Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications

Theodoor Scholte[1], Davide Balzarotti[2], Engin Kirda[2,3]

[1] SAP Research, Sophia Antipolis
theodoor.scholte@sap.com

[2] Institut Eurecom, Sophia Antipolis
{balzarotti,kirda}@eurecom.fr

[3] Northeastern University, Boston
kirda@eurecom.fr

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.

1 Introduction

The web has become part of everyone's 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., [4][15][28]).

Organizations such as MITRE [15], SANS Institute [4] and OWASP [28] 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 [9, 14, 33–35], dynamic tainting [23, 24, 27], combination of dynamic tainting and static analysis [32], prevention by construction or by design [8, 13, 29, 36] and enforcement mechanisms executing within the browser [1, 7, 10, 31]. Some of these techniques have been commercialized and can be found in today's development toolsets. An example is Microsoft's FxCop [6] 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 result 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, by performing an automated analysis, we attempt to answer 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 to 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.

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 Injections 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 [19]. According to MITRE's FAQ [21], 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 [20].
- The name of the affected application, version numbers, and the vendor of the application represented by Common Platform Enumeration (CPE) identifiers [18].
- The impact and severity of the vulnerability according to the Common Vulnerability Scoring System (CVSS) standard [17].

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 need 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 fetch this information by using the disclosure date from the corresponding entry in the Open Source Vulnerability Database (OSVDB) [11].

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).

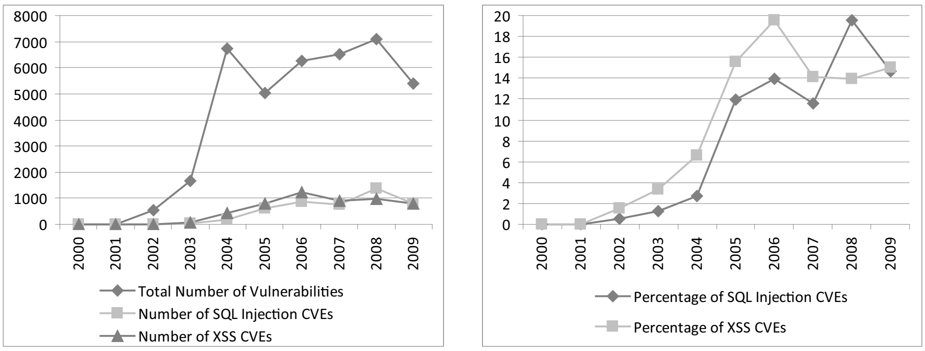
Although NVD provides a mapping between CVEs and CWEs, this mapping is not complete and many CVE entries do not have any classification at all. For this reason, we chose to perform a classification which is based on both the CWE classification and on the description of the CVE entry. In general, a CVE 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



(a) Vulnerability trends in numbers. (b) Vulnerability trends in percentages

Fig. 1: *Cross-Site Scripting and SQL Injection vulnerabilities over time.*

of using placeholders, sometimes, real examples of SQL or Javascript code may also be used. Two examples of attack strings are:

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

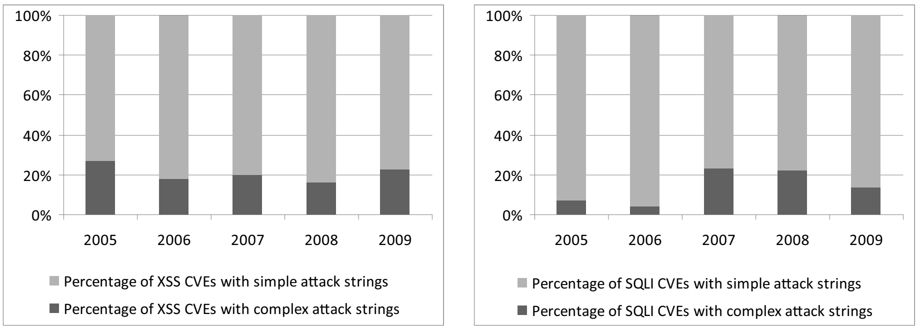
At the end of each line, note the placeholders that can be substituted with arbitrary code by the attacker.

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. 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 5222 Cross-Site Scripting entries and 4810 SQL Injection entries.

Figure 1a shows the number of vulnerability reports over time and figure 1b shows the percentage of reported Cross-Site Scripting and SQL Injection vulnerabilities over the total CVE entries.



(a) Cross-Site Scripting

(b) SQL Injection

Fig. 2: *Complexity exploits over time.*

Our first expectation based on intuition was to observe the number of reported vulnerabilities follow a classical bell shape: beginning with a slow 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, followed by a steep increase of Cross-Site Scripting and SQL Injection vulnerability reports in the years 2004, 2005 and 2006. 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"). Hence, web security threats such as Cross-Site Scripting and SQL Injection started receiving more focus after 2004.

Unfortunately, the number of reported vulnerabilities has not significantly decreased since 2006. In other words, the number of 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. For example, the attacker may have to carefully craft the malicious input in order to reach a subtle vulnerable functionality, or to pass certain input transformations (e.g., uppercase or character replacement). In particular, we are interested in identifying those cases in which the application developers were aware of the threats, but implemented insufficient, easy to evade sanitization routines.

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 extracted 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, more *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 [16][30].

In particular, we classify a Cross-Site Scripting attack string as complex (i.e., 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 contains 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

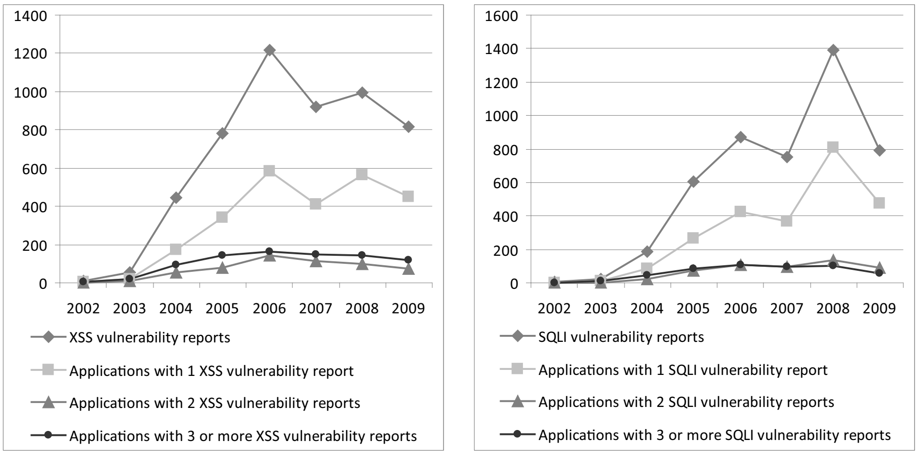
For SQL Injection attack strings, we looked at the following characteristics:

- 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 2a and 2b show the percentage of CVEs having one or more complex attack strings[4]. 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 of 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.

[4] 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.

Fig. 3: *The number of affected applications over time.*

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

3.2 Application Popularity

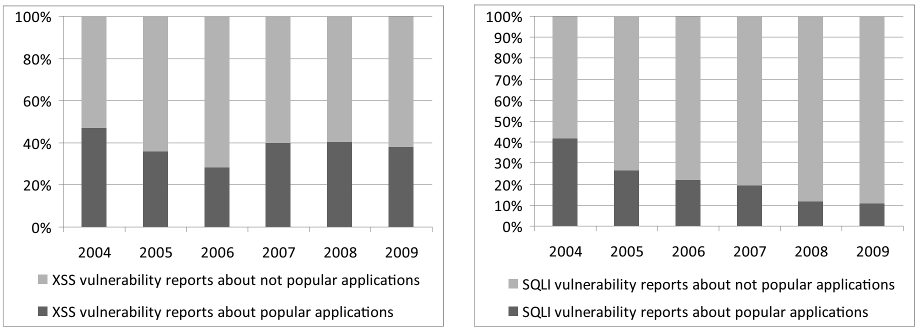
Since the complexity does not seem to explain the increasing number of reported vulnerabilities, we decided to focus on the type of applications. We started by extracting the vulnerable application's name from a total of 8854 SQL Injection and Cross-Site Scripting vulnerability reports in the NVD database that are associated to one or more CPE identifiers.

Figures 3a and 3b plot the number of applications 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. 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.

Based on this finding, 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 they are today. That is, do vulnerability reports nowadays tend to concentrate on less popular, or recently developed applications?



(a) Cross-Site Scripting

(b) SQL Injection

Fig. 4: *Vulnerability reports about applications and their popularity over time.*

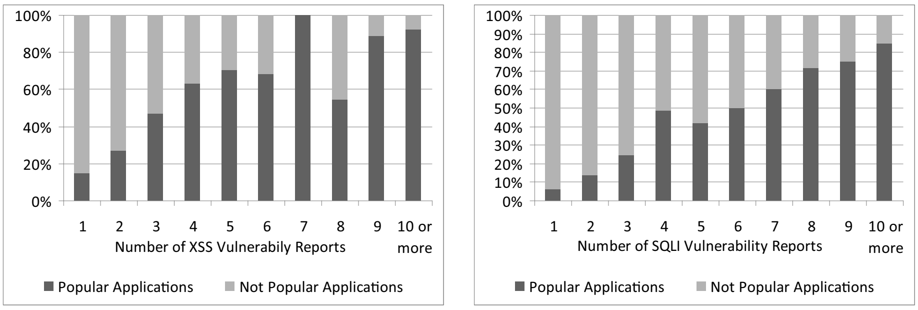
The first step consists of determining the popularity of these applications in order to be able to understand if it is true that popular products are more aware of (and therefore less vulnerable to) Cross-Site Scripting and SQL Injection attacks.

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 contain the name of the vendor or the name of the application, we flag the application as being ‘popular’. Otherwise, the application is 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 describe software products and not something else (e.g., when the product name also corresponds 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 4 shows the percentages of vulnerability reports that are associated with popular applications. 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 product are more security-aware. Unfortunately, according to Figure 4a, 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.

Even though the empirical evidence also does not support our second hypothesis, we noticed one characteristic that is common to both types of vulnerabilities: popular applications, probably because they are analyzed in more detail, typically have a higher number of reported vulnerabilities. The results, shown



(a) Cross-Site Scripting

(b) SQL Injection

Fig. 5: *Popularity of applications across the distribution of the number of vulnerability reports.*

in Figures 5a and 5b, suggest that it would be useful to investigate how these vulnerabilities have evolved in the lifetime of the applications.

3.3 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 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, in particular the well-known products, are becoming more secure. To verify this hypothesis, 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 changelogs, for each application, we extracted in which version a vulnerability was introduced and in which version the vulnerability was fixed. In order to obtain reliable insights into the vulnerabilities lifetime, we excluded the 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, and we also extract information from the version control systems (CVS, or SVN) of the different products.

Table 1a and Table 1b show a total of 145 Cross-Site Scripting and 52 SQL Injection vulnerabilities in the most affected applications. The tables distinguish *foundational* and *non-foundational* vulnerabilities. Foundational vulnerabilities

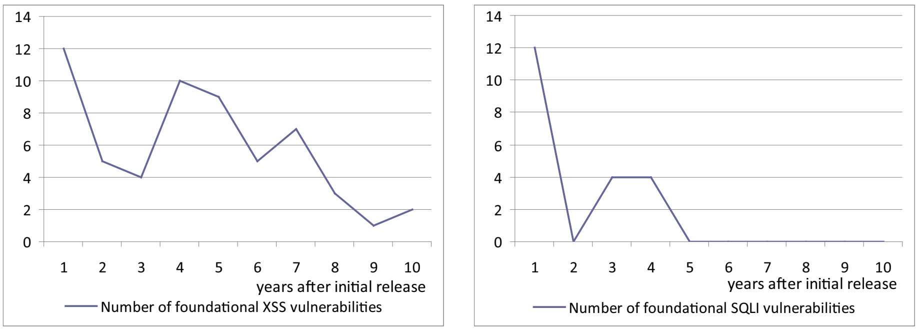
	Foundational	Non-Foundational
bugzilla	4	9
drupal	0	22
joomla	5	3
mediawiki	3	22
mybb	9	2
phorum	3	5
phpbb	4	2
phpmyadmin	14	13
squirrelmail	10	3
wordpress	6	6
Total	58	87

(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	2	1
punbb	3	3
wordpress	0	4
Total	20	32

(b) SQL Injection

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



(a) Cross-Site Scripting

(b) SQL Injection

Fig. 6: *Time elapsed between initial release and vulnerability disclosure.*

are vulnerabilities that were present in the first version of an application, while non-foundational vulnerabilities were introduced after the initial release.

We observed that 40% of the Cross-Site Scripting vulnerabilities are foundational and 60% are non-foundational. For SQL Injection, these percentages are 38% and 61%. 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 *foundational* vulnerabilities. Figure 6a and Figure 6b plot the number of foundational 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 year after the release of the product. For Cross-Site Scripting vulnerabilities, the result is quite different. Many foundational vulnerabilities are discovered even 10 years after the code was initially released. This observation suggests that it is very problematic to find Cross-Site Scripting vulnerabilities compared to SQL Injection vulnerabilities. We believe that this difference is caused by the fact that the attack surface for SQL Injection attacks is much smaller when compared with Cross-Site Scripting . Therefore, it is easier for developers to identify (and protect) all the sensitive entry points in the application code.

The difficulty of finding Cross-Site Scripting vulnerabilities is confirmed by the average elapsed time between the initial software release and the disclosure of foundational vulnerabilities. For SQL Injection vulnerabilities, this value is 2 years, while for Cross-Site Scripting is 4.33 years.

4 Related Work

Our work is not the first study of vulnerability trends based on CVE data. In [2], 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, [22] 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 [3] 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. [5]. 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 [12], 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, but do not consider web applications.

Ozment et al. [26] 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.

5 Discussion and Conclusion

Our findings in this study show that the complexity of Cross Site Scripting and SQL Injection exploits in vulnerability reports have not been increasing. Hence, this finding suggests that the majority of vulnerabilities are not due to sanitization failure, but due to the absence of input validation. Despite awareness programs provided by MITRE [19], SANS Institute [4] and OWASP [28], application developers seem to be neither aware of these classes of vulnerabilities, nor are able to implement effective countermeasures.

Furthermore, our study suggests that a main reason why the number of web vulnerability reports have not been decreasing is because many more applications are now vulnerable to flaws such as Cross-Site Scripting and SQL Injection . In fact, we observe 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. We believe that one of the reasons for this observation could be because SQL Injection problems may be easier to fix (e.g., by using stored procedures).

The empirical data we collected and analyzed for this paper supports the general intuition that web developers are bad at securing 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 about 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 [29] or managed languages such as C# or Java that prevent buffer overflow problems).

Acknowledgments

The research leading to these results was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) from the contract N 216917 (for the project MASTER), N 257129 (for the project PoSecCo) and N 257007. This work has also been supported by the POLE de Competitivite SCS (France) through the MECANOS project and the French National Research Agency through the VAMPIRE project. We would also like to thank Secure Business Austria for their support.

References

1. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 91–100, New York, NY, USA, 2010. ACM.

2. S. M. Christey and R. A. Martin. Vulnerability type distributions in cve. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007.
3. S. Clark, S. Frei, M. Blaze, and 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.
4. Rohit Dhamankar, Mike Dausin, Marc Eisenbarth, and James King. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/, 2009.
5. Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, New York, NY, USA, 2006. ACM.
6. Microsoft Inc. Msdn code analysis team blog. http://blogs.msdn.com/b/codeanalysis/, 2010.
7. Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.
8. Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure code generation for web applications. In Fabio Massacci, Dan S. Wallach, and Nicola Zannone, editors, *ESSoS*, volume 5965 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2010.
9. Nenad Jovanovic, Christopher Kruegel, and Engin 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*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
10. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.
11. Jake Kouns, Kelly Todd, Brian Martin, David Shettler, Steve Tornio, Craig Ingram, and Patrick McDonald. The open source vulnerability database. http://osvdb.org/, 2010.
12. Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang 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*, pages 25–33, New York, NY, USA, 2006. ACM.
13. Benjamin Livshits and Úlfar 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*, pages 95–104, New York, NY, USA, 2007. ACM.
14. V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.
15. Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/, 2010.
16. Ferruh Mavituna. Sql injection cheat sheet. http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/, 2009.
17. Peter Mell, Karen Scarfone, and Sasha Romanosky. A complete guide to the common vulnerability scoring system version 2.0. http://www.first.org/cvss/cvss-guide.html, 2007.
18. MITRE. Common platform enumeration (cpe). http://cpe.mitre.org/, 2010.

19. MITRE. Common vulnerabilities and exposures (cve). http://cve.mitre.org/, 2010.
20. MITRE. Common weakness enumeration (cwe). http://cwe.mitre.org/, 2010.
21. MITRE. Mitre faqs. http://cve.mitre.org/about/faqs.html, 2010.
22. Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, November 2010.
23. James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. The Internet Society, 2005.
24. Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.
25. Computer Security Division of National Institute of Standards and Technology. National vulnerability database version 2.2. http://nvd.nist.gov/, 2010.
26. Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
27. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In Alfonso Valdes and Diego Zamboni, editors, *RAID*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2005.
28. The Open Web Application Security Project. Owasp top 10 - 2010, the ten most critical web application security risks, 2010.
29. W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, pages 283–298. USENIX Association, 2009.
30. RSnake. Xss (cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html, 2009.
31. K. Vikram, Abhishek Prateek, and Benjamin 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*, pages 173–186, New York, NY, USA, 2009. ACM.
32. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni 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.
33. Gary Wassermann and Zhendong 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, June 2007. ACM Press New York, NY, USA.
34. Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.
35. Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
36. Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Better abstractions for secure server-side scripting. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 507–516, New York, NY, USA, 2008. ACM.