Have things changed now? An Empirical Study on Input Validation Vulnerabilities in Web Applications

Theodoor Scholte[a,*], Davide Balzarotti[b], Engin Kirda[c]

[a] *SAP Research Security & Trust, Sophia-Antipolis, France*
[b] *Networking and Security Department, Institute Eurecom, Sophia-Antipolis, France*
[c] *College of Computer and Information Science, Northeastern University, Boston MA, USA*

Abstract

Web applications have become important services in our daily lives. Millions of users use web applications to obtain information, perform financial transactions, have fun, socialize, and communicate. Unfortunately, web applications are also frequently targeted by attackers. Recent data from SANS institute estimates that up to 60% of Internet attacks target web applications.

In this paper, we perform an empirical analysis of a large number of web vulnerability reports with the aim of understanding how input validation flaws have evolved in the last decade. In particular, we are interested in finding out if developers are more aware of web security problems today than they used to be in the past. Our results suggest that the complexity of the attacks have not changed significantly and that many web problems are still simple in nature. Hence, despite awareness programs provided by organizations such as MITRE, SANS Institute and OWASP, application developers seem to be either not aware of these classes of vulnerabilities, or unable to implement effective countermeasures. Therefore, we believe that there is a growing need for languages and application platforms that attack the root of the problem and secure applications by design.

Keywords: Security, Software Engineering, Web Security, Vulnerability Study, Input Validation, Web Application

1. Introduction

The web has become part of daily life, and web applications now support us in many of our daily activities. Unfortunately, web applications are prone to various classes of vulnerabilities. Hence, much effort has been spent on making web applications more secure in the past decade (e.g., [1][2][3]).

☆This document is a collaborative effort.

*Corresponding author

Email addresses: theodoor.scholte@sap.com (Theodoor Scholte), balzarotti@eurecom.fr (Davide Balzarotti), ek@ccs.neu.edu (Engin Kirda)

Organizations such as MITRE [2], SANS Institute [1] and OWASP [3] have emphasized the importance of improving the security education and awareness among programmers, software customers, software managers and chief information officers. These organizations do this by means of regularly publishing lists with the most common programming errors. Also, the security research community has worked on tools and techniques to improve the security of web applications. These techniques include static code analysis [4, 5, 6, 7, 8], dynamic tainting [9, 10, 11], combination of dynamic tainting and static analysis [12], prevention by construction or by design [13, 14, 15, 16] and enforcement mechanisms executing within the browser [17, 18, 19, 20]. Some of these techniques have been commercialized and can be found in today's development tool sets. An example is Microsoft's FxCop [21], which can be integrated into some editions of Microsoft Visual Studio.

Although a considerable amount of effort has been spent by many different stake-holders on making web applications more secure, we lack quantitative evidence that this attention has improved the security of web applications over time. In particular, we are interested in finding out and understanding how two common classes of vulnerabilities, namely SQL injection and cross-site scripting, have evolved in the last decade.

We chose to focus our study on SQL injection and cross-site scripting vulnerabilities as these classes of web application vulnerabilities have the same root cause: improper sanitization of user-supplied input that results from invalid assumptions made by the developer on the input of the application. Moreover, these classes of vulnerabilities are prevalent, well-known and have been well-studied in the past decade. Thus, it is likely that there is a sufficient number of vulnerability reports available to allow an empirical analysis. In this paper, we extend the work presented in [22] in which we conducted an automated analysis with the aim of answering the following questions:

1. *Do attacks become more sophisticated over time?*
   We automatically analyzed over 2600 vulnerabilities and found out that the vast majority of them was not associated with any sophisticated attack techniques. Our results suggest that the exploits do not intend to evade any input validation, escaping or encoding defense mechanisms. Moreover, we do not observe any particular increasing trend with respect to complexity.
2. *Do well-known and popular applications become less vulnerable over time?*
   Our results show that an increasing number of applications have exactly one vulnerability. Furthermore, we observe a shift from popular applications to non-popular applications with respect to SQL injection vulnerabilities, a trend that is, unfortunately, not true for cross-site scripting.
3. *Do the most affected applications become more secure over time?*
   We studied in detail the ten most affected open source applications resulting in two top ten lists – one for cross-site scripting and one for SQL injection. In total, 197 vulnerabilities were associated with these applications. We investigated the difference between *foundational* and *non foundational* vulnerabilities and found that the first class is decreasing over

time. Moreover, an average time of 4.33 years between the initial software release and the vulnerability disclosure date suggests that many of today's reported cross-site scripting vulnerabilities were actually introduced into the applications many years ago.

This paper elaborates on the work published in [22] in the following way:

1. *Do the requirements for an attack become more complex?*
   We performed an automated analysis on a large number of cross-site scripting and SQL injection vulnerability reports to understand whether an attacker nowadays has to meet certain criteria before performing the actual attack – for example, whether an attacker has to be authenticated in order to conduct an attack. Although we did observe an increase in the number of vulnerabilities that required an attacker to fulfill some prerequisites, this increase is not significant.
2. *Do SQL injection vulnerabilities in an application come together with cross-site scripting vulnerabilities?*
   We computed the correlation between applications affected by cross-site scripting and applications affected by SQL injection vulnerabilities over time. Interestingly, we observed a strong negative correlation which is increasing over time.
3. *Do applications become more secure with time?*
   We examined the reporting rates of cross-site scripting and SQL injection found in web applications. We found out that the majority of reports about an application are reported within 2 months of each other. Furthermore, the reporting rate of cross-site scripting and SQL injection decreases slightly when comparing the period 2006-2007 with the period 2008-2009. This might suggest that the situation is improving.
4. *Are there any long term effects of input validation vulnerabilities in web applications?*
   We studied the disclosure duration of *foundational* and *non-foundational* vulnerabilities. Our results show that the disclosure duration of foundational cross-site scripting vulnerabilities is large compared to the disclosure duration of foundational SQL injection and non foundational cross-site scripting vulnerabilities. By measuring the average disclosure duration over the years, we show that with time, the disclosure duration of foundational cross-site scripting vulnerabilities is increasing. This suggests that nowadays we still see the effects of cross-site scripting vulnerabilities that were introduced many years ago.

The rest of the paper is organized as follows: The next section describes our methodology and data gathering technique. Section 3 presents an analysis of the SQL injection and cross-site scripting reports and their associated exploits. In Section 4, we present the related work and then briefly conclude the paper in Section 5.

2. Methodology

To be able to answer how cross-site scripting and SQL injection vulnerabilities have evolved over time, it is necessary to have access to significant amounts of vulnerability data. Hence, we had to collect and classify a large number of vulnerability reports. Furthermore, automated processing is needed to be able to extract the exploit descriptions from the reports. In the next sections, we explain the process we applied to collect and classify vulnerability reports and exploit descriptions.

2.1. Data Gathering

One major source of information for security vulnerabilities is the CVE dataset, which is hosted by MITRE [23]. According to MITRE's FAQ [24], CVE is not a vulnerability database but a vulnerability identification system that 'aims to provide common names for publicly known problems' such that it allows 'vulnerability databases and other capabilities to be linked together'. Each CVE entry has a unique CVE identifier, a status ('entry' or 'candidate'), a general description, and a number of references to one or more external information sources of the vulnerability. These references include a source identifier and a well-defined identifier for searching on the source's website. Vulnerability information is provided to MITRE in the form of *vulnerability submissions*. MITRE assigns a CVE identifier and a candidate status. After the CVE Editorial Board has reviewed the candidate entry, the entry may be assigned the 'Accept' status.

For our study, we used the CVE data from the National Vulnerability Database (NVD) [25] which is provided by the National Institute of Standards and Technology (NIST). In addition to CVE data, the NVD database includes the following information:

- Vulnerability type according to the Common Weakness Enumeration (CWE) classification system [26].
- The name of the affected application, version numbers, and the vendor of the application represented by Common Platform Enumeration (CPE) identifiers [27].
- The impact and severity of the vulnerability according to the Common Vulnerability Scoring System (CVSS) standard [28].

The NIST publishes the NVD database as a set of XML files, in the form: `nvdcve-2.0-year.xml`, where year is a number from 2002 until 2010. The first file, `nvdcve-2.0-2002.xml` contains CVE entries from 1998 until 2002. In order to build timelines during the analysis, we needed to know the discovery date, disclosure date, or the publishing date of a CVE entry. Since CVE entries originate from different external sources, the timing information provided in the CVE and NVD data feeds proved to be insufficient. For this reason, we fetched

this information by using the disclosure date from the corresponding entry in the Open Source Vulnerability Database (OSVDB) [29].

For each candidate and accepted CVE entry, we extracted and stored the identifier, the description, the disclosure date from OSVDB, the CWE vulnerability classification, the CVSS scoring, the affected vendor/product/version information, and the references to external sources. Then, we used the references of each CVE entry to retrieve the vulnerability information originating from the various external sources. We stored this website data along with the CVE information for further analysis.

2.2. Vulnerability Classification

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

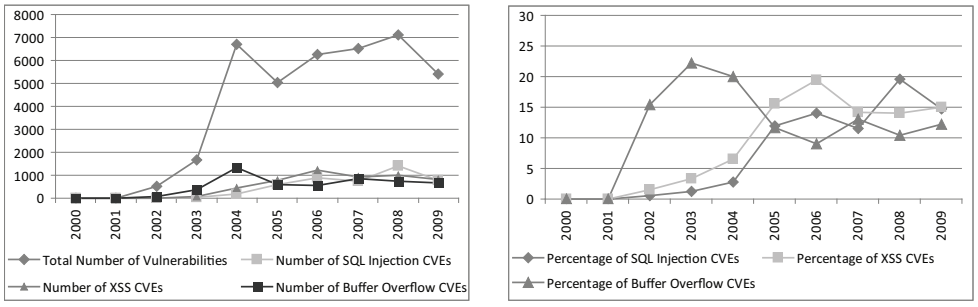
Although NVD provides a mapping between CVEs and CWEs, this mapping is not complete and many CVE entries do not have any classification at all. For this reason, we chose to perform a classification which is based on both the CWE classification and on the description of the CVE entry. In general, we observed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

For fetching the cross-site scripting related CVEs out of the CVE data, 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 the SQL injection related CVEs by using the CWE identifier 'CWE-89' and the keyword 'SQL injection'.

2.3. The Exploit Data Set

To acquire a general view on the security of web applications, we are not only interested in the vulnerability information, but also in the way each vulnerability can be exploited. Some external sources of CVEs that provide information concerning cross-site scripting or SQL injection-related vulnerabilities also provide exploit details. Often, this information is represented by a script or an *attack string*.

An attack string is a well-defined reference to a location in the vulnerable web application where code can be injected. The reference is often a complete URL that includes the name of the vulnerable script, the HTTP parameters, and some characters to represent the placeholders for the injected code. In addition to using placeholders, sometimes, real examples of SQL or Javascript code may also be used. Two examples of attack strings are:



(a) Vulnerability trends in numbers

(b) Vulnerability trends in percentages

Figure 1: *Buffer overflow, cross-site scripting and SQL injection vulnerabilities over time*

```
http://[victim]/index.php?act=delete&dir=&file=[XSS]
http://[victim]/index.php?module=subjects&func=viewpage&pageid=[SQL]
```

At the end of each line, note the placeholders that can be substituted with arbitrary code by the attacker.

The similar structure of attack strings allows our tool to automatically extract, store and analyze the exploit format. Hence, we extracted and stored all the attack strings associated with both cross-site scripting and the SQL injection CVEs.

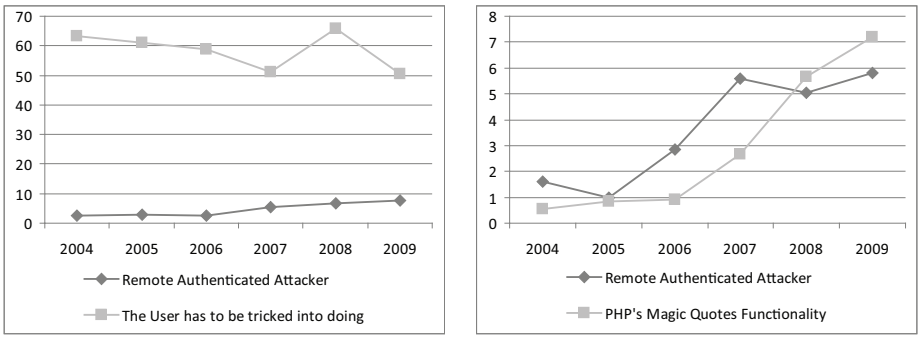
3. Analysis of the Vulnerabilities Trends

The first question we wish to address in this paper is whether the number of SQL injection and cross-site scripting vulnerabilities reported in web applications has been decreasing in recent years. To answer this question, we automatically analyzed the 39,081 entries in the NVD database from 1998 to 2009[1]. We had to exclude 1,301 CVE entries because they did not have a corresponding match in the OSVDB database and, as a consequence, did not have a disclosure date associated with them. For this reason, these CVE entries are not taken into account for the rest of our study. Of the remaining vulnerability reports, we identified a total of 5349 buffer overflow entries, 5413 cross-site scripting entries and 4825 SQL injection entries.

Figure 1a shows the number of vulnerability reports over time and figure 1b shows the percentage of reported vulnerabilities over the total CVE entries.

Our first expectation based on intuition was to observe that the number of reported vulnerabilities follow a classical bell shape, beginning with a slow

[1]At the time of our study, a full vulnerability dataset of 2010 was not available. Hence, our study does not cover 2010.



(a) cross-site scripting (b) SQL injection

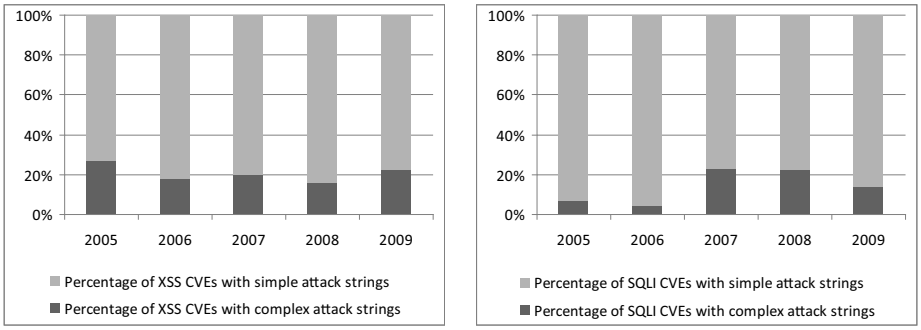
Figure 2: *Prerequisites for successful attacks (in percentages)*

start when the vulnerabilities are still relatively unknown, then a steep increase corresponding to the period in which the attacks are disclosed and studied, and finally a decreasing phase when the developers start adopting the required countermeasures. In fact, the graphs show an initial phase (2002-2004) with very few reports about cross-site scripting and SQL injection vulnerabilities and many reports about buffer overflow vulnerabilities. This phase is followed by a steep increase in input validation vulnerability reports in the years 2004, 2005 and 2006 and overtakes the number of Buffer Overflow vulnerability reports. Note that this trend is consistent with historical developments. Web security started increasing in importance after 2004, and the first XSS-based worm was discovered in 2005 (i.e., "Samy Worm" [30]). Hence, web security threats such as cross-site scripting and SQL injection started receiving more focus after 2004 and, in the meantime, these threats have overtaken buffer overflow problems. Unfortunately, the number of reported cross-site scripting and SQL injection vulnerabilities has not significantly decreased since 2006. In other words, the number of cross-site scripting and SQL injection vulnerabilities found in 2009 is comparable with the number reported in 2006. In the rest of this section, we will formulate and verify a number of hypotheses to explain the possible reasons behind this phenomenon.

3.1. Attack Sophistication

Hypothesis 1. *Simple, easy-to-find vulnerabilities have now been replaced by complex vulnerabilities that require more sophisticated attacks.*

The first hypothesis we wish to verify is whether the overall number of vulnerabilities is not decreasing because the simple vulnerabilities discovered in the early years have now been replaced by new ones that involve more complex attack scenarios. In particular, we are interested in finding out whether the prerequisites for an attack have changed over time. We were inspired by bug



(a) cross-site scripting (b) SQL injection

Figure 3: *Exploit complexity over time*

reports corresponding to the vulnerabilities to look at the prerequisites for attacks. By investigating the bug reports of web applications, we found out that in some of these cases, software developers are aware of a vulnerability but are unwilling to fix it because the vulnerability is only exploitable in certain scenarios and the risk is minimal. One example of such a scenario is a vulnerability in the administration interface of a web application which is only expoitable by an administrator. Moreover, some vulnerabilities are only exploitable when the user is tricked into performing some action via a phishing attack, for example. Software developers may also decide not to fix SQL injection vulnerabilities if certain configuration settings can prevent them.

To determine the prerequisites for successful attacks, we searched for particular phrases in the descriptions of the CVE entries. For cross-site scripting vulnerabilities, we looked at the occurrence of the following phrases:

- 'remote authenticated' to identify whether the attacker needs to be authenticated.
- 'trick', 'tricked', 'tricking', 'crafted link', 'crafted url', 'malicious url', 'malicious link', 'malicious website', 'crafted website', 'malicious email', 'malicious e-mail', 'crafted email', 'crafted e-mail', 'malicious message', 'crafted message' to identify whether the attacker needs to deceive the victim into performing some action.

For SQL injection vulnerabilities, we looked for occurrence of the following keywords:

- 'remote authenticated' to identify whether the attacker needs to be authenticated.
- 'without magic_quotes_gpc enabled', 'magic_quotes_gpc is disabled' to determine whether disabling PHP's magic_quotes functionality allows a SQL injection attack.

Figures 2a and 2b plot the percentage of vulnerabilities requiring the given prerequisite over the total number of cross-site scripting or SQL injection vulnerabilities in the given year, respectively. Figure 2a suggests that at least 50 percent of the cross-site scripting vulnerabilities require some involvement from the victim. We observe that since 2005, there has been a slight increase of SQL injection vulnerabilities that can only be exploited when the controversial magic_quotes feature is disabled. This trend is consistent with PHP's development roadmap, which intends to deprecate the feature in PHP version 5.3.0 and remove it in version 6.0. Another trend we observed is a slight increase in vulnerabilities that require an attacker to be authenticated. Although the trend is not significant, it may suggest that developers have started to pay attention to the security of functionality accessible by everyone but fail to secure the functionality used by (website) administrators. Since the trends on prerequisites are not significant, we do not consider them as being the reason behind the steadily increasing input validation vulnerabilities trends. We are also interested in discovering whether the complexity of exploits has increased. Our purpose in doing this is to identify those cases in which the application developers were aware of threats but implemented insufficient, easy-to-evade sanitization routines. In these cases, an attacker has to craft the malicious input more carefully or has to perform certain input transformations (e.g., uppercase or character replacement).

One way to determine the "complexity" of an exploit is to analyze the attack string and to look for evidence of possible evasion techniques. As mentioned in Section 2.3, we automatically extract the exploit code from the data provided by external vulnerability information sources. Sometimes, these external sources do not provide exploit information for every reported cross-site scripting or SQL injection vulnerability, do not provide exploit information in a parsable format, or do not provide any exploit information at all. As a consequence, not all CVE entries can be associated with an *attack string*. On the other hand, in some cases, there exist several ways of exploiting a vulnerability, and, therefore, many *attack strings* may be associated with a single vulnerability report. In our experiments, we collected attack strings for a total of 2632 distinct vulnerabilities.

To determine the exploit complexity, we looked at several characteristics that may indicate an attempt from the attacker to evade some form of input sanitization. The selection of the characteristics is inspired by so-called injection cheat sheets that are available on the Internet [31][32].

In particular, we classify a cross-site scripting attack string as complex (in contrast to simple) if it contains one or more of the following characteristics:

- Different cases are used within the script tags (e.g., `ScRiPt`).
- The script tags contain one or more spaces (e.g., `< script>`)
- The attack string contains 'landingspace-code' which is the set of attributes of HTML-tags (e.g., `onmouseover`, or `onclick`)
- The string contains encoded characters (e.g., `&#41;`)

- The string is split over multiple lines

For SQL injection attack strings, we looked at the following characteristics:

- The use of comment specifiers (e.g., `/**/`) to break a keyword
- The use of encoded single quotes (e.g., '`%27`', '`&#x27`'; '`&#39`', '`Jw==`')
- The use of encoded double quotes (e.g., '`%22`', '`&#x22;`', '`&#34`', '`Ig==`')

If none of the previous characteristics is present, we classify the exploit as "simple". Figures 3a and 3b show the percentage of CVEs having one or more complex attack strings[2]. The graphs show that the majority of the available exploits are, according to our definition, not sophisticated. In fact, in most of the cases, the attacks were performed by injecting the simplest possible string, without requiring any tricks to evade input validation.

Interestingly, while we observe a slight increase in the number of SQL injection vulnerabilities with sophisticated attack strings, we do not observe any significant increase in cross-site scripting attack strings. This may be a first indication that developers are now adopting (unfortunately insufficient) defense mechanisms to prevent SQL injection, but that they are still failing to sanitize the user input to prevent cross-site scripting vulnerabilities.

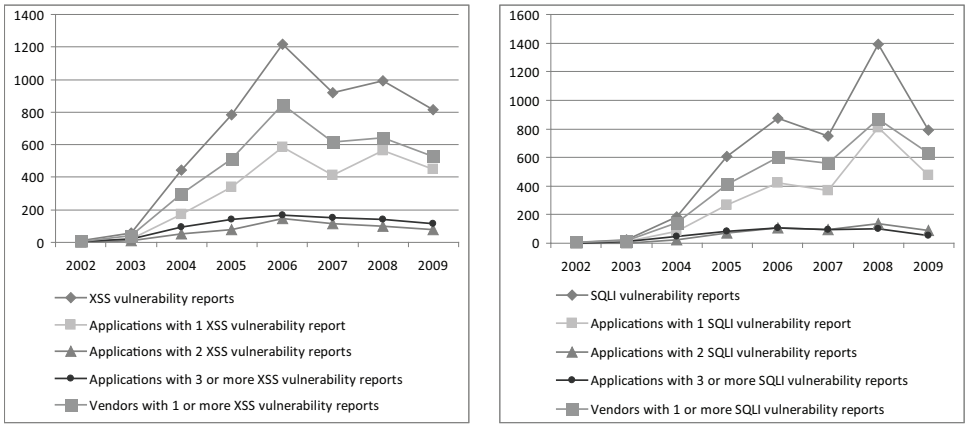
To conclude, the available empirical data suggests that increased attack complexity *is not* the reason behind the steadily increasing number of vulnerability reports.

3.2. Application Popularity

Since complexity does not seem to explain the increasing number of reported vulnerabilities, we decided to focus on the type of applications. We started by extracting vulnerable application and vendor names from a total of 8854 SQL injection and cross-site scripting vulnerability reports in the NVD database that are associated to one or more CPE identifiers. Figures 4a and 4b plot the number of applications and vendors that are affected by a certain number of vulnerabilities over time. Both graphs clearly show how the increase in the number of vulnerabilities is a direct consequence of the increasing number of vulnerable applications and their vendors. In fact, the number of web applications with more than one vulnerability report over the whole time frame is quite low, and it has been slightly decreasing since 2006.

Since cross-site scripting and SQL injection vulnerabilities have the same root cause, it is interesting to investigate whether there is a relationship between the occurrence of the two types of vulnerabilities in web applications. This gives an answer to the question whether a developer who fails to implement countermeasures against SQL injection also fails to implement countermeasures against

[2]The graph starts from 2005 because there were less than 100 vulnerabilities having exploit samples available before that year. Hence, results before 2005 are statistically less significant.



(a) cross-site scripting affected applications (b) SQL injection affected applications

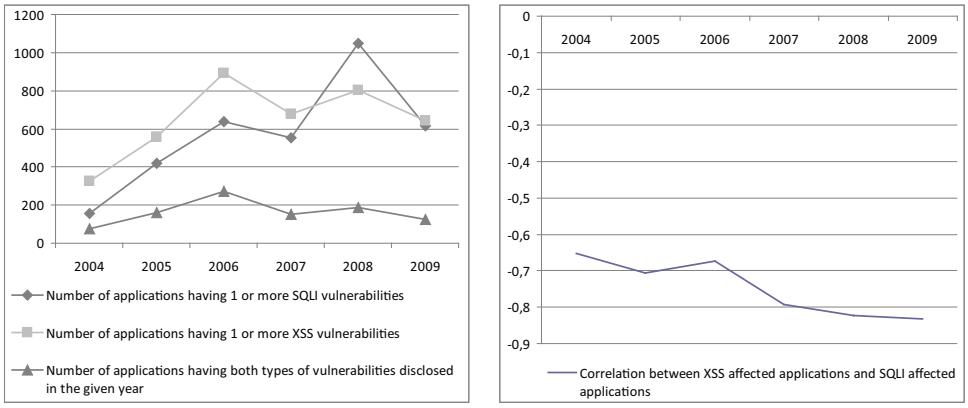
Figure 4: *The number of affected applications over time*

cross-site scripting. In order to answer this question, we measured the correlation between cross-site scripting and SQL injection vulnerabilities. Although correlation cannot be used to infer a causal relationship between SQL injection and cross-site scripting vulnerabilities, it can indicate the potential existence of this causal relationship. Figure 5 shows the correlation between applications affected by both cross-site scripting and SQL injection vulnerabilities. More specifically, we measured the number of applications affected by both types of vulnerabilities (Figure 5a) and the correlation coefficient over time (Figure 5b). Figure 5b plots $\rho(X, Y)$ with $X = 0$ or 1 indicating whether the application was affected by a cross-site scripting vulnerability and $Y = 0$ or 1 indicating whether the application was affected by an SQL injection vulnerability. The graph shows a strong negative correlation, meaning that the occurrence of cross-site scripting vulnerabilities is correlated with an absence of SQL injection vulnerabilities in an application. As Figure 5b shows, the negative correlation tends to become stronger over time. This might indicate that developers are aware of implementing countermeasures against SQL injection but fail to do so for cross-site scripting vulnerabilities.

Based on these findings, we formulated our second hypothesis:

Hypothesis 2. *Popular applications are now more secure while new vulnerabilities are discovered in new, less popular, applications.*

The idea behind this hypothesis is to test whether more vulnerabilities were reported about well-known, popular applications in the past than are today. That is, do vulnerability reports nowadays tend to concentrate on less popular, or recently developed applications?



(a) The number of applications over time

(b) Correlation coefficient over time

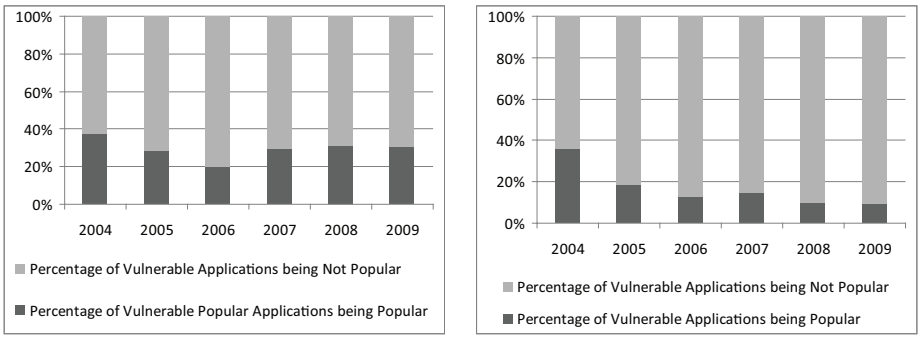
Figure 5: *Applications having cross-site scripting and SQL injection Vulnerabilities over time*

The first step in exploring this hypothesis consists of determining the popularity of these applications in order to be able to understand if it is true that popular products are more aware of (and therefore less vulnerable to) cross-site scripting and SQL injection attacks.

We determined the popularity of applications through the following process:

1. Using Google Search, we performed a search on the vendor and application names within the `Wikipedia` domain.
2. When one of the returned URLs contained the name of the vendor or the name of the application, we flagged the application as being 'popular'. Otherwise, the application was classified as being 'unpopular'.
3. Finally, we manually double-checked the list of popular applications in order to make sure that the corresponding Wikipedia entries described software products and not something else (e.g., when the product name also corresponded to a common English word).

After the classification, we were able to identify 676 popular and 2573 unpopular applications as being vulnerable to cross-site scripting. For SQL injection, we found 328 popular and 2693 unpopular vulnerable applications. Figure 6 shows the percentages of popular applications associated with one or more vulnerability reports. The trends support the hypothesis that SQL injection vulnerabilities are indeed moving toward less popular applications – maybe as a consequence of the fact that well-known products are more security-aware. Unfortunately, according to Figure 6a, the same hypothesis is not true for cross-site scripting: in fact, the ratio of well-known applications vulnerable to cross-site scripting has been relatively constant in the past six years.



(a) cross-site scripting (b) SQL injection

Figure 6: *Vulnerable applications and their popularity over time*

Even though the empirical evidence also does not support our second hypothesis, we noticed one characteristic that is common to both types of vulnerabilities: as shown in Figures shown in Figures 7a and 7b, popular applications typically have a higher number of reported vulnerabilities. There may be many possible reasons to explain this. For example, one possible explanation might be that popular applications are more frequently targeted by attackers, as the application has more impact on potential victims and thus more vulnerabilities are being reported. Another possible explanation could be that developers of popular applications are more security aware or that these applications are better analyzed. Hence, the application meets higher security standards.

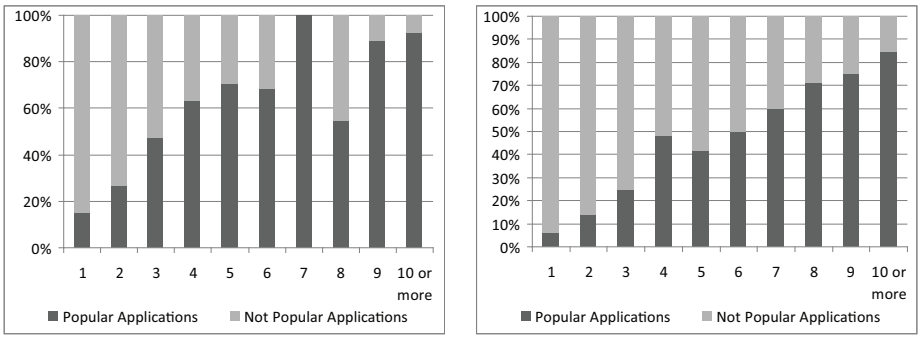
The results, shown in Figures 7a and 7b, suggest that it would be useful to investigate how these vulnerabilities have evolved in the lifetime of the applications.

3.3. Application and Vulnerability lifetime

So far, we determined that a constant, large number of simple, easy-to-exploit vulnerabilities are still found in many web applications today. Also, we determined that the high number of reports is driven by an increasing number of vulnerable applications and not by a small number of popular applications. Based on these findings, we formulate our third hypothesis:

Hypothesis 3. *Even though the number of reported vulnerable applications is growing, each application is becoming more secure over time.*

This hypothesis is important, because, if true, it would mean that web applications (the well-known products in particular) are becoming more secure. To verify this hypothesis, we studied the frequency of vulnerability reports of applications affected by cross-site scripting and SQL injection vulnerabilities.



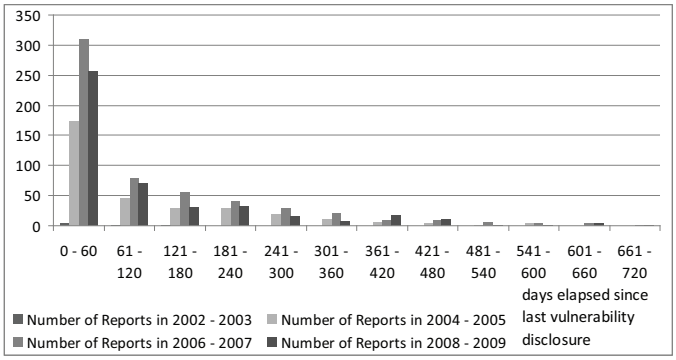
(a) cross-site scripting

(b) SQL injection

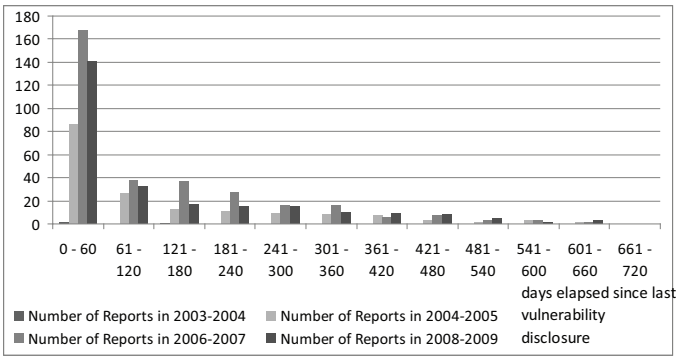
Figure 7: *Popularity of applications across the distribution of the number of vulnerability reports*

One way to examine the security of an application is to measure the rate at which vulnerabilities are being reported. The frequency of vulnerability reports about an application can be estimated by measuring the time between them. We applied an analogous metric from *reliability engineering*, the time between-failures (TBF), by defining a vulnerability report as a failure. Figures 8a and Figure 8b plot the reporting rates of cross-site scripting vulnerabilities and SQL injection vulnerabilities, respectively. The graphs clearly show a steep increase in the reporting rates between 2002 and 2007 and a slight decrease after 2007. In order to verify the hypothesis more precisely, it is also necessary to look at the duration or lifetime of cross-site scripting and SQL injection vulnerabilities. We studied the lifetimes of cross-site scripting and SQL injection vulnerabilities in the ten most-affected open source applications according to the NIST NVD database. By analyzing the change logs for each application, we extracted the version in which a vulnerability was introduced and the version in which a vulnerability was fixed. In order to obtain reliable insights into the vulnerability's lifetime, we excluded vulnerability reports that were not confirmed by the respective vendor. For our analysis, we used the CPE identifiers in the NVD database, the external vulnerability sources, the vulnerability information provided by the vendor. We also extracted information from the version control systems (CVS, or SVN) of the different products.

Table 1a and Table 1b show a total of 147 cross-site scripting and 52 SQL injection vulnerabilities in the most affected applications. The tables distinguish between *foundational* and *non-foundational* vulnerabilities. Foundational vulnerabilities are vulnerabilities that were present in the first version of an application, while non-foundational vulnerabilities were introduced after the initial release.



(a) cross-site scripting



(b) SQL injection

Figure 8: *Reporting rate of vulnerabilities*

We observed that 39% of the cross-site scripting vulnerabilities are foundational and 61% are non-foundational. For SQL injection, these percentages are 42% and 58%. These results suggest that most of the vulnerabilities are introduced by new functionality that is built into new versions of a web application.

Finally, we investigated how long it took to discover the vulnerabilities. Figure 9a and Figure 9b plot the number of vulnerabilities that were disclosed after a certain amount of time had elapsed after the initial release of the applications. The graphs show that most SQL injection vulnerabilities are usually discovered in the first few years after the release of the product. For cross-site scripting vulnerabilities, the result is quite different. Many foundational vulnerabilities are disclosed even 10 years after the code was initially released. This observation suggests that it is very problematic to find foundational cross-site scripting vulnerabilities compared to SQL injection vulnerabilities. This is supported by the fact that the average elapsed time between the software release and the disclosure of foundational vulnerabilities is 2 years for SQL injection vulnerabilities,

	Foundational	Non-Foundational
bugzilla	4	7
drupal	0	22
joomla	5	4
mediawiki	3	21
mybb	9	2
phorum	3	5
phpbb	4	2
phpmyadmin	14	13
squirrelmail	10	4
wordpress	6	9
Total	58	89

(a) cross-site scripting

	Foundational	Non-Foundational
bugzilla	1	8
coppermine	1	3
e107	0	3
joomla	4	0
moodle	0	3
mybb	9	3
phorum	0	4
phpbb	3	0
punbb	4	2
wordpress	0	4
Total	22	30

(b) SQL injection

Table 1: *Foundational and non-foundational vulnerabilities in the ten most affected open source web applications*

Vulnerable applications reporting about scripts:	1871
Vulnerable scripts:	2499
Average (vulnerable scripts / applications):	1.34
Vulnerable applications reporting about parameters:	1905
Vulnerable parameters:	9304
Average (vulnerable parameters / applications):	4.88

(a) cross-site scripting

Vulnerable applications reporting about scripts:	2759
Vulnerable scripts:	3548
Average (vulnerable scripts / applications):	1.29
Vulnerable applications reporting about parameters:	1902
Vulnerable parameters:	6556
Average (vulnerable parameters / applications):	3.45

(b) SQL injection

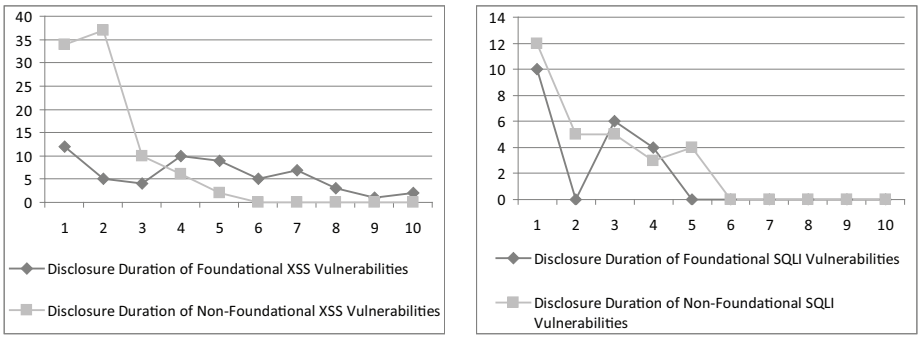
Table 2: The attack surface

while for cross-site scripting this value is 4.33 years.

Figures 10a and 10b plot the average elapsed time between software release and the disclosure of vulnerabilities over time. These results show that cross-site scripting vulnerabilities are indeed harder to find than SQL injection vulnerabilities and that foundational cross-site scripting vulnerabilities become even more difficult to find over time. Also note, that there are no foundational SQL injection vulnerabilities reported in 2009.

We believe that difference between cross-site scripting and SQL injection vulnerabilities concerning the lifetime is caused by the fact that the attack surface for SQL injection attacks is much smaller when compared with cross-site scripting attacks. Therefore, it is interesting to further investigate the size of the attack surface of vulnerable applications.

From the CVE descriptions, we extracted the scripts and parameters that are vulnerable to cross-site scripting or SQL injection and we counted them.



(a) cross-site scripting

(b) SQL injection

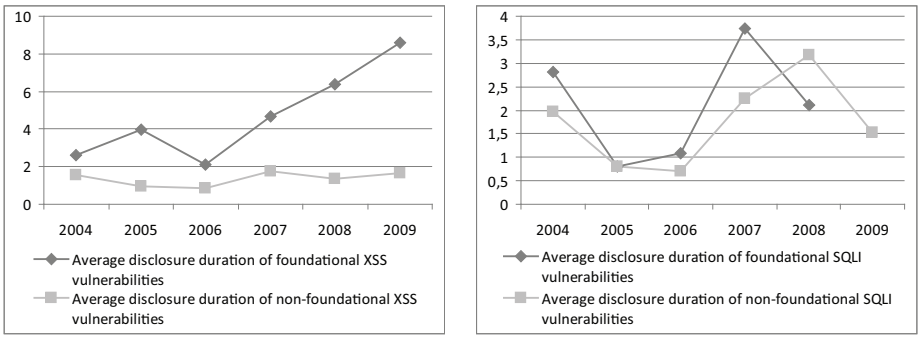
Figure 9: *Time elapsed between software release and vulnerability disclosure in years.*

By measuring the average number of vulnerable scripts and parameters per application for both cross-site scripting and SQL injection vulnerabilities, we get insights into the size of the attack surface. Tables 2a and 2b show the number of applications that are associated with vulnerabilities related to the affected scripts and/or parameters for cross-site scripting and SQL injection vulnerabilities, respectively. In addition, the number of affected scripts and parameters is shown. We observe that the average number of vulnerable scripts and parameters per application is indeed larger for cross-site scripting vulnerabilities than for SQL injection vulnerabilities. Thus, the results confirm the intuition that the difference in vulnerability lifetime between cross-site scripting and SQL injection vulnerabilities is caused by the size of the attack surface. We believe that the attack surface of SQL injection vulnerabilities is smaller because it is easier for developers to identify (and protect) all the sensitive entry points in the code (e.g. code concerning database access) of the web application than for cross-site scripting vulnerabilities.

4. Threats to Validity

The vulnerability classification and analysis on vulnerability complexity is based on static lexical matching. Identifying and analyzing vulnerabilities by searching for combinations of keywords can be difficult because of the possibility of having vulnerability reports containing semantically equivalent but lexically divergent expressions for the same concepts. Moreover, languages do change over time. Neologisms may be accepted and after a period of time these are succeeded by other terms. Both phenomena skew data and might impact the results of our analysis.

In order to get a better understanding of the impact on our analysis, we measured the number of cross-site scripting and SQL injection vulnerabilities



(a) cross-site scripting (b) SQL injection

Figure 10: *Average duration of vulnerability disclosure in years over time*

that are incorrectly classified as cross-site scripting or SQL injection vulnerabilities (false positive rate). The false positive rate was measured by manually analyzing a sample of 50 vulnerabilities for each year between 2002 and 2009. In total, we manually analyzed 358 cross-site scripting vulnerabilities and 324 SQL injection vulnerabilities. This sample represents 6.6 % and 6.7 % of the cross-site scripting and SQL injection classified vulnerabilities, respectively.

In the sample, we found 10 false positives for cross-site scripting vulnerabilities (2,8 %) and 7 false positives for SQL injection (2,2 %). While manually analyzing the vulnerabilities, we did not find any evidence that the keywords we searched for in the vulnerability complexity analysis were used in other contexts than in the contexts we were looking for. Thus, our vulnerability classification and complexity analysis did not result in a significant number of false positives.

Since the NVD dataset is a large dataset, we were unable to perform a manual analysis to measure the false negative rate – that is, the number of vulnerabilities that could not be classified as cross-site scripting or SQL injection vulnerability. Some of the main causes are the lexically divergent expressions for the same concepts and/or language evolution. We expect that the false positives are likely to occur more often in earlier years than in recent years, as the language to describe cross-site scripting and SQL injection vulnerability reports becomes more regular over the years.

5. Related Work

Our work is not the first study of vulnerability trends based on CVE data. In [33], Christey et al. present an analysis of CVE data covering the period 2001 - 2006. The work is based on manual classification of CVE entries using the CWE classification system. In contrast, [34] uses an unsupervised learning technique on CVE text descriptions and introduces a classification system

called *'topic model'*. While the works of Christey et al. and Neuhaus et al. focus on analysing general trends in vulnerability databases, our work specifically focuses on web application vulnerabilities, and, in particular, cross-site scripting and SQL injection. We have investigated the reasons behind the trends.

Clark et al. present in [35] a vulnerability study with a focus on the early existence of a software product. The work demonstrates that re-use of legacy code is a major contributor to the rate of vulnerability discovery and the number of vulnerabilities found. In contrast to our work, the paper does not focus on web applications, and it does not distinguish between particular types of vulnerabilities.

Another large-scale vulnerability analysis study was conducted by Frei et al. [36]. The work focuses on zero-day exploits and shows that there has been a dramatic increase in such vulnerabilities. Also, the work shows that there is a faster availability of exploits than of patches.

In [37], Li et al. present a study on how the number of software defects evolve over time. The data set of the study consists of bug reports of two Open Source software products that are stored in the Bugzilla database. The authors show that security-related bugs are becoming increasingly important over time in terms of absolute numbers and relative percentages. However, they do not consider web applications.

Ozment et al. [38] studied how the number of security issues relate to the number of code changes in OpenBSD. The study shows that 62 percent of the vulnerabilities are *foundational*; they were introduced prior to the release of the initial version and have not been altered since. The rate at which foundational vulnerabilities are reported is decreasing, somehow suggesting that the security of the same code is increasing. In contrast to our study, Ozment el al.'s study does not consider the security of web applications.

To the best of our knowledge, we present the first vulnerability study that takes a closer, detailed look at how two popular classes of web vulnerabilities have evolved over the last decade.

6. Discussion and Conclusion

Our findings in this study show that the complexity of cross-site scripting and SQL injection attacks related to the vulnerabilities in the NVD database has not been increasing. Neither the prerequisites to attacks nor the complexity of exploits have changed significantly. Hence, this finding suggests that the majority of vulnerabilities are not due to sanitization failure, but rather due to the absence of input validation. Despite awareness programs provided by MITRE [23], SANS Institute [1] and OWASP [3], application developers are still not implementing effective countermeasures. Furthermore, our study suggests that a major reason why the number of web vulnerability reports has not been decreasing is because many more applications of different vendors are now vulnerable to flaws such as cross-site scripting and SQL injection. Although cross-site scripting and SQL injection vulnerabilities share the same root cause,

we could not find any significant correlation between applications affected by cross-site scripting and by SQL injection vulnerabilities. In fact, the small negative correlation tends to become stronger. By measuring the popularity of the applications, we observed a trend that SQL injection vulnerabilities occur more often in an increasing number of unpopular applications.

Finally, when analyzing the most affected applications, we observe that years after the initial release of an application, cross-site scripting vulnerabilities concerning the initial release are still being reported. Note that this is in contrast to SQL injection vulnerabilities. By measuring the attack surface of cross-site scripting and SQL injection vulnerabilities, we found out that the attack surface of SQL injection vulnerabilities is much smaller than for cross-site scripting vulnerabilities. Hence, SQL injection problems may be easier to find because only a relatively small part of the application's code is used for database access.

The empirical data we collected and analyzed for this paper supports the general intuition that web developers consistently fail to secure their applications. The traditional practice of writing applications and then testing them for security problems (e.g., static analysis, blackbox testing, etc.) does not seem be working well in practice. Hence, we believe that more research is needed in securing applications by design. That is, the developers should not be concerned with problems such as cross-site scripting or SQL injection. Rather, the programming language or the platform should make sure that the problems do not occur when developers produce code (e.g., similar to solutions such as in [15] or managed languages such as C# or Java that prevent buffer overflow problems).

References

[1] R. Dhamankar, M. Dausin, M. Eisenbarth, J. King, The top cyber security risks, http://www.sans.org/top-cyber-security-risks/ (2009).

[2] B. Martin, M. Brown, A. Paller, D. Kirby, 2010 cwe/sans top 25 most dangerous software errors, http://cwe.mitre.org/top25/ (2010).

[3] OWASP, Owasp top 10 - 2010, the ten most critical web application security risks (2010).

[4] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities (short paper), in: SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2006, pp. 258–263. doi:http://dx.doi.org/10.1109/SP.2006.29.

[5] V. B. Livshits, M. S. Lam, Finding security errors in Java programs with static analysis, in: Proceedings of the 14th Usenix Security Symposium, 2005, pp. 271–286.

[6] G. Wassermann, 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, ACM Press New York, NY, USA, San Diego, CA, 2007.

[7] G. Wassermann, Z. Su, Static Detection of Cross-Site Scripting Vulnerabilities, in: Proceedings of the 30th International Conference on Software Engineering, ACM Press New York, NY, USA, Leipzig, Germany, 2008.

[8] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2006.

[9] J. Newsome, D. X. Song, Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software, in: NDSS, The Internet Society, 2005.

[10] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically hardening web applications using precise tainting, in: R. Sasaki, S. Qing, E. Okamoto, H. Yoshiura (Eds.), SEC, Springer, 2005, pp. 295–308.

[11] T. Pietraszek, C. V. Berghe, Defending against injection attacks through context-sensitive string evaluation, in: A. Valdes, D. Zamboni (Eds.), RAID, Vol. 3858 of Lecture Notes in Computer Science, Springer, 2005, pp. 124–145.

[12] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis, in: In Proceedings of 14th Annual Network and Distributed System Security Symposium (NDSS 2007), 2007.

[13] M. Johns, C. Beyerlein, R. Giesecke, J. Posegga, Secure code generation for web applications, in: F. Massacci, D. S. Wallach, N. Zannone (Eds.), ESSoS, Vol. 5965 of Lecture Notes in Computer Science, Springer, 2010, pp. 96–113.

[14] B. Livshits, U. Erlingsson, Using web application construction frameworks to protect against code injection attacks, in: PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security, ACM, New York, NY, USA, 2007, pp. 95–104. doi:http://doi.acm.org/10.1145/1255329.1255346.

[15] W. Robertson, G. Vigna, Static enforcement of web application integrity through strong typing, in: Proceedings of the 18th conference on USENIX security symposium, USENIX Association, 2009, pp. 283–298.

[16] D. Yu, A. Chander, H. Inamura, I. Serikov, Better abstractions for secure server-side scripting, in: WWW '08: Proceeding of the 17th international conference on World Wide Web, ACM, New York, NY, USA, 2008, pp. 507–516. doi:http://doi.acm.org/10.1145/1367497.1367566.

[17] D. Bates, A. Barth, C. Jackson, Regular expressions considered harmful in client-side xss filters, in: WWW '10: Proceedings of the 19th international conference on World wide web, ACM, New York, NY, USA, 2010, pp. 91–100. doi:http://doi.acm.org/10.1145/1772690.1772701.

[18] T. Jim, N. Swamy, M. Hicks, Defeating script injection attacks with browser-enforced embedded policies, in: WWW '07: Proceedings of the 16th international conference on World Wide Web, ACM, New York, NY, USA, 2007, pp. 601–610. doi:http://doi.acm.org/10.1145/1242572.1242654.

[19] E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Noxes: a client-side solution for mitigating cross-site scripting attacks, in: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, ACM, New York, NY, USA, 2006, pp. 330–337. doi:http://doi.acm.org/10.1145/1141277.1141357.

[20] K. Vikram, A. Prateek, B. Livshits, Ripley: automatically securing web 2.0 applications through replicated execution, in: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security, ACM, New York, NY, USA, 2009, pp. 173–186. doi:http://doi.acm.org/10.1145/1653662.1653685.

[21] Microsoft, Msdn code analysis team blog, http://blogs.msdn.com/b/codeanalysis/ (2010).

[22] T. Scholte, D. Balzarotti, E. Kirda, Quo vadis? a study of the evolution of input validation vulnerabilities in web applications, in: Fifteenth International Conference on Financial Cryptography and Data Security, 2011.

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

[24] MITRE, Mitre faqs, http://cve.mitre.org/about/faqs.html (2010).

[25] NIST, National vulnerability database version 2.2, http://nvd.nist.gov/ (2010).

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

[27] MITRE, Common platform enumeration (cpe), http://cpe.mitre.org/ (2010).

[28] P. Mell, K. Scarfone, S. Romanosky, A complete guide to the common vulnerability scoring system version 2.0, http://www.first.org/cvss/cvss-guide.html (2007).

[29] J. Kouns, K. Todd, B. Martin, D. Shettler, S. Tornio, C. Ingram, P. McDonald, The open source vulnerability database, http://osvdb.org/ (2010).

[30] N. Mook, Cross-site scripting worm hits myspace, http://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/ (October 2005).

[31] F. Mavituna, Sql injection cheat sheet, http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/ (2009).

[32] RSnake, Xss (cross site scripting) cheat sheet esp: for filter evasion, http://ha.ckers.org/xss.html (2009).

[33] S. M. Christey, R. A. Martin, Vulnerability type distributions in cve, http://cwe.mitre.org/documents/vuln-trends/index.html (2007).
URL `http://cwe.mitre.org/documents/vuln-trends/index.html`

[34] S. Neuhaus, T. Zimmermann, Security trend analysis with cve topic models, in: Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering, 2010.

[35] S. Clark, S. Frei, M. Blaze, J. Smith, Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities, in: Annual Computer Security Applications Conference, 2010.

[36] S. Frei, M. May, U. Fiedler, B. Plattner, Large-scale vulnerability analysis, in: LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense, ACM, New York, NY, USA, 2006, pp. 131–138. doi:http://doi.acm.org/10.1145/1162666.1162671.

[37] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, C. Zhai, Have things changed now?: an empirical study of bug characteristics in modern open source software, in: ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, ACM, New York, NY, USA, 2006, pp. 25–33. doi:http://doi.acm.org/10.1145/1181309.1181314.

[38] A. Ozment, S. E. Schechter, Milk or wine: does software security improve with age?, in: USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2006.