using to find vulnerabilities in an application. The first lists constraints, pros, and cons of several tools, including source code analysis, dynamic analysis, and black-box scanners. The second article discusses differences between white- and black-box approaches to finding vulnerabilities. Overall, these articles provide a gentle introduction to the space of options available for finding vulnerabilities. Our studies, on the other hand, gather evidence to suggest how well various tools work in practice.

Basili and Selby [32] 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 [71] 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 study in Chapter 4, we did not specify how the reviewers should review the code as long as they did not use any automated tools.

Austin and Williams evaluated four different techniques for vulnerability discovery on two health record systems: "systematic and exploratory manual penetration testing, static analysis, and automated penetration testing" [28], finding that very few vulnerabilities are in fact found by multiple techniques and that automatic automated penetration testing is the most effective in terms of vulnerabilities found per hour.

2.3 Vulnerability ecosystem

This section surveys work that uses vulnerability data sets to discover how the nature of vulnerabilities (e.g., the number of vulnerabilities present, the severity of vulnerabilities, or how long vulnerabilities remain latent) has changed over time within a single application or across multiple applications. This work is particularly relevant to Chapter 5, which uses a data set of vulnerabilities in two web browsers to understand how the security of these browsers has changed over time.

Vulnerability rates within software

Rescorla gathered data from NIST's ICAT database (which has since been updated and renamed to NVD [8]) to analyze whether vulnerability rates tend to decrease over time [102]. He found no evidence that it is in fact worthwhile for software vendors to attempt to find vulnerabilities in their own software because there is no evidence that such efforts are reducing vulnerability rates.

Ozment and Schechter used the OpenBSD CVS repository to ask and answer similar questions as Rescorla [95]. They find that the rate of discovery of what they call *foundational* vulnerabilities—those present since the beginning of the study period—had decreased over the study period.

Neuhaus and Plattner use vulnerability reports for Mozilla, Apache httpd, and Apache Tomcat to evaluate whether vulnerability fix rates have changed over time [91]. They conclude that the supply of vulnerabilities is not declining, and therefore that attackers and/or vulnerability researchers have not hit diminishing returns in looking for vulnerabilities.

Evolution of vulnerabilities over time

Scholte et al. use the NVD to evaluate how cross-site scripting and SQL injection vulnerabilities have evolved over time [104]. They find that the complexity of such vulnerabilities does not appear to have changed over time and that many foundational cross-site scripting vulnerabilities are still being discovered.

Chapter 3

Web application development tools

3.1 Introduction

The web has become the dominant platform for new software applications. As a result, new web applications are being developed all the time, causing the security of such applications to become increasingly important. Web applications manage users' personal, confidential, and financial data. Vulnerabilities in web applications can prove costly for organizations; costs may include direct financial losses, increases in required technical support, and tarnished image and brand.

Security strategies of an organization often include developing processes and choosing tools that reduce the number of vulnerabilities present in live web applications. These software security measures are generally focused on some combination of (1) building secure software, and (2) finding and fixing security vulnerabilities in software after it has been built.

How should managers and developers in charge of these tasks decide which tools – languages, frameworks, debuggers, etc. – to use to accomplish these goals? What basis of comparison do they have for choosing one tool over another? Common considerations for choosing (e.g.,) one programming language over another include:

- How familiar staff developers are with the language.
- If new developers are going to be hired, the current state of the market for developers with knowledge of the language.
- Interoperability with and re-usability of existing in-house and externally-developed components.

This chapter was previously published at the 2nd USENIX Conference on Web Application Development [62].

- Perceptions of security, scalability, reliability, and maintainability of applications developed using that language.

Similar considerations exist for deciding which web application development framework to use and which process to use for finding vulnerabilities.

This work begins an inquiry into how to improve one part of the last of these criteria: the basis for evaluating a tool's inclination (or disinclination) to contribute to application security.

Past research and experience reveal that different tools *can* have different effects on application security. The software engineering and software development communities have seen that an effective way to preclude buffer overflow vulnerabilities when developing a new application is to simply use a language that offers automatic memory management. We have seen also that even if other requirements dictate that the C language must be used for development, using the safer `strlcpy` instead of `strcpy` can preclude the introduction of many buffer overflow vulnerabilities [86], and this is merely one of many such functions to consider.

This research is an exploratory study into the security properties of some of the tools and processes that organizations might choose to use during and after they build their web applications. We seek to understand whether the choice of language, web application development framework, or vulnerability-finding process affects the security of the applications built using these tools.

We study the questions by analyzing 9 independent implementations of the same web application. We collect data on (1) the number of vulnerabilities found in these implementations using both a manual security review and an automatic black-box penetration testing tool, and (2) the level of security support offered by the frameworks. We look in these data sets for patterns that might indicate differences in application security between programming languages, frameworks, or processes for finding vulnerabilities. These patterns allow us to generate and test hypotheses regarding the security implications of the various tools we consider.

This chapter's main contributions are as follows:

- We develop a methodology for studying differences in the effect on application security that different web application development tools may have. The tools we consider are programming languages, web application development frameworks, and processes for finding vulnerabilities.

- We generate and test hypotheses regarding the differences in security implications of these tools.

- We develop a taxonomy for framework-level defenses that ranges from *always on* framework support to no framework support.

- We find evidence that automatic framework-level defenses work well to protect web applications, but that even the best manual defenses will likely continue to fall short of their goals.
- We find evidence that manual source code analysis and automated black-box penetration testing are complementary.

3.2 Goals

Programming language

We want to measure the influence that programming language choice has on the security of the software developed using that language. If such an influence exists, software engineers (or their managers) could take it into account when planning which language to use for a given job. This information could help reduce risk and allocate resources more appropriately.

We have many reasons to believe that the features of a programming language could cause differences in the security of applications developed using that language. For example, research has shown that type systems can statically find (and therefore preclude, by halting compilation) certain types of vulnerabilities [105, 103]. In general, static typing can find bugs (any of which could be a vulnerability) that may not have been found until the time of exploitation in a dynamically-typed language.

Also, one language's standard libraries might be more usable, and therefore less prone to error, than another's. A modern exception handling mechanism might help developers identify and recover from dangerous scenarios.

But programming languages differ in many ways beyond the languages themselves. Each language has its own community, and these often differ in their philosophies and values. For example, the Perl community values TMTOWTDI ("There's more than one way to do it") [9], but the Zen of Python [97] states, "[t]here should be one – and preferably, only one – obvious way to do it." Clear documentation could play a role as well.

Therefore, we want to test whether the choice of language measurably influences overall application security. If so, it would be useful to know whether one language fares better than another for any specific class of vulnerability. If this is the case, developers could focus their efforts on classes for which their language is lacking good support, and not worry so much about those classes in which data show their language is strong.

Web application development framework

Web application development frameworks provide a set of libraries and tools for performing tasks common in web application development. We want to evaluate the role that they play in the development of secure software. This can help developers make more informed decisions when choosing which technologies to use.

Recently, we have seen a trend of frameworks adding security features over time. Many modern frameworks take care of creating secure session identifiers (e.g., Zend, Ruby on Rails), and some have added support for automatically avoiding cross-site scripting (XSS) or cross-site request forgery (CSRF) vulnerabilities (e.g., Django, CodeIgniter). It is natural to wonder whether frameworks that are proactive in developing security features yield software with measurably better security, but up to this point we have no data showing whether this is so.

Vulnerability-finding tool

Many organizations manage security risk by assessing the security of software before they deploy or ship it. For web applications, two prominent ways to do so are: (1) black-box penetration testing, using automated tools designed for this purpose, and (2) manual source code analysis by an analyst knowledgeable about security risks and common vulnerabilities. The former has the advantage of being mostly automated and being cheaper; the latter has a reputation as more comprehensive but more expensive. However, we are not aware of quantitative data to measure their relative efficacy. We work toward addressing this problem by comparing the effectiveness of manual review to that of automated black-box penetration testing. Solid data on this question may help organizations make an informed choice between these assessment methods.

3.3 Methodology

In order to address these questions, we analyze several independent implementations of the same web application specification, written using different programming languages and different frameworks. We find vulnerabilities in these applications using both manual source code review and automated black-box penetration testing, and we determine the level of framework support each implementation has at its disposal to help it contend with various classes of vulnerabilities. We look for associations between: (1) programming language and number of vulnerabilities, (2) framework support and number of vulnerabilities, and (3) number of vulnerabilities found by manual source code analysis and by automated black-box penetration testing.

We analyze data collected in a previous study called Plat_Forms [100]. In that work, the researchers devised and executed a controlled experiment that gave 9 professional programming teams the same programming task for a programming contest. Three of the teams used Perl, three used PHP, and the remaining three used Java.

The contest rules stated that each team had 30 hours to implement the specification of a web application called People By Temperament [99]. Each team chose which frameworks they were going to use. There was little overlap in the set of frameworks used by teams using the same programming language. Table 3.1 lists the set of frameworks used by each team.

Team Number	Language	Frameworks used
1	Perl	DBIx::DataModel, Catalyst, Template Toolkit
2	Perl	Mason, DBI
5	Perl	Gantry, Bigtop, DBIx::Class
3	Java	abaXX, JBoss, Hibernate
4	Java	Spring, Spring Web Flow, Hibernate, Acegi Security
9	Java	Equinox, Jetty, RAP
6	PHP	Zend Framework, OXID framework
7	PHP	proprietary framework
8	PHP	Zend Framework

Table 3.1: Set of frameworks used by each team.

The researchers collected the 9 programs and analyzed their properties. While they were primarily concerned with metrics like performance, completeness, size, and usability, we re-analyze their data to evaluate the security properties of these 9 programs.

Each team submitted a complete source code package and a virtual machine image. The VM image runs a web server, which hosts their implementation of People by Temperament. The source code packages were trimmed to remove any code that was not developed specifically for the contest, and these trimmed source code packages were released under open source licenses.[1]

For our study, we used the set of virtual machine images and the trimmed source code packages. The Plat_Forms study gathered other data (e.g., samples at regular intervals of the current action of each developer) that we did not need for the present study. The data from our study are publicly available online.[2]

People by Temperament

We familiarized ourselves with the People by Temperament application before beginning our security analysis. The application is described as follows:

> PbT (People by Temperament) is a simple community portal where members can find others with whom they might like to get in contact: people register to become members, take a personality test, and then search for others based on criteria such as personality types, likes/dislikes, etc. Members can then get in contact with one another if both choose to do so. [99]

[1] `http://www.plat-forms.org/sites/default/files/platforms2007solutions.zip`

[2] `http://www.cs.berkeley.edu/~finifter/datasets/`

Integer-valued	Stored XSS Reflected XSS SQL injection Authentication or authorization bypass
Binary	CSRF Broken session management Insecure password storage

Table 3.2: The types of vulnerabilities we looked for. We distinguish binary and integer-valued vulnerability classes. Integer-valued classes may occur more than once in an implementation. For example, an application may have several reflected XSS vulnerabilities. The binary classes represent presence or absence of an application-wide vulnerability. For example, in all implementations in this study, CSRF protection was either present throughout the application or not present at all (though this could have turned out differently).

People by Temperament is a small but realistic web application with a non-trivial attack surface. It has security goals that are common amongst many web applications. We list them here:

- **Isolation between users.** No user should be able to gain access to another user's account; that is, all information input by a user should be integrity-protected with respect to other users. No user should be able to view another user's confidential information without approval. Confidential information includes a user's password, full name, email address, answers to personality test questions, and list of contacts. Two users are allowed to view each other's full name and email address once they have agreed to be in contact with one another.

- **Database confidentiality and integrity.** No user should be able to directly access the database, since it contains other users' information and it may contain confidential web site usage information.

- **Web site integrity.** No user should be able to vandalize or otherwise modify the web site contents.

- **System confidentiality and integrity.** No user should be able to gain access to anything on the web application server outside of the scope of the web application. No user should be able to execute additional code on the server.

The classes of vulnerabilities that we consider are presented in Table 3.2. A vulnerability in any of these classes violates at least one of the application's security goals.

Review Number	**Dev. Team Number**	**Language**	**SLOC**	**Review Time (min.)**	**Review Rate (SLOC/hr)**
1	6	PHP	2,764	256	648
2	3	Java	3,686	229	966
3	1	Perl	1,057	210	302
4	4	Java	2,021	154	787
5	2	Perl	1,259	180	420
6	8	PHP	2,029	174	700
7	9	Java	2,301	100	1,381
8	7	PHP	2,649	99	1,605
9	5	Perl	3,403	161	1,268

Table 3.3: Time taken for manual source code reviews, and number of source lines of code for each implementation.

Vulnerability data

We gathered vulnerability data for each implementation in two distinct phases. In the first phase, a reviewer performed a manual source code review of the implementation. In the second phase, we subjected the implementation to the attacks from an automated black-box web penetration testing tool called Burp Suite Pro [98].

We used both methods because we want to find as many vulnerabilities as we could. We hope that any failings of one method will be at least partially compensated by the other. Although we have many static analysis tools at our disposal, we chose not to include them in this study because we are not aware of any that work equally well for all language platforms. Using a static analysis tool that performs better for one language than another would have introduced systematic bias into our experiment.

Manual source code review

We manually reviewed all implementations. We believe we were adequately qualified to do so given our knowledge about security issues and previous involvement in security reviews.

Using one reviewer for all implementations avoids the problem of subjectivity between different reviewers that would arise if the reviewers were to examine disjoint sets of implementations. We note that having multiple reviewers would be beneficial if each reviewer was able to review all implementations independently of all other reviewers.

We followed the Flaw Hypothesis Methodology [36] for conducting the source code reviews, which involves devising a prioritized list of hypothesized flaws based on an understanding of the system in question. We used the People by Temperament specification and knowledge of flaws common to web applications to develop a list of types of vulnerabilities to look for. We performed two phases of review, first looking for specific types of flaws from the list, then comprehensively reviewing the implementation. We confirmed each suspected vulnerability by developing an exploit.

We randomly generated the order in which to perform the manual source code reviews in order to mitigate any biases that may have resulted from choosing any particular review order. Table 3.3 presents the order in which the reviewer reviewed the implementations as well as the amount of time spent on each review.

We spent as much time as we felt necessary to perform a complete review. As shown in Table 3.3, the number of source lines of code reviewed per hour varies widely across implementations; the minimum is 302 and the maximum is 1,605. Cohen [42] states that "[a]n expected value for a meticulous inspection would be 100-200 LOC/hour; a normal inspection might be 200-500." It is unclear upon what data or experience these numbers are based, but we expect the notion of "normal" to vary across different types of software. For example, we expect a review of a kernel module to proceed much more slowly than that of a web application. Additionally, we note that the number of source lines of code includes both static HTML content and auto-generated code, neither of which tends to require rigorous security review.

To help gauge the validity of our data for manual source code review, we test the following hypotheses:

- Later reviews take less time than earlier reviews.
- More vulnerabilities were found in later reviews.
- Slower reviews find more vulnerabilities.

If we find evidence in support of either of the first two hypotheses, this may indicate that the reviewer gained experience over the course of the reviews, which may have biased the manual review data. A more experienced reviewer can be expected to find a larger fraction of the vulnerabilities in the code, and if this fraction increases with each review, we expect our data to be biased in showing those implementations reviewed earlier to be more secure. Spearman's rho (a statistical test of correlation) for these two hypotheses is $\rho = 0.633$ ($p = 0.0671$) and $\rho = -0.0502$ ($p = 0.8979$), respectively, which means that we do not find evidence in support of either of these hypotheses.

If we find evidence in support of the third of these hypotheses, this may indicate that the reviewer did not allow adequate time to complete one or more of the reviews. This would bias the data to make it appear that those implementations reviewed more quickly are more secure than those for which the review proceeded more slowly. The correlation coefficient between review rate and number of vulnerabilities found using manual analysis is $r = 0.0676$ ($p = 0.8627$), which means we do not find evidence in support of this hypothesis. The lack of support for these hypotheses modestly increases our confidence in the validity of our manual analysis data.

Black-box testing

We used Portswigger's Burp Suite Professional version 1.3.08 [98] for black box testing of the implementations. We chose this tool because a previous study has shown it to be among the best of the black box testing tools [50] and because it has a relatively low cost.

We manually spidered each implementation before running Burp Suite's automated attack mode (called "scanner"). All vulnerabilities found by Burp Suite were manually verified and de-duplicated (when necessary). Manual verification was necessary because Burp Suite's results often included findings that were not true vulnerabilities. We counted separate findings as duplicates when they were the result of an error in the same line of code.

Framework support data

We devised a taxonomy to categorize the level of support a framework provides for protecting against various vulnerability classes. We distinguish levels of framework support as follows.

The strongest level of framework support is *always on*. Once a developer decides to use a framework that offers always-on protection for some vulnerability class, a vulnerability in that class cannot be introduced unless the developer stops using the framework. An example of this is the CSRF protection provided automatically by Spring Web Flow [49], which Team 4 used in its implementation. Spring Web Flow introduces the notion of tokens, which define flows through the UI of an application, and these tokens double as CSRF tokens, a well-known protection mechanism for defending against CSRF vulnerabilities. Since they are integral to the functionality the framework provides, they cannot be removed or disabled without ceasing to use the framework entirely.

The next strongest level of framework support is *opt-out* support. This level of support provides protection against a vulnerability class by default, but it can be disabled by the developer if he so desires. Team 2's custom ORM framework provides opt-out support for SQL injection. If the framework is used, SQL injection cannot occur, but a developer can opt out by going around the framework to directly issue SQL queries.

Opt-in support refers to a defense that is disabled by default, but can be enabled by the developer to provide protection throughout the application. Enabling the protection may involve changing a configuration variable or calling into the framework code at initialization time. Once enabled, opt-in support defends against all subsequent instances of that vulnerability class. Acegi Security, used by Team 4, provides a `PasswordEncoder` interface with several different implementations. We consider this opt-in support because a developer can select an implementation that provides secure password storage for his application.

Manual support is the weakest level of framework support. This term applies if the framework provides a useful routine to help protect against a vulnerability class, but that routine must be utilized by the developer *each time protection is desired.* For example, many frameworks provide XSS filters that can be applied to untrusted data before it is included in the HTML page. These filters spare a developer the burden of writing a correct filter, but the developer must still remember to invoke the filter every time untrusted data is output to a user. Manual support is weak because a developer has many opportunities to make an error of omission. Forgetting to call a routine (such as an XSS filter) even once is enough to introduce a vulnerability. We use the term *automatic* support to contrast with manual support; it refers to any level of support stronger than manual support.

For each implementation, we looked at the source code to discover which frameworks were used. We read through the documentation for each of the frameworks to find out which protection mechanisms were offered for each vulnerability class we consider. We defined the implementation's level of support for a particular vulnerability class to be the highest level of support offered by any framework used by the implementation.

Individual vulnerability data

We gather data about each individual vulnerability to deepen our understanding of the current framework ecosystem, the reasons that developers introduce vulnerabilities, and the limitations of manual review. For each vulnerability, we determine how far the developers would have had to stray from their chosen frameworks in order to find manual framework support that could have prevented the vulnerability. Specifically, we label each vulnerability with one of the following classifications:

1. **Framework used.** Framework support that could have prevented this vulnerability exists in at least one of the frameworks used by the implementation.

2. **Newer version of framework used.** Framework support exists in a newer version of one of the frameworks used by the implementation.

3. **Another framework for language used.** Framework support exists in a different framework for the same language used by the implementation.

4. **Some framework for some language.** Framework support exists in some framework for some language other than the one used by the implementation.

5. **No known support.** We cannot find framework support in any framework for any language that would have stopped the vulnerability.

We label each vulnerability with the *lowest* level at which we are able to find framework support that could have prevented the vulnerability. We do so using our awareness and knowledge of state-of-the-art frameworks as well as the documentation frameworks provide.

Similarly, for each vulnerability, we determine the level at which the developers could have found automatic (i.e., opt-in or better) framework support. We evaluate this in the same manner as we did for manual support, but with a focus only on automatic protection mechanisms.

	Java	**Perl**	**PHP**
Number of programmers	9	9	9
Mean age (years)	32	32	32
Mean experience (years)	7.1	8.7	9.8

Table 3.4: Statistics of the programmers.

Threats to validity

Experimental design. The Plat_Forms data were gathered in a non-randomized experiment. This means that the programmers chose which language to use; the language was not randomly assigned to them by the researchers. This leaves the experiment open to selection bias; it could be the case that more skilled programmers tend to choose one language instead of another. As a result, any results we find represent what one might expect when hiring new programmers who choose which language to use, rather than having developers on staff and telling them which language to use.

Programmer skill level. If the skill level of the programmers varies from team to team, then the results represent the skills of the programmers, not inherent properties of the technologies they use for development. Fortunately, the teams had similar skills, as shown in Table 3.4.

Security awareness. Security was not explicitly mentioned to the developers, but all were familiar with security practices because their jobs required them to be [109]. It may be that explicitly mentioning security or specifying security requirements would have changed the developers' focus and therefore the security of the implementations, but we believe that the lack of special mention is realistic and representative of many programming projects. In the worst case, this limits the external validity of our results to software projects in which security is not explicitly highlighted as a requirement.

Small sample size. Due to the cost of gathering data of this nature, the sample size is necessarily small. This is a threat to external validity because it makes it difficult to find statistically significant results. In the worst case, we can consider this a case study that lets us generate hypotheses to test in future research.

Generalization to other applications. People by Temperament is one web application, and any findings with respect to it may not hold true with respect to other web applications, especially those with vastly different requirements or of much larger scale. The teams had only 30 hours to complete their implementation, which is not representative of most real software development projects. Despite these facts, the application does have a significant amount of functionality and a large enough attack surface to be worth examining.

Number of vulnerabilities. We would like to find the total number of vulnerabilities present in each implementation, but each analysis (manual and black-box) finds only some fraction of them. If the detection rate of our manual analysis is better for one language or one implementation than it is for another, this is a possible threat to validity. However, we have no reason to expect a systematic bias of this nature, as the reviewer's level of experience in manual source code review is approximately equivalent for all three languages. At no time did the reviewer feel that any one review was easier or more difficult than any other.

Similarly, if the detection rate of our black-box tool is better for one language or implementation than it is for another, this could pose a threat to validity. We have no reason to believe this is the case. Because black-box testing examines only the input-output behavior of a web application and not its implementation, it is inherently language- and implementation-agnostic, which leads us to expect that it has no bias for any one implementation over any other.

Vulnerability severity. Our analysis does not take into account any differences in vulnerability severity. Using our analysis, an implementation with many low-severity vulnerabilities would appear less secure than an implementation with only a few very high-severity vulnerabilities, though in fact the latter system may be less secure overall (e.g., expose more confidential customer data). We have no reason to believe that average vulnerability severity varies widely between implementations, but we did not study this in detail.

Vulnerabilities introduced later in the product cycle. Our study considers only those vulnerabilities introduced during initial product development. Continued development brings new challenges for developers that simply were not present in this experiment. Our results do not answer any questions about vulnerabilities introduced during code maintenance or when features are added after initial product development.

Framework documentation. If a framework's documentation is incomplete or incorrect, or if we misread or misunderstood the documentation, we may have mislabeled the level of support offered by the framework. However, the documentation represents the level of support a developer could reasonably be expected to know about. If we were unable to find documentation for protection against a class of vulnerabilities, we expect that developers would struggle as well. On the other hand, it is possible that a framework could claim it has support, but that this support is buggy or not actually present. We have not seen this occur in any framework, so we do not expect it to be widespread enough to affect our conclusions.

Awareness of frameworks and levels of support. There may exist frameworks that we are not aware of that provide strong framework support for a vulnerability class. If this is the case, our labeling of vulnerabilities with the nearest level at which framework support exists (Section 3.3) may be incorrect. We have made every effort to consider all frameworks with a significant user base in order to mitigate this problem, and we have consulted several lists of frameworks (e.g., [70]) in order to make our search as thorough as reasonably possible.

3.4 Results

We look for patterns in the data and analyze it using statistical techniques. We note that we do not find many statistically significant results due to the limited size of our data set.

Total number of vulnerabilities

Figure 3.1 displays the total number of vulnerabilities found in each implementation, including both integer-valued and binary vulnerability classes (we count a binary vulnerability as one vulnerability in these aggregate counts).

Every implementation had at least one vulnerability. This suggests that building secure web applications is difficult, even with a well-defined specification, and even for a relatively small application.

One of the Perl implementations has by far the most vulnerabilities, primarily due to its complete lack of XSS protection.[3] This does not seem to be related to the fact that Perl is the language used, however, since the other two Perl implementations have only a handful of vulnerabilities, and few XSS vulnerabilities.

The Java implementations have fewer vulnerabilities than the PHP implementations. In fact, every Java implementation contains fewer vulnerabilities than each PHP implementation.

A one-way ANOVA test reveals that the overall relationship in our data set between language and total number of vulnerabilities is not statistically significant ($F = 0.57$, $p = 0.592$). We also perform a Student's t-test for each pair of languages, using the Bonferroni correction to account for the fact that we test multiple (3) hypotheses. As expected, we do not find a significant difference between PHP and Perl or between Perl and Java. We find a statistically significant difference between PHP and Java ($p = 0.033$).

Vulnerability classes

Figure 3.2 breaks down the total number of vulnerabilities into the separate integer-valued vulnerability classes, and the shaded rows in Table 3.5 present the data for the binary vulnerability classes.

XSS. A one-way ANOVA test reveals that the relationship between language and number of stored XSS vulnerabilities is not statistically significant ($F = 0.92$, $p = 0.4492$). The same is true for reflected XSS ($F = 0.43$, $p = 0.6689$).

[3]None of our conclusions would differ if we were to exclude this apparent outlier.

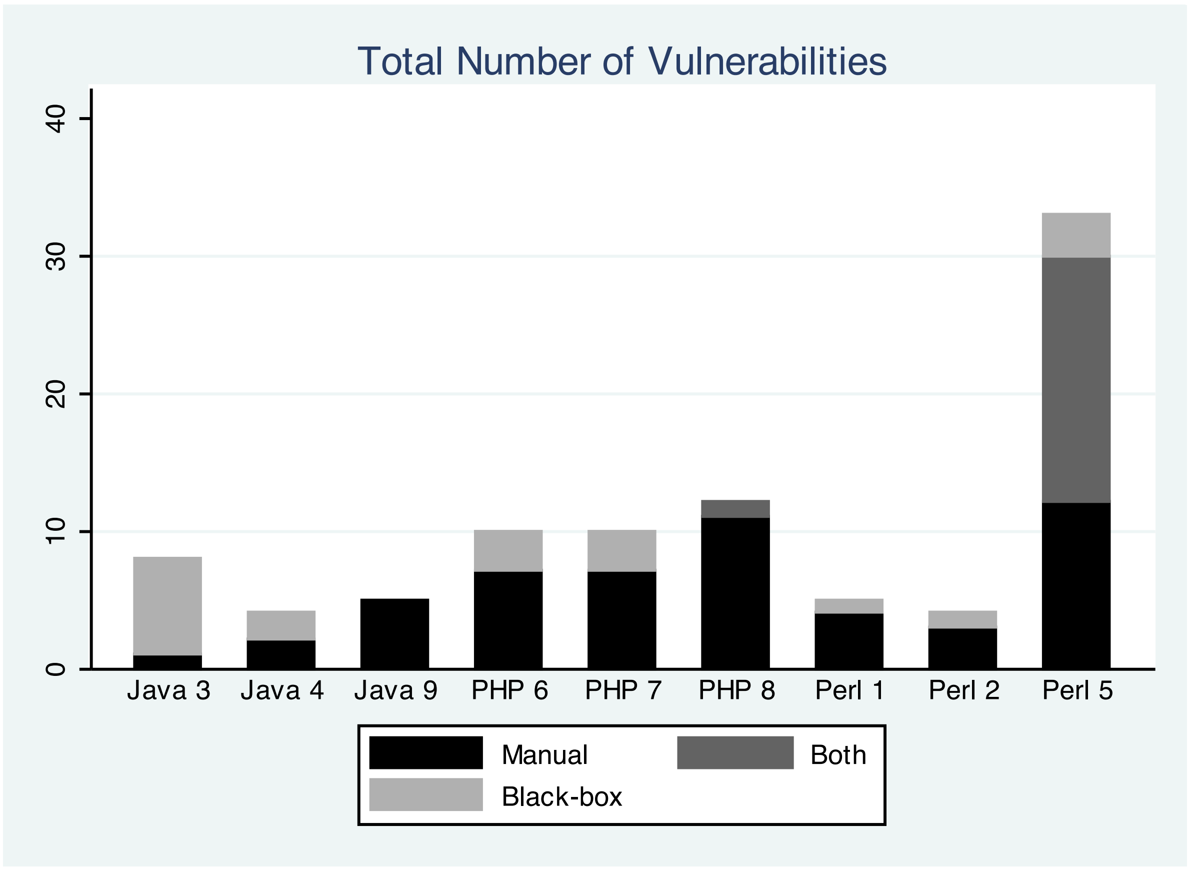


Figure 3.1: The total number of vulnerabilities found in the 9 implementations of People by Temperament. The x-axis is labeled with the language and team number.

		CSRF		Session Management		Password Storage	
Team Number	**Language**	Vulnerable?	Framework Support	Vulnerable?	Framework Support	Vulnerable?	Framework Support
1	Perl	•	none		opt-in	•	opt-in
2	Perl	•	none	•	none	•	none
5	Perl	•	none	•	none		opt-out
3	Java		manual		opt-out	•	none
4	Java		always on		opt-in	•	opt-in
9	Java	•	none		opt-in		none
6	PHP	•	none		opt-out	•	opt-in
7	PHP	•	none		opt-out	•	none
8	PHP	•	none		opt-out	•	opt-in

Table 3.5: Presence or absence of binary vulnerability classes, and framework support for preventing them.

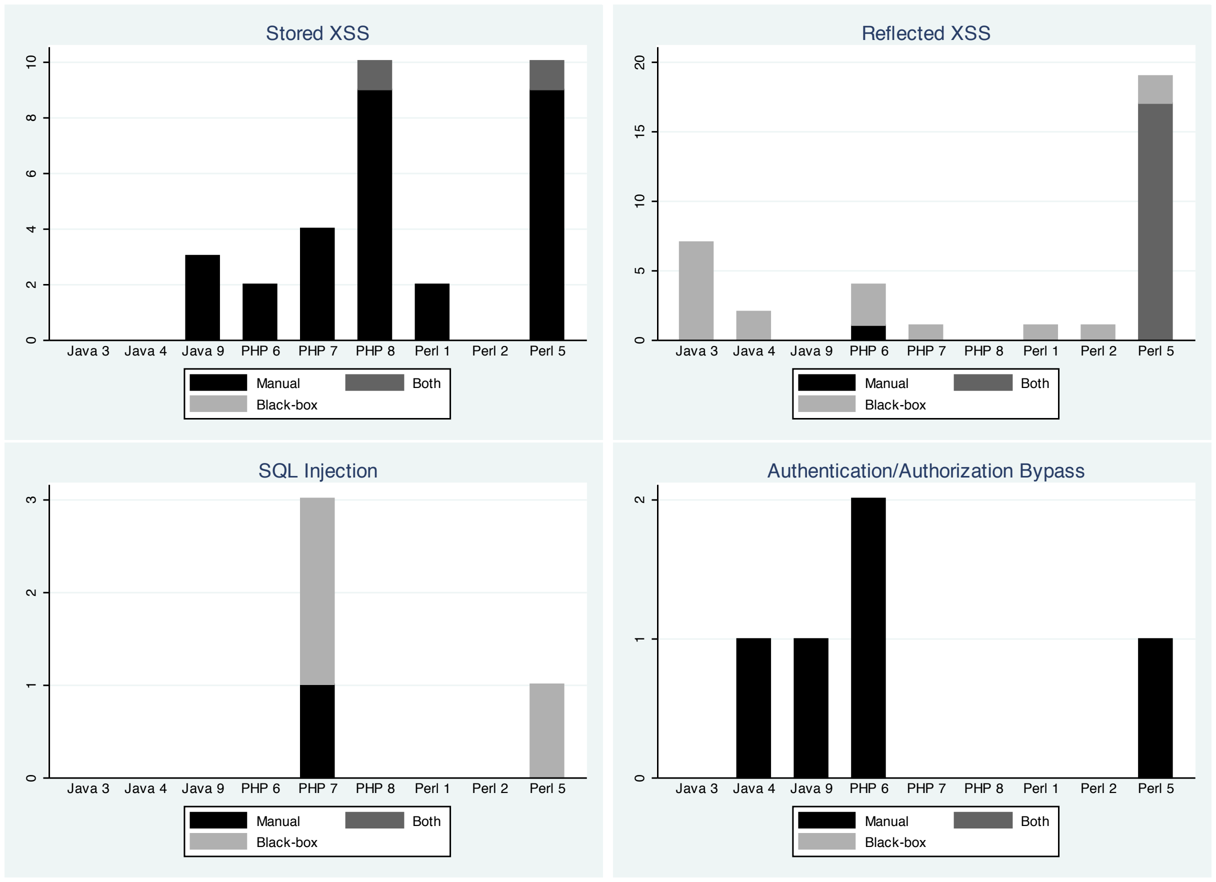


Figure 3.2: Vulnerabilities by vulnerability class.

SQL injection. Very few SQL injection vulnerabilities were found. Only two implementations had any such vulnerabilities, and only 4 were found in total. The difference between languages is not statistically significant ($F = 0.70$, $p = 0.5330$).

Authentication and authorization bypass. No such vulnerabilities were found in 5 of the 9 implementations. Each of the other 4 had only 1 or 2 such vulnerabilities. The difference between languages is not statistically significant ($F = 0.17$, $p = 0.8503$).

CSRF. As seen in Table 3.5, all of the PHP and Perl implementations, and 1 of 3 Java implementations were vulnerable to CSRF attacks. Fisher's exact test reveals that the difference between languages is not statistically significant ($p = 0.25$).

Session management. All implementations other than 2 of the 3 Perl implementations were found to implement secure session management. That is, the Perl implementations

		Vulnerabilities found by			
Team Number	**Language**	**Manual only**	**Black-box only**	**Both**	**Total**
1	Perl	4	1	0	5
2	Perl	3	1	0	4
5	Perl	12	3	18	33
3	Java	1	7	0	8
4	Java	2	2	0	4
9	Java	5	0	0	5
6	PHP	7	3	0	10
7	PHP	7	3	0	10
8	PHP	11	0	1	12

Table 3.6: Number of vulnerabilities found in the implementations of People by Temperament. The "Vulnerabilities found by" columns display the number of vulnerabilities found only by manual analysis, only by black-box testing, and by both techniques, respectively. The final column displays the total number of vulnerabilities found in each implementation.

were the only ones with vulnerable session management. Fisher's exact test reveals that the difference is not statistically significant ($p = 0.25$).

Insecure password storage. Most of the implementations used some form of insecure password storage, ranging from storing passwords in plaintext to not using a salt before hashing the passwords. One Perl and one Java implementation did not violate current best practices for password storage. There does not, however, appear to be any association between programming language and insecure password storage. Fisher's exact test does not find a statistically significant difference ($p = 0.999$).

Manual review vs. black-box testing

Table 3.6, Figure 3.1, and Figure 3.3 list how many vulnerabilities were found only by manual analysis, only by black-box testing, and by both techniques. All vulnerabilities in the binary vulnerability classes were found by manual review, and none were found by black-box testing.

We observe that manual analysis fared better overall, finding 71 vulnerabilities (including the binary vulnerability classes), while black-box testing found only 39. We also observe that there is very little overlap between the two techniques; the two techniques find different vulnerabilities. Out of a total of 91 vulnerabilities found by either technique, only 19 were found by both techniques (see Figure 3.3). This suggests that they are complementary, and that it may make sense for organizations to use both.

Organizations commonly use only black-box testing. These results suggest that on a smaller budget, this practice makes sense because either technique will find some vulnera-

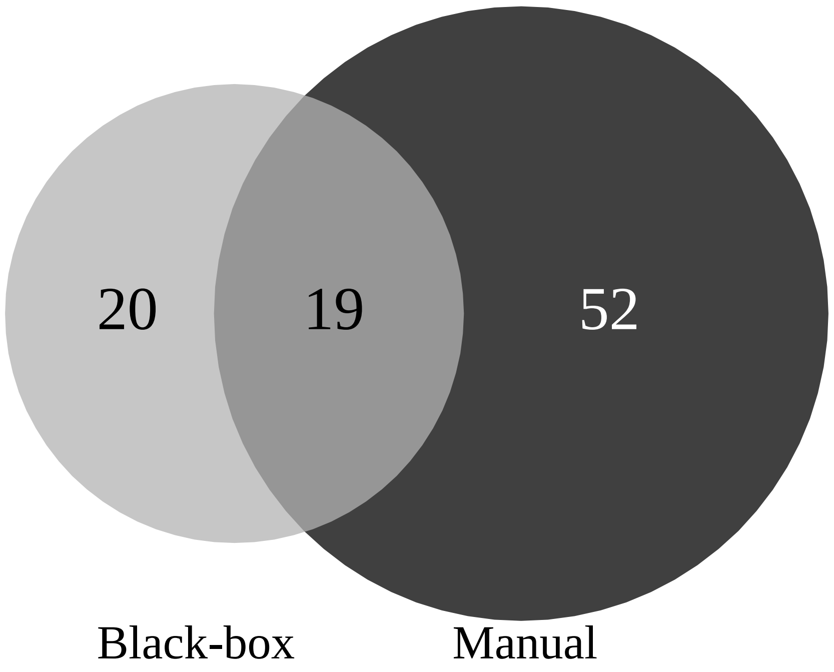


Figure 3.3: Vulnerabilities found by manual analysis and black-box penetration testing.

bilities that the other will miss. If, however, an organization can afford the cost of manual review, *it should supplement this with black-box testing.* The cost is small relative to that of review, and our results suggest that black-box testing will find additional vulnerabilities.

Figure 3.2 reveals that the effectiveness of the two techniques differs depending upon the vulnerability class. Manual review is the clear winner for authentication and authorization bypass and stored XSS vulnerabilities, while black-box testing finds more reflected XSS and SQL injection vulnerabilities. This motivates the need for further research and development of better black-box penetration testing techniques for stored XSS and authentication and authorization bypass vulnerabilities. We note that recent research has made progress toward finding authentication and authorization bypass vulnerabilities [46, 59], but these are not black-box techniques.

Reviewer ability. We now discuss the 20 vulnerabilities that were not found manually. Our analysis of these vulnerabilities further supports our conclusion that black-box testing complements manual review.

For 40% (8) of these, the reviewer found at least one similar vulnerability in the same implementation. That is, there is evidence that the reviewer had the skills and knowledge required to identify these vulnerabilities, but overlooked them. This suggests that we cannot expect a reviewer to have the consistency of an automated tool.

For another 40%, the vulnerability detected by the tool was in framework code, which was not analyzed by the reviewer. An automated tool may find vulnerabilities that reviewers are not even looking for.

The remaining 20% (4) represent vulnerabilities for which no similar vulnerabilities were found by the reviewer in the same implementation. It is possible that the reviewer lacked the necessary skills or knowledge to find these vulnerabilities, since there is no evidence that the reviewer was able to find anything similar.

Framework support

We examine whether stronger framework support is associated with fewer vulnerabilities. Figure 3.4 displays the relationship for each integer-valued vulnerability class between the level of framework support for that class and the number of vulnerabilities in that class. If for some vulnerability class there were an association between the level of framework support and the number of vulnerabilities, we would expect most of the points to be clustered around (or below) a line with a negative slope.

For each of the three[4] classes, we performed a one-way ANOVA test between framework support for the vulnerability class and number of vulnerabilities in the class. None of these results are statistically significant.

As shown in Figure 3.4, our data set allows us to compare only frameworks with no support to frameworks with manual support because the implementations in our data set do not use frameworks with stronger support (with one exception). We found no significant difference between these levels of support. However, this data set does not allow us to examine the effect of opt-in, opt-out, or always-on support on vulnerability rates. In future work, we would like to analyze implementations that use frameworks with stronger support for these vulnerability classes. Example frameworks include CodeIgniter's `xss_clean` [3], Google Ctemplate [7], and Django's `autoescape` [5], all of which provide opt-out support for preventing XSS vulnerabilities. A more diverse data set might reveal relationships that cannot be gleaned from our current data.

Table 3.5 displays the relationship between framework support and vulnerability status for each of the binary vulnerability classes.

There does not appear to be any relationship for password storage. Many of the implementations use frameworks that provide opt-in support for secure password storage, but they do not use this support and are therefore vulnerable anyway. This highlights the fact that manual framework support is only as good as developers' awareness of its existence.

Session management and CSRF do, on the other hand, appear to be in such a relationship. Only the two implementations that lack framework support for session management have vulnerable session management. Similarly, only the two implementations that have framework support for CSRF were not found to be vulnerable to CSRF attacks. Both results were

[4]The level of framework support for stored XSS and reflected XSS is identical in each implementation, so we combined these two classes.

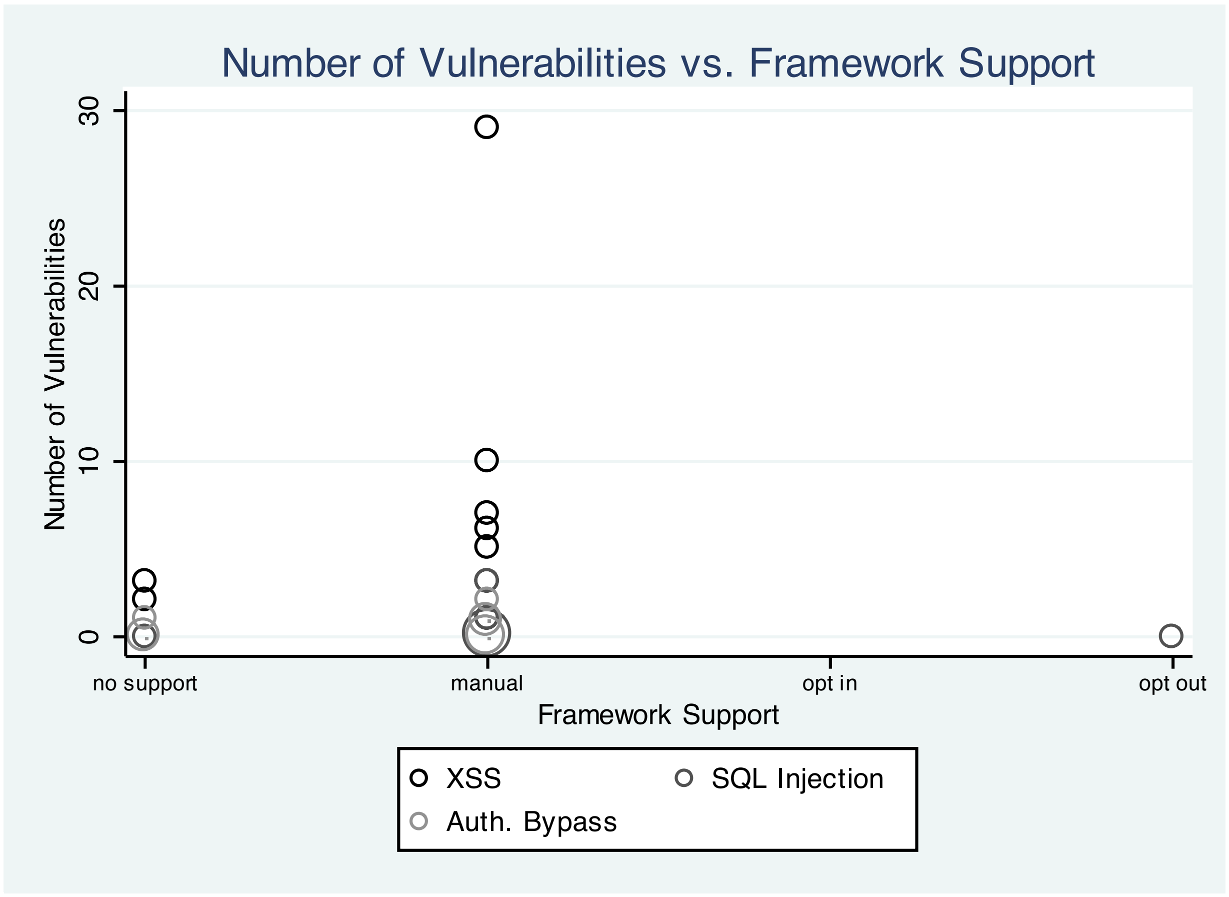


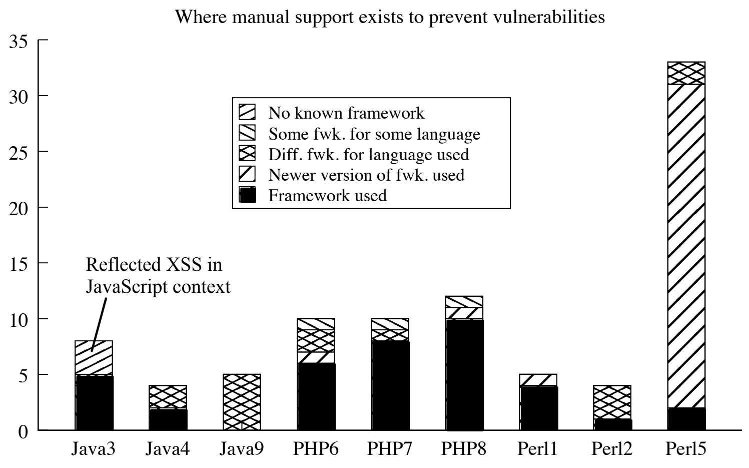
Figure 3.4: Level of framework support vs. number of vulnerabilities for integer-valued vulnerability classes. The area of a mark scales with the number of observations at its center.

found to be statistically significant using Fisher's exact test ($p = 0.028$ for each).

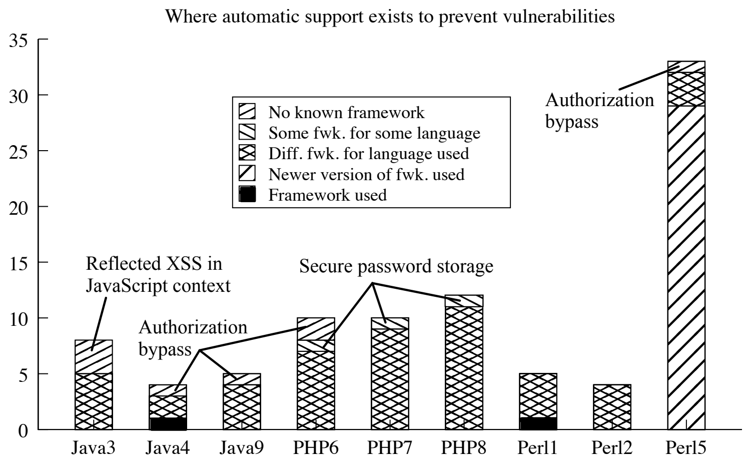
The difference in results between the integer-valued and binary vulnerability classes suggests that manual support does not provide much protection, while more automated support is effective at preventing vulnerabilities. During our manual source code review, we frequently observed that developers were able to correctly use manual support mechanisms in some places, but they forgot or neglected to do so in other places.

Figure 3.5 presents the results from our identification of the lowest level at which framework support exists that could have prevented each individual vulnerability (as described in Section 3.3).

It is rare for developers not to use available automatic support (the darkest bars in Figure 3.5b show only 2 such vulnerabilities), but they commonly fail to use existing manual support (the darkest bars in Figure 3.5a, 37 vulnerabilities). In many cases (30 of the 91 vulnerabilities found), the existing manual support was correctly used elsewhere. This suggests that no matter how good manual defenses are, they will never be good enough;



(a) Manual framework support



(b) Automatic framework support

Figure 3.5: For each vulnerability found, how far developers would have to stray from the technologies they used in order to find framework support that could have prevented each vulnerability, either manually (left) or automatically (right).

developers can forget to use even the best manual framework support, even when it is evident that they are aware of it and know how to use it correctly.

For both manual and automatic support, the majority of vulnerabilities could have been prevented by support from another framework for the same language that the implementation used. That is, it appears that strong framework support exists for most vulnerability classes for each language in this study.

The annotations in Figure 3.5 point out particular shortcomings of frameworks for different vulnerability classes. We did not find any framework that provides any level of support for sanitizing untrusted output in a JavaScript context, which Team 3 failed to do repeatedly, leading to 3 reflected XSS vulnerabilities. We were also unable to find a PHP framework that offers automatic support for secure password storage, though we were able to find many tutorials on how to correctly (but manually) salt and hash passwords in PHP. Finally, we are not aware of any automatic framework support for preventing authorization bypass vulnerabilities. Unlike the other vulnerability classes we consider, these require correct policies; in this sense, this vulnerability class is fundamentally different, and harder to tackle, as acknowledged by recent work [46, 59].

Limitations of statistical analysis

We caution the reader against drawing strong, generalizable conclusions from our statistical analysis, and we view even our strongest results as merely suggestive but not conclusive. Although we entered this study with specific goals and hypotheses (as described in Section 3.2), results that appear statistically significant may not in fact be valid – they could be due to random chance.

When testing 20 hypotheses at a 0.05 significance level, we expect one of them to appear significant purely by chance. We tested 19 hypotheses in this study, and 3 of them appeared to be significant. Therefore, we should not be surprised if one or two of these seemingly-significant associations are in fact spurious and due solely to chance. We believe more powerful studies with larger data sets are needed to convincingly confirm the apparent associations we have found.

3.5 Conclusion and future work

We have analyzed a data set of 9 implementations of the same web application to look for security differences associated with programming language, framework, and method of finding vulnerabilities. Each implementation had at least one vulnerability, which indicates that it is difficult to build a secure web application – even a small, well-defined one.

Our results provide little evidence that programming language plays a role in application security, but they do suggest that the level of framework support for security may influence application security, at least for some classes of vulnerabilities. Even the best manual support is likely not good enough; frameworks should provide automatic defenses if possible.

In future work, we would like to evaluate more modern frameworks that offer stronger support for preventing vulnerabilities. We are aware of several frameworks that provide automatic support for avoiding many types of XSS vulnerabilities (e.g., Django [48], CodeIgniter [3], and GWT SafeHtml [17]).

We have found evidence that manual code review is more effective than black-box testing, but combining the two techniques is more effective than using either one by itself. We found that the two techniques fared differently for different classes of vulnerabilities. Black-box testing performed better for reflected XSS and SQL injection, while manual review performed better for stored XSS, authentication and authorization bypass, session management, CSRF, and insecure password storage. We believe these findings warrant future research with a larger data set, more reviewers, and more black-box tools.

We believe it will be valuable for future research to test the following hypotheses, which were generated from this exploratory study.

- *H1: The practical significance of the difference in security between applications that use different programming languages is negligible.* If true, programmers need not concern themselves with security when choosing which language to use (subject to the support offered by frameworks available for that language).
- *H2: Stronger, more automatic, framework support for vulnerabilities is associated with fewer vulnerabilities.* If true, recent advances in framework support for security have been beneficial, and research into more framework-provided protections should be pursued.
- *H3: Black-box penetration testing tools and manual source code review tend to find different sets of vulnerabilities.* If true, organizations can make more informed decisions regarding their strategy for vulnerability remediation.

We see no reason to limit ourselves to exploring these hypotheses in the context of web applications; they are equally interesting in the context of mobile applications, desktop applications, and network services.

Finally, we note that future work in this area may benefit from additional data sources, such as source code repositories. These rich data sets may help us answer questions about (e.g.,) developers' intentions or misunderstandings when introducing vulnerabilities and how vulnerabilities are introduced into applications over time. A deeper understanding of such issues will aid us in designing new tools and processes that will help developers write more secure software.

Acknowledgments

We thank Adrienne Felt, Erika Chin, and the anonymous reviewers for their thoughtful comments on earlier drafts of this work. We also thank the Plat_Forms team for their hard work in putting together the Plat_Forms contest. This research was partially supported by National Science Foundation grants CNS-1018924 and CCF-0424422.

Additionally, thanks to my co-author of this work, David Wagner [62] for working with me and giving his permission for this work to appear in this thesis.

Chapter 4

Manual security review

4.1 Introduction

In this chapter we focus on 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 with varying amounts of security experience to conduct a security code review of a simple web application. These reviewers 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.

4.2 Goals

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

This chapter was previously published at the 5th International Symposium on Engineering Secure Software and Systems [52].

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?

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?

Predicting Effectiveness

We asked each reviewer about the following factors (see Appendix B for the complete questionnaire), 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.

4.3 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.

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 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 reviewers 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 (seven after one additional vulnerability was discovered by participants, which we describe further in Section 4.4). 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 4.4 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.

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 [110]. We used oDesk to post the code review task, hire reviewers that met our requirements, and pay our subjects for their work. The protocol for our experiment was approved by the UC Berkeley Institutional Review Board (IRB).

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 code base, 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.

Task

We gave participants directions on how to proceed with the code review, an example vulnerability report, and the TestCMS code base. The instructions specified that no automated code review tools should be used. Also, the reviewers 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 [94]. We designed a template that participants were instructed to use to report each vulnerability. The template has the following sections for the reviewer to fill in accordingly:

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

The type, location, description, and impact gave us the information needed for us to understand the vulnerability. The last section, "Steps to Exploit", was intended to encourage reviewers to report only exploitable vulnerabilities as opposed to poor security practices in the code.

The reviewers 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.

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 4.4. After running statistical tests on the data, we were able to quantify how well the reviewers conducted their reviews.

Threats to Validity

oDesk Population.

As stated previously, we hired reviewers 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 size and volume of activity on oDesk (1.5 million jobs posted in 2012, 3.1 million registered contractors [23]), 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 4.3, we introduced two SQL Injection vulnerabilities. Adding these artificial vulnerabilities creates an artificially flawed code base 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 code base were modeled after vulnerabilities found in other CMSs, which we hope will serve to minimize the artificiality of the code base 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 reviewers 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 reviewer 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 reviewers 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 reviewers 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.

4.4 Results

Vulnerability Report Classification

In order to understand the reviewers' 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 reviewer 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 reviewer 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 reviewer 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. We did not validate reported weaknesses. Weaknesses contributed to the total number of reports each reviewer submitted, but were considered neither correct nor false.

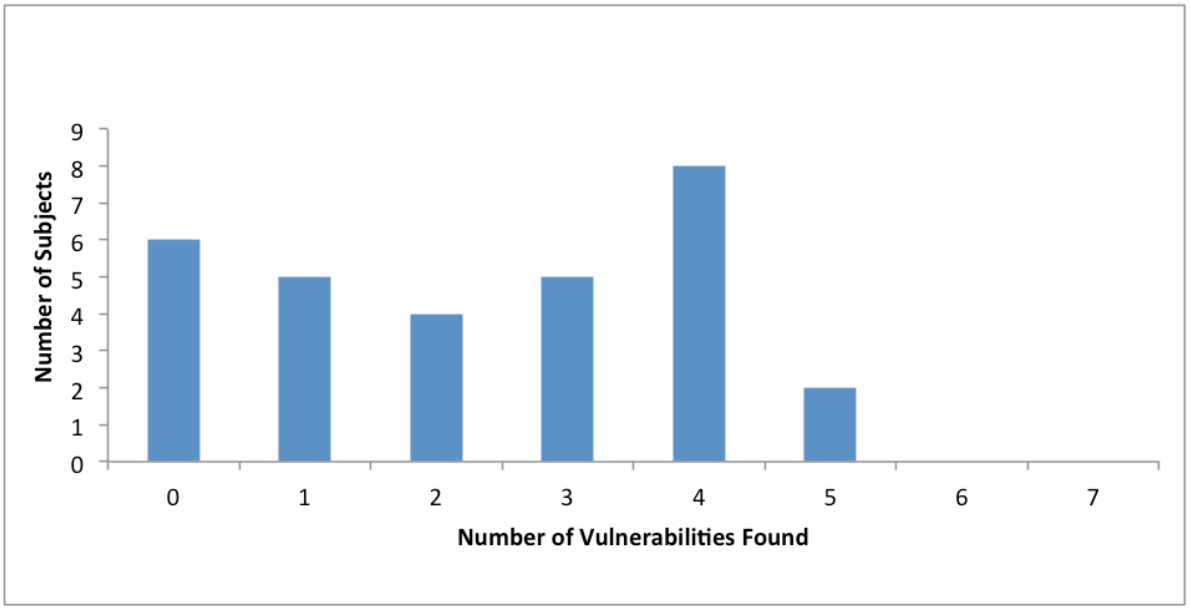


Figure 4.1: The number of vulnerabilities found by individual reviewers.

Reports involving vulnerabilities in the administrative interface were placed in the out of scope category, as we asked reviewers not to report these vulnerabilities. Reports in this category as well as duplicate reports were ignored; they were not considered when counting the reviewer's total reports, valid reports, or invalid reports.

As we manually checked through all reports from all reviewers, 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.

Reviewer Effectiveness

We measured the relative effectiveness of each reviewer by counting the total number of valid reports from that reviewer 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 vulnerabilities was found by the majority of subjects, while only 17% of the reviewers found the lack of Cross-Site Request Forgery protection. Table 4.1 shows the percentage of reviewers who reported each vulnerability. The average number of correct vulnerabilities found was 2.33 with a standard deviation of 1.67. Figure 4.1 shows a histogram of the number of vulnerabilities found by each reviewer. This data shows that some reviewers are more effective than others, which

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%

Table 4.1: The proportion of reviewers who reported each vulnerability.

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

Table 4.2: Pairs of vulnerabilities for which finding one correlates with finding the other.

addresses questions (2) and (3) in Section 4.2. The histogram also shows that none of the reviewers found more than five of the seven vulnerabilities and about 20% did not find any vulnerabilities.

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 4.2 contains pairs that were significantly correlated, which is shown by their corresponding correlation coefficient (r) and p-value. The first three rows of Table 4.2 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; reviewers may have been specifically looking for SQL Injection vulnerabilities.

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 reviewer. The average number of reported vulnerabilities is 6.29 with a standard deviation of 5.87. Figure 4.2 shows a histogram of the percentage 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

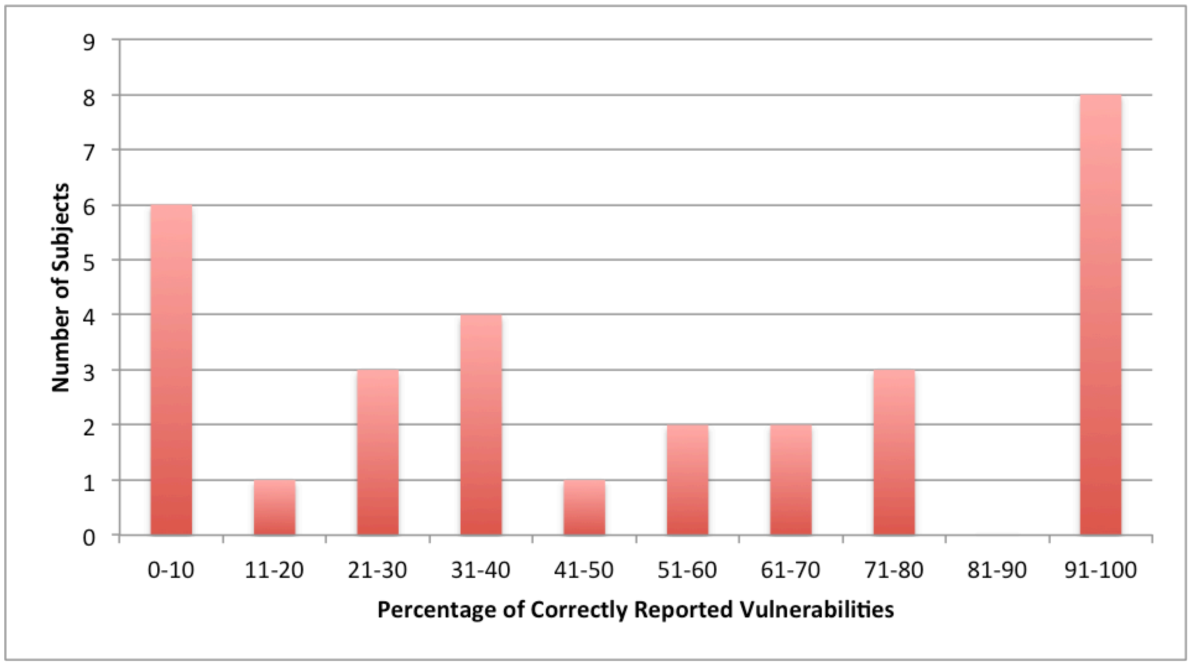


Figure 4.2: The percentage of reported vulnerabilities that were correct.

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 4.2.

Figure 4.3 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 reviewer 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.

Optimal Number of Reviewers

In order to determine the optimal number of reviewers, we simulated hiring various numbers of reviewers. In each simulation, we chose X random 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;

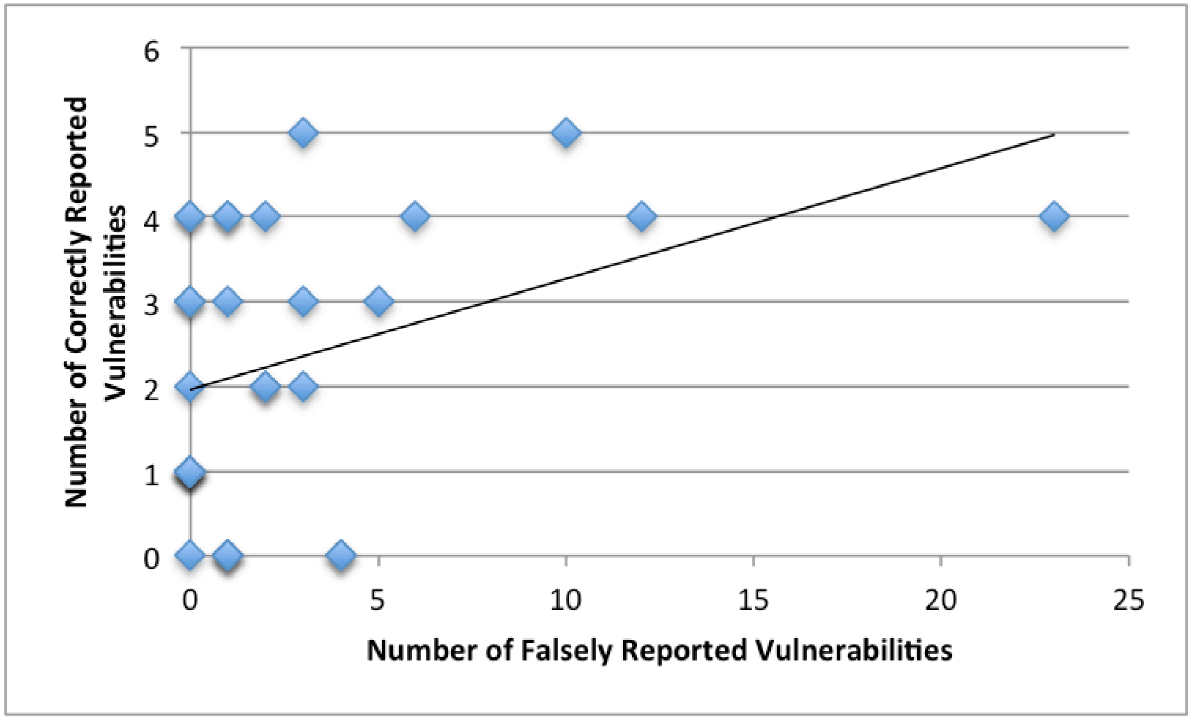


Figure 4.3: The relationship between correctly reported vulnerabilities and falsely reported vulnerabilities.

this estimates the probability of finding all vulnerabilities with X reviewers. Figure 4.4 shows the probability of finding all vulnerabilities based on the number of reviewers 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 4.2.

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 review. We were surprised to find these criteria to be of little use in predicting reviewer 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 used in a way that diverges from the

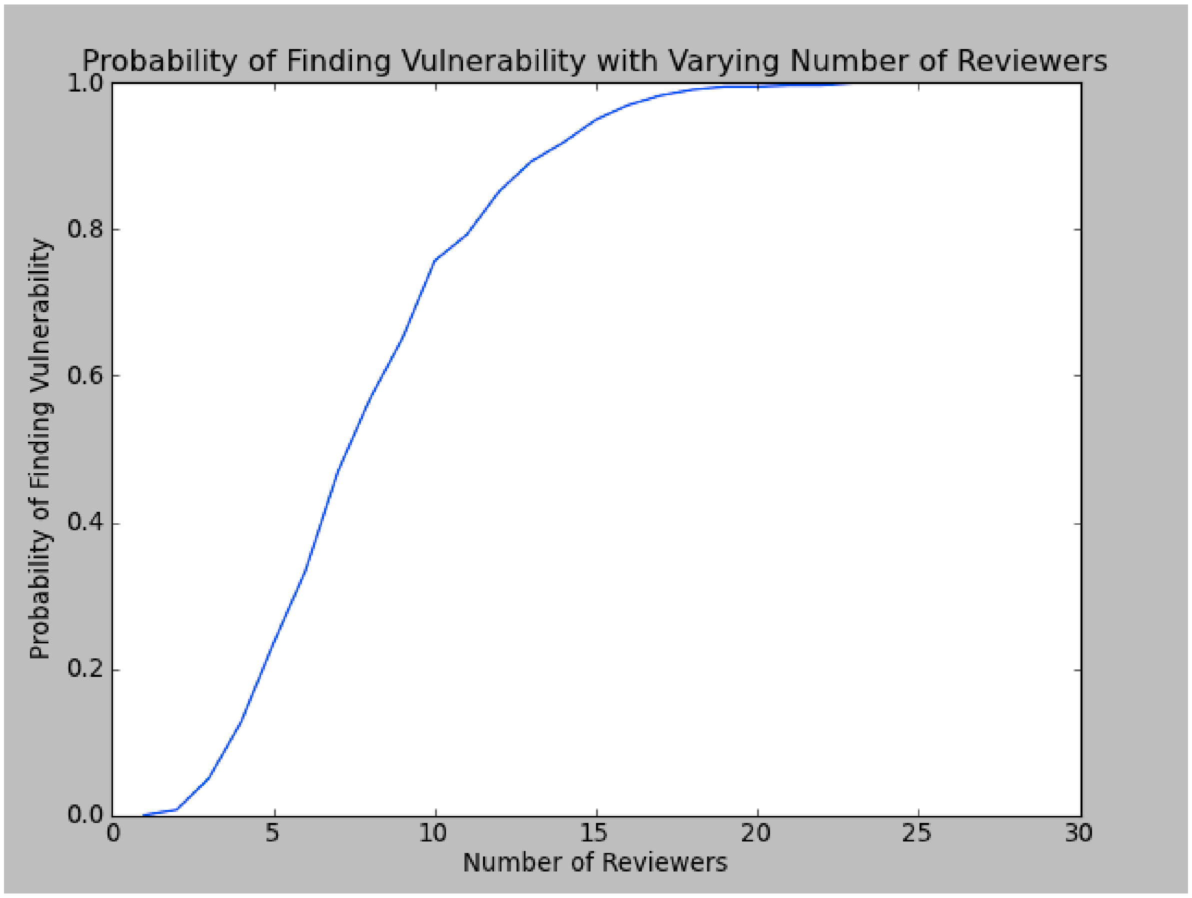


Figure 4.4: Probability of finding all vulnerabilities depending on the number of reviewers hired.

specification [29]. 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 reviewer's reports. We define accuracy as the fraction of correctly reported vulnerabilities out of the total reported vulnerabilities. Figure 4.5 shows this relationship with a correlation coefficient of -0.4141 and a p-value of

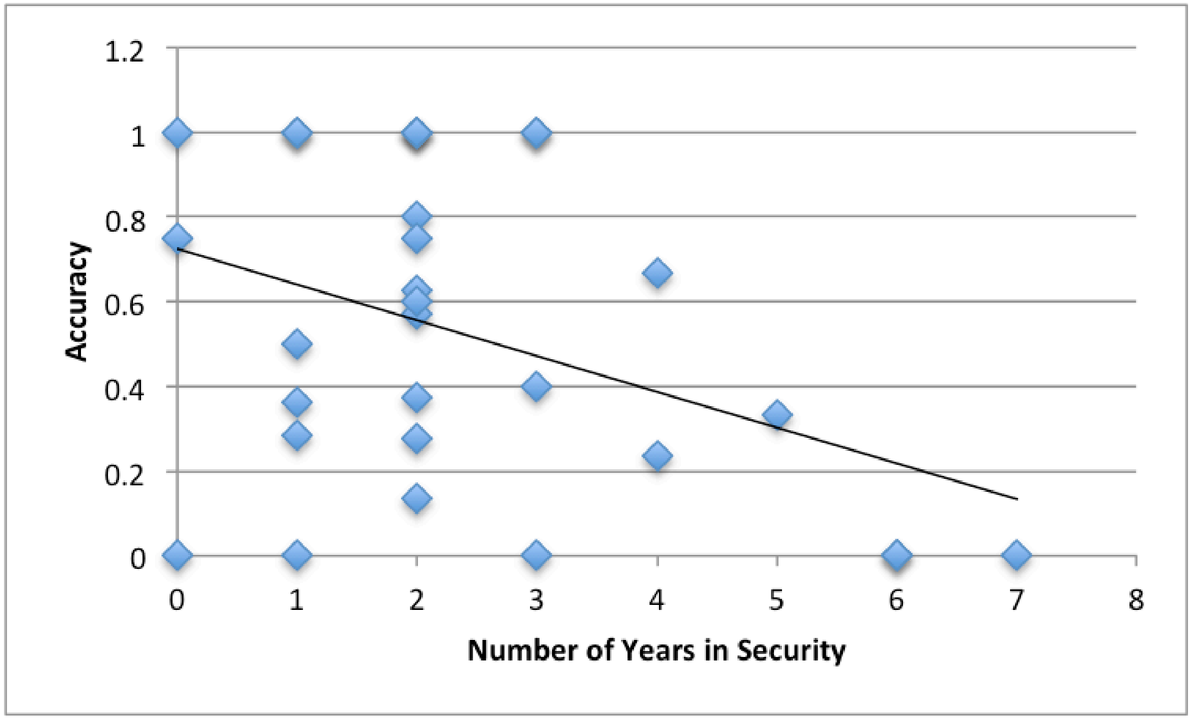


Figure 4.5: The relationship between years of experience in computer security and accuracy of reports. Accuracy is defined as the fraction of correctly reported vulnerabilities out of the total number of reported vulnerabilities.

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

Table 4.3: The p-values for correlation tests between experience in different areas and the number of correct vulnerabilities reported.

.0229. While this is statistically significant in our data set, it is not what would be expected because it indicates that the more years of experience a reviewer has, the lower the reviewer's accuracy.

We did not find significant correlations between the number of correct vulnerabilities reported and reviewer experience with software development, web development, or computer security. Table 4.3 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 4.6 shows this relationship.

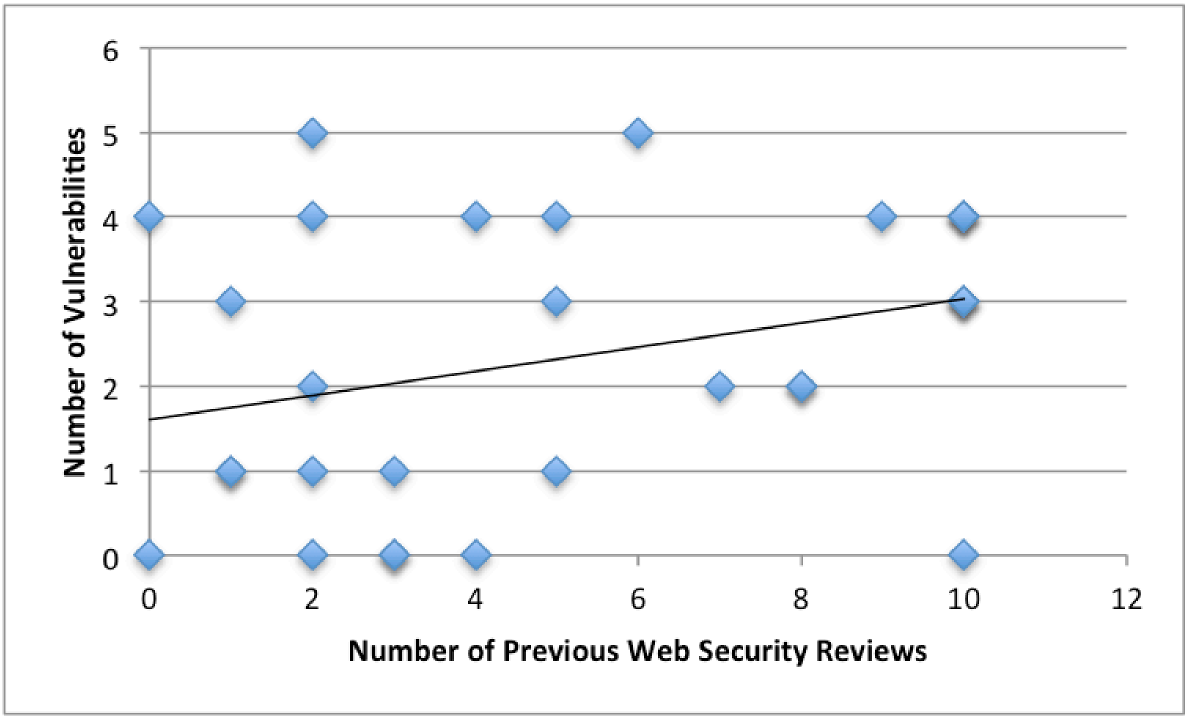


Figure 4.6: The correlation between the number of previously conducted security code reviews and the number of correctly reported vulnerabilities.

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.

4.5 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. 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 reviewer 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.

Additionally, thanks to my co-authors of this work, Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Adrian Mettler, and David Wagner [52] for working with me and giving their permission for this work to appear in this thesis.

Chapter 5

Vulnerability rewards programs

5.1 Introduction

Some software vendors pay security researchers for the responsible disclosure of a security vulnerability. Programs implementing the rules for this exchange are known as vulnerability rewards programs (VRPs) or bug bounty programs. The last couple of years have seen an upsurge of interest in VRPs, with some vendors expanding their existing programs [26, 53], others introducing new programs [24, 30, 87], and some companies offering to act as an intermediary between security researchers and vendors offering VRPs [10].

VRPs offer a number of potential attractions to software vendors. Offering adequate incentives entices security researchers to look for vulnerabilities, and this increased attention improves the likelihood of finding latent vulnerabilities.[1] Second, coordinating with security researchers allows vendors to more effectively manage vulnerability disclosures, reducing the likelihood of unexpected and costly zero-day disclosures. Monetary rewards provide an incentive for security researchers not to sell their research results to malicious actors in the underground economy or the gray world of vulnerability markets. Third, VRPs may make it more difficult for black hats to find vulnerabilities to exploit. Patching vulnerabilities found through a VRP increases the difficulty and therefore cost for malicious actors to find zero-days because the pool of latent vulnerabilities has been diminished. Additionally, experience gained from VRPs (and exploit bounties [57, 64]) can yield improvements to mitigation techniques and help identify other related vulnerabilities and sources of bugs. Finally, VRPs often engender goodwill amongst the community of security researchers. Taken together, VRPs provide an attractive tool for increasing product security and protecting customers.

Despite their potential benefits, there is an active debate over the value and effectiveness of VRPs. A number of vendors, notably Microsoft, Adobe, and Oracle, do not maintain a VRP, with Microsoft arguing that VRPs do not represent the best return on investment on a per-bug basis [63]. Further, it is also not clear if the bounties awarded are a sufficient attraction for security researchers motivated by money—underground economy prices for

[1]For instance, Linus's Law suggests "Given enough eyeballs, all bugs are shallow." [101]

vulnerabilities are far higher than those offered by VRPs [85, 54].

Given the emergence of VRPs as a component of the secure development lifecycle and the debate over the efficacy of such programs, we use available data to better understand existing VRPs. We focus on the Google Chrome and Mozilla Firefox web browsers, both of which are widely considered to have mature and successful VRPs, as case studies. We analyze these VRPs along several dimensions with the intention of better understanding the characteristics, metrics, and trajectory of a successful VRP.

We make the following contributions:

- We collect and analyze data on vulnerability rewards over the last 3 years for the Google Chrome VRP and the Mozilla Firefox VRP (Section 5.3).
- We assess the state of these VRPs along several dimensions, including costs, benefits, popularity, and efficacy (Section 5.4).
- We make concrete recommendations for software vendors aiming to start or evolve their own VRP (Section 5.5).

5.2 Background

A *secure software development lifecycle (SDLC)* aims to address software security throughout the entire software development process, from before specifications are developed to long after software has been released [47]. A *vulnerability remediation strategy* is any systematic approach whose goal is to reduce the number of software vulnerabilities [14]. Vulnerability remediation strategies are one important part of an SDLC, complemented by things like incident response [13], operational security considerations [12], and defense in depth [4].

Potential vulnerability remediation strategies include:

- **Code reviews.** These can range from informal, as-needed requests for code review to systematized, formal processes for mandatory code inspection. Typically, SDLCs also include an extra security review for security critical features.
- **Penetration testing.** Software vendors may perform in-house penetration testing or may hire external companies who specialize in this service. Penetration testing ranges from manual to automated.
- **Use of dynamic and static analysis tools.** Specialized tools exist for catching a wide range of flaws, e.g., memory safety vulnerabilities, configuration errors, and concurrency bugs.
- **Vulnerability rewards programs.** The focus of our study, VRPs have recently received increased attention from the security community.

How such strategies are systematized and realized varies widely between software vendors. One company might require mandatory code reviews before code check-in, while another might hire outside penetration testing experts a month before product release. Vendors often combine or innovate on existing strategies.

Vulnerability rewards programs (VRPs) appear to be emerging as a viable vulnerability remediation strategy. Many companies have them, and their popularity continues to grow [40, 37]. But VRPs have gone largely unstudied. For a company considering the use of a VRP in their SDLC, guidance is limited.

By studying successful VRPs, we aim to provide guidance on the development of new VRPs and the evolution and maturation of existing VRPs. Vendors looking to grow their VRPs can benefit from an improved understanding of successful VRPs: for instance, they can gauge where they stand by comparing their program to other successful VRPs.

Toward this end, we measure, characterize, and discuss the Google Chrome and Mozilla Firefox VRPs. We choose these VRPs in particular because browsers are a popular target for malicious actors today. Their ubiquitous nature and their massive, complex code base with significant legacy components make them especially vulnerable. Complex, high-performance components with a large attack surface such as JavaScript JITs also provide an alluring target for malicious actors. For the same reasons, they are also widely studied by security researchers; they therefore provide a large sample size for our study. In addition, browser vendors were among the first to offer rewards for vulnerabilities: Mozilla's VRP started in 2004 and Google introduced the Chrome VRP in 2010, before the security community at large adopted VRPs as a viable vulnerability remediation strategy.

Google Chrome VRP

The Google Chrome VRP[2] is widely considered an exemplar of a mature, successful VRP. When first introduced in January 2010, the Google Chrome VRP offered researchers rewards ranging from \$500 for high- and critical-severity bugs, with a special \$1337 reward for particularly critical or clever bugs. Over time, the typical payout increased to a \$1000 minimum with a maximum payout of \$3133.7 for high-impact vulnerabilities. Additionally, the Chrome team, at its discretion, sometimes provides rewards of up to \$30,000 [55].

Google also sponsors a separate exploit bounty program, the bi-annual "pwnium" competition [57]. This program focuses on full exploits; a mere vulnerability is not enough. In return, it awards higher bounties (as high as \$150,000) for these exploits [39]. Reliable exploits for modern browsers typically involve multiple vulnerabilities and significant engineering effort. For example, the two winning entries in a recent "pwnium" contest required six and ten vulnerabilities in addition to "impressive" engineering in order to achieve a successful exploit [93, 38]. Our focus is on programs that provide bounties for vulnerabilities; we do not consider exploit bounties in this work.

[2]The program is officially the Chromium VRP with prizes sponsored by Google. We refer to it as the Google Chrome VRP for ease of exposition.

The severity of a vulnerability plays a key role in deciding reward amounts. Google Chrome uses a clear guideline for deciding severity [11]. In short, a critical vulnerability allows an attacker to run arbitrary native code on the user's machine; for instance, web-accessible memory corruption vulnerabilities that appear in the Chrome kernel,[3] are typically critical severity. A high-severity vulnerability is one that allows an attacker to bypass the same-origin policy, e.g., via a Universal XSS vulnerability (which enables an attacker to mount an XSS attack on any web site) or a memory corruption error in the sandbox. A vulnerability is of medium severity if achieving a high/critical status requires user interaction, or if the vulnerability only allows limited disclosure of information. Finally, a low-severity vulnerability refers to all the remaining security vulnerabilities that do not give the attacker control over critical browser features. Medium-severity vulnerabilities typically receive rewards of $500, and low-severity vulnerabilities typically do not receive rewards.

Mozilla Firefox VRP

Mozilla's VRP is, to the best of our knowledge, the oldest VRP in the industry. First started in 2004 and based on a similar project at Netscape in the '90s, the Mozilla VRP rewarded researchers with $500 for high- or critical-severity security bugs. Starting July 1, 2010 Mozilla expanded its program to award all high/critical vulnerabilities $3000 [26].

Mozilla's security ratings are similar to that of Chrome. Critical vulnerabilities allow arbitrary code execution on the user's computer. Vulnerabilities that allow an attacker to bypass the same-origin policy or access confidential information on the user's computer are high severity. Due to the absence of privilege separation in the Firefox browser, *all* memory corruption vulnerabilities are critical, regardless of the component affected. Mozilla is currently investigating a privilege-separated design for Firefox [15, 81].

Mozilla's VRP also qualitatively differs from the Google program. First, Mozilla awards bounties even if the researcher publicly discusses the vulnerability instead of reporting it to Mozilla.[4] Second, Mozilla also explicitly awards vulnerabilities discovered in "nightly" (or "trunk") versions of Firefox. In contrast, Google discourages researchers for using "canary" builds and only awards bounties in canary builds if internal testing would miss those bugs [108].

Goals

We intend to improve our understanding of the following characteristics of a successful VRP: (1) Expected cost, (2) expected benefits, (3) incentive levels effective for encouraging and sustaining community participation, and (4) volume of VRP activity (e.g., number of patches coming out of VRP reports).

[3]Chrome follows a privilege-separated design [31]. The Chrome kernel refers to the privileged component.

[4]But Mozilla reports that this was a rare occurrence over the period of time we consider, possibly because the VRP became widely known [111].

We do so by studying available data coming out of two exemplars of successful VRPs, that of Google Chrome and Mozilla Firefox. Understanding these VRPs will allow these vendors to evaluate and improve their programs, and it will suggest targets for other vendors to strive toward with their VRPs. At minimum, we hope to arrive at a better understanding of the current state of VRPs and how they have evolved. At best, we aim to make concrete suggestions for the development and improvement of VRPs.

5.3 Methodology

For both browsers, we collect all bugs for which rewards were issued through the browser vendor's VRP. To evaluate the impact of the VRP as a component of the SDLC, we also collected all security bugs affecting stable releases. We chose to look only at bugs affecting stable releases to ignore the impact of transient bugs and regressions caught by internal testing.

For each bug in the above two data sets, we gathered the following details: (1) severity of the bug, (2) reward amount, (3) reporter name, (4) report date. For bugs affecting stable releases, we also aimed to gather the date a release patching the bug became available for download. As we discuss below, we were able to gather this data for only a subset of all bugs.

For all bugs, we mark a bug as internally or externally reported via a simple heuristic: if a reward was issued, the reporter was external, and otherwise the reporter was internal. Because low and medium severity vulnerabilities usually do not receive bounties, we only look at critical/high vulnerabilities when comparing internal and external bug reports. While all high/critical vulnerabilities are eligible for an award, a researcher can still refuse an award, in which case, our heuristic falsely marks the bug "internal." We are aware of a handful of such instances, but there are not enough of these in our data set to affect our analysis.

In this section, we present how we gathered this data set for Chrome and Firefox. We first discuss how we identify the list of bugs affecting stable releases and bugs awarded bounties, followed by a discussion on how we identified, for each bug, other details such as severity. Finally, we discuss threats to the validity of our measurement study.

Gathering the Google Chrome data set

We gathered data from the official Chromium bug tracker [20] after confirming with Google employees that the bug tracker contained up-to-date, authoritative data on rewards issued through their VRP. We search the bug tracker for bugs marked with the special "Reward" label to collect bugs identified through the VRP. Next, we searched the bug tracker for bugs marked with the special "Security-Impact: Stable" to collect bugs affecting stable releases. Next, we remove the special Pwnium [57] rewards from all data sets because Pwnium rewards *exploits* instead of vulnerabilities as in the regular VRP. This provides us with 501 bugs identified through the VRP and 1347 bugs affecting stable releases.

The Chromium Bug tracker provides a convenient interface to export detailed bug metadata, including severity, reporter, and report date, into a CSV file, which we use to appropriately populate our data set. We identify the reward amounts using the "Reward" label.

Unfortunately, the Chromium bug tracker does not include the release date of bug fixes. Instead, we gather this data from the Google Chromium release blog [1]. For each stable release of the Chromium browser, Google releases a blog post listing security bugs fixed in a release. For the subset of bugs mentioned in these release notes, we extract the release date of the stable version of Chrome that patches the bug.

Gathering the Mozilla Firefox data set

Similar to Google Chrome, we searched Bugzilla, the Firefox bug tracker, for an attachment used to tag a bug bounty.[5] We identified 190 bugs via this search.

Unlike the Chrome bug tracker, Bugzilla does not provide a convenient label to identify bugs affecting stable releases. Instead, Mozilla releases Mozilla Foundation Security Advisories (MFSA) with every stable release of Mozilla Firefox [88]. We scraped these advisories for a list of bugs affecting stable releases. We also use the MFSAs to identify the release date of a patched, stable version of Firefox. We gathered 613 unique bugs from the MFSA advisories dating back to March 22, 2010 (Firefox 3.6).

Similar to the Chromium Bug tracker, the Bugzilla website provides a convenient interface to export detailed bug data into a CSV file for further analysis. We used Bugzilla to collect, for each bug above, the bug reporter, the severity rating, and the date reported. The security severity rating for a bug is part of the Bugzilla keywords field and not Bugzilla's severity field. We do not separately collect the amount paid because, as previously discussed, Mozilla's expanded bounty program awards $3,000 for all critical/high vulnerabilities.[6]

Threats to validity

In this section, we document potential threats to validity so readers can better understand and take into account the sources of error and bias in our study.

It is possible that our data sets are incomplete, i.e., there exist patched vulnerabilities that do not appear in our data. For example, for both Chrome and Firefox, we rely heavily on the keyword/label metadata to identify bugs; since this labeling is a manual process, it is possible that we are missing bugs. To gather the list of bugs affecting stable releases, we use the bug tracker for Chrome but use security advisories for Mozilla, which could be incomplete. Given the large number of vulnerabilities we do have in our data sets, we expect

[5]The existence of this attachment is not always visible to the public. We acknowledge the support of Mozilla contributor Dan Veditz for his help gathering this data.

[6]8% of the 190 bugs in our rewards data-set received bounties of less than $3,000. Since the details of bounties are not public, we cannot distinguish these bugs. We do not believe this affects the conclusions of our analysis.

Severity	Chrome		Firefox	
	Stable	Bounty	Stable	Bounty
Low	226	1	16	1
Medium	288	72	66	9
High	793	395	79	38
Critical	32	20	393	142
Unknown	8	13	59	0
Total	1347	501	190	613

Table 5.1: Number of observations in our data set.

that a small number of missing observations would not materially influence the conclusions we draw.

We treat all rewards in the Firefox VRP as $3,000 despite knowing that 8% of the rewards were for less than this amount [111]. Data on which rewards were for less money and what those amounts were is not publicly available. Any results we present regarding amounts paid out for Firefox vulnerabilities may therefore have as much as 8% error, though we expect a far lower error, if any.

Parts of our analysis also compare Firefox and Chrome VRPs in terms of number of bugs found, which assumes that finding security vulnerabilities in these browsers requires comparable skills and resources. It could be the case that it is just easier to find bugs in one over the other, or one browser has a lower barrier to entry for vulnerability researchers. For example, the popular Address Sanitizer tool only worked on Google Chrome until Mozilla developers tweaked their build infrastructure to enable support for the same [68].

We study only two VRPs; our results do not necessarily generalize to any other VRPs. We caution the reader to avoid generalizing to other VRPs, but instead take our results as case studies of two successful VRPs.

5.4 Results

We study the VRPs along three axes: benefits to product security, costs to the vendor, and incentives for security researchers. As previously discussed, the two VRPs we study differ in their design. We focus on these differences to identify actionable information, if any, for Google and Mozilla as well as lessons for prospective VRPs. Our data set also provides the opportunity for insights into the state of the SDLC of Chrome and Firefox; we discuss these results last.

Table 5.1 presents information about the final data set we used for our analysis. We have made our data set available online for independent analysis [61].

Benefits to product security

Because VRPs form a part of the SDLC, the critical success metric of such a program is improvement in product security. Because directly measuring improvement to product security is difficult, we rely on secondary indicators like the number of vulnerabilities patched, severity of bugs, and the number of researchers involved in finding and sharing vulnerabilities.

Number of vulnerabilities

We find that the Chrome VRP uncovers about 2.6 times as many vulnerabilities as that of Firefox (501 vs. 190), despite the fact that Chrome's total number of advisories is only 2.2 times that of Firefox (Table 5.1). 27.5% of Chrome advisories originate from VRP contributions (371 of 1347), and 24.1% of Firefox advisories (148 of 613) result from VRP contributions.

Discussion. Both VRPs yield a significant fraction of the total number of security advisories, which is a clear benefit. Chrome is seeing approximately 1.14 times the benefit of Firefox by our metric of fraction of advisories resulting from VRP contributions.

Vulnerability severity

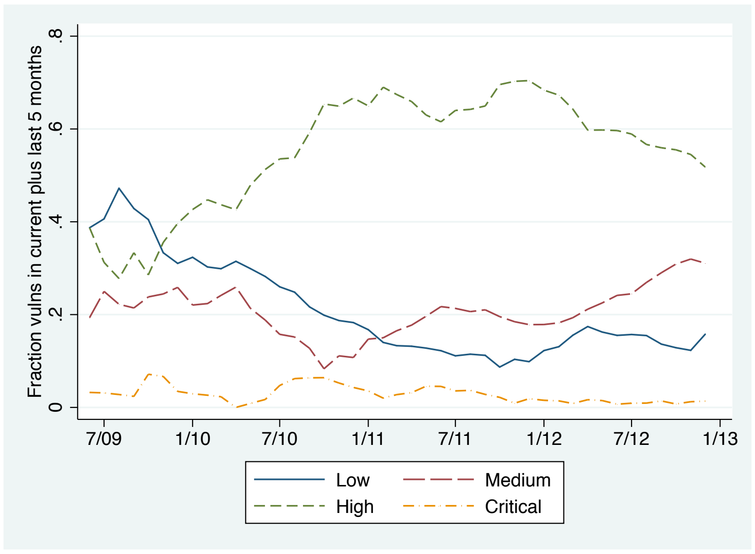
VRPs typically award bounties only for critical- and high- severity vulnerabilities, but they aim to encourage security research and responsible disclosure in general. We investigate the breakdown of the severity of vulnerabilities awarded a bounty under the VRPs in order to better understand the broader SDLC of the browsers.

Table 5.1 lists the total number of vulnerabilities by severity for Firefox and Chrome. Figure 5.1 plots the fraction of vulnerabilities at each severity level over the current plus 5 previous months.

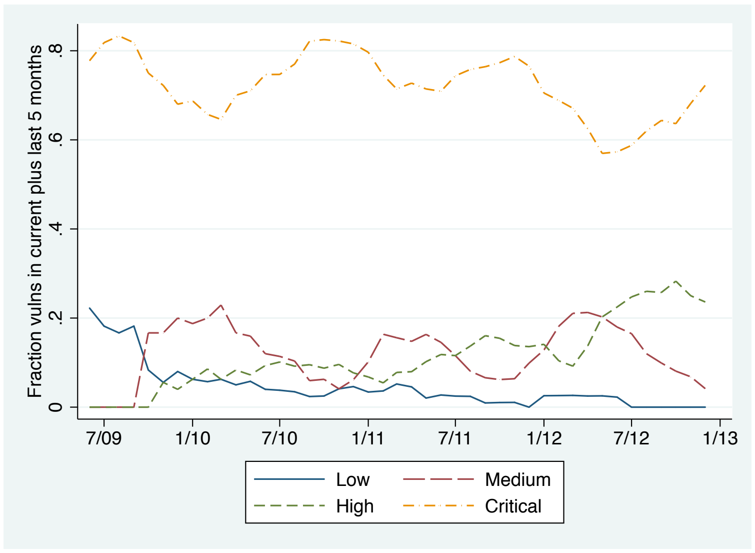
Discussion. Firefox has a much higher ratio of critical vulnerabilities to high vulnerabilities than Chrome. We expect that many of Firefox's critical vulnerabilities would instead be high severity if, like Chrome, it also had a privilege-separated architecture. The lack of such an architecture means that any memory corruption vulnerability in Firefox is a critical vulnerability. We hypothesize that this architectural difference is the most influential factor in causing such a large difference in vulnerability severity classification.

The fraction of critical severity bugs has remained relatively constant for Chrome. We also notice the start of a trend in Chrome—the fraction of high severity vulnerabilities is declining and the fraction of medium severity vulnerabilities is increasing.

Chrome's privilege-separated architecture means that a critical vulnerability indicates malicious code execution in the privileged process. We see that there continue to be new bugs resulting in code execution in the privileged process. Further investigation into these bugs can help understand how and why they continue to surface.



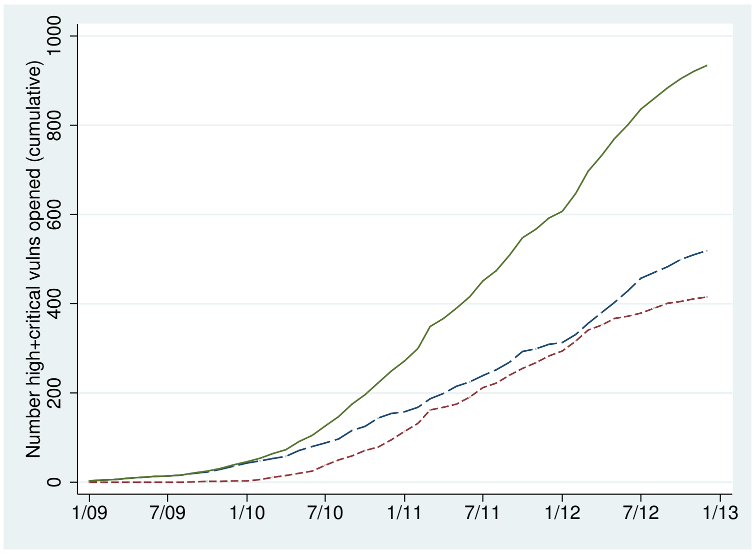
(a) Chrome



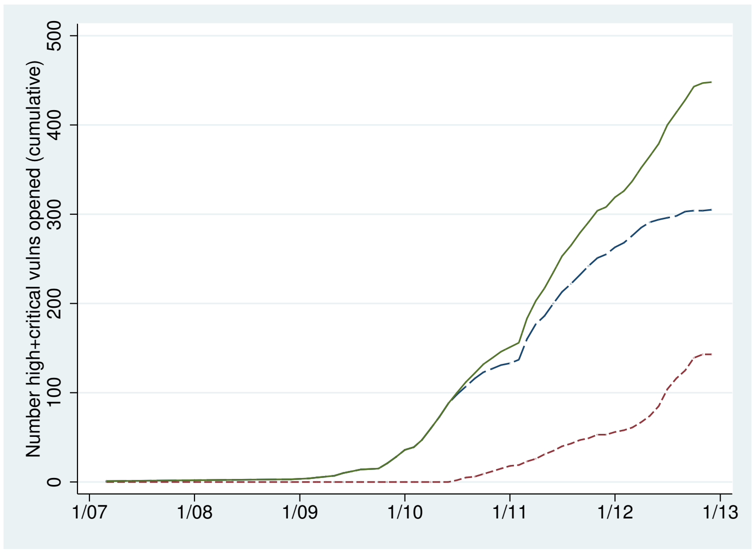
(b) Firefox

Figure 5.1: Moving average over the current plus 5 previous months of the percentage of vulnerabilities at each severity level (low is blue solid line, medium is red long-dashed line, high is green short-dashed line, and critical is yellow dash-dotted line).

Low-severity vulnerabilities in Google Chrome make up a significant fraction of all vulnerabilities reported. In contrast, the fraction of low- and medium-severity vulnerabilities in Firefox remains negligible.



(a) Chrome

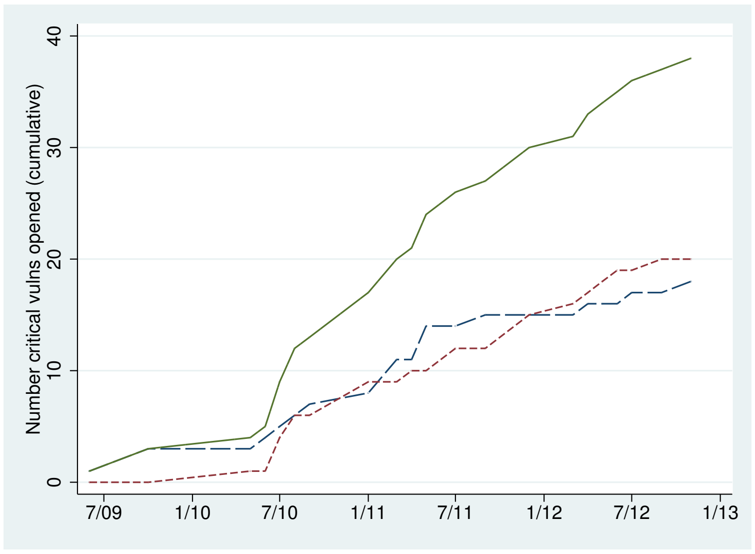


(b) Firefox

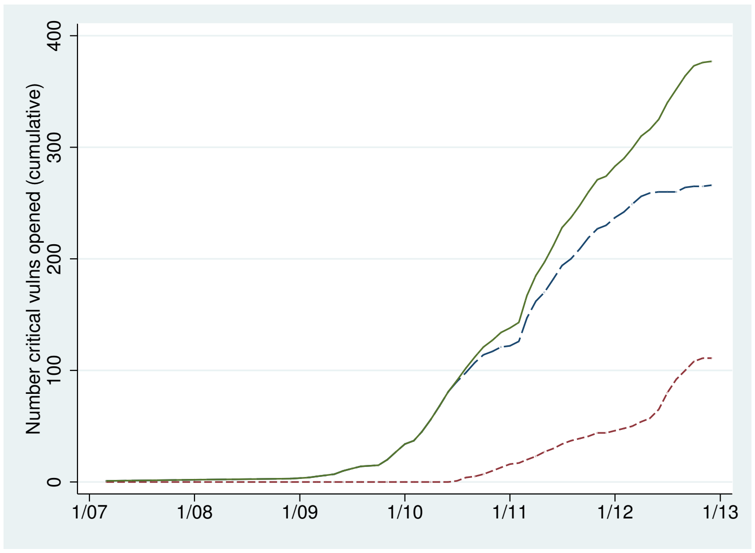
Figure 5.2: Number of high- plus critical-severity vulnerabilities reported over time, discovered internally (blue long-dashed line), externally (red short-dashed line), and total (green solid line).

Community engagement

One advantage of VRPs is engagement with the broader security community. We studied this engagement along two axes: (1) the contribution of internal and external researchers towards identifying security vulnerabilities, and (2) the number of external participants.



(a) Chrome



(b) Firefox

Figure 5.3: Number of critical-severity vulnerabilities reported over time, discovered internally (blue long-dashed line), externally (red short-dashed line), and total (green solid line).

Figure 5.2 depicts the cumulative number of high- and critical-severity vulnerabilities patched and Figure 5.3 depicts the same, but for only critical vulnerabilities. Table 5.2 shows the distribution of the total number of vulnerabilities reported by each external participant in each of the two VRPs. Although a few external participants submit many bugs, there is

# Bugs	Freq.
1	45
3	2
4	1
6	1
10	1
12	2
13	3
16	1
17	1
22	1
24	1
27	1
35	1
48	1
92	1
Total	63

(a) Chrome

# Bugs	Freq.
1	46
2	9
3	4
5	1
6	1
9	1
10	1
12	1
14	1
47	1
Total	66

(b) Firefox

Table 5.2: Frequency distribution of number of high- or critical-severity vulnerabilities found by external contributors.

a clear long tail of participants in both VRPs. Table 5.3 shows the same distribution, but for internal (i.e., employee) reporters of vulnerabilities.

Discussion. For both browsers, internal contributions for high- and critical-severity vulnerabilities have consistently yielded the majority of patches. The number of external contributions to Chrome has nearly caught up with the number of internal contributions (i.e., around 4/11 and 3/12, in Figure 5.2a) at various times, and as of the end of our study, these two quantities are comparable. Considering only critical-severity vulnerabilities, external contributions have exceeded internal contributions as of the end of our study. For Firefox, on the other hand, the number of external contributions has consistently been far lower than the number of internal contributions.

We observe an increase in the rate of external contributions to Chrome starting around July 2010, six months after the inception of the VRP. As seen in Figure 5.3a, this is more pronounced when considering only critical-severity vulnerabilities. We conjecture that this change corresponds to increased publicity for the Chrome VRP after Google increased reward amounts [53].

Linus's Law, defined by Eric Raymond as "Given enough eyes, all bugs are shallow," suggests that it is in the interests of the software vendor to encourage more people to par-

# Bugs	Freq.
1	67
2	10
3	10
4	2
5	2
14	1
20	2
67	1
263	1
Total	96

(a) Chrome

# Bugs	Freq.
1	43
2	10
3	7
4	3
5	2
6	2
7	1
12	1
13	1
15	1
17	2
18	1
21	1
23	1
44	1
Total	77

(b) Firefox

Table 5.3: Frequency distribution of number of high- or critical-severity bugs found by internal contributors.

ticipate in the search for bugs. The distributions of bugs found by external participants indicate that both VRPs have been successful in encouraging broad community participation. The existence of a long tail of contributors holds for internal contributors as well as external contributors.

VRP costs

Though the number of bounties suggests that VRPs provide a number of benefits, a thorough analysis necessarily includes an analysis of the costs of these programs. In this section, we examine whether VRPs provide a *cost-effective* mechanism for software vendors. We analyze one ongoing cost of the VRP: the amount of money paid to researchers as rewards for responsible disclosure. Running a VRP has additional overhead costs that we do not measure.

Cost of rewards

Figure 5.4 displays the total cost of paying out rewards for vulnerabilities affecting stable releases. We find that over the course of three years, the costs for Chrome and Firefox are similar: approximately $400,000.

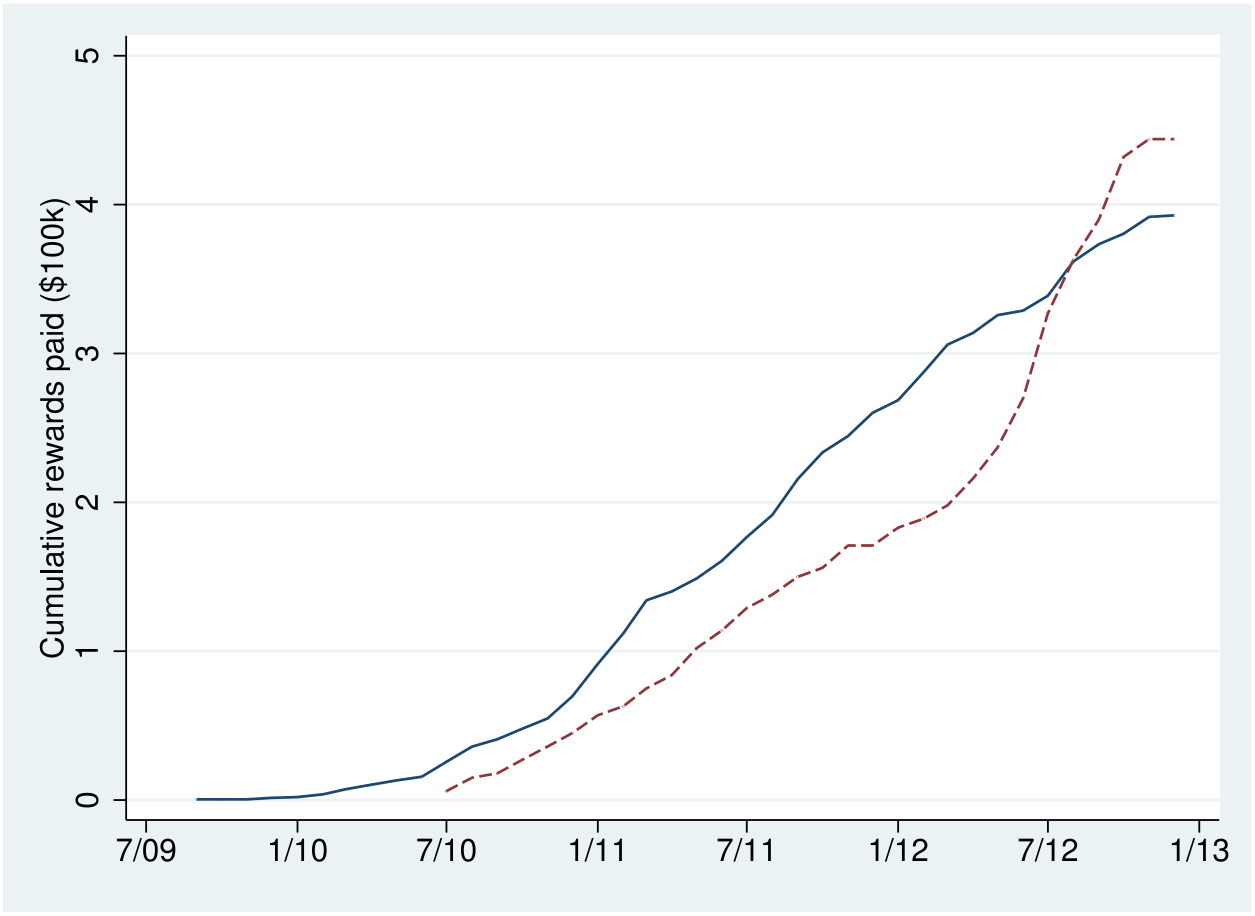
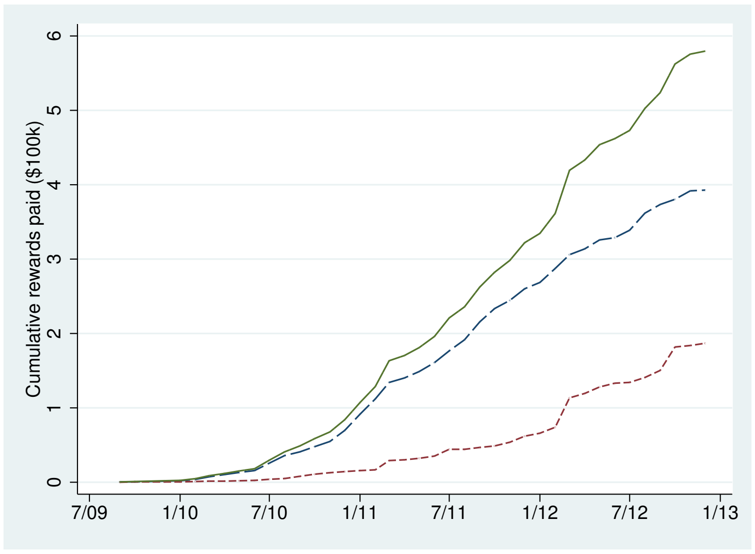


Figure 5.4: Cumulative rewards paid out for Chrome (blue solid line) and Firefox (red dashed line), excluding rewards for vulnerabilities not affecting stable versions.

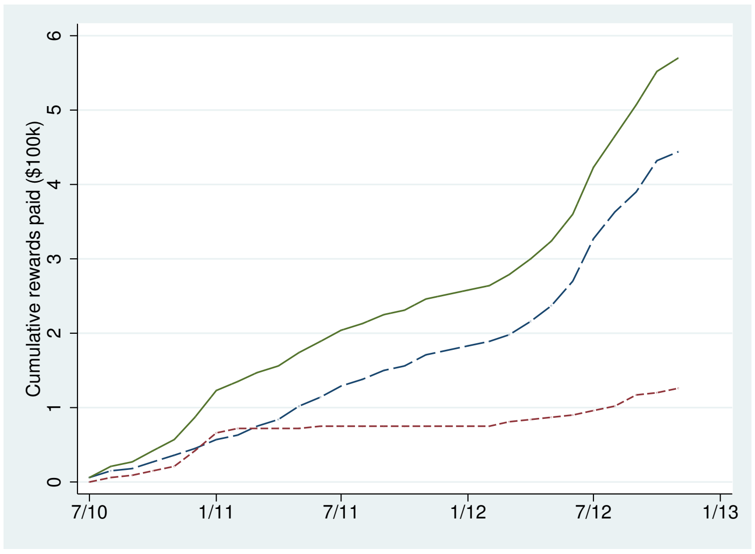
Rewards for Development Releases. Both Firefox and Chrome issue rewards for vulnerabilities that do not affect stable release versions, increasing the *total* cost of the VRP beyond the cost of rewarding vulnerabilities affecting stable releases. One potential drawback of such rewards is that the VRPs awards transient bugs that may never make their way into a user-facing build in the first place. On the other hand, such rewards could catch bugs earlier in the development cycle, reducing the likelihood of expensive out-of-cycle releases.[7]

Figure 5.5 shows the cumulative rewards issued by each of the two VRPs for vulnerabilities affecting stable releases, vulnerabilities not affecting stable releases, and the sum of the two. We observe that the Chrome VRP has paid out $186,839, 32% of its total cost of $579,605 over the study period for vulnerabilities not affecting a stable release. The Firefox VRP has paid out $126,000, 22% of its total cost of $570,000, over the study period for such vulnerabilities.

[7]An out-of-cycle release typically only occurs for vulnerabilities being actively exploited in the wild.



(a) Chrome



(b) Firefox

Figure 5.5: Cumulative rewards paid out for vulnerabilities affecting a stable release (blue long-dashed line), vulnerabilities not affecting any stable release (red short-dashed line), and the sum of the two (green solid line).

Discussion. The total cost of each of the two VRPs is remarkably similar. Both spend a significant fraction of the total cost on vulnerabilities not present in stable release versions.

Average daily cost

Figure 5.6 plots the average daily cost to date of each VRP over time. We see that Chrome's VRP has cost $485 per day on average, and that of Firefox has cost $658 per day.

Discussion. If we consider that an average North American developer on a browser security team (i.e., that of Chrome or Firefox) would cost the vendor around $500 per day (assuming a $100,000 salary with a 50% overhead), we see that the cost of either of these VRPs is comparable to the cost of just one member of the browser security team. On the other hand, the *benefit* of a VRP far outweighs that of a single security researcher because each of these VRPs finds many more vulnerabilities than any one researcher is likely to be able to find. The Chrome VRP has paid 501 bounties, and the most prolific internal security researcher has found 263 vulnerabilities. For Firefox, these numbers are 190 and 48, respectively. Based on this simple cost/benefit analysis, it appears that both of these VRPs have been cost-effective mechanisms for finding security vulnerabilities.

There is the potential added benefit that the wide variety of external participants may find *different* types of vulnerabilities than internal members of the security team. A few pieces of anecdotal evidence support this. Chrome awards bounty amounts of $1,337, $2,337, $3,133.7, and $7,331 for bugs that they consider clever or novel [55], and our data set contains 31 such awards. Additionally, one of PinkiePie's Pwnium exploits led to a full review of the Chrome kernel file API, which resulted in the discovery of several additional vulnerabilities [55, 21]. The Chrome security team missed all these issues until PinkiePie discovered and exploited one such issue [18].

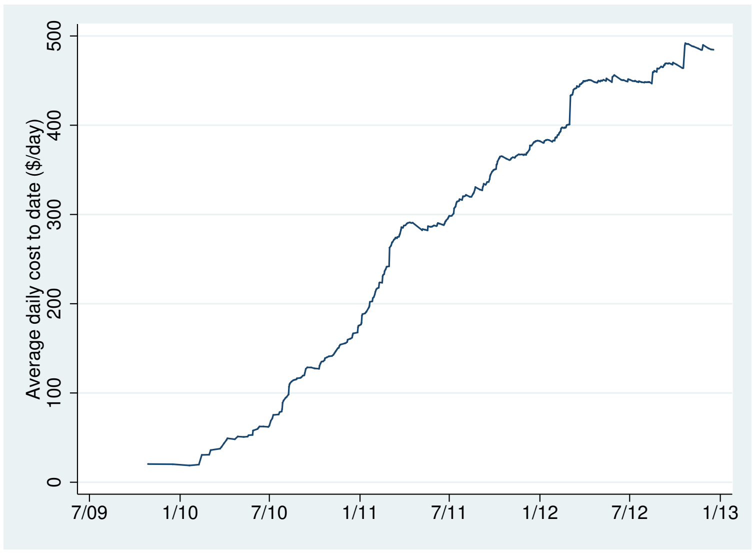
Participant incentives

Firefox offers a standard reward of $3,000 for all vulnerabilities. In contrast, the Chrome VRP's incentive structure provides different reward levels based on a number of subjective factors like difficulty of exploit, presence of a test case, novelty, and impact, all of which is at the discretion of Google developers.

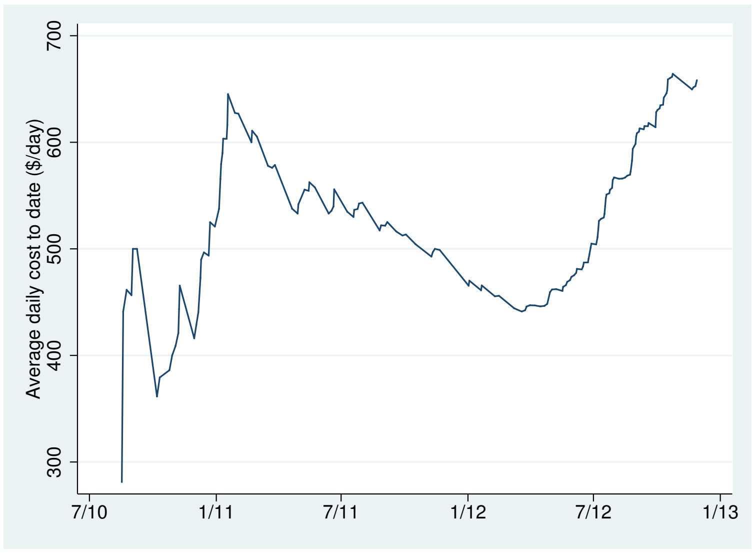
VRPs as employment

Table 5.4a displays the total amounts of money earned by each external contributor to the Chrome VRP. Only three external contributors (out of eighty two) have earned over $80,000 over the lifetime of the VRP, and an additional five have earned over $20,000.

In contrast to Google Chrome, we see in Table 5.4b that a single Firefox contributor has earned $141,000 since the beginning of our study period. Ten of this individual's rewards, representing $30,000, were for vulnerabilities that did not impact a stable release. Six contributors have earned more than $20,000 via the Mozilla VRP.



(a) Chrome



(b) Firefox

Figure 5.6: Average daily cost to date since first reward.

Discussion. Contributing to a single VRP is, in general, not a viable full-time job, though contributing to multiple VRPs may be for unusually successful vulnerability researchers. Anecdotal evidence also suggests a similar conclusion: Google hired two researchers who first came to light via the Chrome VRP [55] and Mozilla hired at least three researchers [111].

$ earned	Freq.
500	26
1,000	25
1,337	6
1,500	2
2,000	1
3,000	2
3,133	1
3,500	2
4,000	1
5,000	1
7,500	1
11,000	1
11,500	1
11,837	1
15,000	1
17,133	1
18,337	1
20,633	1
24,133	1
28,500	1
28,633	1
37,470	1
80,679	1
85,992	1
105,103	1
Total	82

(a) Chrome

$ earned	Freq.
3,000	46
6,000	12
9,000	4
12,000	1
15,000	1
21,000	1
27,000	1
30,000	1
36,000	1
42,000	1
141,000	1
Total	70

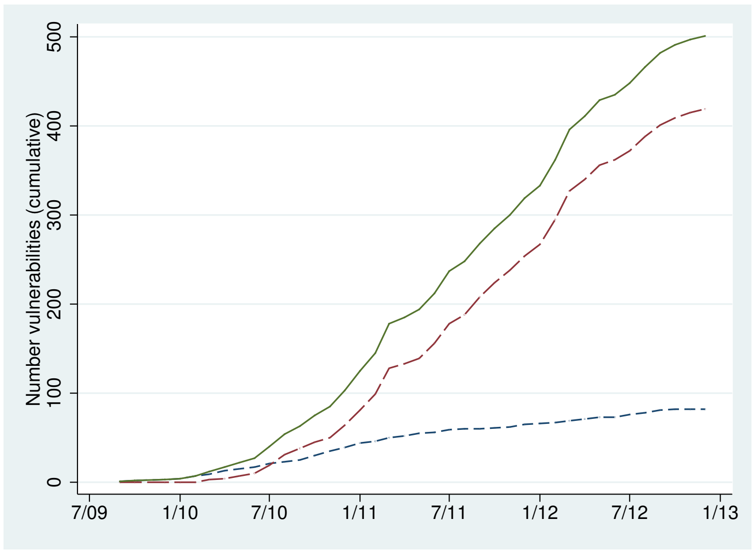
(b) Firefox

Table 5.4: Frequency distributions of total amounts earned by external VRP contributors.

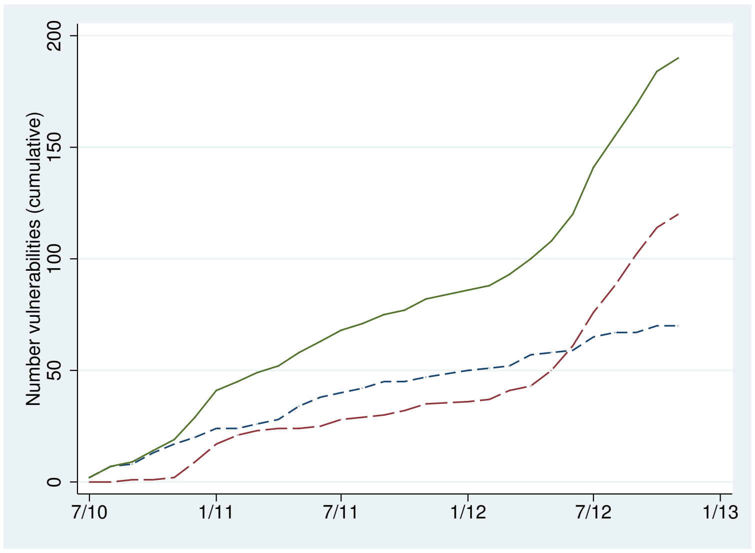
Repeat contributors

Figure 5.7 shows the cumulative number of vulnerabilities patched due to reports from first-time VRP participants and repeat participants. For both programs, first-time participant rewards are steadily increasing, and repeat participant rewards are increasing even more quickly.

Discussion. Both VRP incentive structures are evidently sufficient for both attracting new participants and continuing to entice existing participants, though we do note differences be-



(a) Chrome



(b) Firefox

Figure 5.7: Cumulative number of vulnerabilities rewarded, as reported by (1) first-time VRP contributors (blue short-dashed line), (2) repeat contributors (red long-dashed line), and (3) all contributors (green solid line).

tween Chrome and Firefox. Until recently, repeat participants in Firefox's VRP represented a relatively small fraction of the number of awards issued. Chrome, on the other hand, has seen the majority of its reports come from repeat participants for nearly the whole lifetime of its VRP.

Amount ($)	Frequency (%)
500	24.75
1,000	60.08
1,337	3.59
1,500	2.99
2,000	2.99
2,337	0.60
2,500	0.60
3,000	0.20
3,133	1.80
3,500	0.20
4,000	0.20
4,500	0.20
5,000	0.20
7,331	0.20
10,000	1.40

Table 5.5: Percentage of rewards given for each dollar amount in Chrome VRP.

Reward amounts

Table 5.5 depicts the reward amounts paid to external researchers by the Chrome VRP. The majority of the rewards are for only $500 or $1,000. Large rewards, such as $10,000 rewards, are infrequent.

Discussion. We hypothesize that high maximum rewards entice researchers to participate, but low ($500 or $1,000) rewards are typical, and the total cost remains low. The median (mean) payout for Chrome bug bounty is $1,000 ($1,156.9), suggesting that a successful VRP can be inexpensive with a low expected individual payout. Much like the lottery, a large maximum payout ($30,000 for Chrome), despite a small expected return (or even negative, as is the case of anyone who searches for bugs but never successfully finds any) appears to suffice in attracting enough participants. Bhattacharyya and Garrett [34] explain this phenomenon as follows:

> Lotteries are instruments with negative expected returns. So when people buy lottery tickets, they are trading off negative expected returns for skewness. Thus, if a lottery game has a larger prize amount, then a buyer will be willing to accept a lower chance of winning that prize.

Severity	Median, Chrome	Std. dev. Chrome	Median, Firefox	Std. dev., Firefox
Low	58.5	110.6	114	256.1
Medium	45.5	78.9	106	157.6
High	28.0	35.3	62.5	85.7
Critical	20.0	26.6	76	116.5

Table 5.6: Median and standard deviation of number of days between vulnerability report and release that patches the vulnerability, for each severity level.

Other factors

Our data set provides an additional opportunity to better understand the state of the SDLC (software development lifecycle) at Mozilla and Google. In particular, we analyze (1) the elapsed time between a vulnerability report and the release of a patched browser version that fixes the vulnerability, and (2) how often vulnerabilities are independently discovered, and what the potential implications are of this rediscovery rate.

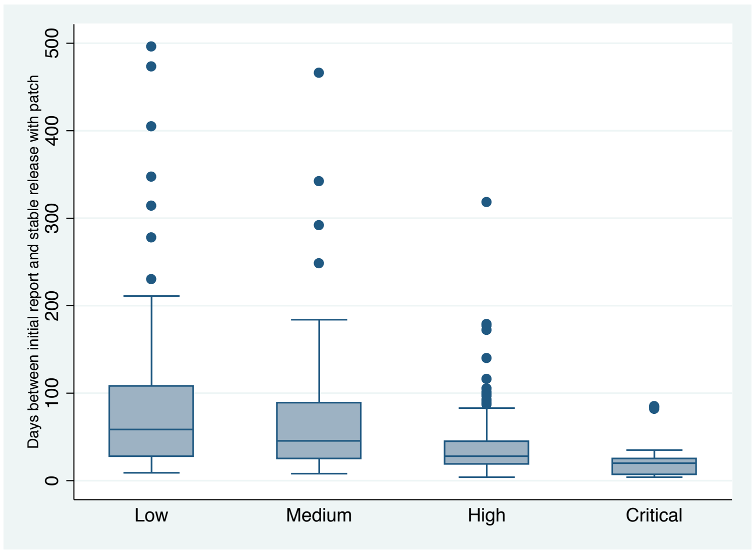
Time to patch

As previously discussed, we choose to study time to release a patched version, *not* time to commit a patch. Although relying on time to release a patch means we analyze only a subset of the data (Section 5.3), we believe the time to release a patched version of the browser is the more useful metric for end users. Mozilla Firefox and Google Chrome both follow a rapid-release cycle, with a new release every 6 or 7 weeks [2, 22]. In some cases, browser vendors release an out-of-band (or "chemspill") release for vulnerabilities with active exploits in the wild. Such out-of-band releases are one of the most expensive incidents for software companies, with costs running into millions of dollars [67]. Our metric awards the engineering and management commitment required in choosing to release such versions.

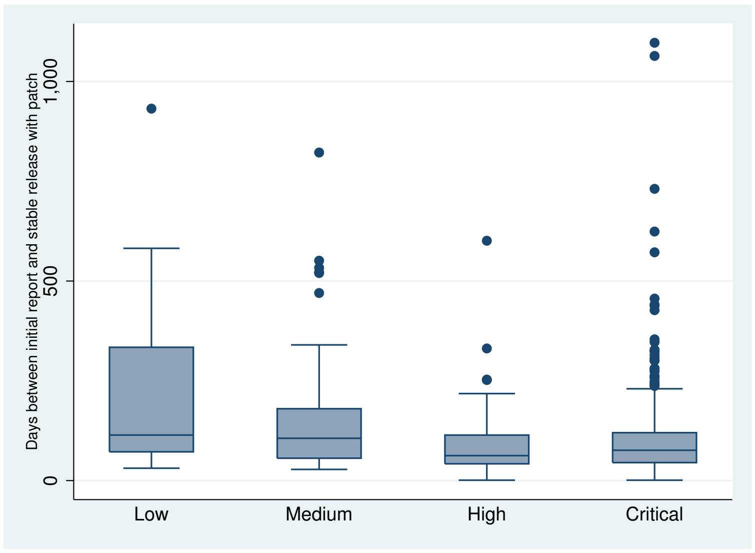
Figure 5.8 depicts the time between initial report of a vulnerability and the stable release that patches it. Table 5.6 gives summary statistics for these distributions.

Figure 5.9 is a scatter plot of the same data, which allows us to see changes in time to patch over time. Figure 5.10 shows the change in standard deviation of time to patch over time. More specifically, for a given release date, the y-value is the standard deviation for all bugs patched in that release or up to five prior releases. These graphs indicate that the standard deviation in time to patch critical vulnerabilities has slowly dropped for Firefox, while Chrome's time to patch critical vulnerabilities has remained relatively constant over time.

Discussion. For Chrome, both the median time to patch and the variance are lower for higher severity vulnerabilities. This is an important parameter for a VRP because responsible disclosure depends critically on vendor response time [16, 56]. If a vendor does not patch



(a) Chrome

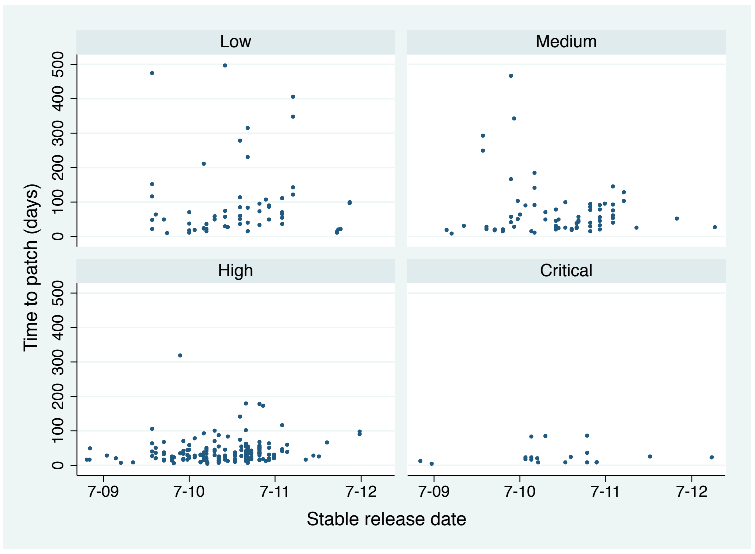


(b) Firefox

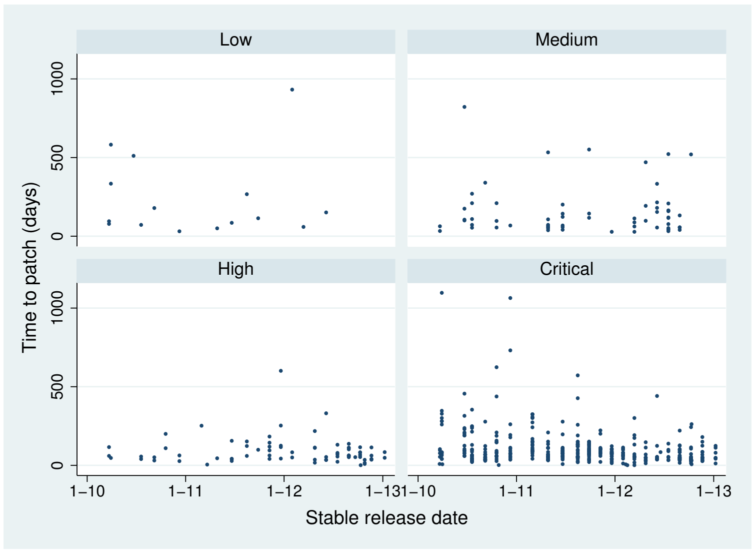
Figure 5.8: Box and whisker plots depicting the distributions of time between vulnerability report and release that patches the vulnerability, for each severity level.

in a reasonable time frame, security researchers are less likely to exercise responsible disclosure. Accordingly, this may be a contributing factor in Firefox's lower degree of community participation (as compared to Chrome), given that the time to patch critical vulnerabilities in Firefox is longer and has very high variance.

In Chrome, the time to patch is faster for critical vulnerabilities than it is for high severity



(a) Chrome

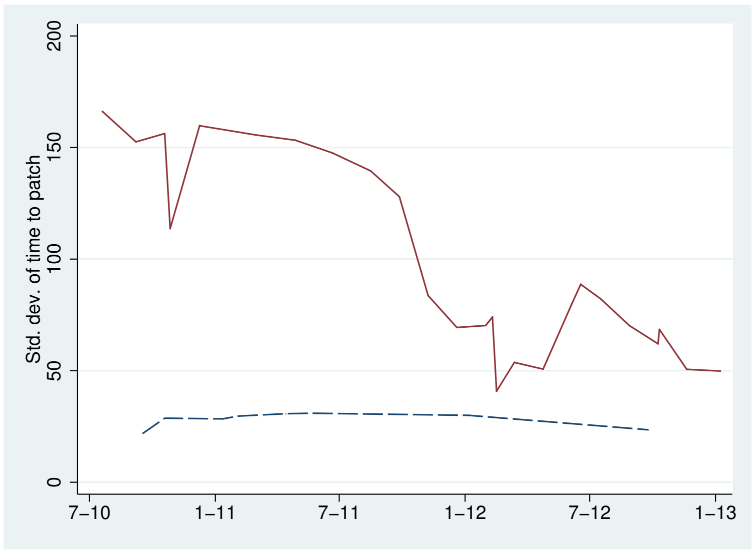


(b) Firefox

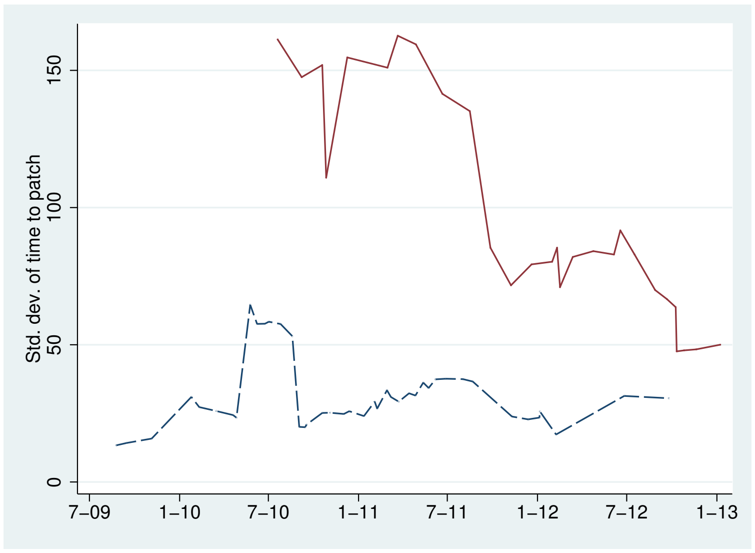
Figure 5.9: Scatter plots depicting the time between vulnerability report and release that patches the vulnerability vs. time, for each severity level.

vulnerabilities. This trend continues for medium- and low-severity vulnerabilities as well. This indicates correct prioritization of higher-severity vulnerabilities by Chrome security engineers. The same cannot be said for Firefox; high and critical severity vulnerabilities tend to take about the same amount of time to fix.

The high variance in Firefox's time to patch critical vulnerabilities may be partly at-



(a) Critical-severity vulnerabilities.



(b) Critical and high severity vulnerabilities.

Figure 5.10: Standard deviation of time to patch over time. For a given release date, the y-value is the standard deviation of all bugs patched in that release or up to five prior releases. The red solid line is for Firefox, and the blue dashed line is for Chrome.

tributable to the lack of privilege separation in Firefox, since a larger TCB for critical vulnerabilities means that there is a larger pool of engineers owning code that might hold a critical vulnerability. However, it is an encouraging sign that Firefox has gradually reduced this variance. Nonetheless, the variance in patch times and typical time to patch for Firefox both remain far higher than we see for Chrome, suggesting the need for a concerted effort at reducing this.

Independent discovery

Using the Chromium release blog, we manually coded an additional variable `independent`. This variable represents the number of times a vulnerability was independently discovered. We coded it using the text of the `credit` variable, which mentions "independent discovery" of a vulnerability in the case of multiple independent discoveries.

Our Chrome data set indicates when a vulnerability was independently discovered by multiple parties, identifies the parties, and in some cases, gives an upper bound on the time between discovery and rediscovery. Of the 668 vulnerabilities in our Chrome VRP data set, fifteen (2.25%) of them had at least two independent discoveries, and two of these had three independent discoveries. This is a lower bound on the number of independent discoveries of these vulnerabilities, since it represents only those known to the vendor.

Figure 5.11 displays the independent rediscovery rates for individuals. Each dot represents an individual contributor in our data set. Its x-value gives the number of vulnerabilities discovered by this individual, and its y-value gives the number of these vulnerabilities independently rediscovered by another contributor in our data set. Of those individuals who reported five or more vulnerabilities, the highest rediscovery rate is 25% and the mean is 4.6%.

Our Firefox data set does not indicate independent rediscovery, but we have limited data from personal communication with a Firefox security engineer [111]. He indicated that there had been at least 4-7 vulnerabilities reported through the VRP for which there had been two independent discoveries, a rate of 2.7% to 4.7%, which is consistent with what we see in our Chrome data set.

Discussion. Recent discoveries of sophisticated malware indicate that there are parties capable of finding and holding onto multiple zero-day vulnerabilities. Stuxnet used four zero-days, while Flame and Duqu each used one [84]; if the same actor developed all of these, as reported in the media [96], this indicates that a stash of six zero-days is not unheard of and the existence of even larger vulnerability stashes is not outside the realm of possibility.

A zero-day loses its value when the vendor becomes aware of it, which happens via independent discovery of the vulnerability. Thus, a stash of zero-days will decay at some rate. We use the data available to us to better understand how a vulnerability stash could decay. If we assume these rediscovery rates are similar to those of black-hats, then we can infer that a black hat's vulnerability stash decays slowly.

The time between rediscovery in the Chrome data set (upper bounds) range from 9 to 115 days, with a median of 26 days (but we do caution that $n = 9$). This suggests that after a vulnerability has been discovered, stashed, and not rediscovered after a month or two; a black hat can rest fairly easy in knowing that the vulnerability will remain viable for a long time.

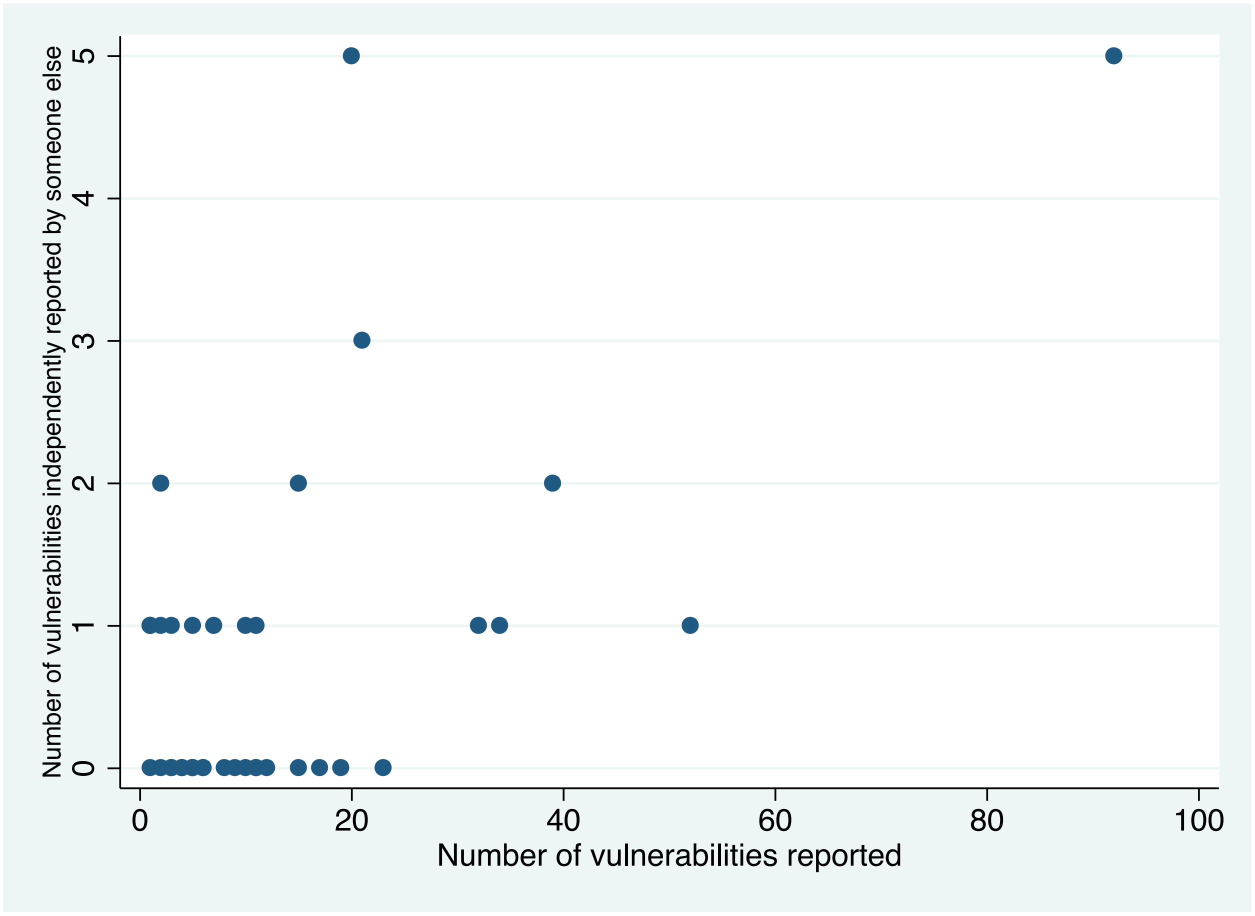


Figure 5.11: Independent vulnerability discoveries within the Chrome VRP data set. Each dot represents an individual contributor in our data set. Its x-value gives the number of vulnerabilities contributed by this individual, and its y-value gives the number of these contributions that were independently discovered by another contributor in our data set.

5.5 Discussion and recommendations

In this section, we synthesize what we have learned and present concrete recommendations for software vendors based on our data analysis.

Mozilla Firefox vs. Google Chrome

Despite costing approximately the same as the Mozilla program, the Chrome VRP has identified more than three times as many bugs, is more popular and shows similar participation from repeat and first-time participants. There is a stark difference between the levels of external participation in the two VRPs (Figure 5.2).

Despite having the oldest bounty program, external contributions lag far behind internal contributions to Firefox's security advisories. In contrast, external contributions to Chrome's security advisories closely rival internal contributions. Investigating further, we find three key differences between the two programs:

Tiered structure with large special rewards. Mozilla's program has a fixed payout of $3,000, which is approximately equal to the normal maximum payout for Chrome ($3,1337). Nonetheless, Chrome's tiered structure, with even higher payouts (e.g., $10,000) possible for clever bugs and special cases appears to be far more effective in encouraging participation. This makes sense with an understanding of incentives in lotteries: the larger the potential prize amount, the more willing participants are to accept a lower expected return, which, for VRPs, means the program can expect more participants [34].

Time to patch. We see a far higher variance in the time-to-release-patch metric for critical vulnerabilities in Mozilla Firefox. It is generally accepted that the viability of responsible disclosure depends on a reasonable vendor response time [16]. Thus, the high variance in Mozilla's response time could affect responsible disclosure through the VRP.

Higher profile. Chrome's VRP has a higher profile, with annual competitions like Pwnium providing particularly high rewards (up to $60,000). Chrome authors also provide extra reward top-ups for "interesting" bugs. We believe this sort of "gamification" leads to a higher profile for the Chrome VRP, which may help encourage participation, particularly from researchers interested in wider recognition.

Our methodology does not provide insight into the motivations of security researchers and the impact of VRP designs on the same—a topic we leave for future work. Nevertheless, we hypothesize that these three factors combined explain the disparity in participation between the Firefox and Chrome VRPs. Accordingly, we recommend Mozilla change their reward structure to a tiered system like that of Chrome. We urge Mozilla to do whatever it takes to continue to reduce the variance in time to release a patch for critical vulnerabilities, though we also realize the difficulty involved in doing so. Mozilla may consider holding its own annual competitions or otherwise increasing the PR surrounding its VRP.

Recommendations for vendors

Our study of the Chrome and Firefox VRPs yield a number of observations that we believe can guide vendors interested in launching or evolving their own VRPs.

We find that VRPs appear to provide an economically efficient mechanism for finding vulnerabilities, with a reasonable cost/benefit trade-off (Section 5.4). In particular, they appear to be 2-100 times more cost-effective than hiring expert security researchers to find vulnerabilities. We therefore recommend that more vendors consider using them to their (and their users') advantage. The cost/benefit trade-off may vary for other types of (i.e., non-browser) software vendors; in particular, the less costly a security incident is for a vendor, the less useful we can expect a VRP to be. Additionally, we expect that the higher-profile the software project is (among developers and security researchers), the more effective a VRP will be.

Response time, especially for critical vulnerabilities, is important. (Section 5.4). High variance in time-to-patch is not appreciated by the security community. It can reasonably be expected to reduce participation because it makes responsible disclosure through the VRP a less attractive option than the other options available to security researchers.

VRP incentive design is important and should be carefully considered. Chrome's tiered incentive structure appears more effective at encouraging community participation than Firefox's fixed-amount incentive structure (Section 5.4). Additionally, both Chrome and Firefox have increased their rewards over time. Doing so increases publicity, entices participants, and signals that a vendor is betting that their product has become more secure over time.

Our analysis demonstrates the impact of privilege separation on the Chrome VRP (Section 5.4). Privilege separation also provides flexibility to the Chrome team. For example, a simple way for Chrome to cut costs while still increasing participation could be to reduce reward amounts for high-severity vulnerabilities and increase reward amounts for critical-severity vulnerabilities. Mozilla does not have this flexibility. Vendors should consider using their security architecture to their advantage.

5.6 Conclusion and future work

We examined the characteristics of successful vulnerability rewards programs (VRPs) by studying two successful, well-known VRPs. Both programs appear economically efficient, comparing favorably to the cost of hiring full-time security researchers. The Chrome VRP features low expected payouts accompanied by high potential payouts, a strategy that appears to be effective in engaging a broad community of vulnerability researchers.

We hope that our study of these two VRPs serves as a valuable reference for software vendors aiming to evolve an existing VRP or start a new one. Potential future work on understanding VRPs includes economic modeling of VRPs; identifying typical patterns, trajectories, or phases in a VRP; and studying failed or unsuccessful VRPs to get a better sense of possible pitfalls in VRP development. Gathering and analyzing data from more VRPs will surely paint a more complete picture of their potential costs and benefits.

Acknowledgments

Thanks to my co-authors of this work, Devdatta Akhawe, and David Wagner, for working with me and giving their permission for this work to appear in this thesis.

Chapter 6

Conclusion

We have performed three empirical studies, each of which has enhanced our understanding of the effectiveness of some tool or technique intended to improve software security.

In Chapter 3, we analyzed a set of 9 implementations of the same web application to look for security differences associated with programming language, framework, and method of finding vulnerabilities. Our study suggested that the level of framework support available for avoiding certain classes of vulnerabilities does appear to have an impact on application security. We also found evidence that even the best manual framework support will continue to fall short, and therefore continued research effort should focus on automated framework defenses against security vulnerabilities. Manual code review found more vulnerabilities than black-box testing, but combining the two techniques was more effective than using either one by itself.

In Chapter 4, we designed and executed an empirical study to ask questions about the effectiveness of manual source code review for security. 30 subjects analyzed a web application with known vulnerabilities and delivered vulnerability reports to us, along with a post-completion survey regarding their level of experience and education. None of the factors on our survey were correlated with reviewer effectiveness on the code review. Overall effectiveness of the code reviewers was low: twenty percent found no vulnerabilities, and no reviewer found more than 5 of the 7 known vulnerabilities. There was significant variation amongst the reviewers, with 6 subjects delivering all or almost all false positives and 8 subjects delivering all or mostly all true positives. Our simulation based on our data set revealed that a random set of 10 reviewers had approximately an 80% chance of finding all vulnerabilities, 15 reviewers had approximately a 95% chance of finding all vulnerabilities, and hiring more than 20 reviewers would almost certainly be a waste of money.

In Chapter 5, we gathered and analyzed data from two well-known vulnerability rewards programs: those of Google Chrome and Mozilla Firefox. Our analysis revealed that both programs appear economically efficient, comparing favorably to the cost of hiring full-time security researchers. Both programs have successfully encouraged broad community participation, with both a steady stream of first-time contributors and repeat contributors. Chrome's program has uncovered more than three times as many vulnerabilities as that of

Firefox despite having a remarkably similar total cost. Chrome's average time to patch critical vulnerabilities (as well as the variance) is lower than that of Firefox. We hypothesize that Chrome has seen more benefit from its VRP as compared to that of Firefox due to these differences in time-to-patch and rewards structure.

In addition to achieving our broad goal of demonstrating that empirical techniques are useful toward improving our understanding of various tools, we have:

- Developed methodologies for testing the security implications of various software development and vulnerability remediation techniques. We developed three different experimental methodologies over the course of our three studies, each of which can be re-used or extended for future studies.

- Supplemented opinions, recommendations, anecdotes, and "best practices" with empirical evidence that some technique tends to yield more secure software than some other technique.

- Improved the state of software security by providing actionable recommendations regarding which tools or techniques to choose. Our three studies can serve as references for software developers or software vendors choosing which tools to use to prevent or find vulnerabilities in their software.

Appendix A

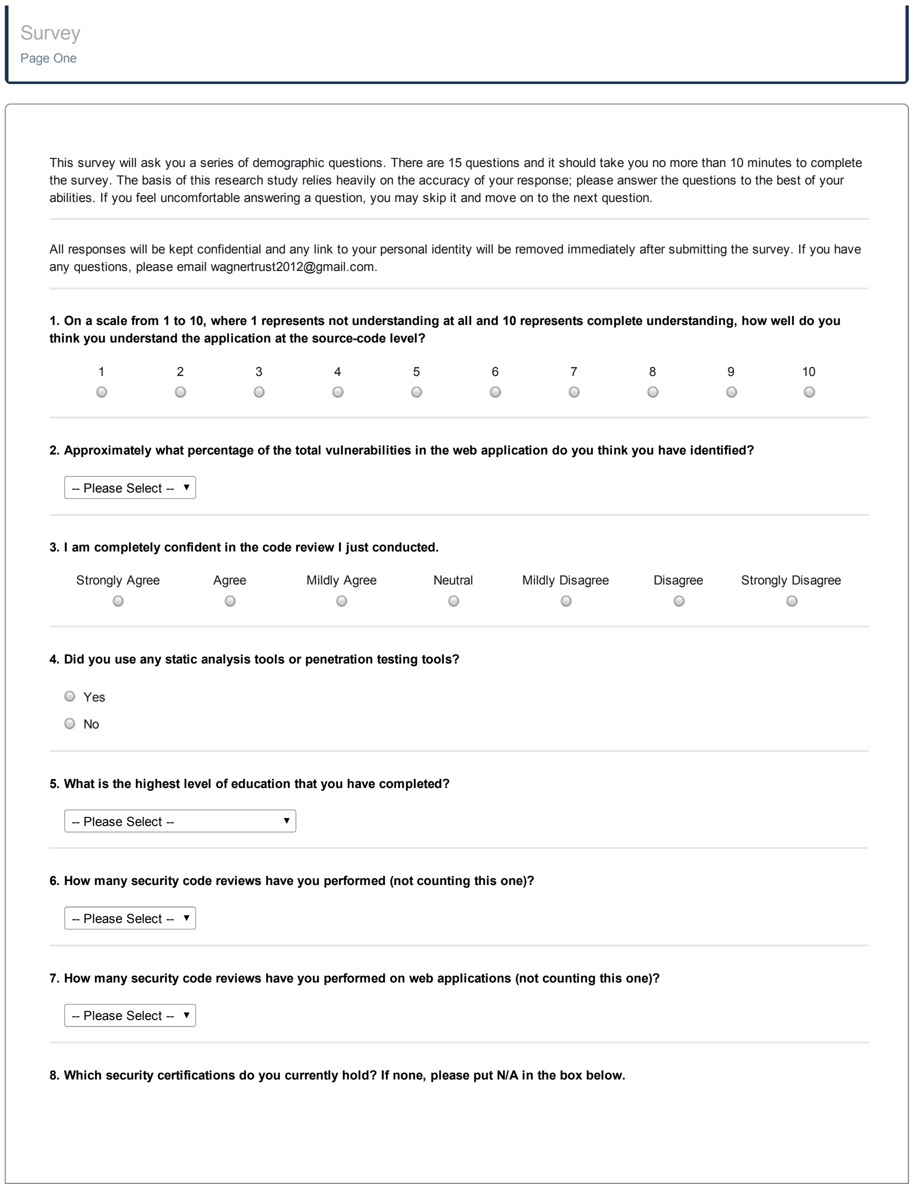
Demographic and Other Factors in Code Review Study

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

Appendix B

Post-Review Questionnaire Administered in Code Review Study



Survey
Page One

This survey will ask you a series of demographic questions. There are 15 questions and it should take you no more than 10 minutes to complete the survey. The basis of this research study relies heavily on the accuracy of your response; please answer the questions to the best of your abilities. If you feel uncomfortable answering a question, you may skip it and move on to the next question.

All responses will be kept confidential and any link to your personal identity will be removed immediately after submitting the survey. If you have any questions, please email wagnertrust2012@gmail.com.

1. On a scale from 1 to 10, where 1 represents not understanding at all and 10 represents complete understanding, how well do you think you understand the application at the source-code level?

1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○

2. Approximately what percentage of the total vulnerabilities in the web application do you think you have identified?

-- Please Select --

3. I am completely confident in the code review I just conducted.

Strongly Agree	Agree	Mildly Agree	Neutral	Mildly Disagree	Disagree	Strongly Disagree
○	○	○	○	○	○	○

4. Did you use any static analysis tools or penetration testing tools?

○ Yes
○ No

5. What is the highest level of education that you have completed?

-- Please Select --

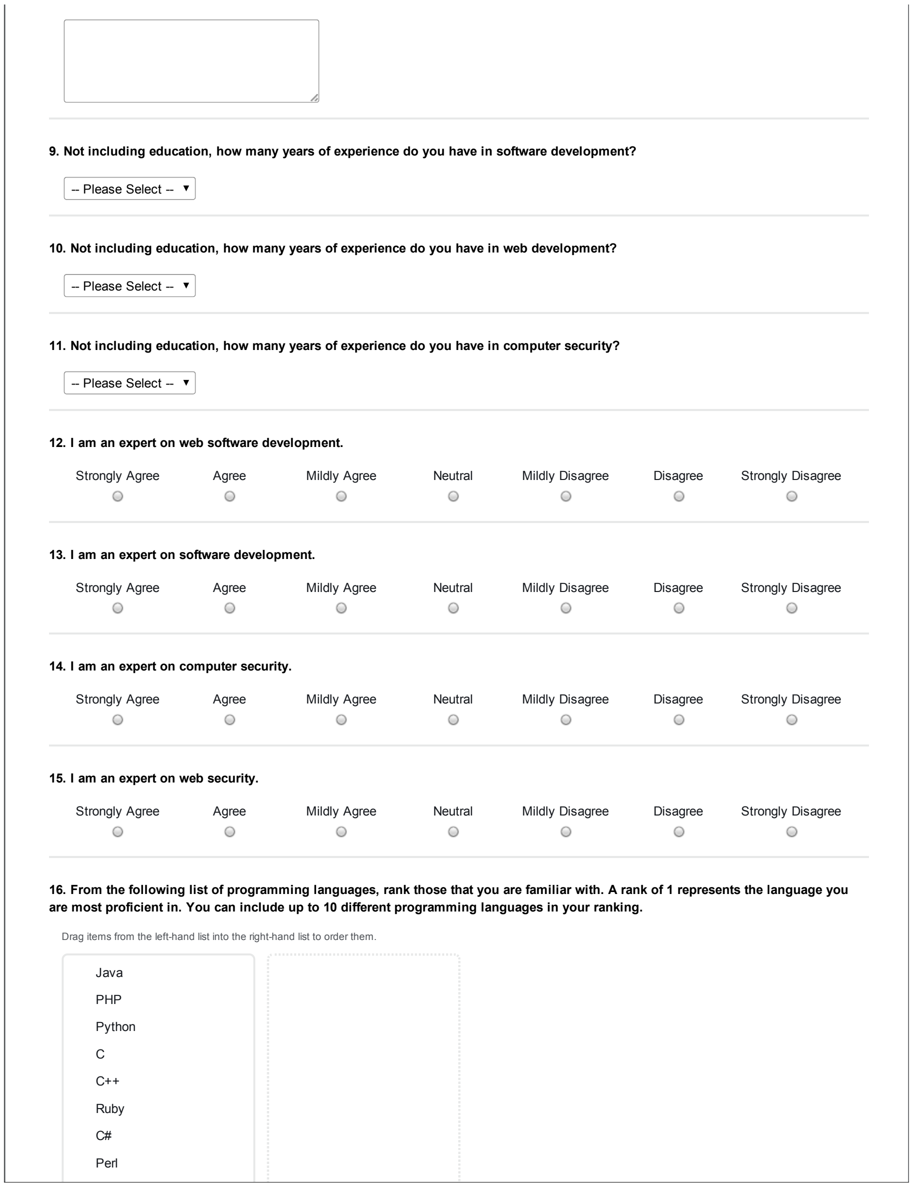
6. How many security code reviews have you performed (not counting this one)?

-- Please Select --

7. How many security code reviews have you performed on web applications (not counting this one)?

-- Please Select --

8. Which security certifications do you currently hold? If none, please put N/A in the box below.



9. Not including education, how many years of experience do you have in software development?

-- Please Select --

10. Not including education, how many years of experience do you have in web development?

-- Please Select --

11. Not including education, how many years of experience do you have in computer security?

-- Please Select --

12. I am an expert on web software development.

Strongly Agree	Agree	Mildly Agree	Neutral	Mildly Disagree	Disagree	Strongly Disagree
○	○	○	○	○	○	○

13. I am an expert on software development.

Strongly Agree	Agree	Mildly Agree	Neutral	Mildly Disagree	Disagree	Strongly Disagree
○	○	○	○	○	○	○

14. I am an expert on computer security.

Strongly Agree	Agree	Mildly Agree	Neutral	Mildly Disagree	Disagree	Strongly Disagree
○	○	○	○	○	○	○

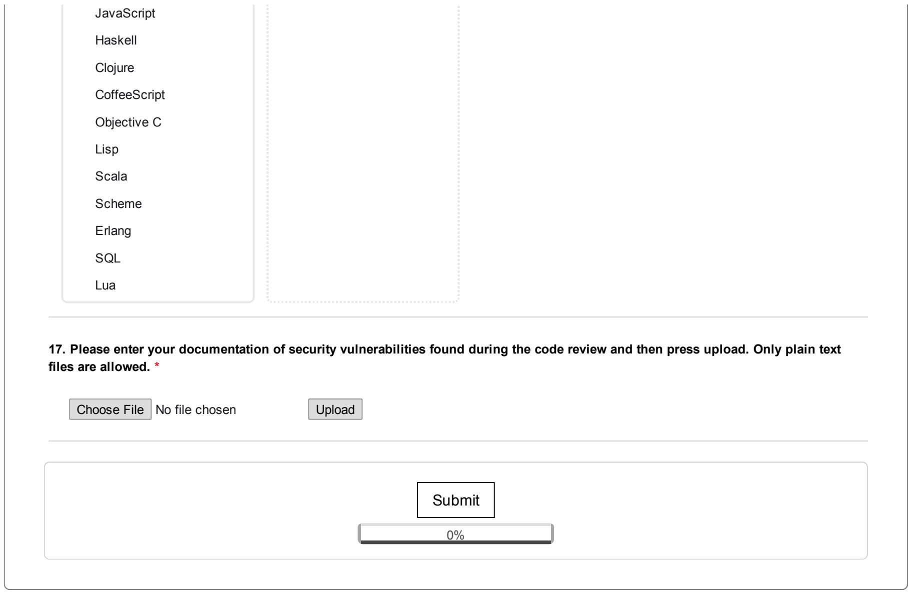
15. I am an expert on web security.

Strongly Agree	Agree	Mildly Agree	Neutral	Mildly Disagree	Disagree	Strongly Disagree
○	○	○	○	○	○	○

16. From the following list of programming languages, rank those that you are familiar with. A rank of 1 represents the language you are most proficient in. You can include up to 10 different programming languages in your ranking.

Drag items from the left-hand list into the right-hand list to order them.

Java
PHP
Python
C
C++
Ruby
C#
Perl


JavaScript
Haskell
Clojure
CoffeeScript
Objective C
Lisp
Scala
Scheme
Erlang
SQL
Lua

17. Please enter your documentation of security vulnerabilities found during the code review and then press upload. Only plain text files are allowed. *

Choose File No file chosen Upload

Submit

0%

Bibliography

[1] Chrome Releases: Stable Updates. http://googlechromereleases.blogspot.com/search/label/Stable%20updates.

[2] Chromium Development Calendar and Release Info. http://www.chromium.org/developers/calendar.

[3] CodeIgniter User Guide Version 1.7.3: Input Class. http://codeigniter.com/user_guide/libraries/input.html.

[4] Defense in Depth. http://www.nsa.gov/ia/_files/support/defenseindepth.pdf.

[5] django: Built-in template tags and filters. http://docs.djangoproject.com/en/dev/ref/templates/builtins.

[6] git. http://git-scm.com/.

[7] google-ctemplate. http://code.google.com/p/google-ctemplate/.

[8] National Vulnerability Database. http://nvd.nist.gov/.

[9] perl.org glossary. http://faq.perl.org/glossary.html#TMTOWTDI.

[10] Secunia Vulnerability Coordination Reward Program (SVCRP). https://secunia.com/community/research/svcrp/.

[11] Severity Guidelines for Security Issues. https://sites.google.com/a/chromium.org/dev/developers/severity-guidelines.

[12] Understanding Operational Security. http://www.cisco.com/web/about/security/intelligence/opsecurity.html.

[13] Creating a Computer Security Incident Response Team: A Process for Getting Started, February 2006. https://www.cert.org/csirts/Creating-A-CSIRT.html.

[14] Vulnerability Remediation, September 2010. https://www.cert.org/vuls/remediation.html.

[15] MozillaWiki: Electrolysis, April 2011. https://wiki.mozilla.org/Electrolysis.

[16] CERT/CC Vulnerability Disclosure Policy, November 2012. https://www.cert.org/kb/vul_disclosure.html.

[17] Developer's Guide – SafeHtml, October 2012. https://developers.google.com/web-toolkit/doc/latest/DevGuideSecuritySafeHtml.

[18] Security: Pwnium 2 tcmalloc profile bug, 2012. http://crbug.com/154983.

[19] Bugzilla, February 2013. http://www.bugzilla.org/.

[20] Chromium Bug Tracker, 2013. http://crbug.com.

[21] Chromium bug tracker: Sandbox bypasses found in review, 2013. http://goo.gl/13Z1R.

[22] MozillaWiki: RapidRelease/Calendar, January 2013. https://wiki.mozilla.org/RapidRelease/Calendar.

[23] oDesk at a Glance, April 2013. https://www.odesk.com/info/about/.

[24] The MEGA Vulnerability Reward Program, February 2013. https://mega.co.nz/#blog_6.

[25] Update google chrome, 2013. https://support.google.com/chrome/bin/answer.py?hl=en&answer=95414.

[26] Lucas Adamski. Refresh of the Mozilla Security Bug Bounty Program, July 2010. https://blog.mozilla.org/security/2010/07/15/refresh/.

[27] Özlem Albayrak and David Davenport. Impact of Maintainability Defects on Code Inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 50:1–50:4, 2010.

[28] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 97–106. IEEE, 2011.

[29] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg. Static code analysis to detect software security vulnerabilities—does experience matter? In *Availability, Reliability and Security, 2009. International Conference on*, pages 804–810. IEEE, 2009.

[30] Michael Barrett. PayPal "Bug Bounty" Program for Security Researchers, June 2012. https://www.thepaypalblog.com/2012/06/paypal-bug-bounty-program/.

[31] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, 2008.

[32] V.R. Basili and R.W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, Dec 1987.

[33] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, May 2010.

[34] Nalinaksha Bhattacharyya and Thomas A. Garrett. Why People Choose Negative Expected Return Assets - An Empirical Examination of a Utility Theoretic Explanation. *Federal Reserve Bank of St. Louis Working Paper Series*, March 2006. `http://research.stlouisfed.org/wp/2006/2006-014.pdf`.

[35] S. Biffl. 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)*, pages 136–145, 2000.

[36] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, Boston, MA, 2003.

[37] Eusebiu Blindu. Vulnerabilities reward programs, July 2012. `http://www.testalways.com/2012/07/13/vulnerabilities-reward-programs/`.

[38] Ken Buchanan, Chris Evans, Charlie Reis, and Tom Sepez. A Tale of Two Pwnies (Part 2), June 2012. `http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html`.

[39] Ken Buchanan, Chris Evans, Charlie Reis, and Tom Sepez. Show off Your Security Skills: Pwn2Own and Pwnium 3, January 2013. `http://blog.chromium.org/2013/01/show-off-your-security-skills-pwn2own.html`.

[40] Luca Carettoni. "No More Free Bugs" Initiative, October 2011. `http://blog.nibblesec.org/2011/10/no-more-free-bugs-initiatives.html`.

[41] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[42] Jason Cohen. *Best Kept Secrets of Peer Code Review*, page 117. Smart Bear, Inc., Austin, TX, 2006.

[43] Steve Coll. The rewards (and risks) of cyber war, June 2012. `http://www.newyorker.com/online/blogs/comment/2012/06/the-rewards-and-risks-of-cyberwar.html`.

[44] Coverity. Coverity security advisor, 2013. `http://www.coverity.com/products/security-advisor.html`.

[45] Mark Curphey and Rudolph Araujo. Web application security assessment tools. *IEEE Security and Privacy*, 4:32–41, 2006.

[46] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, pages 267–282. USENIX Association, 2009.

[47] Noopur Davis. Secure Software Development Life Cycle Processes, July 2012. `https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/sdlc/326-BSI.html`.

[48] Django Software Foundation. Django, 2013. `https://www.djangoproject.com/`.

[49] Keith Donald, Erwin Vervaet, and Ross Stoyanchev. Spring Web Flow: Reference Documentation, October 2007. `http://static.springsource.org/spring-webflow/docs/1.0.x/reference/index.html`.

[50] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, July 2010.

[51] Venkatreddy Dwarampudi, Shahbaz Singh Dhillon, Jivitesh Shah, Nikhil Joseph Sebastian, and Nitin Satyanarayan Kanigicharla. Comparative study of the Pros and Cons of Programming languages: Java, Scala, C++, Haskell, VB.NET, AspectJ, Perl, Ruby, PHP & Scheme. `http://arxiv.org/pdf/1008.3431`.

[52] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An Empirical Study on the Effectiveness of Security Code Review. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, March 2013.

[53] Chris Evans. Celebrating Six Months of Chromium Security Rewards, July 2010. `http://blog.chromium.org/2010/07/celebrating-six-months-of-chromium.html`.

[54] Chris Evans. Bug bounties vs. black (& grey) markets, May 2011. `http://scarybeastsecurity.blogspot.com/2011/05/bug-bounties-vs-black-grey-markets.html`.

[55] Chris Evans. Personal Communication, March 2013.

[56] Chris Evans, Eric Grosse, Neel Mehta, Matt Moore, Tavis Ormandy, Julien Tinnes, Michal Zalewski, and Google Security Team. Rebooting Responsible Disclosure: a focus on protecting end users, July 2010. `http://googleonlinesecurity.blogspot.com/2010/07/rebooting-responsible-disclosure-focus.html`.

[57] Chris Evans and Justin Schuh. Pwnium: rewards for exploits, February 2012. `http://blog.chromium.org/2012/02/pwnium-rewards-for-exploits.html`.

[58] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[59] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, Washington, DC, August 2010.

[60] A.L. Ferreira, R.J. Machado, L. Costa, J.G. Silva, R.F. Batista, and M.C. Paulk. An Approach to Improving Software Inspections Performance. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–8, Sept 2010.

[61] Matthew Finifter, Devdatta Akhawe, and David Wagner. Chrome and Firefox VRP datasets, February 2013. `http://usenixsecurity13.weebly.com/`.

[62] Matthew Finifter and David Wagner. Exploring the Relationship Between Web Application Development Tools and Security. In *Proceedings of the 2nd USENIX Conference on Web Application Development*. USENIX, June 2011.

[63] Dennis Fisher. Microsoft Says No to Paying Bug Bounties, July 2010. `http://threatpost.com/microsoft-says-no-paying-bug-bounties-072210/`.

[64] Brian Gorenc. Pwn2Own 2013, January 2013. `http://dvlabs.tippingpoint.com/blog/2013/01/17/pwn2own-2013`.

[65] L. Hatton. Testing the Value of Checklists in Code Inspections. *IEEE Software*, 25(4):82–88, July–Aug 2008.

[66] Les Hatton. Predicting the Total Number of Faults Using Parallel Code Inspections. `http://www.leshatton.org/2005/05/total-number-of-faults-using-parallel-code-inspections/`, May 2005.

[67] Chris Hofmann. Personal Communication, March 2013.

[68] Christian Holler. Trying new code analysis techniques, January 2012. `https://blog.mozilla.org/decoder/2012/01/27/trying-new-code-analysis-techniques/`.

[69] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on the World Wide Web*, pages 40–52, 2004.

[70] Jaspal. The best web development frameworks, June 2010. `http://www.webdesignish.com/the-best-web-development-frameworks.html`.

[71] C. Jones. Software Defect-Removal Efficiency. *IEEE Computer*, 29(4):94–95, Apr 1996.

[72] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 263–268, May 2006.

[73] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on the World Wide Web*, pages 247–256, 2006.

[74] Sean Michael Kerner. HP Fortifies Security with Static and Dynamic Analysis, February 2010. `http://www.developer.com/security/article.php/3866371/HP-Fortifies-Security-with-Static-and-Dynamic-Analysis.htm`.

[75] A. Kieyzun, P.J. Guo, K. Jayaraman, and M.D. Ernst. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In *31st IEEE International Conference on Software Engineering*, pages 199–209, May 2009.

[76] Brian Krebs. Tagging and tracking espionage botnets, July 2012. `https://krebsonsecurity.com/2012/07/tagging-and-tracking-espionage-botnets/`.

[77] Brian Krebs. Crimeware author funds exploit buying spree, January 2013. `https://krebsonsecurity.com/2013/01/crimeware-author-funds-exploit-buying-spree/`.

[78] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. 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*, pages 3–12, 2008.

[79] P. McCarthy, A. Porter, H. Siy, and Jr. Votta, 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*, pages 100–111, Mar 1996.

[80] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2:81–85, 2004.

[81] Ian Melven. MozillaWiki: Features/Security/Low rights Firefox, August 2012. `https://wiki.mozilla.org/Features/Security/Low_rights_Firefox`.

[82] Andrew Meneely and Laurie Williams. Secure Open Source Collaboration: an Empirical Study of Linus' Law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462. ACM, 2009.

[83] Andrew Meneely and Laurie Williams. Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software Security. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 9. ACM, 2010.

[84] Bill Miller and Dale Rowe. A survey of scada and critical infrastructure incidents. In *Proceedings of the 1st Annual conference on Research in information technology*, RIIT '12, pages 51–56, New York, NY, USA, 2012. ACM.

[85] Charles Miller. The legitimate vulnerability market: the secretive world of 0-day exploit sales. In *WEIS*, 2007.

[86] Todd C. Miller and Theo de Raadt. strlcpy and strlcat - consistent, safe, string copy and concatenation. `http://www.courtesan.com/todd/papers/strlcpy.html`.

[87] Elinor Mills. Facebook launches bug bounty program, July 2011. `http://news.cnet.com/8301-27080\_3-20085163-245/facebook-launches-bug-bounty-program/`.

[88] Mozilla Foundation. Mozilla Foundation Security Advisories, January 2013. `https://www.mozilla.org/security/announce/`.

[89] Liam Murchu. Stuxnet using three additional zero-day vulnerabilities, September 2010. `http://www.symantec.com/connect/blogs/stuxnet-using-three-additional-zero-day-vulnerabilities`.

[90] Atif Mushtaq. Man in the browser, February 2010. `http://www.fireeye.com/blog/technical/malware-research/2010/02/man-in-the-browser.html`.

[91] Stephan Neuhaus and Bernhard Plattner. Software security economics: Theory, in practice. In *WEIS*, 2012.

[92] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[93] Jorge Lucangeli Obes and Justin Schuh. A Tale of Two Pwnies (Part 1), May 2012. `http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html`.

[94] OWASP Foundation. Code Review Metrics. `https://www.owasp.org/index.php/Code_Review_Metrics`, 2010.

[95] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age. In *In USENIX-SS06: Proceedings of the 15th conference on USENIX Security Symposium*. USENIX Association, 2006.

[96] Nicole Perlroth. Researchers Find Clues in Malware, May 2012. `http://www.nytimes.com/2012/05/31/technology/researchers-link-flame-virus-to-stuxnet-and-duqu.html`.

[97] Tim Peters. PEP 20 – The Zen of Python. `http://www.python.org/dev/peps/pep-0020/`.

[98] PortSwigger Ltd. Burp Suite Professional. `http://www.portswigger.net/burp/editions.html`.

[99] Lutz Prechelt. Plat_Forms 2007 task: PbT. Technical Report TR-B-07-10, Freie Universität Berlin, Institut für Informatik, Germany, January 2007.

[100] Lutz Prechelt. Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties. *IEEE Transactions on Software Engineering*, 99, 2010.

[101] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[102] Eric Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[103] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[104] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. *Financial Cryptography and Data Security*, pages 284–298, 2012.

[105] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[106] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50. ACM, 2008.

[107] Larry Suto. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, February 2010. `http://www.ntobjectives.com/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf`.

[108] The Chromium Authors. Vulnerability Rewards Program: Rewards FAQ, 2010. `http://goo.gl/m1MdV`.

[109] Florian Thiel. Personal Communication, November 2009.

[110] TopSite. 10 Best Outsourcing Websites. `http://www.topsite.com/best/outsourcing`.

[111] Dan Veditz. Personal Communication, February 2013.

[112] Stefan Wagner, Jan Jrjens, Claudia Koller, Peter Trischberger, and Technische Universitt Mnchen. Comparing bug finding tools with reviews and tests. In *Proceedings of the 17th International Conference on Testing of Communicating Systems (TestCom05), volume 3502 of LNCS*, pages 40–55. Springer, 2005.

[113] James Walden, Maureen Doyle, Robert Lenhof, and John Murray. Java vs. PHP: Security Implications of Language Choice for Web Applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, February 2010.

[114] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, June 2007.

[115] WhiteHat Security. WhiteHat Website Security Statistic Report: 9th Edition, May 2010. `http://www.whitehatsec.com/home/resource/stats.html`.

[116] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 421–428. IEEE, 2010.