Exploring the Relationship Between Web Application Development Tools and Security

Matthew Finifter
University of California, Berkeley
finifter@cs.berkeley.edu

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

Abstract

How should software engineers choose which tools to use to develop secure web applications? Different developers have different opinions regarding which language, framework, or vulnerability-finding tool tends to yield more secure software than another; some believe that there is no difference at all between such tools. This paper adds quantitative data to the discussion and debate.

We use manual source code review and an automated black-box penetration testing tool to find security vulnerabilities in 9 implementations of the same web application in 3 different programming languages. We explore the relationship between programming languages and number of vulnerabilities, and between framework support for security concerns and the number of vulnerabilities. We also compare the vulnerabilities found by manual source code review and automated black-box penetration testing.

Our findings are: (1) we do not find a relationship between choice of programming language and application security, (2) automatic framework protection mechanisms, such as for CSRF and session management, appear to be effective at precluding vulnerabilities, while manual protection mechanisms provide little value, and (3) manual source code review is more effective than automated black-box testing, but testing is complementary.

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.
- 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.

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 paper'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.

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 [21, 20]. 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") [4], but the Zen of Python [16] 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 if frameworks that are pro-active 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

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 1: Set of frameworks used by each team.

more comprehensive but more expensive. However, we are not aware of quantitative data to measure their relative effectiveness. 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 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 [19]. 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 [18]. 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 1 lists the 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 a lot of 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]

3.1 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. [18]

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.

[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 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.

Review Number	Dev. Team Number	Lang.	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: Time taken for manual source code reviews, and number of source lines of code for each implementation.

- **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 2. A vulnerability in any of these classes violates at least one of the application's security goals.

3.2 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 [17].

We used both methods because we want to find as many vulnerabilities as we can. 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.

3.2.1 Manual source code review

One reviewer (Finifter) reviewed all implementations. This reviewer is knowledgeable about security and had previously been involved 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.

The reviewer followed the Flaw Hypothesis Methodology [6] for conducting the source code reviews. He 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. He performed two phases of review, first looking for specific types of flaws from the list, then comprehensively reviewing the implementation. He 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 presents the order in which the reviewer reviewed the implementations as well as the amount of time spent on each review.

The reviewer spent as much time as he felt necessary to perform a complete review. As shown in Table 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 [7] 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 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.

3.2.2 Black-box testing

We used Portswigger's Burp Suite Professional version 1.3.08 [17] 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 [11] 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).

3.3 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 [10], 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.

3.4 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:

	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 4: Statistics of the programmers.

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.

3.5 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 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 [23]. 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.

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 3.4) 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., [14]) in order to make our search as thorough as reasonably possible.

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.

4.1 Total number of vulnerabilities

Figure 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.

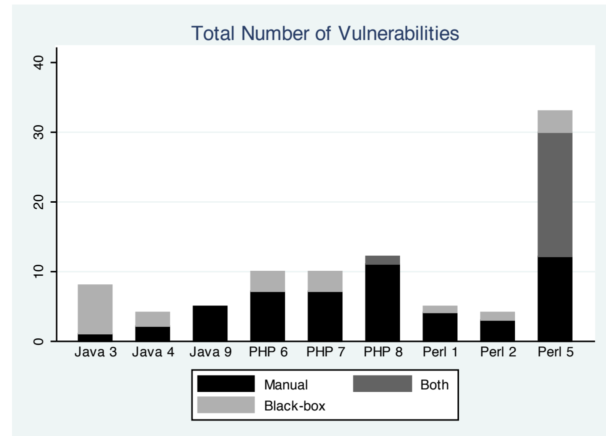


Figure 1: The total number of vulnerabilities found in the 9 implementations of People by Temperament. The x-axis is labeled with the language and team number.

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 ($F = 0.57$, $p = 0.592$). Because this appears to be largely due to the obvious lack of a difference between Perl and each of the other two languages, we also perform a Student's t-test for each pair of languages, using the Bonferroni correction to correct for the fact that we test 3 separate 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 ($p = 0.033$).

4.2 Vulnerability classes

Figure 2 breaks down the total number of vulnerabilities into the separate integer-valued vulnerability classes, and the shaded rows in Table 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.

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

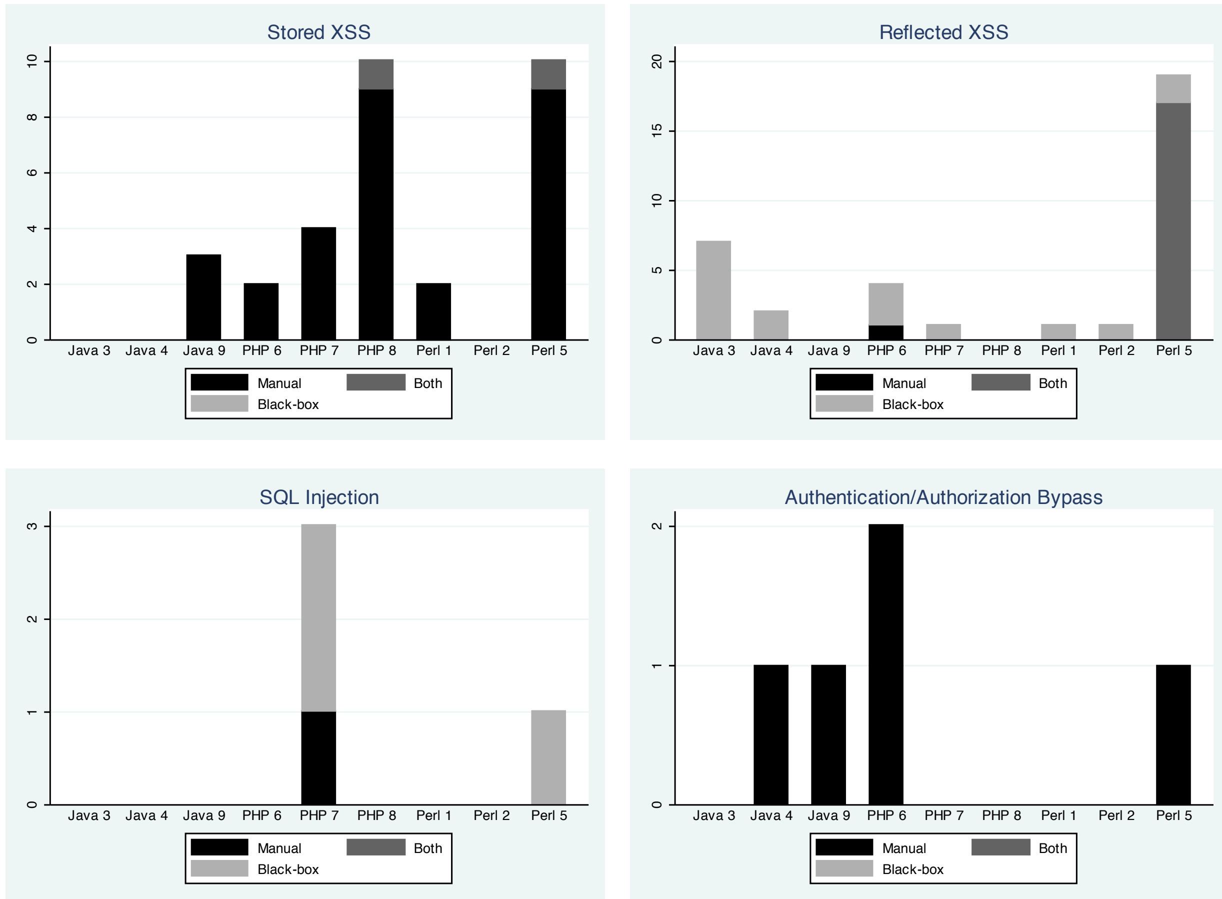


Figure 2: Vulnerabilities by vulnerability class.

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

Team Number	Language	Vulnerabilities found by: 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 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.

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 ($p = 0.999$).

4.3 Manual review vs. black-box testing

Table 6 and Figure 1 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). 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 vulnerabilities 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 2 reveals that the effectiveness of the two techniques differs depending upon the vulnerability class.

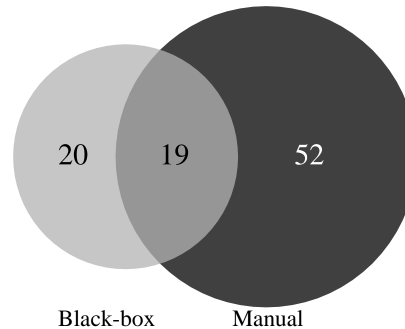


Figure 3: Vulnerabilities found by manual analysis and black-box penetration testing.

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 [9, 13], 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.

4.4 Framework support

We examine whether stronger framework support is associated with fewer vulnerabilities. Figure 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 vulnerabil-

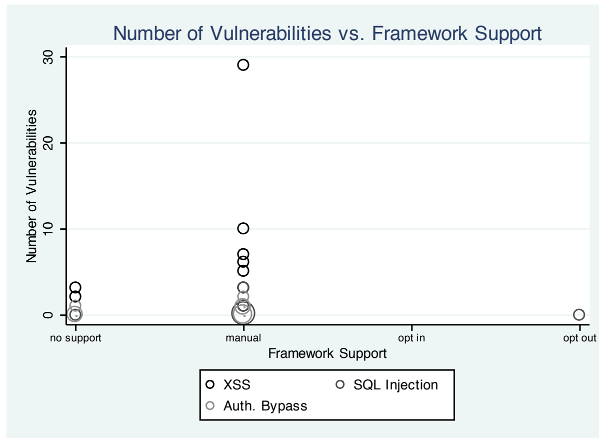


Figure 4: Level of framework support vs. number of vulnerabilities for integer-valued vulnerability classes. The area of a mark scales with the number of observations at its center.

ities, 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.

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` [1], Google Ctemplate [3], and Django's `autoescape` [2], 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 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 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.

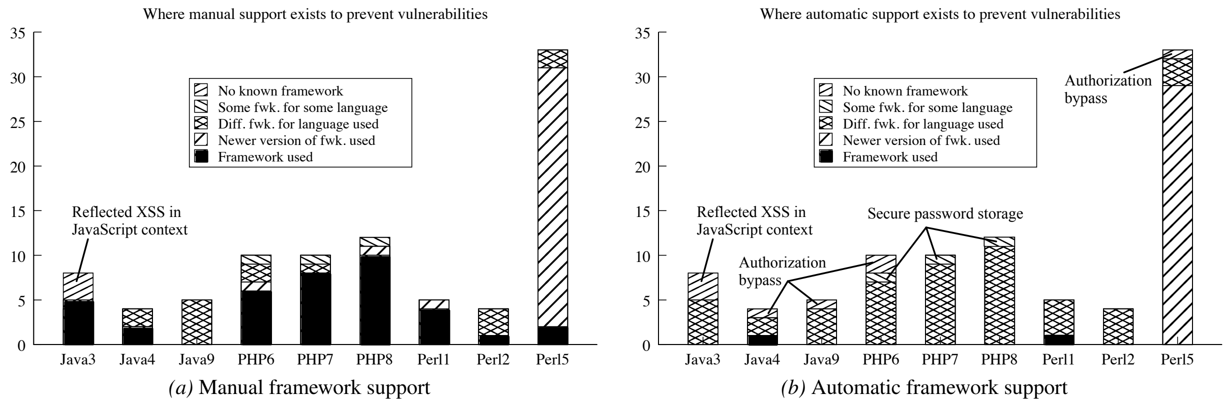
Figure 5 presents the results from our identification of the lowest level at which framework support exists that could have prevented each individual vulnerability (as described in Section 3.4).

It is rare for developers not to use available automatic support (the darkest bars in Figure 5b show only 2 such vulnerabilities), but they commonly fail to use existing manual support (the darkest bars in Figure 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; 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 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 [9, 13].

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



(a) Manual framework support

(b) Automatic framework support

Figure 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).

4.5 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 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.

5 Related work

In this section, we survey related work, which falls into 3 categories: (1) studies of the relationship between programming languages and application security, (2) comparisons of the effectiveness of different automated black-box web application penetration testing tools, and (3) comparisons of different bug- and vulnerability-finding techniques.

Programming languages and security. The 9th edition of the WhiteHat Website Security Statistic Report [26] offers what we believe is the best insight to date regarding 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 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. The report makes no mention of statistical significance, but given the size of their data set, one can expect all of their findings to be statistically significant (though not necessarily practically significant).

Walden et al. [25] 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, ours samples across 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. [12] 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.

Finally, the Plat_Forms [19] study (from which the present study acquired its data) performed a shallow security analysis of the 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. We greatly extended their study using both white- and black-box techniques to find vulnerabilities.

Automated black-box penetration testing. We are aware of three separate efforts to compare the effectiveness of different automated black-box web application security scanners. Suto [22] 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. [5] 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, but poorly on second-order vulnerabilities (e.g., stored XSS). 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.

Doupé et al. [11] 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, 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. Our choice regarding which black-box scanner to use was based in part on these studies.

Bug- and vulnerability-finding techniques. Wagner et al. [24] 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 focuses specifically on security vulnerabilities in web applications, and we use a different type of tool in our study than they use in theirs.

Two short articles [8, 15] 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.

6 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.

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 paper. 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. Matthew Finifter was also supported by a National Science Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

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

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

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

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

[5] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 332–345.

[6] BISHOP, M. *Computer Security: Art and Science*. Addison-Wesley Professional, Boston, MA, 2003.

[7] COHEN, J. *Best Kept Secrets of Peer Code Review*. Smart Bear, Inc., Austin, TX, 2006, p. 117.

[8] CURPHEY, M., AND ARAUJO, R. Web application security assessment tools. *IEEE Security and Privacy 4* (2006), 32–41.

[9] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium* (2009), USENIX Association, pp. 267–282.

[10] DONALD, K., VERVAET, E., AND STOYANCHEV, R. Spring Web Flow: Reference Documentation, October 2007. http://static.springsource.org/spring-webflow/docs/1.0.x/reference/index.html.

[11] DOUPÉ, A., COVA, M., AND VIGNA, G. 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).

[12] DWARAMPUDI, V., DHILLON, S. S., SHAH, J., SEBASTIAN, N. J., AND KANIGICHARLA, N. S. 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.

[13] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2010).

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

[15] MCGRAW, G., AND POTTER, B. Software security testing. *IEEE Security and Privacy 2* (2004), 81–85.

[16] PETERS, T. PEP 20 – The Zen of Python. http://www.python.org/dev/peps/pep-0020/.

[17] PORTSWIGGER LTD. Burp Suite Professional. http://www.portswigger.net/burp/editions.html.

[18] PRECHELT, L. Plat_Forms 2007 task: PbT. Tech. Rep. TR-B-07-10, Freie Universität Berlin, Institut für Informatik, Germany, January 2007.

[19] PRECHELT, L. Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties. *IEEE Transactions on Software Engineering 99* (2010).

[20] ROBERTSON, W., AND VIGNA, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium* (Montreal, Canada, August 2009).

[21] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium* (2001), pp. 201–220.

[22] SUTO, L. 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.

[23] THIEL, F. Personal Communication, November 2009.

[24] WAGNER, S., JRJENS, J., KOLLER, C., TRISCHBERGER, P., AND MNCHEN, T. U. Comparing bug finding tools with reviews and tests. In *In Proc. 17th International Conference on Testing of Communicating Systems (TestCom05), volume 3502 of LNCS* (2005), Springer, pp. 40–55.

[25] WALDEN, J., DOYLE, M., LENHOF, R., AND MURRAY, J. Java vs. PHP: Security Implications of Language Choice for Web Applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (February 2010).

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