An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages

Theodoor Scholte
SAP Research
Sophia Antipolis, France
theodoor.scholte@sap.com

Davide Balzarotti
Institute Eurecom
Sophia Antipolis, France
balzarotti@eurecom.fr

William Robertson
Northeastern University
Boston
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
Boston
ek@ccs.neu.edu

ABSTRACT

Web applications have become an integral part of the daily lives of millions of users. Unfortunately, web applications are also frequently targeted by attackers, and attacks such as XSS and SQL injection are still common. In this paper, we present an empirical study of more than 7000 input validation vulnerabilities with the aim of gaining deeper insights into how these common web vulnerabilities can be prevented. In particular, we focus on the relationship between the specific programming language used to develop web applications and the vulnerabilities that are commonly reported. Our findings suggest that most SQL injection and a significant number of XSS vulnerabilities can be prevented using straight-forward validation mechanisms based on common data types. We elaborate on these common data types, and discuss how support could be provided in web application frameworks.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures, product metrics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures, frameworks*

General Terms

Security vulnerabilities

Keywords

input validation, web application, programming language, security

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

1. INTRODUCTION

Web applications have become essential in our daily lives, and millions of users access these applications to communicate, socialize, and perform financial transactions. Unfortunately, due to their increased popularity as well as the high value data they expose, web applications have also become common targets for attackers.

In the past decade, much effort has been spent on making web applications more secure, and much of this work has focused on mitigating input validation vulnerabilities. The security research community has proposed numerous tools and techniques to detect and prevent such vulnerabilities, including static code analysis [8, 11, 27, 28, 31], dynamic tainting [18, 19, 20], prevention by construction or by design [7, 10, 21, 32], and client-side mechanisms executing within the browser [2, 6, 9, 26]. Some of these techniques have been integrated in developer toolsets (e.g., [13, 5]). Moreover, organizations such as OWASP [25] and the SANS Institute [22] have started to offer training programs that aim to educate application developers on how to detect, prevent, or avoid these classes of vulnerabilities.

Despite these efforts, several studies have shown that many web applications are still prone to these well-known and well-studied classes of vulnerabilities [12, 30]. In addition, Scholte et al. [23] have shown that the complexity of the attacks have not been increasing over time, and that most exploits are still simple in nature. Clearly, web developers often fail to apply existing countermeasures, and a new class of solutions is required to help improve the security situation on the web.

An important property of a programming language is the type system that is used. A type system classifies program statements and expressions according to the values they can compute, and it is useful for statically reasoning about possible program behaviors. Some popular web languages such as PHP and Perl are weakly-typed, meaning that the language implicitly converts values when operating with expressions of a different type.

The advantage of weakly-typed languages from a web developer's point of view is that they are often easy to learn and use. Furthermore, they allow developers to create applications quickly as they do not have to worry about declaring data types for the input parameters of a web application. Hence, most parameters are treated as generic "strings" even

though they might actually represent an integer value, a boolean, or a set of specific characters (e.g., an e-mail address). As a result, attacks are often possible if the validation is poor. For example, an attacker could inject scripting code (i.e., a string) into the value of a parameter that is normally used by the application to store an integer.

In order to gain deeper insights into the reasons behind common vulnerabilities in web applications, we analyzed around 3,933 cross-site scripting (XSS) and 3,758 SQL injection vulnerabilities affecting applications written in popular languages such as PHP, Python, ASP, and Java. For more than 800 of these vulnerabilities, we manually extracted and analyzed the code responsible for handling the input, and determined the type of the affected parameter (e.g., boolean, integer, or string). Furthermore, we studied 79 web application frameworks available for many popular programming languages. Our results suggest that most SQL injection and a significant number of XSS vulnerabilities can be prevented using straight-forward validation mechanisms based on common data types.

This paper makes the following contributions:

- We have analyzed in detail more than 3500 XSS and 3500 SQL injection vulnerability reports. We have also studied the validation functions provided by 79 web application frameworks. To our knowledge, this is the largest empirical study of its kind conducted to date.
- Our study suggests that many SQL injection and XSS vulnerabilities can be prevented if web languages and frameworks would support the enforcement of common data types such as integer, boolean, and specific types of strings such as an e-mail or a delivery address.

The rest of this paper is organized as follows. The next section describes our experimental methodology. Section 3 presents an analysis of the SQL injection and XSS vulnerabilities affecting popular web applications. In addition, we also present analysis results on the validation functionality typically provided by web application frameworks. In Section 4, we discuss our key findings and summarize the insights we distilled. We present related work in Section 5, and briefly conclude the paper in Section 6.

2. DATA COLLECTION AND METHODOLOGY

In order to study the characteristics of vulnerable web applications, it is necessary to have access to a significant amount of vulnerability data. Hence, we collected and classified a large number of vulnerability reports. These vulnerability reports were used to identify the programming language each web application was developed in. Furthermore, we used the vulnerability reports we gathered to semi-automatically extract vulnerable input parameters from the source code of web applications. Finally, by automatically collecting data from a number of open source project hosting services, we were able to estimate the popularity of web programming languages.

In the following, we discuss our methodology for collecting, classifying, and extracting information from vulnerability reports, project hosting services, and open source web applications.

2.1 Vulnerability Reports

2.1.1 Data Gathering

Our baseline source of vulnerability information is the publicly available data from the National Vulnerability Database (NVD) provided by NIST [17]. The Common Vulnerabilities and Exposures (CVE) initiative hosted by MITRE [15] supplies data for the NVD. Each CVE entry has a unique CVE identifier, a status ("entry" or "candidate"), and a general description. In addition to CVE data, the NVD database includes the following information that is relevant for our study:

- The vulnerability type according to the Common Weakness Enumeration (CWE) classification system [16].
- The name of the affected application, version numbers, and the vendor of the application represented by Common Platform Enumeration (CPE) identifiers [14].

For each candidate and accepted CVE entry, we extracted and stored the identifier, the description, and the CWE vulnerability classification.

2.1.2 Vulnerability Classification

Since our study focuses on particular classes of vulnerabilities, it is essential to classify the vulnerability reports. As mentioned in the previous section, CVE entries in the NVD database are classified according to the Common Weakness Enumeration classification system, which aims to be a comprehensive taxonomy of software weaknesses. NVD uses only a small subset of 19 CWEs for mapping CVEs to CWEs; among those are XSS (CWE-79) and SQL injection (CWE-89).

Although NVD provides a mapping between CVEs and CWEs, this mapping is unfortunately far from complete, and many CVE entries do not have any classification at all. For this reason, we chose to perform a classification based on both the CWE classification as well as the description of the CVE entry. In general, a CVE description is formatted according to the following pattern: {description of vulnerability} {affected application} *allows* {description of attacker} {impact and location description}. Thus, the CVE description includes the vulnerability type.

For fetching XSS–related CVEs from the total set of CVEs, we selected the CVEs associated with CWE identifier "CWE-79". Then, we added the CVEs having the text "cross-site scripting" in their description by performing a case-insensitive query. Similarly, we classified SQL injection–related CVEs by using the CWE identifier "CWE-89" or the "SQL injection" keyword.

2.1.3 Programming Language Classification

To study the relationship between programming language and vulnerable web applications, we automatically analyzed XSS and SQL injection–related CVEs. As mentioned in the previous section, many CVE entries contain a description of the location of a vulnerability in the web application. In general, vulnerability reports use fully–qualified filenames to identify the vulnerable script. We used the filename extension to classify whether an application is written in PHP

(.php), ASP/ASP.NET[1] (.asp, .aspx), ColdFusion (.cfm), Java (.jsp), Perl (.pl), and Python (.py).

The programming language of some applications could not be determined in an automated fashion as the corresponding vulnerability reports did not provide any information concerning the vulnerable scripts. In these cases, we manually determined the programming language by performing search queries and analyzing the source code of the web application.

2.2 Attack Vectors

We analyzed the source code of a significant number of vulnerable web applications with the aim of understanding to what extent data typing and validation mechanisms could help in preventing XSS and SQL injection vulnerabilities. In order to obtain a test set of applications with a high number of vulnerable input parameters, we chose to focus our study on 20 popular open source PHP web applications that contained the highest incidence of XSS vulnerabilities, and on 20 with the highest incidence of SQL injection vulnerabilities. The 28 applications belonging to the two, largely overlapping, sets are: claroline, coppermine, deluxebb, drupal, e107, horde, jetbox, joomla, mambo, mantis, mediawiki, moodle, mybb, mybloggie, papoo, phorum, phpbb, phpfusion, phpmyadmin, pligg, punbb, runcms, serendipity, squirrelmail, typo3, webspell, wordpress, and xoops.

For each of these applications, we manually examined the corresponding vulnerability reports to identify the specific application version and any example of attack inputs. Given this information, we downloaded the source code of each application and linked the input vectors to the application's source code to determine an appropriate data type. We repeated this process for a total of 809 vulnerability reports.

In the process of linking vulnerability reports to source code, we first used the version of the source code that was known to be vulnerable. Then, we repeated the process and linked the vulnerability reports to source code in which the vulnerabilities were patched. To determine the data type of the vulnerable input parameter, we manually analyzed how each vulnerability was patched and how the value of the input parameter was used throughout the web application.

3. ANALYSIS

In this section, we present the results of our empirical study, and draw conclusions from an analysis of the data. In particular, we first examine whether certain languages are more prone to XSS and SQL injection vulnerabilities. Then, we analyze the type of the input parameters that commonly serve as attack vectors, and we compare them with particular features provided by the web programming languages, or by the application frameworks available to them.

3.1 Language Popularity and Reported Vulnerabilities

A central question of this paper concerns whether the choice of programming language used to develop web applications influences the exposure of those applications to XSS and SQL injection vulnerabilities. To that end, we performed a comparison of the distribution of popular web programming languages to the distribution of reported XSS and SQL injection vulnerabilities for each of those languages.

[1]We chose to determine the platform instead of the language as we could not automatically identify whether an application was implemented in C# or Visual Basic.

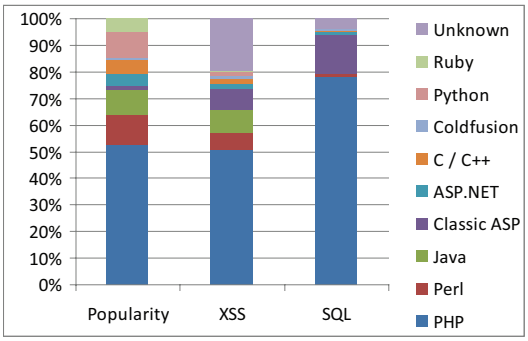


Figure 1: Distributions of popularity, reported XSS vulnerabilities, and reported SQL injection vulnerabilities for several web programming languages. The language popularity distribution was calculated by crawling open source project hosting services, while vulnerability data was drawn from the NVD [17].

Language popularity was calculated by crawling open source project hosting services such as Google Code, Sourceforge, and Freshmeat. For each of these services, we identified web application projects by filtering on project tags using values such as "cms", "dynamic content", and "message board". These projects were classified according to the primary development language by using each service's built-in search functionality.

The vulnerability data was drawn from the NVD [17] by automatically classifying reports according to the language of the affected application. For each report, our analysis checked whether the report concerned an XSS or SQL injection vulnerability. Our analysis identified 5,361 XSS and 4,773 SQL injection CVE entries out of a total of 39,081 entries. Because 142 of these entries did not correspond to any CPE values identifying vulnerable applications, these were excluded from that set. The resulting set of CVE entries used in our analysis was composed of 3,933 XSS and 3,758 SQL injection reports.

An automated classification of this set of vulnerability reports was able to classify 3,254 and 2,187 of the XSS and SQL injection CVE entries, respectively, as corresponding to a particular programming language. The remainder of the CVE entries were manually classified.

The distributions of language popularity and vulnerability reports are shown in Figure 1. The graph shows the statistics for 9 popular programming languages. Unfortunately, for 3.8% of SQL injection and 19% of XSS vulnerability reports, we were not able to automatically determine the primary development language of the application. These cases, represented by the "Unknown" category in Figure 1, are often related to commercial products for which the software companies do not provide information about the development language on their websites.

Under the null hypothesis — that is, that the choice of programming language does not influence the exposure of applications to XSS and SQL injection vulnerabilities — one would expect the relative distributions of popularity and reported vulnerabilities to be roughly equivalent. However, the histogram shows that this is not always the case.

The result for PHP-based applications is an illustrative

example. 52% of the applications in our test set were developed in PHP, a value that also corresponds to the share of reported XSS vulnerabilities in PHP applications. Therefore, it may seem that PHP is intrinsically no more or less vulnerable than other languages to this kind of attack. However, PHP-related vulnerabilities comprised almost 80% of the SQL injection reports from our collection of CVE entries. A similar, but opposite trend mismatch can be observed for many other languages. For example, although 10% of the applications in our dataset were written in Java, we found that only 0.5% of SQL injection vulnerabilities are associated with Java-based applications. Clearly, Java applications seem to be less prone to both XSS and SQL injection vulnerabilities. These differences are too common and too large to be considered statistically insignificant.

This is also shown by Pearson's chi-square tests which we used to assess the goodness of fit. We tested the hypothesis that the language popularity and the number of vulnerabilities per language is not significantly different. For XSS, we found chi-square 48.4 and for SQL injection we found chi-square 138. The degrees of freedom is 8. Looking these numbers up in the chi-square table shows that both probabilities are less than 0.001 meaning that the hypothesis is not true. Thus, the number of XSS respectively SQL injection vulnerabilities for a given language is not determined by the popularity of that language.

Note that there may be many possible reasons to explain the discrepancies. For example, one possible explanation might be that Java developers are simply more careful than those that favor other languages, and that PHP developers are instead worse, on average, at applying known defense techniques to prevent SQL injection. On the other hand, it might also be that certain languages, as well as the web development frameworks available for those languages, are intrinsically more resistant to — or provide better defenses against — XSS and SQL injection vulnerabilities.

Another possible reason could be that the web development frameworks available for a certain language provide a better set of functionality (or a better API for that functionality) to properly sanitize the user inputs, thus making the life easier for the web developers. In the rest of the section, we explore in more detail these possibilities by analyzing the impact of the language type system on the security of the application, and the functionality provided by common application frameworks that can be used to prevent XSS and SQL injection vulnerabilities.

3.2 Language Choice and Input Validation

As we saw from Figure 1, the choice of programming language clearly has an influence on the exposure of applications developed in those languages to XSS and SQL injection vulnerabilities. While there are several plausible explanations for this phenomenon, one likely hypothesis is that some programming languages are intrinsically more robust against the introduction of web application vulnerabilities. In the following, we examine a particular mechanism by which a language or framework might mitigate the potential for web application attacks.

3.2.1 Input Validation.

One defensive mechanism that is critical for the correct functioning of applications is *input validation*. In the abstract, input validation is the process of assigning semantic meaning to unstructured and untrusted inputs to an application, and ensuring that those inputs respect a set of constraints describing a well-formed input. For web applications, inputs take the form of key-value pairs of strings. The validation of these inputs may be performed either in the browser using Javascript, or on the server. Since there is currently no guarantee of the integrity of computation in the browser, security-relevant input validation should be performed on the server, and, therefore, we restrict our discussion of input validation to this context.

```
POST /payment/submit HTTP/1.1
Host:  example.com
Cookie:  SESSION=cbb8587c63971b8e
[...]

cc=1234567812345678&month=8&year=2012&
save=false&token=006bf047a6c97356
```

Figure 2: Example HTTP request.

To elucidate the input validation process for server-side web applications, consider the pedagogical HTTP request shown in Figure 2. This figure shows a typical structure for a payment submission request to a fictional e-commerce application. As part of this request, there are several input parameters that the controller logic for `/payment/submit` must handle: `cc`, a credit card number; `month`, a numeric month; `year`, a numeric year; `save`, a flag indicating whether the payment information should be saved for future use; `token`, an anti-CSRF token; and `SESSION`, a session identifier. Each of these request parameters requires a different type of input validation. For instance, the credit card number should be a certain number of characters and pass a Luhn check. The month parameter should be an integer value between 1 and 12 inclusive. The year parameter should also be an integer value, but can range from the current year to an arbitrary year in the near future. The save flag should be a boolean value, but as there are different representations of logical true and false (e.g., $\{\text{true}, \text{false}\}, \{1, 0\}, \{\text{yes}, \text{no}\}$), the application must consistently recognize a fixed set of possible values.

Input validation, in addition to its role in facilitating program correctness, is a helpful tool to prevent the introduction of vulnerabilities into web applications. Were an attacker to supply the value

```
year=2012'; INSERT INTO admins(user, passwd)
 VALUES('foo', 'bar');--
```

to our fictional e-commerce application as part of a SQL injection to escalate privileges, proper input validation would recognize that the malicious value was not a valid year, with the result that the application would refuse to service the request.

Input validation can occur in multiple ways. Validation can be performed implicitly — for instance, through typecasting a string to a primitive type like a boolean or integer. For the example attack shown above, a cast from the input string to an integer would result in a runtime cast error, since the malicious value is not a well-formed integer. On the other hand, input validation can be performed explicitly, by invoking framework-provided validation routines. Explicit validation is typically performed for input values exhibiting

complex structure, such as email addresses, URLs, or credit card numbers.

In this respect, the choice of programming language and framework for developing web applications plays an important role in the security of those applications. First, if a language features a strong type system such that typecasts of ill-formed input values to certain primitive types will result in runtime errors, the language can provide an implicit defense against the introduction of vulnerabilities like XSS and SQL injection. Second, if a language framework provides a comprehensive set of input validation routines for complex data such as email addresses or credit card numbers, the invocation of these routines can further improve the resilience of a web application to the introduction of common vulnerabilities.

3.3 Typecasting As an Implicit Defense

To quantify the extent to which typecasting of input values to primitive types might serve as a layer of defense against XSS and SQL injection vulnerabilities, we performed an analysis over vulnerability reports for a test set of web applications. Specifically, we examined the source code of these applications to determine the whether the vulnerable input has a primitive type. Where we could not directly identify the type, we looked at the modifications made to the source code to resolve the vulnerability.

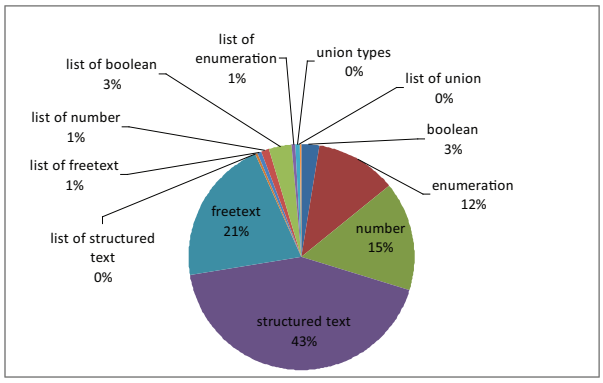
We extracted all the input parameters of the applications through the approach described in Section 2.2. Following this approach, we were able to link 270 parameters corresponding to XSS attack vectors, and 248 parameters corresponding to SQL injection vectors to the source of of the test set of applications.[2]

Figures 3a and 3b show an overview of the types corresponding to input parameters vulnerable to XSS and SQL injection. Most of the vulnerable parameters had one of the following types: boolean, numeric, structured text, free text, enumeration, or union. Booleans can take either logical true or false values. Examples of numeric types are integers or floating point numbers. By "structured text", we mean that the parameter is a string and, additionally, there is an expected structure to the string. A real name, URL, email address, or a username in a registration form are examples of this type. In contrast, the "free text" type denotes arbitrary, unstructured strings. Input parameters corresponding to the enumeration type should only accept a finite set of values that are known in advance. Examples are genders, country names, or a select form field. Finally, a union type denotes a variable that combines multiple types (e.g., a value that should either be a numeric value or a boolean value).

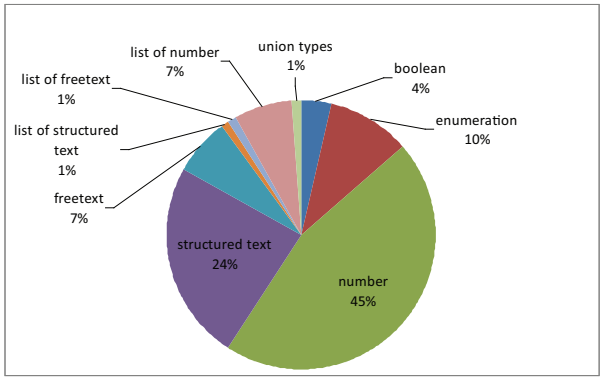
Only about 20% of the input validation vulnerabilities are associated with the free text type. This means that in these cases, the application should accept an arbitrary text input. Hence, an input validation vulnerability of this type can only be prevented by sanitizing the user-supplied input.

Interestingly, 35% of the input parameters vulnerable to XSS are actually numeric, enumeration, or boolean types (including lists of values of these types), while 68% of the input parameters vulnerable to SQL injections correspond to these simple data types. Thus, the majority of input validation vulnerabilities for these applications could have been prevented by enforcing the validation of user-supplied input based on simple data types. We believe that the large number of parameters vulnerable to SQL injection that correspond to the numeric type is caused by the phenomenon that many web applications use integers to identify objects and use these values in the application logic that interacts with the backend database (e.g. to identify users, messages, or blog posts).

[2]Many CVE reports do not mention any attack vectors. Hence, we excluded them for this analysis.



(a) XSS vulnerabilities.



(b) SQL injection vulnerabilities.

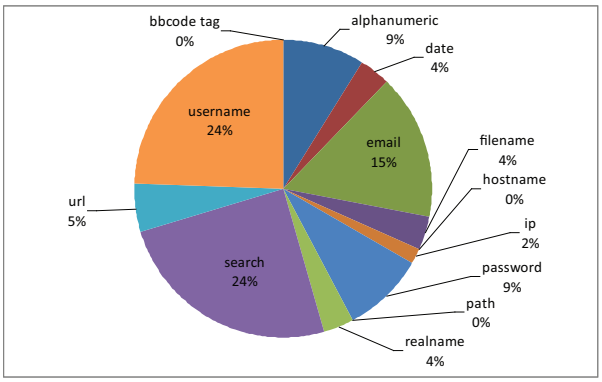
Figure 3: Data types corresponding to vulnerable input parameters.

3.4 Input Validation As an Explicit Defense

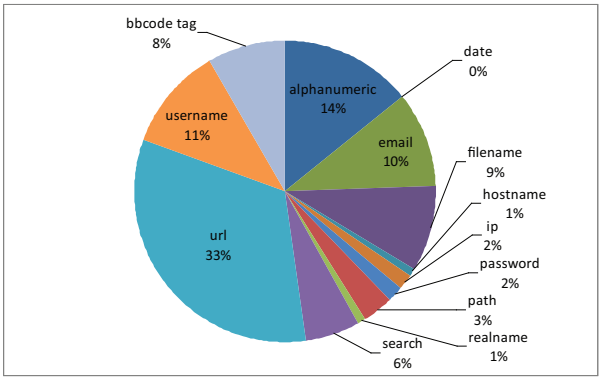
In Section 3.2 we argued that a comprehensive support for input validation in popular web application frameworks can improve the resilience of a language to the introduction of XSS and SQL injection vulnerabilities. Even though developers must remember to explicitly use these validation functions for every possible input of the application, the fact that the right functions are already provided by the framework greatly simplifies the process.

To verify the extension of the support offered by common frameworks, we first need to extract the different classes of structured text responsible for most of the attack against web applications. Figures 4b and 4a show a detailed overview of which particular "structured text" is responsible for most of the XSS and SQL injection vulnerabilities. The graph shows that web applications would benefit from input validation routines that are able to sanitize complex data such as URLs, usernames, and email addresses, since these data classes are often used as attack vectors.

However, implementing them and systematically analyz-



(a) XSS vulnerabilities.



(b) SQL injection vulnerabilities.

Figure 4: Structured string corresponding to vulnerable input parameters.

ing them for correctness and safety is not a simple task. Therefore, one should expect this functionality to be provided by many application frameworks. In our experiments, we analyzed 78 open source web application frameworks for several web programming languages, including: PHP, Perl, Python, Ruby, .NET, and Java. These frameworks were selected on the basis of factors such as popularity as well as the size and activity level of the developer and user communities. For each framework, we classified the kinds of validation functions for complex input values that are exposed to developers.

Partial results of this classification are shown in Table 1. We observe that almost 20% of the frameworks we studied do not provide any validation functionality at all. In fact, of the 78 frameworks we analyzed, only 37 provided any support for validation of complex input data, though these frameworks supported a wide variety of different data types — 31 in all.[3]

Unfortunately, there is a mismatch between the set of validation functions normally provided by these frameworks and the common attack vectors reported in Figures 4a and 4b. For example, 43% of the frameworks do not provide any way to automatically validate URLs that are often used for XSS attacks.

[3]In the interests of space, in Table 1 we summarize only those validation functions that appeared in five or more frameworks.

4. DISCUSSION

Our empirical results from Section 3 indicate that the implicit validation resulting from casting input data to primitive types observed in applications written in strongly-typed languages is, indeed, correlated to a decreased exposure to XSS and SQL injection vulnerabilities. Additionally, the data indicates that the availability of explicit validation functions is also correlated with a reduced count of reported vulnerabilities.

As a result, we can conclude that there is empirical evidence to support the general intuition that input validation serves as an effective layer of defense against XSS and SQL injection vulnerabilities. In fact, it is likely that the increased usage of strongly-typed languages and explicit input validation functions for web programming would have similar benefits for other classes of vulnerabilities, as well as more general software faults.

Note, however, that we observe that input validation is not a panacea for eradicating vulnerabilities in web applications. For example, a particular drawback of the explicit input validation for complex input data is that the developer is responsible for applying the appropriate validator to each and every possible user input that is not already covered by implicit typecasting. Unfortunately, this is, as operational experience has demonstrated in the case of web application output sanitization, an arduous and error-prone task [23].

Therefore, we advocate that, analogous to the case of framework support for automatic output sanitization, web development languages and frameworks should support the automatic validation of web application inputs as an additional security measure against both XSS and SQL injection vulnerabilities, as well as other security-relevant application logic flaws.

An automatic input validation policy can take several concrete forms. One such instantiation would be to enrich the type system of an appropriate strongly-typed web programming language, such that the language could infer the proper validation routines to apply for a wide variety of common input data. Another possibility would be framework support for a centralized policy description that explicitly enumerates the possible input vectors to an application, as well as the appropriate validation functions to apply. The investigation of these avenues for automatic input validation is promising research work.

5. RELATED WORK

Our work is not the first study of vulnerabilities in web applications. Fonseca et al. studied how software faults relate to web application security [4, 24]. Their results show that only a small set of software fault types is responsible for most of the XSS and SQL injection vulnerabilities in web applications. Moreover, they empirically demonstrated that the most frequently occurring fault type is that of missing function calls to sanitization or input validation functions. Our work partially corroborates this finding, but also focuses on the potential for automatic input validation as a means of improving the effectiveness of existing input validation methods.

Independent from our work, Weinberger et al. studied in detail how effective web application frameworks are in sanitizing user-supplied input to defend applications against XSS attacks [29]. In their work, they compare the sanitiza-

Language	PHP	Perl	Python	Ruby	.NET	Java	Total
Frameworks	21	4	2	0	3	7	37 (100%)
Email	16	2	1	0	3	7	29 (78%)
Date	13	4	2	0	2	3	24 (64%)
URL	11	1	2	0	2	5	21 (57%)
Alphanumeric	10	2	1	0	1	0	14 (38%)
Phone	7	1	0	0	0	1	9 (24%)
Time	6	1	2	0	0	0	9 (24%)
Password	4	3	0	0	0	2	9 (24%)
IP Address	6	1	1	0	0	0	8 (22%)
Filename	4	2	1	0	0	0	7 (19%)
Credit card	3	0	0	0	1	3	7 (19%)

Table 1: Framework support for various complex input validation types across different languages.

tion functionality provided by web application frameworks and the features that popular web applications require. In contrast to our work, their focus is on output sanitization as a defense mechanism against XSS, while we investigate the potential for input validation as an additional layer of defense against both XSS and SQL injection.

While much research effort has been spent on applying taint-tracking techniques [8, 11, 18, 19, 20, 27, 28, 31] to ensure that untrusted data is sanitized before its output, less effort has been spent on the correctness of input validation and sanitization. Wassermann proposed an approach based on string-taint analysis to determine the set of strings an application may generate [28]. Balzarotti et al. investigated the application of static and dynamic taint analysis techniques to check the correctness of the sanitization process [1]. The evaluation of the two tools shows that input validation vulnerabilities can be caused by incorrect implementations of validation and sanitization routines.

Finifter et al. also studied the relationship between the choice of development tools and the security of the resulting web applications [3]. Their study focused in-depth on nine applications written to an identical specification, with implementations using several languages and frameworks, while our study examined a broader selection of applications, languages, and frameworks. While their study did not find a relationship between the choice of development tools and application security, their work shows that automatic, framework-provided mechanisms are preferable to manual mechanisms for mitigating vulnerabilities related to Cross-Site Request Forgery, broken session management and insecure password storage.

6. CONCLUSION

Web applications have become an important part of the daily lives of millions of users. Unfortunately, web applications are also frequently targeted by attacks such as XSS and SQL injection.

In this paper, we presented our empirical study of more than 7000 web application vulnerabilities and more than 70 web application development frameworks with the aim of gaining deeper insights into how common web vulnerabilities can be prevented. In the study, we have focused on the relationship between the specific programming language used to develop web applications, and the vulnerabilities that are commonly reported.

Our findings suggest that many SQL injection and XSS could easily be prevented if web languages and frameworks would be able to automatically enforce common data types such as integer, boolean, and specific types of strings such as e-mails and URLs.

Acknowledgments

The research leading to these results was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) from the PoSecCo project (contract N 216917) and contract N 257007, by the FFG - Austrian Research Promotion Agency from the COMET K1-project and by National Science Foundation grant CNS-1116777. This work has also been supported by the French National Research Agency through the CESSA and VAMPIRE projects. We would also like to thank Secure Business Austria for their support.

7. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Saner: composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[2] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on the World Wide Web*, pages 91–100, Raleigh, NC, USA, 2010. ACM.

[3] M. Finifter and D. Wagner. Exploring the Relationship Between Web Application Development Tools and Security. In *USENIX Conference on Web Application Development (WebApps)*. USENIX Association, June 2011.

[4] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *Proceedings of the IEEE Internation Conference on Dependable Systems and Networks*, pages 257–266, June 2008.

[5] Fortify Software. Fortify Software Security Assurance Products. `http://www.fortify.com`, 2011.

[6] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on the World Wide Web*, pages 601–610, Banff, AB, CA, 2007. ACM.

[7] M. Johns, C. Beyerlein, R. Giesecke, and J. Posegga. Secure Code Generation for Web Applications. In *International Symposium on Engineering Secure Software and Systems*, LNCS 5965, pages 96–113. Springer, 2010.

[8] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Oakland, CA, USA, 2006. IEEE Computer Society.

[9] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 330–337, Dijon, FR, 2006. ACM.

[10] B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 95–104, San Diego, CA, USA, 2007. ACM.

[11] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, Aug 2005.

[12] B. Martin, M. Brown, A. Paller, and D. Kirby. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25/`, 2011.

[13] Microsoft Inc. MSDN Code Analysis Team Blog. `http://blogs.msdn.com/b/codeanalysis/`, 2010.

[14] MITRE. Common Platform Enumeration (CPE). `http://cpe.mitre.org/`, 2010.

[15] MITRE. Common vulnerabilities and exposures (cve). `http://cve.mitre.org/`, 2010.

[16] MITRE. Common weakness enumeration (cwe). `http://cwe.mitre.org/`, 2010.

[17] National Institute of Standards and Technology. National Vulnerability Database Version 2.2. `http://nvd.nist.gov/`, 2010.

[18] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the ISOC Network and Distributed Systems Symposium*, 2005.

[19] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *SEC*, pages 295–308, 2005.

[20] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[21] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th USENIX Security Symposium*, pages 283–298. USENIX Association, 2009.

[22] SANS Intitute. Information Security Training, Certification & Research. `http://www.sans.org/`, 2011.

[23] T. Scholte, D. Balzarotti, and E. Kirda. Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, Bay Gardens Beach Resort, Saint Lucia, 2011.

[24] N. Seixas, J. Fonseca, M. Vieira, and H. Madeira. Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 129–135. IEEE Computer Society, 2009.

[25] The Open Web Application Security Project. OWASP – The Open Web Application Security Project. `http://www.owasp.org/`, 2011.

[26] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 173–186, Chicago, IL, USA, 2009. ACM.

[27] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, Jun 2007. ACM.

[28] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, GER, May 2008. ACM.

[29] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical report, UC Berkeley, 2011.

[30] WhiteHat Security. WhiteHat Security Statistics Report. `http://www.whitehatsec.com`, 2011.

[31] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, 2006. USENIX Association.

[32] D. Yu, A. Chander, H. Inamura, and I. Serikov. Better abstractions for secure server-side scripting. In *Proceedings of the 17th International Conference on the World Wide Web*, pages 507–516, Beijing, CN, 2008. ACM.