# Understanding Vulnerabilities in Plugin-based Web Systems: An Exploratory Study of WordPress

Oslien Mesa
Pontifical Catholic University of Rio de Janeiro Rio de Janeiro
orodriguez@inf.puc-rio.br

Reginaldo Vieira
Federal University of São João del-Rei
reginaldo.vieira@ufsj.edu.br

Marx Viana
Pontifical Catholic University of Rio de Janeiro Rio de Janeiro
mleles@inf.puc-rio.br

Vinicius H. S. Durelli
Federal University of São João del-Rei
durelli@ufsj.edu.br

Elder Cirilo
Federal University of São João del-Rei
elder@ufsj.edu.br

Marcos Kalinowski
Pontifical Catholic University of Rio de Janeiro Rio de Janeiro
kalinowski@inf.puc-rio.br

Carlos Lucena
Pontifical Catholic University of Rio de Janeiro Rio de Janeiro
lucena@inf.puc-rio.br

## ABSTRACT

A common software product line strategy involves plugin-based web systems that support simple and quick incorporation of custom behaviors. As a result, they have been widely adopted to create web-based applications. Indeed, the popularity of ecosystems that support plugin-based development (e.g., WordPress) is largely due to the number of customization options available as community-contributed plugins. However, plugin-related vulnerabilities tend to be recurrent, exploitable and hard to be detected and may lead to severe consequences for the customized product. Hence, there is a need to further understand such vulnerabilities to enable preventing relevant security threats. Therefore, we conducted an exploratory study to characterize vulnerabilities caused by plugins in web-based systems. To this end, we went over WordPress vulnerability bulletins cataloged by the National Vulnerability Database as well as associated *patches* maintained by the WordPress plugins repository. We identified the main types of vulnerabilities caused by plugins as well as their impact and the size of the patch to fix the vulnerability. Moreover, we identified the most common security-related topics discussed among WordPress developers. We observed that, while plugin-related vulnerabilities may have severe consequences and might remain unnoticed for years before being fixed, they can commonly be mitigated with small and localized changes to the source code. The characterization helps to provide an understanding on how typical plugin-based vulnerabilities manifest themselves in practice. Such information can be helpful to steer future research on plugin-based vulnerability detection and prevention.

*SPLC'18, 10–14 September, 2018, Gothenburg, Sweden*
© 2018 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

## CCS CONCEPTS

• **Security and privacy** → **Web application security**;

## KEYWORDS

Software Product Lines, Plugin-based Web Systems, Security, Exploratory Study

**ACM Reference format:**
Oslien Mesa, Reginaldo Vieira, Marx Viana, Vinicius H. S. Durelli, Elder Cirilo, Marcos Kalinowski, and Carlos Lucena. 2018. Understanding Vulnerabilities in Plugin-based Web Systems: An Exploratory Study of WordPress. In *Proceedings of 22nd International Conference on Software Product Line, Gothenburg, Sweden, 10–14 September, 2018 (SPLC'18),* 11 pages.
https://doi.org/10.475/123_4

## 1 INTRODUCTION

Plugins are extensions that complement software applications with new custom-tailored behaviors. Plugin-based web systems provide developers with a number of benefits; offering a wide range of plugins to choose from is the hallmark of moderns web systems as WordPress[1], Drupal[2], and Joomla[3]. Notably, WordPress offers a myriad of plugins (more than 52,000) and has been widely used to build thousands of customized web applications. Currently, it is used by 28.9% of all web applications monitored by the Web Technology Surveys.[4] Thus, plugin-based web systems have become mainstream and developers have turned their attention to exploring their potential to conceive web applications by composing, configuring, and extending a set of plugins.

The ability to use plugins to extend web systems with additional and custom behaviors has both positive and negative implications for developers. First, there are many customization options from which to choose, thus developers are able to quickly and easily build web applications that are custom-tailored to the users' needs,

[1]wordpress.org
[2]www.drupal.org
[3]www.joomla.org
[4]w3techs.com

allowing for rapid prototyping and development. However, web applications can also be conceived as a complex composition of multiple plugins from different sources, which raises a number of concerns related to plugins security. Indeed, several WordPress-based applications may be exposed to many types of vulnerability due to security problems in existing plugins. For example, the TimTumb Plugin that was used by millions of WordPress-based applications has reportedly been open to an exploit where attackers could deliberately execute remote code[5].

As evidenced in [14], it is somewhat common for web applications to be rife with security vulnerabilities, some of which might lead to critical problems. Approximately 15% of the vulnerabilities in web applications can be regarded as critical according to the National Vulnerability Database (NVD)[6] and almost 35% of them are easily exploitable vulnerabilities [2]. Moreover, as pointed out by Telang and Wattal [43], vendors are negatively affected by security-related vulnerability announcements in their software applications. According to Telang and Wattal, the more severe the vulnerability, the greater the adverse impact on a software company's stock price. Additionally, the fall in stock price is exacerbated if developers fail to release a path at the time of disclosure. Therefore, vendors have an incentive to strive for creating more secure software systems. In order to cope with this problem, an in-depth characterization of vulnerabilities can help researchers and practitioners to develop proper vulnerability prevention and detection tools. To this end, we set out to investigate security vulnerabilities found in the "wild" and reported over the past 12 years in order to provided a broad understanding of the consequences and implications associated with vulnerabilities caused by plugins in web systems. We probed into the characteristics of vulnerabilities (e.g., most common types and severity) and the effort (mostly in terms of code churn [40]) required to fix these security threats. We also looked at the amount of time plugin-related vulnerabilities remain on the code util to be fixed. Furthermore, to gain insights into the WordPress development community, we decided to analyze the most common security-related topics discussed among developers. Our exploratory study makes the following contributions:

- We identified the main types of security vulnerabilities caused by plugins and found that most of them are critical and might tamper with the integrity of web applications. The most common vulnerabilities are Cross-site Scripting, SQL Injection, and Cross-site Request Forgery. Additionally, our results would seem to indicate that, unlike other types of vulnerabilities, the occurrence of these vulnerabilities has not declined over time.
- According to the NVD bulletins we analyzed, the most dangerous vulnerability type seems to be SQL Injection, which can be easily exploited by attackers. In this context, the results of our investigation also suggest that database security is by far the most popular topic discussed among plugin developers, with more than 290 questions.
- Considering the patches we analyzed, we found that most changes occur in a single line and mostly entail invoking a sanitization method. Moreover, our results show that it takes a considerable number of days for fixing a vulnerability: on average, it could remain unnoticed for 653 days before being identified and then fixed.

[5]nvd.nist.gov/vuln/detail/CVE-2011-4106
[6]cve.mitre.org

The remainder of this paper is organized as follows: Section 2 presents background on plugin-based web systems and web security. Section 3 details how we carried out our study. Section 4 outlines the results of our study. Section 5 provides a discussion aimed at further analyzing our results. Section 6 describes threats to the validity of our study. Finally, Section 7 presents some concluding remarks.

## 2 BACKGROUND

In order to lay the foundation for the rest of our paper, in this section we provide some background on plugin-based systems and give an overview of vulnerabilities in web applications, highlighting what has already been investigated in this area.

### 2.1 Plugin-based Web Systems

Internet and web technologies have become an essential part of modern-day society. Recent advances in web technologies and the rapid expansion of the Internet have made it possible for web applications to handle the storage, exchange, and retrieval of a sizable amount of information, some of which may be sensitive. This rapid expansion also offered opportunities for developers to embrace software product line engineering methodologies such as plugin-based development [42] [12].

The idea behind plugin-based development is that plugins offer additional features to extend the core capabilities provided by the software systems. The core is usually maintained by a set of long-term developers while third parties develop a number of plugins around the core of the software system [12]. Therefore, over the last years, web development has evolved into a process of amalgamation of many extensions developed by different people [21]. This is beneficial for both the provider of the plugin-based system and the plugin provider. The software system provider can focus on the core functionality and the plugin provider does not have to be concerned about the plugin infrastructure, as this is provided by the system's core functionality.

It turns out that a plugin-based development is a promising engineering methodology to build complex web applications that may present several variability and have to be easily customizable [29]. Due to these benefits, plugin-based web systems (e.g., WordPress, Joomla,Drupal) have been extensively used to create a vast range of web applications. In particular, Wordpress provides a plugin infrastructure to create a variety of software applications which is used by 28.9% of all web applications monitored by the Web Technology Surveys. Moreover, WordPress is also supported by a global community of thousands of commercial and open source software developers as well as enthusiasts, which makes this specific plugin-based system an interesting case to be investigated.

### 2.2 Security Vulnerabilities

The complexity of web applications has been increasing at a rapid pace accompanied by the disclosure of many more security vulnerabilities. Accordingly, security has become an increasing recognition of non-functional requirements [35] [3]. In this study, we consider

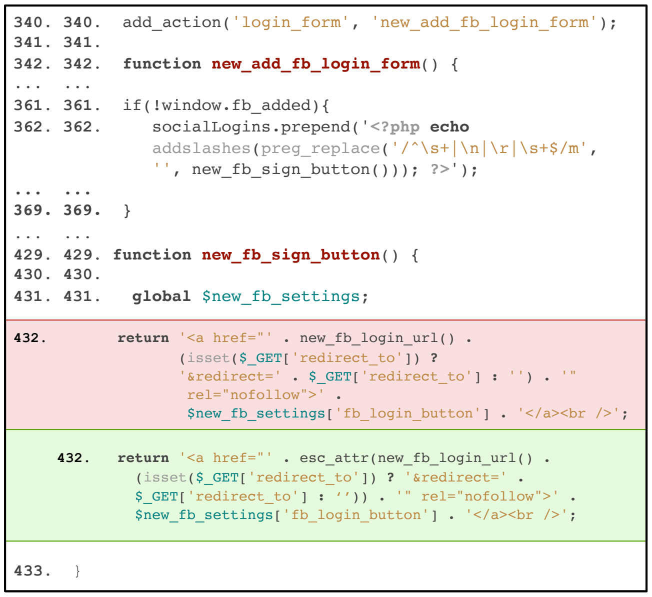


Figure 1: Illustrative Example

```
340. 340.   add_action('login_form', 'new_add_fb_login_form');
341. 341.
342. 342.   function new_add_fb_login_form() {
...  ...
361. 361.   if(!window.fb_added){
362. 362.      socialLogins.prepend('<?php echo
           addslashes(preg_replace('/^\s+|\n|\r|\s+$/m',
           '', new_fb_sign_button())); ?>');
...  ...
369. 369.   }
...  ...
429. 429.  function new_fb_sign_button() {
430. 430.
431. 431.   global $new_fb_settings;

432.        return '<a href="' . new_fb_login_url() .
                  (isset($_GET['redirect_to']) ?
                  '&redirect=' . $_GET['redirect_to'] : '') . '"
                   rel="nofollow">' .
                   $new_fb_settings['fb_login_button'] . '</a><br />';

     432.   return '<a href="' . esc_attr(new_fb_login_url() .
             (isset($_GET['redirect_to']) ? '&redirect=' .
             $_GET['redirect_to'] : '')) . '" rel="nofollow">' .
             $new_fb_settings['fb_login_button'] . '</a><br />';

433.   }
```

a vulnerability as a failure that allows attackers to exploit the software application in a way not foreseen by the developer [4]. That is, a vulnerability opens up opportunities to potential misuse of a given software application. Since web applications contain sensitive information, vulnerabilities can be exploited to gain access to information or to sabotage their operation. The OWASP Top 10 [7] raises awareness about the majority of security vulnerabilities in web applications. Some common types of vulnerabilities are Improper Input Validation and Unauthorized Access. It is worth mentioning, however, that web applications can be exposed to many other types of vulnerabilities [20] [38].

Due to their complexity, plugin-based web systems are also plagued by vulnerabilities [44]. For example, Wordpress provides a infrastructure to which plugins can contribute new features by modifying shared global state or running a function at a specific point in the execution of the core. A plugin vulnerability can arise when the plugin sends malicious input data to core components without proper sanitization or input validation. The Figure 1 illustrates a vulnerability[8] introduced by a WordPress plugin (line 432) and the respective patch[9]. In our example, in Figure 1, the plugin introduces a new action hook that is triggered through the "password" field in the login form, allowing for the customization of the built-in WordPress login form and letting visitor to manage web applications using their social profiles (e.g., Facebook, Google, Twitter). In this example, the `new_fb_sign_button` function (line 342) leads to a Cross-site Scripting (XSS) vulnerability, which allows remote attackers to inject arbitrary script via the `redirect_to` parameter.

The Wordpress login form works properly in isolation, but the introduction of the aforementioned plugin causes a vulnerability as shown in Figure 1. Given that WordPress allows millions of developers across the world to create their own plugins, it is unrealistic to anticipate all possible plugin installations, interactions, and combinations in a specific web application. As a result, it is clear that security can be violated by unanticipated plugins interactions with the core, creating vulnerabilities that can be potentially exploited by attackers [20] [45] [26]. Thus, the security of these web systems hinges on keeping both core and plugin vulnerabilities in check.

Walden et al. [44] carried out a study to observe whether plugins are the source of vulnerabilities present in web systems and they confimed that most of the vulnerabilities (92%) appear in plugin code. In addition, according to their results, plugin code differs from core code in terms of the types of vulnerabilities present in the results. Vulnerabilities such as Cross-site Scripting and Cross-site Request Forgery were more prevalent in plugin code, while others vulnerabilities were more prevalent in core code. Similarly, Fonseca and Vieira [20] suggest that many plugins have Cross-site Scripting and SQL injection vulnerabilities that can be easily exploited. Moreover, they found that the coverage analysis performed by static analysis tools is inefficient and these tools often report false positives. Koskinen et al. [26] analyzed 322 randomly selected WordPress plugins. More specifically, they compared the amount of potential vulnerabilities and vulnerability density to the user ratings in hopes to determine if user ratings can be used to select secure plugins. The results suggest that the quality of plugins varies and that there is no correlation between the ratings of plugins and the amount of vulnerabilities found in them.

We studied a large set of vulnerabilities (875), taking into account other perspectives. We included in our study the frequency of the most recurrent types of vulnerability, the severity of plugin-related vulnerabilities in terms of their access complexity and impact on the availability, integrity, and confidentiality. Moreover, we also studied the complexity of the changes required to remove a vulnerability in terms of change types and amount of modified files, and we also looked at the survival time of vulnerabilities. Finally, we present additional insights collected from the WordPress development community. The results from our analysis enabled us to obtain an in-depth characterization of plugin-related vulnerabilities.

## 3 STUDY DESIGN

We conducted an exploratory study to quantitatively characterize plugin-related vulnerabilities on web systems. In particular, we investigated the occurrence, severity, complexity, and survival time of vulnerabilities caused by the multitude of plugins available for WordPress. We also looked into the most common security-related topics discussed among developers in Q&A websites.

### 3.1 Goal and Research Questions

Our goal is to shed some light on the plugin-related security vulnerabilities found in the "wild" and reported over the past 12 years. We used the organization proposed by the Goal/Question/Metric (GQM) [9] to define the goals of our study. According to the proposed goal definition template, the scope of our study can be summarized as described below. In addition, based on our goal, we came up with the a set of research questions (RQs), which are also presented below.

[7]www.owasp.org/index.php/Top_10_2017-Top_10
[8]nvd.nist.gov/vuln/detail/CVE-2015-4413
[9]plugins.trac.wordpress.org/changeset/1178736

**Analyze** plugin-related security vulnerabilities
**with the purpose of** characterization
**with respect to** vulnerability types, criticality, patch size, survival time, and discussed topics
**from the point of view of the** developers
**in the context of** plugin-related security vulnerabilities reported for WordPress (cataloged by the National Vulnerability Database), associated patches (maintained by the WordPress plugins repository), and discussions among WordPress developers (questions and answers posted on Stack Exchange targeted at WordPress developers).

- **RQ$_1$:** What are the main types of security vulnerabilities caused by WordPress plugins?
- **RQ$_2$:** How critical are the security vulnerabilities caused by WordPress plugins?
- **RQ$_3$:** What is the patch size to fix WordPress plugin vulnerabilities?
- **RQ$_4$:** How long does a vulnerability survive in the WordPress plugins code?
- **RQ$_5$:** What are the most common security-related topics discussed among WordPress developers?

## 3.2 Data Extraction and Analysis

In this subsection we outline our data extraction procedure and the analysis method employed.

*3.2.1 Vulnerability Bulletins.* As mentioned, our study was based on 895 vulnerability bulletins registered in NVD. The information was collected in September 2017 by downloading the entire NVD data as JSON Feeds. We then build a parser that went over all bulletin descriptions and collected the ones in which the keywords "*Wordpress*" and "*Plugin*" occurred together in a given sentence. After reviewing the bulletin descriptions for a second time, our parser extracted the type and severity of the vulnerabilities in each bulletin. Finally, we examined the resulting data to answer RQ$_1$ and RQ$_2$.

To classify the vulnerabilities in the bulletins, NVD employs the Common Weakness Enumeration Specification (CWE) [27]. CWE offers a community-developed list of security vulnerabilities that appear in architecture, design, or code. For example, as mentioned, Cross-site Scripting is a common type of vulnerability related to the improper neutralization of input in web applications, which can be introduced at design and implementation time. Therefore, CWE classifies Cross-site Scripting (CWE-79) as a *Data Processing Error*, falling into the category of *Improper Encoding or Escaping of Output*. Using CWE as a common language to classify and describe vulnerabilities, we analyzed the vulnerabilities aspects (Type) and counted the number of times each vulnerability type (e.g., SQL Injection, Path Traversal) appeared in the bulletins.

Similarly, NVD uses the Common Vulnerability Scoring System (CVSS) [28] as a quantitative model to consistently measure and communicate vulnerability severity scores. Currently, the CVSS captures the severity of vulnerabilities by considering how they can be accessed and whether or not extra conditions are required to exploit them. In our study we considered the following relevant measures to exam and answer RQ$_2$: *Integrity Impact* – the impact to integrity when a vulnerability is exploited successfully; *Availability Impact* – impact to availability of a successful exploited vulnerability, which evaluates the impact of attacks that consume computational resources (e.g., network bandwidth use, disk space, or processor cycles), and *Complexity Impact* – the complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target application.

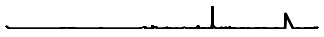
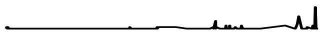
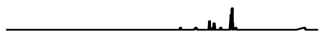
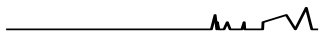
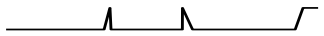
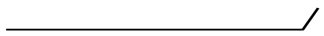
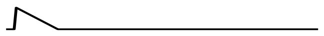
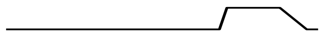
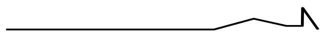
Based on the CVSS model, we analyzed vulnerability severity according to the data registered in the NVD bulletins:

- **Integrity**: **None** - there is no impact on integrity; **Partial** - it is possible to make modifications to files or gain access to application information; however, either the attacker has no control over what can be modified or the attacker's scope is limited; **Complete** - there is a total compromise of application integrity.
- **Availability**. **None** - no impact on web application availability; **Partial** - reduction in performance, or partial interruptions in available resources; **Complete** - stop of affected resources.
- **Confidentiality**. **None** - no impact on web application confidentiality; **Partial** - considerable informational disclosure; **Complete** - total information disclosure.
- **Complexity**. **Low** - does not demand expertise; **Medium** - requires a little expertise; and **High** - demands high expertise.

*3.2.2 Patches.* NVD also promotes the incorporation of links to other web sites with complementary information of interest to developers and users. Thus, in order to measure the size of the path to fix plugin-related vulnerabilities and their survival time, in the third inspection of the vulnerability bulletins, we extracted information related to patches posted in the WordPress Plugin repository. This repository supports the development of over 52,000 WordPress-related plugins and provides a set of development tools – a wiki, a version control system, and a bug tracker.

From the 895 bulletins previously collected, we found that 109 report on patches. In the context of our study, we consider a patch as a collection of source code modifications designed to fix or improve a WordPress plugin. Given that some patches also include improvement-related changes, to answer RQ$_3$ and RQ$_4$, two authors manually went over all patches to sort out the vulnerability-related changes from the improvement-related ones. Afterwards, we computed the number of modified, added, or deleted line of code. We ignored empty lines (i.e.m, black lines) and statements spanning over multiples lines were considered as single lines. These metrics have been successfully employed in the literature to measure and predict software system defect density and vulnerabilities [30] [13] [32] [40]. To compute the survival time of vulnerabilities, we proceeded as follow: *(i)* for each file identified as involved in the vulnerability-fixing commit we extracted the file revision before the vulnerability fixing; *(ii)* starting from such revision, for each source code line changed to fix the vulnerability, we went over all previous revisions until we noticed the introduction of these

**Table 1: Common vulnerability types with their frequency, time of introduction and CWE code**

| Type | Total | Time of Introduction | Frequency |
|---|---|---|---|
| Cross-site Scripting (CWE-79) | 397 | Implementation |  |
| SQL Injection (CWE-89) | 166 | Implementation / Operation |  |
| Cross-Site Request Forgery (CWE-352) | 120 | Implementation |  |
| Path Traversal (CWE-22) | 48 | Implementation |  |
| Management of Permissions (CWE-264) | 28 | Design |  |
| Improper Input Validation (CWE-20) | 21 | Implementation |  |
| Code Injection (CWE-94) | 20 | Implementation |  |
| Information Exposure (CWE-200) | 19 | Implementation |  |
| Unrestricted Upload (CWE-434) | 11 | Implementation |  |

respective lines. This revision is considered as the vulnerability-introducing commit. Finally, we calculated the number of days between the vulnerability-introducing commit and the vulnerability-fixing commit. Essentially, the survival time represents a major indicator of the risk of not fixing a vulnerability within a given time frame [16] [18].

*3.2.3 Discussions from Q&A Websites.* To answer RQ$_5$ we analyzed questions posted on Stack Exchange targeted at WordPress developers.[10] WordPress Stack Exchange allows users to post questions concerning theme and plugin development, best practices, and server configuration. Each question has a title, which summarizes the main concern, and a body detailing the problem. Users can also attach up to five tags when asking a question. A tag is a keyword that categorizes similar questions and helps users to find and answer relevant questions.

Using the Stack Exchange application programming interface (API) [11], we extracted questions and answers posted between August 2010 and October 2017. To collect questions related to security issues we limited our search to questions in which the keyword "plugin" appeared along with "vulnerability", "hacked", "security", "insecure", or "safe". We looked for these keywords both in the title and in the body of questions. Next, we manually eliminated non-related questions. We extracted information from the titles, bodies, number of views, scores, and lists of answers from each of the selected questions. Prior to topic modeling, the extracted information was pre-processed: we removed any code snippet enclosed in `<code>...</code>`. In the context of our research, code snippets do not convey much helpful information. Additionally, we removed all HTML tags, all stop words, and applied Porter Stemming to map each word to their base form.

[10]wordpress.stackexchange.com
[11]api.stackexchange.com

*Topic Modeling.* In hopes of extracting the most common security-related topics discussed among developers, we applied the Latent Dirichlet Allocation (LDA) technique [11]. The LDA is a statistical topic modeling technique that describes documents as mixtures of topics, wherein a topic is a bag of connected words. To train LDA, we used MALLET Toolkit [12]. We adopted the common guidelines recommended by Griffiths and Steyvers [23], by running LDA for 1,000 Gibbs sampling iterations. We also allowed some topics to be more prominent than others, which led to better and cohesive topics. We chose more coarse-grained topics, so that the topics could properly capture the general, perceived, and potential security concerns discussed by WordPress developers. As the number of topics is an option that controls the granularity of topics, we configured the number of topics to be equal to 10. Finally, we also assigned a label to each topic based on the most representative keywords [8].

## 4 RESULTS

In this section we frame our discussion around the RQs described in the previous sections. In Section 5, we discuss the results in a broader context.

### 4.1 RQ$_1$ - What are the main vulnerabilities caused by WordPress plugins?

As mentioned, to answer RQ$_1$, we analyzed some aspects reported on vulnerability bulletins. First, we probed the vulnerability types and whether their frequency has been increasing or decreasing over the years. Second, we also investigated when they are normally introduced during the development life cycle. Table 1 lists the most common vulnerability types, their frequency, and time of introduction. In the context of our study, vulnerabilities that occurred at least ten times are considered "prevalent".

[12]mallet.cs.umass.edu

**Table 2: The impact of vulnerabilities in accordance with the CVSS model by type. L means Low, M means Medium, H means High, N means None, P means Partial and C means Complete**

| Type | Complexity | | | Integrity | | | Availability | | | Confidentiality | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | L | M | H | N | P | C | N | P | C | N | P | C |
| Data Processing Errors (CWE-19) | 1 | - | - | - | 1 | - | 1 | - | - | - | 1 | - |
| Improper Input Validation (CWE-20) | 12 | 9 | - | 1 | 18 | 2 | 9 | 10 | 2 | 5 | 14 | 2 |
| Information Exposure (CWE-200) | 19 | - | - | 19 | - | - | 19 | - | - | - | 19 | - |
| Path Traversal (CWE-22) | 46 | 2 | - | 43 | 4 | 1 | 44 | 4 | - | 3 | 43 | 2 |
| Security Features (CWE-254) | 1 | - | - | - | 1 | - | 1 | - | - | 1 | - | - |
| Access Controls (CWE-264) | 23 | 5 | - | 5 | 21 | 2 | 13 | 13 | 2 | 11 | 15 | 2 |
| Improper Access Control (CWE-284) | 6 | - | - | 2 | 4 | - | 4 | 2 | - | 3 | 3 | - |
| Improper Authorization (CWE-285) | 2 | - | - | - | 2 | - | 2 | - | - | 2 | - | - |
| Improper Authentication (CWE-287) | 4 | 1 | - | - | 5 | - | 1 | 4 | - | 1 | 4 | - |
| Cross-Site Request Forgery (CWE-352) | 2 | 117 | 1 | 2 | 117 | 1 | 7 | 112 | 1 | 5 | 114 | 1 |
| Unrestricted Upload of File (CWE-434) | 11 | - | - | - | 10 | 1 | 3 | 7 | 1 | 3 | 7 | 1 |
| Files Accessible to External Parties (CWE-552) | 1 | - | - | 1 | - | - | 1 | - | - | - | 1 | - |
| Link Following (CWE-59) | 1 | - | - | 1 | - | - | 1 | - | - | - | 1 | - |
| Open Redirect (CWE-601) | - | 1 | - | - | 1 | - | 1 | - | - | - | 1 | - |
| Command Injection (CWE-77) | 2 | - | - | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 |
| OS Command Injection (CWE-78) | 1 | - | - | - | 1 | - | - | 1 | - | - | 1 | - |
| Cross-site Scripting (CWE-79) | 1 | 388 | 8 | - | 397 | - | 397 | - | - | 397 | - | - |
| SQL Injection (CWE-89) | 160 | 6 | - | - | 166 | - | - | 166 | - | - | 166 | - |
| Code Injection (CWE-94) | 14 | 6 | - | - | 18 | 2 | - | 18 | 2 | - | 18 | 2 |
| NOINFO | 11 | - | - | - | 8 | 3 | 4 | 4 | 3 | 4 | 4 | 3 |
| OTHER | 16 | 16 | 1 | 2 | 29 | 2 | 8 | 23 | 2 | 2 | 29 | 2 |
| Total | 334 | 551 | 10 | 78 | 804 | 15 | 516 | 365 | 14 | 437 | 442 | 16 |

Our results indicate that an extremely limited number of vulnerability types occur and most of them are introduced at implementation time. Out of the 705 software vulnerabilities in CWE, only 9 are regularly caused by WordPress plugins. As shown in Table 1, the three most common types of vulnerability are Cross-site Scripting with 397 occurrences (43.72%); SQL Injection with 166 occurrences (18.28%), and Cross-Site Request Forgery with 120 occurrences (13.21%). The least common type is Unrestricted Upload (1.21%), which allows arbitrary code to be uploaded and executed by the web application. Interestingly, although PHP is well-known for supporting the execution of arbitrary code on the server side without directly invoking the host operating system, we observed that this type of vulnerability is not so common in web system plugins.

The results also show that the frequency of some types of vulnerability can change over time. When investigating Table 1, we discovered four evolution patterns of vulnerability caused by plugins in web systems: *(i) persistent vulnerability*, which manifest itself often along the years (P1); *(ii) occasional vulnerability*, which occurs at irregular intervals over the years and thus has been reappearing from time to time (P2); *(iii) once occasional vulnerability but now lays "dormant"*, which is a vulnerability that manifested itself then almost disappeared, after which it has not manifested itself again over and over (P3); and *(iv) occasional vulnerability that has become prevalent*, which is a vulnerability that appeared recently and since then has become prevalent in plugins (P4).

Notably, the majority of the vulnerability types caused by plugins in WordPress falls into pattern P1: meaning that 76.31% of these vulnerabilities keep cropping up through the years and that security is still not increasing over time. We conjecture that vulnerabilities that follow pattern P1 are the easiest to be introduced. For instance, Cross-site Scripting and SQL Injection are types of vulnerability in pattern P1. The former, according to [2], is very common in web applications because it requires a great deal of knowledge about program construction and thus it should be avoided altogether. Indeed, whenever developers are negligent during the PHP code writing process, they are prone to lapses that may result in the introduction of Cross-site Scripting vulnerabilities into plugin code. Regarding SQL Injection, we surmise it as a common vulnerability because it is easily detected and exploited. Moreover, as observed in [2], any web application with even a minimal user base is likely to be subject to an attempted attack of SQL Injection.

We also noted reduced number of vulnerabilities (10.66%) associated with patterns P2, P3, and P4. Code injection is the only type in P3. We are convinced that code injection was once common and then almost disappeared because developers started avoiding unsafe resources as `eval()`. However, in contrast, other vulnerability types associated with improper input validation as Path Traversal and Unrestricted Upload have become prevalent. We understand that proper input validation is still a major challenge faced by plugin developers, specially because many of the recently developed plugins have been created by not-so-skilled developers [41].

Management of Permissions is the only type in P2. Although, vulnerabilities related to permission management have appeared from time to time, our results indicate that recently there seems to be a peak in this type of vulnerability.

## 4.2 $RQ_2$ - How critical are the vulnerabilities caused by WordPress plugins?

The main reason to be concerned about security is because a vulnerability may negatively influence web application in a number of ways. For example, successful exploitation of a vulnerability could lead to a complete loss of confidentiality, but no loss of availability. Thus, as presented in Section 4, we analyzed the impact of vulnerabilities taking into account the degree to which they lead to loss of confidentiality, integrity, and availability in accordance with the CVSS model. In addition, we also investigated the complexity of the attack required to successfully exploit the vulnerability. Table 2 summarizes the impact of the vulnerabilities examined in this study.

Out of the 895 vulnerability bulletins analyzed, only 78 (8.71%) indicated that the vulnerability would not lead to a significant compromise of the web application integrity. However, 804 bulletins (89.83%) suggested that the vulnerability had the potential to allow attackers unrestricted access to files or to information maintained by the vulnerable web application, even when the scope the attackers could have had access to was limited. Only 15 reports (1.67%) described vulnerabilities that had the potential to jeopardize the integrity of the web application as a whole. Therefore, our results suggest that the majority of the vulnerabilities have to potential to lead to data breaches while a few vulnerabilities do not expose web application to serious security risks. Moreover, fewer vulnerabilities (less than 2%) had the potential to render the whole application vulnerable.

We also looked at the impact of the vulnerabilities on the availability and confidentiality of web application. As shown in Table 2, the number of vulnerabilities that do not affect, or that only partially affect the web application are somewhat similar. The results show that 516 (57.65%) vulnerabilities do not compromise availability. Nevertheless, according to the reports, 365 (40.78%) vulnerabilities have the potential to negatively affect the performance of the web application or partially render resources unavailable. Similar results can also be observed when we consider confidentiality. A considerable number of vulnerabilities, that is, 437 (48.82%) do not affect the confidentiality of the web application while 442 (49.38%) of them enable the attacker to access some system files. Only in 16 (1.78%) reports the vulnerability poses a security breach that allows the attacker to take over the vulnerable application and access sensitive data.

As shown in Table 2, 551 (61.56%) vulnerabilities require some special access conditions. However, in 334 (37.31%) cases, the vulnerabilities introduced by plugins do not impose any restriction to attackers nor hamper their access to certain files. That is, any attack can be performed manually and requires little, to no skill to be carried out. Indeed, SQL Injection (CWE-89) seems to be the most dangerous vulnerability type. Although, the reports indicate that SQL Injection attacks only partially affect the integrity, availability, and confidentially of the systems, most of the cases suggest that these vulnerabilities can be easily exploited by attackers.

**Table 3: Descriptive Statistics: Patches**

| | Lines Added | Lines Changed | Lines Removed | Files |
|---|---|---|---|---|
| Min | 0 | 0 | 0 | 1 |
| Max | 20 | 35 | 284 | 18 |
| Mean | 1.76 | 2.51 | 5.21 | 1.49 |
| Median | 0 | 1 | 0 | 1 |
| Std. Dev. | 3.62 | 4.26 | 30.90 | 1.76 |

## 4.3 $RQ_3$ - What is the patch size to fix WordPress plugin vulnerabilities?

We analyzed the impact of maintaining vulnerabilities caused by plugins based on the patches available on the WordPress plugins repository. Among the patches, 42% involved the addition of one or more lines of code, 67% the change of one or more existing lines of code, and 18% removing code. Therefore, our results (see Table 3) seem to suggest that, on average, only one change is needed to fix a vulnerability. While removing actions are frequently used to remove some files that are not needed but introduced a vulnerability, the majority of the changes were mostly simple ones like a call to a sanitization method, as illustrated in Figure 1. Moreover, according to the standard deviation, the size of the patches in terms of the number of files does not seem to vary considerably. Therefore, considering the 119 patches analyzed, we noted that most changes occur in only a few files. On average, fixing a vulnerability entails changing only one file. According to our analysis, the worst case scenario involves having to change up to 18 files.

Analyzing the top three patches we observed that they are large because their fix entailed sanitizing a large volume of parameters and creating supporting procedures to avoid SQL Injections. For example, the multiple SQL Injection vulnerabilities in the Simple Ads Manager plugin[13] was fixed in patch 1136202[14], which entailed adding and modifying approximately 35 lines of code spread across 2 files to avoid remote attackers to execute arbitrary SQL commands via several input parameters. Similarly, to fix two vulnerabilities[15] [16] in the Video Embed & Thumbnail Generator plugin – one allowing attackers to obtain the installation path via unknown vectors and another allowing remote attackers to execute arbitrary commands – the developers in patch 507924 [17] had to change almost 72 lines of code by implementing the Media Upload Form and Generator functionalities to basically process input data. This more in-depth analysis, however, indicates that the patch size of fixing post-release security vulnerabilities is significant only in some very specific cases.

[13]nvd.nist.gov/vuln/detail/CVE-2015-2824
[14]plugins.trac.wordpress.org/changeset/1136202
[15]nvd.nist.gov/vuln/detail/CVE-2012-1786
[16]nvd.nist.gov/vuln/detail/CVE-2012-1785
[17]plugins.trac.wordpress.org/changeset/507924

**Table 4: Descriptive Statistics: Survivability**

| | Number of Days |
|---|---|
| Min | 1 |
| Max | 1710 |
| Mean | 563.41 |
| Median | 470 |
| Std. Dev. | 1.34 |

## 4.4 RQ$_4$ - How long does a vulnerability survive in the WordPress plugins code?

In general, the lifetime of a vulnerability appears to transit through distinct states, variating from the introduction, passing by the disclosure, until the release of a patch fixing the security threat [5]. The introduction (e.g. the birth of the vulnerability) occurs unintentionally during development. We consider the vulnerability to be introduced only after the release of the code. The vulnerability is disclosed when the discoverer reveals details of the security threat to a wider audience. Finally, a vulnerability is fixed when the developer releases a patch that corrects the underlying security threat. We consider the number of days between the vulnerability introduction and its fixing as its survival time. The Table 4 reports the survival time of security vulnerability in the plugins code extracted from the 119 patch analyzed.

Although disclosure might accelerates patch release, as observed by Arora et. al. [6], the number of days until a vulnerability is fixed after it is introduced in the code of a WordPress plugin is on the other hand, considerable. On average, it takes 653 days until the fixing of a vulnerability after its introduction. This means that a vulnerability could remain unnoticed in the web application for years before being identified and then fixed. For example, the vulnerability CVE-2015-4413[18] as illustrated in Figure 1 has been fixed in the commit 1178736[19] made on 6th November 2015. The commit 603491[20] performed on 25th September 2012 was identified as the vulnerability-introducing commit. Therefore, a simple to be corrected but very severe vulnerability survived in the code for 989 days (2 years, 8 months, 10 days).

## 4.5 RQ$_5$ - What are the most common vulnerability related topics discussed among developers?

Table 5 shows the distribution of questions (Q) and answers (A) per topic as well as the ratio of answers per question (A/Q), means of views ($M_v$), and means of score ($M_s$). Table 5 provides us with interesting results. For example, *Server Security* is the topic with the least amount of questions and answers and *Database Security* is the one with most questions and answers (291 questions and 410 answers). Questions about *Server Security* usually focus on how server administrators decisions may prevent an attacker from exploiting a vulnerability and from gaining access to sensitive internal information. Although the topic *Server Security* has the highest Score (5.09),

[18]nvd.nist.gov/vuln/detail/CVE-2015-4413
[19]plugins.trac.wordpress.org/changeset/1178736
[20]plugins.trac.wordpress.org/changeset/603491

**Table 5: The 10 topics discovered by LDA**

| Topics Name | Q | A | A/Q | Views | Score |
|---|---|---|---|---|---|
| Server Security | 11 | 19 | 1.72 | 440.09 | 5.09 |
| Cryptography | 38 | 54 | 1.42 | 991.02 | 1.84 |
| Authentication | 43 | 59 | 1.37 | 1947.55 | 1.37 |
| Plugin Update | 27 | 33 | 1.22 | 386.77 | 1.37 |
| Permissions | 34 | 51 | 1.5 | 1801.32 | 1.32 |
| File Upload | 29 | 36 | 1.24 | 1835.62 | 1.93 |
| Content Security | 43 | 60 | 1.39 | 855.04 | 1.16 |
| Programming | 50 | 63 | 1.26 | 517.08 | 2.4 |
| SSL Certificate | 42 | 45 | 1.07 | 1209.28 | 0.57 |
| Database Security | 291 | 410 | 1.63 | 794.16 | 1.63 |

we believe that this category has a low number of questions because plugin developers are not concerned about server-side security. Indeed, we found an outlier question with a Score of 48 and 3,108 views, which alone is responsible for 64.20% of the *Server Security* topic popularity. Analyzing this question[21], we found that it very popular because it brought up an appealing discussion regarding a collaboratively-edited list of "high-end WordPress webhost" for those "WordPress site that needs really hardened security".

In contrast, the large number of questions about *Database Security* corroborates our observation that web applications are subject to vulnerabilities such as SQL Injection (see Section 4.1). Moreover, while SQL Injection can be addressed with solutions such as web firewalls, we agree that they are best dealt with in source code by carefully employing secure coding practices. This observation and the data in Table 5, suggest that developers are interested in *Programming*, which has one of the highest question Score (2.4). Despite the low number of questions, we believe that *Programming* is the most important topic to guide the overall progress towards proactive security.

It is also worth mentioning that the number of views can vary significantly. For instance, *Authentication* is the topic that has the highest average number of views (1,947.55) and *Server Security* has the lowest (440.0). Upon analyzing the most visualized questions in *Authentication*, we noted that they are very general and recurrent questions about best practices on authentication and authorization. In contrast, we did not observe a considerable discrepancy among question scores. In 2 out of 8 cases, the question scores are close to 1.50, which means that the number of "up-votes" are not more than two times bigger than the number of "down-votes". Therefore, we can conclude that in general *Server Security* contains few, but relevant, questions and although *SSL Certificate* is a very popular topic, averaging more than 1200 views, some of the questions are poorly elaborated.

Finally, we investigated how tags are used in StackExchange WordPress, which emphasizes the use of tags to categorize and group questions. We found a total of 325 distinct tags. The five most common tags are: *(i)* login – 35 occurrences; *(ii)* updates – 29 occurrences; *(iii)* ssl – 28 occurrences; *(iv)* multisite – 26 occurrences, and *(v)* database – 25 occurrences. These results corroborates to indicate that *Authentication*, *Authorization*, *SSL Certificate*,

[21]wordpress.stackexchange.com/questions/9231

and *Database Security* are indeed the most common security-related topics discussed among WordPress developers.

## 5 IMPLICATIONS

In this section we further discuss the results presented in the previous sections.

### 5.1 Mitigations and Detection Methods

As the results indicate, if plugin developers manage to fully understand, detect, and eliminate Cross-site Scripting, SQL Injection and Cross-site Request Forgery, they will be developing plugins with 76.31% less vulnerabilities. However, it is worth mentioning that the aforementioned vulnerabilities are not the only vulnerabilities that can be exploited by attackers. Rather, these are simply the most common vulnerabilities we identified in our study. Following the typical methods to mitigate vulnerabilities does not guarantee that a given plugin is secure, but it does, however, set a good security baseline. Next we summarize some of the recommended mitigation methods that can be used to make plugins code more reliable.

- Understand how data is used and the encoding expected by the core parts of the plugin-based web systems (e.g. WordPress, Drupal). This is especially important because plugins have to send input data in specific patterns to many internal components. In this case, developers should assume all input as malicious and reject any input that does not strictly conform to the required encoding pattern, or transform it into a valid input. For example, WordPress provides some helpfully PHP functions such as: `esc_html` – escapes HTML to safely output processed inputs; `esc_js` – escapes Javascript to be used in tag attributes; `sanitize_sql_orderby` – sanitize a SQL order by clause; `sanitize_text_field` – sanitizes a string checking for invalid characters.
- There are other recommended mitigation methods that are easier to apply. For instance, to protect databases from SQL Injection, developers should use parameterized SQL statements by separating code from data manipulation. Moreover, it is important to check the headers to verify whether the request is from the same origin. Also, it is useful to avoid exposing data to unauthenticated attackers. Another good practice is avoiding the GET method for any request that triggers a state change.

Unfortunately, there is no method able to detect the most common code weakness with 100% of accuracy and coverage. Even modern techniques that use data flow analysis to detect Cross-site Scripting [39], sometimes give false positives when examining for some vulnerabilities. Indeed, Fonseca and Vieira [20] set out to study the effectiveness of static code analysis tools to detect vulnerabilities in plugins of web systems and understand the potential impact of those vulnerabilities in the security of the core web system. Their results suggest the coverage analysis performed by static analysis tools is inefficient and these tools report false positives often. In contrast, SQL Injection vulnerabilities are usually more effectively detected by dynamic analysis methods. However, it is well-known that even dynamic analysis methods have drawbacks [1]. It is also worth mentioning that not even a thorough software testing process can guarantee the detection of all possible vulnerabilities. Although software testing can be seen as a process aimed at making sure software does what it was designed to do and also that it does not do anything that was not planned [31], in most cases testing every possible input is not possible or impractical. Also, creating and maintaining a test suite demands many human resources and executing large test suites might take too long. Therefore, creating secure plugins (devoid of vulnerabilities) poses a complex challenge to developers as testing all possible plugins combination is in general impractical [22] [33] [34].

Other manual analysis methods (e.g., code review) can be more effective than strictly automated methods. This is especially true for vulnerabilities related to design. For example, Cross-site Request Forgery is specially hard to detect reliably using automated methods. This is because each web system has its own security policy and manner to indicates which requests require strong confidence that the user intends to make the request. Manual analysis can be useful for finding SQL Injection, but it might not achieve the desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large. Consequently, some organizations as OWASP stress the importance of a balanced approach that includes both manual reviews and automated analysis that covers all phases of the web system development process. Code recommenders techniques [15] [7] seems to be useful in this context as it can be used in some cases to present a list of all possible sanitization methods, allowing a developer to browse the proposal and to select the appropriate one from the list. However, as sanitization methods should be recommended with respect to specific contexts (vulnerable codes) we still be missing a set of metrics that could help tools to effectively recommend the places most susceptible to have a vulnerability and the most appropriated sanitation methods [19] [45].

### 5.2 Developing Secure Plugins

We agree that plugin developers strive to write secure code, but as we could observe they still do not know how, mainly because they are not-so-skilled developers. Indeed, given the nature of the most common vulnerabilities, it is clear that developers do not understand common attacks, the security functions provided by the language and platform they use, and how certain easily applicable practices can mitigate security problems. We found that the time window of addressing security vulnerabilities post-release can be high is some cases. Overall, it takes less time to address vulnerabilities when they are found earlier in the plugin development lifecycle. Thus, our recommendation is to develop new methodologies tailored to plugin developers, which promote security in the requirements and design phases. In this case, we have to consider that developers are the most important entities in software security. As we mentioned, automated solutions tend to fail to detect some vulnerabilities and inexperienced software developers exhibit a lack of texting skills [37]. Worse, strategies as social transparency [36] seems to not influence the testing behavior of plugin development teams as observed in others context [17] [46]. As we could observe, the quality of open source plugins does not increase over time as the frequency of disclosed vulnerability does not decrease over time.

The results also indicate that we need tools that enable novice developers to reliably build secure plugins. That is, those tools need to be usable by developers without any security background. Unfortunately, the well-established security tools cannot operate at the speed required to achieve a more proactive security. These tools rely on existing vulnerability disclose and are incompatible with modern software development.

## 6 THREATS TO VALIDITY

In this section we present the constraints of the study. We list the possible threats and procedures we took to mitigate those issues.

**External validity**. The study was limited to the analysis of 895 NVD vulnerability bulletins associated with WordPress plugins. Therefore, the results cannot be generalized to other plugin-based web systems (e.g., Drupal and Joomla). Given that plugin-based web systems projects vary on characteristics as amount of participants, community structure, and governance, we cannot draw general conclusions about all projects (i.e., the whole population). To build reliable empirical knowledge, we need a family of experiments [10] that include plugin-based web systems of all types. However, with its 52,000 plugins, WordPress is a relevant and highly popular ecosystem. Moreover, we think the impact of this threat is minimal for three reasons: *(i)* most plugin-based web system provide the same basic programming model; *(ii)* the focus of the study was on security vulnerabilities rather than on the specifics of the software systems themselves, and *(iii)* none of the other plugin-based systems focus specifically on promoting secure coding, so there is no clear reason that one would be favored over the other in our study.

Another threat is the number of patches: we found that only 109 bulletins report on patches. Therefore, we cannot ensure that the results would be the same if we had taken into account vulnerabilities that did not have available patches. While it is possible that the results would not be the same with vulnerabilities without patches, we have no reason to believe that any specific set of patches would have a different size and include vulnerabilities that have different survival time.

**Internal Validity**. The data analysis method is another aspect that can negatively affect the results: the study was based on a manual analysis. To mitigate this threat, we conducted a double manual verification of both the NVD vulnerability bulletins and the patches collected from the WordPress plugins repository. We also manually analyzed and eliminated non-related questions extracted from the WordPress Stack Exchange Q&A Website. Independent validation of our results can also be performed once we made all of our data publicly available. Another threat is the validity of our survival time measure (i.e. number of days until a vulnerability was fixed properly). Because most patch include large changes rather than only vulnerability-fixing changes, measuring the extent of the modifications is challenging. While the use of other measure methods (e.g., SZZ [25]) could have produced different results, we believe that our manual analysis is a reasonable representation of the survival time of WordPress plugin vulnerabilities [24].

Another construct validity threat is that if the keyword set was incomplete, the automatic extraction could have missed some vulnerability bulletins or Stack Exchange questions. Section 3 describes our set of keywords. To validate the completeness of the set of keywords, two authors independently inspected randomly chosen NVD bulletins and Stack Exchange questions that did not contain any of the keywords used. The manual review of those assets found any relevant missed case. We applied this analysis to increase our confidence in the soundness of the keyword set.

## 7 CONCLUDING REMARKS

This paper describes the implications of plugin vulnerabilities. To do so, we analyzed 805 WordPress vulnerability bulletins listed by NVD as well as the respective patches of each reported vulnerability. We also analyzed the most common security-related topics discussed among WordPress plugin developers. The key value to this research is that our results provide an overview of the main types of vulnerabilities, severity of these vulnerabilities, and complexity of the changes needed to remove vulnerabilities. To the best of our knowledge, this is the first study to look at several facets of Wordpress plugin vulnerabilities. We show that most common vulnerabilities are Cross-site Scripting, SQL Injection, and Cross-site Request Forgery. Our results would seem to suggest that the frequency with which these vulnerabilities appear do not seem to be declining over time: that is, these common vulnerabilities continue to appear for a number of reasons. We conjecture that the main reason is that plugin developers are somewhat underskilled in secure programming. In a way, most end-users are drawn to WordPress because it is easy to learn and use even for non-tech savvy people.

According to the vulnerabilities reports that we analyzed, SQL Injection tends to partially impact the integrity and availability of web systems and can be easily exploitable. In effect, our results seem to indicate that the most commonly asked questions are centered around database security and associated vulnerabilities. Our investigation shows that this is by far the most popular topic discussed among plugin developers while SSL Certificate seems to be the least covered and the most challenging topic discussed among developers.

Several avenues for future exploration are possible. As mentioned, our analysis suggests that most vulnerabilities appear in a single file. Therefore, as future work, it would be interesting to investigate if there is a correlation between the amount of files in which the vulnerability is present and the size of the plugins. Additionally, since our results show that most vulnerabilities span a single line of code, we believe that code recommenders could help mitigate the introduction of vulnerabilities by providing context-aware code completion features, which would reduce common vulnerability-introducing programming mistakes [15]. The complete material and data related to our study can be found in our support website.[22]

## REFERENCES

[1] Ashish Aggarwal and Pankaj Jalote. 2006. Integrating static and dynamic analysis for detecting vulnerabilities. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, Vol. 1. IEEE, 343–350.

[22] omesa1984.github.io/SPLC18

[2] Omar H Alhazmi, Yashwant K Malaiya, and Indrajit Ray. 2007. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security* 26, 3 (2007), 219–228.

[3] Edward Amoroso. 2018. Recent Progress in Software Security. *IEEE Software* 35, 2 (2018), 11–13.

[4] Chris Anley. 2007. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (2nd ed.). Wiley. 744 pages.

[5] William A. Arbaugh, William L. Fithen, and John McHugh. 2000. Windows of Vulnerability: A Case Study Analysis. *Computer* 33, 12 (2000), 52–59.

[6] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. 2010. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research* 21, 1 (2010), 115–132.

[7] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. 2014. Cscc: Simple, efficient, context sensitive code completion. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 71–80.

[8] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering* 19, 3 (2014), 619–654.

[9] Victor R Basili and H Dieter Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on software engineering* 14, 6 (1988), 758–773.

[10] Victor R Basili, Forrest Shull, and Filippo Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473.

[11] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[12] Jan Bosch. 2009. From software product lines to software ecosystems. In *Proceedings of the 13th international software product line conference*. Carnegie Mellon University, 111–119.

[13] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 257–268.

[14] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2010. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 105–114.

[15] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 213–222.

[16] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. 2011. How long does a bug survive? an empirical study. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 191–200.

[17] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 1277–1286.

[18] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51, 10 (2009), 1469–1484.

[19] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 65–73.

[20] José Fonseca and Marco Vieira. 2014. A Practical Experience on the Impact of Plugins in Web Security. In *The 33rd IEEE Symposium on Reliable Distributed Systems*. IEEE.

[21] Giampaolo Garzarelli. 2013. Open source software and the economics of organization. In *Markets, information and communication*. Routledge, 63–78.

[22] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2012. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 244–254.

[23] Thomas L Griffiths and Mark Steyvers. 2004. Finding scientific topics. *Proceedings of the National academy of Sciences* 101, suppl 1 (2004), 5228–5235.

[24] Matthieu Jimenez, Mike Papadakis, Tegawendé F Bissyandé, and Jacques Klein. 2016. Profiling android vulnerabilities. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. IEEE, 222–229.

[25] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, and others. 2006. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 81–90.

[26] Teemu Koskinen, Petri Ihantola, and Ville Karavirta. 2012. Quality Of WordPress Plug-Ins: An Overview of Security and User Ratings. In *ASE/IEEE International Conference on Social Computing and ASE/IEEE International Conference on Privacy, Security, Risk and Trust*. IEEE, 834–837.

[27] Robert A Martin, Steven M Christey, and Joe Jarzombek. 2005. The case for common flaw enumeration. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*.

[28] Peter Mell, Karen Scarfone, and Sasha Romanosky. 2006. Common vulnerability scoring system. *IEEE Security & Privacy* 4, 6 (2006).

[29] Marc H Meyer and Robert Seliger. 1998. Product platforms in software development. *MIT Sloan Management Review* 40, 1 (1998), 61.

[30] J. C. Munson and S. G. Elbaum. 1998. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 24–31.

[31] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley. 240 pages.

[32] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 284–292. https://doi.org/10.1145/1062455.1062514

[33] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 907–918.

[34] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. 2008. Feature interaction: The security threat from within software systems. *Progress in Informatics* 5 (2008), 75–89.

[35] Birgit Penzenstadler, Ankita Raturi, Debra Richardson, and Bill Tomlinson. 2014. Safety, security, now sustainability: The nonfunctional requirement for the 21st century. *IEEE software* 31, 3 (2014), 40–47.

[36] Raphael Pham. 2014. Improving the software testing skills of novices during onboarding through social transparency. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 803–806.

[37] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. 2014. Enablers, inhibitors, and perceptions of testing in novice software teams. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 30–40.

[38] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361.

[39] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361.

[40] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

[41] Igor Steinmacher, Marco Aurelio Gerosa, and David Redmiles. 2014. Attracting, Onboarding, and Retaining Newcomer Developers in Open Source Software Projects. In *Workshop of Global Software Development*.

[42] MM Syeed, Alexander Lokhman, Tommi Mikkonen, and Imed Hammouda. 2015. Pluggable systems as architectural pattern: an ecosystemability perspective. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ACM, 42.

[43] R. Telang and S. Wattal. 2007. An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price. *IEEE Transactions on Software Engineering* 33, 8 (2007), 544–557.

[44] James Walden, Maureen Doyle, Rob Lenhof, John Murray, and Andrew Plunkett. 2010. Impact of Plugins on the Security of Web Applications. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*. ACM, 1:1–1:8.

[45] James Walden, Maureen Doyle, Grant A Welch, and Michael Whelan. 2009. Security of open source web applications. In *Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 545–553.

[46] Shundan Xiao, Jim Witschey, and Emerson Murphy-Hill. 2014. Social influences on secure development tool adoption: why security tools spread. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*. ACM, 1095–1106.