A Practical Experience on the Impact of Plugins in Web Security

José Fonseca
CISUC, University of Coimbra /
Polytechnic Institute of Guarda, Portugal
josefonseca@ipg.pt

Marco Vieira
CISUC, University of Coimbra
Coimbra, Portugal
mvieira@dei.uc.pt

***Abstract*—In an attempt to support customization, many web applications allow the integration of third-party server-side plugins that offer diverse functionality, but also open an additional door for security vulnerabilities. In this paper we study the use of static code analysis tools to detect vulnerabilities in the plugins of the web application. The goal is twofold: 1) to study the effectiveness of static analysis on the detection of web application plugin vulnerabilities, and 2) to understand the potential impact of those plugins in the security of the core web application. We use two static code analyzers to evaluate a large number of plugins for a widely used Content Management System. Results show that many plugins that are currently deployed worldwide have dangerous Cross Site Scripting and SQL Injection vulnerabilities that can be easily exploited, and that even widely used static analysis tools may present disappointing vulnerability coverage and false positive rates.**

Keywords-Web applications; security; vulnerabilities; static analysis; plugins

I. INTRODUCTION

There is nowadays an increasing dependency on web applications. Ranging from individuals to large organizations, almost everything is stored, available or traded on the web. Web applications can be personal web sites, blogs, news, social networks, web mails, bank agencies, forums, e-commerce applications, etc. The omnipresence of web applications in our way of life and in our economy is so important that they have turned into a natural target for malicious minds.

To allow customization and thus fit the requirements of diverse scenarios, many web applications support the integration of server-side plugins that offer multiple functionalities and may be provided by different parties. Well-known examples are Content Management Systems (CMSs) that allow individuals and/or communities of users to easily create and administrate web sites that publish a variety of contents. The sites created using CMSs can go from personal web pages and community portals to large corporate and e-commerce applications.

Although plugin-based web applications assure extensibility and customizability, the possibility of integrating third-party software opens an additional door for security vulnerabilities, regardless of the security assurance activities conducted on top of the core application. In fact, other works show a predominance of security exploits due to vulnerabilities in the external plugins, when compared to the core application [5][22]. This is mostly due to the typically uncontrolled development processes and poor quality assurance activities applied during the development of such plugins, which are not able to prevent security vulnerabilities from being shipped into the field.

Penetration testing and static analysis are examples of well-known techniques frequently used by web developers to identify security vulnerabilities in their code. Penetration testing consists in stressing the application from the point of view of an attacker ("black-box" approach) using specific malicious inputs. On the other hand, static analysis is a "white-box" approach based on the analysis of the source code of the application (without executing it) looking for potential vulnerabilities. A key difference between penetration testing and static analysis is that the first does not require access to the code while the second does. On the other hand, performing extensive testing may be unfeasible (e.g. due to the typically large number of plugins available and of potential configurations), whereas static analysis theoretically allows covering 100% of the code. For this reason, static analysis is frequently considered the most efficient way to detect vulnerabilities in web applications [1]. Trusting static analysis tools is thus of utmost importance when analyzing code that is used for vital processes or with an economic impact, in particular when security is the issue under discussion.

In this paper we study the use of static analysis tools to detect vulnerabilities in a plugin-based web application. In practice, the goal is to study two key questions:

1. *How effective are free static analysis tools detecting vulnerabilities in web application plugins?*
2. *What is the real importance and impact of plugins in the security of a web application?*

To provide insights on these questions, this paper presents an experimental study in which we used two static analysis tools to detect security vulnerabilities in a comprehensive set of widely used plugins for a major player in the PHP CMS market. The static analyzers used are RIPS, a well-known tool for PHP source code analysis, and phpSAFE, a follow-up of a project whose development was requested by Automattic [3], the developer of WordPress, to improve the security of its plugins. The CMS considered is WordPress, which is used by millions of users around the world, and has a reported market share of approximately 60%, among all CMSs available [38]. We analyzed 35 plugins from the extremely large number of almost 30 thousand plugins available. The plugins have diverse characteristics

PEst-OE/EGE/UI4056/2011 – project financed by Science and Technology Foundation project ICIS - Intelligent Computing in the Internet of Services (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER

concerning the function they execute, the size of the code, the complexity, and the number of known downloads.

Results show that plugins that are currently being used in thousands of WordPress installations have dangerous Cross Site Scripting (XSS) and SQL Injection (SQLi) vulnerabilities. In fact, we disclosed more than 360 vulnerabilities in the plugins analyzed. Another observation is that RIPS presents disappointing results both in coverage and false positives. Compared to phpSAFE, RIPS detected 60% less vulnerabilities.

The outline of this paper is as follows. The next section introduces background concepts. Section III presents the experimental methodology used in the study. Section IV details the concept of static analysis in PHP applications and presents the static code analyzers used. Section V presents the results and discusses the lessons learned. Finally, Section VI concludes the paper.

II. Web Security Testing and Plugins

Previous works and practice suggest that external server-side plugins are a major source of security vulnerabilities. For example, the field study presented in [22], which included 312 real exploits used by hackers to attack web applications, shows the prevalence of security exploits that target the plugins (58%), when compared to the core application.

Another relevant work is [5], which analyzed the security of the 50 most popular and 10 most popular e-commerce WordPress plugins. Besides XSS and SQLi, the vulnerabilities considered include Cross Site Request Forgery (CSRF), Remote/Local File Inclusion, and Path Traversal vulnerabilities. The study reports vulnerabilities in 20% of the top 50 plugins and in 70% of the e-commerce plugins. It also states that only six plugins fixed their vulnerabilities within six months after the vulnerabilities had been discovered. There are, however, questions about how the results were obtained that are not explained in the work presented in [5]. In fact, they do not detail the methodology and tools used, or even if they conducted any kind of manual analysis to confirm the vulnerabilities found. There is also a lack of detail about the vulnerabilities found, like the total number, false positives and negatives, and coverage. Our work addresses these questions. Furthermore, we also make use of plugins to understand the effectiveness of two static code analyzers.

A vulnerability is a weakness (an internal bug) that may be exploited to cause harm, although its presence does not cause harm by itself [25]. In practice, a vulnerability is a precondition for an attack (a malicious external fault) to cause an error and possibly subsequent failures [2]. Well-known examples of dangerous vulnerabilities in web applications are XSS and SQLi.

The search for common software bugs as well as for security problems can be grouped into white-box approaches (e.g., static analysis), black-box approaches (e.g., penetration testing) and a blend of both (gray-box). White-box consists of source code analysis (inspection or static analysis). It uncovers security problems by looking at the code of the application without executing it, so it has no run-time overhead and it may virtually achieve 100% code coverage, as it is able to analyze all the possible execution paths (unlike testing in which code coverage is a well-known problem). It also has the advantage of being applicable early in the software development lifecycle, even when only part of the code is available. Common problems are the high number of false positives (safe code constructs that are seen as vulnerable by the detection mechanism) and false negatives (vulnerable code that is seen as safe). White-box analysis is considered by many as the most efficient way to locate vulnerabilities in a web application [1]. For example, at Microsoft it is believed that code review is around 20 to 30 times more effective in finding bugs than software testing [25] and it can uncover around half of the existing bugs when applied the most adequate manner [4].

III. Experimental Methodology

In this section we propose a methodology for the detection of security vulnerabilities in the code of PHP web application plugins using static analysis. We first present the generic process, and then focus on introducing the types of vulnerabilities addressed, the target web application, and the plugins analyzed.

A. Overall process

The process proposed is based on a set of straightforward phases and steps:

1. **Preparation of the experiments**: create the conditions for running the static analyzers on top of relevant plugins. Two steps are needed:
   a. Identify a *representative web application* that allows the integration of plugins, and select a large set of widely used *plugins* for that application;
   b. Decide on the *types of vulnerabilities* to be the target of the study and select representative *static analyzers* able to detect those vulnerabilities;
2. **Execution of the static code analyzers**: analyze the plugins using the tools. This includes two steps, whose results are later processed and compared:
   a. Perform a *generic analysis* of the plugins using the typical configuration of the analyzers, i.e. not taking into account the fact that the target files are plugins for a specific web application;
   b. Run a *targeted analysis* in which the configuration of the analyzers is tuned for the specific context of the target web application;

   The correlation of the results from these two steps allows studying the performance of the static code analyzers in detecting vulnerabilities when they are configured for the specific context of the web application plugins and when not, with respect to two key figures of merit: *coverage* and *false positives*. This may give insights to how these tools should evolve in the future, e.g. helping to understand whether the tools should be explicitly prepared to handle the analysis of plugins or just need to follow a more generic approach.
3. **Analysis of the results**: collect the reports of the tools and process the information gathered. This includes two steps:
   a. *Manual verification* of the vulnerabilities reported to

confirm the true vulnerabilities and discard the false positives;

b. Analysis of data to *understand the impact of plugins* in the application security and study the relative effectiveness, strengths and weaknesses, of the static analyzer tools.

B. Chosen web vulnerabilities

The list of types of vulnerabilities affecting web applications is enormous, but XSS and SQLi are at the top of that list, accounting for 32% of the vulnerabilities observed [36][34]. Other studies also found XSS and SQLi as the most prevalent [33][18][40]. Figure 1 depicts the yearly percentage of disclosed XSS and SQLi among all the causes of web application vulnerabilities, showing that they have been increasing over time [37]. Many referenced works, such as [34][27][7], have discussed details about the most common vulnerabilities, along with the reasons of their existence, attacks, best practices to avoid, detect and mitigate them.

A XSS attack consists of a malicious injection of HTML and/or other scripting code (frequently Javascript) in a vulnerable web page. Many web applications are built in a way that they use the values supplied by users directly in the HTML displayed in the web browser. Without being properly sanitized, this input can be crafted to change the contents of the web page displayed to the victim, therefore giving the attacker control. In practice, by tweaking the input, the attacker is able to change how the web application executes some of its functions, allowing him to take advantage of users visiting that web page in the future. Note that this input may come from a text field, but also from a file, a database record or from another application. XSS attacks exploit the users' (victims) trust of a web site, allowing the attacker to impersonate them and even execute other types of attacks such as CSRF. The injection of XSS can also be persistent if the malicious input is stored in the back-end database of the web application. This way it can affect every user that visits a web page built from this data, therefore potentiating dramatically the malicious effects of the attack.

SQLi attacks take advantage of unchecked input fields in the web application interface to maliciously tweak the SQL query sent to the back-end database. This allows the attacker to retrieve sensible data or even alter database records. An SQLi attack can be dormant for a while and only be triggered by a specific event, such as the periodic execution of some procedures in the database (e.g., the scheduled database cleaning function). SQLi attacks may also allow privileged access to the back-end server and to the inner network of the enterprise, free of the typical firewall and IDS/IPS barriers.

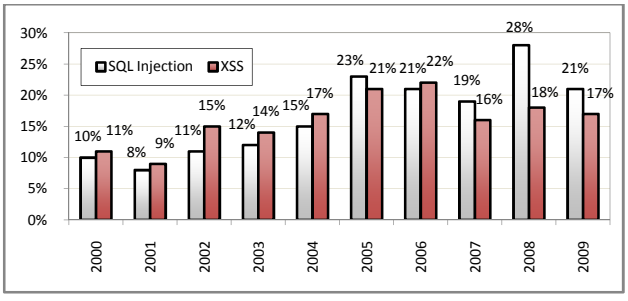

Figure 1. Evolution of disclosed reports of SQLi and XSS [37]

In practice, XSS and SQLi vulnerabilities open the door for attackers to access unauthorized data (read, insert, change or delete), gain access to privileged database accounts, impersonate other users (such as the administrator), mimic web applications, deface web pages, view and manipulate remote files on the server, inject and execute server side programs that allow the creation of botnets controlled by the attacker, etc. Due to the prevalence and the severity of the attacks that exploit these two important vulnerabilities (XSS and SQLi), we have chosen them as the target of our study.

C. Target web application

The information digitally available on the web and stored in back-end databases (the so-called hidden web) is rapidly increasing. In the last three years, from November 2010 to November 2013, the number of web sites grew 212% to 780 million [30]. CMS applications are used by 35% of the web, according to W^{3}Techs [38] and they are normally built on top of third-party plugins. Such large utilization justifies the focus of our work about this type of web application.

The Top 5 CMS used nowadays is presented in TABLE I. WordPress is the most widely used CMS, supporting the creation of web sites like TED, NBC, CNN, The New York Times, Forbes, eBay, Best Buy, Sony, TechCrunch, UPS, National Football League, CBS Radio, etc. [12]. There are over 72 million WordPress sites, which is 9% of the web. Over 400 million people view more than 14 billion pages each month, some of these users produce about 36 million new posts and 63 million new comments each month. WordPress has, currently, 28 thousand plugins that have been downloaded over 550 million times [12]. In an ecosystem so widespread, any security breach may have a significant impact on a huge number of users, enterprises and their overall business.

WordPress is developed in PHP (a dynamically-typed language used by 81.4% of all web sites [39]). A key aspect is that, in the last years, it has been absent from significant attacks that exploit vulnerabilities in the core application. However, the same cannot be said about its third-party plugins, as seen in recent events where the vulnerabilities in just four plugins may have affected over 3.5 million blog sites and many more of their users [13][16][11][9].

D. Plugins selection

Having decided that the core web application was WordPress, we selected a set of 35 plugins, a reasonable number that allowed both the execution of the experiments, including a manual analysis of all the vulnerabilities reported by the static analysis tools, and that could be representative enough to obtain meaningful results. In practice, the reasoning behind the selection of the plugins was to include a very

TABLE I. TOP 5 CMS [38]

CMS	CMS market share	Language	OOP
WordPress	59.4%	PHP	V
Joomla	9.3%	PHP	V
Drupal	5.6%	PHP	X
Blogger	3.4%	Java	V
Magento	2.6%	PHP	V

TABLE II. WORDPRESS PLUGIN SUMMARY DATA

Plugin	Category	Downloads	Rate	Files	LOC	OOP
all-in-one-webmaster v8.2.3	Webmaster	513,234	4.5	1	416	
calendar v1.3.2	Calendar	439,957	3.9	1	3,050	
content-slide v1.4.2	Image	206,833	4.5	3	499	
contextual-related-posts v1.8.6	Content management	309,111	4.3	3	1,157	
digg-digg v5.3.4	Social media	782,548	3.5	13	5,437	6
easy-adsense-lite v6.06	Advertising	289,838	3.3	8	1,128	
events-manager v5.3.8	Events	777,260	4.3	17	5,135	2
external-video-for-everybody v2.0	Video	8,247	4.0	1	394	
feedweb v1.8.8	Feedback	57,706	4.7	12	4,793	
foursquare-checkins v1.2	Checkin	1,688	5.0	1	232	
funcaptcha v0.3.7	Captcha	11,684	4.6	4	1,314	8
ga-universal v1.0	Webmaster	275		1	136	
jaspreetchahals-coupons-lite v2.1	e-commerce	7,082	4.9	3	1,189	18
login-with-ajax v3.0.4	Login	227,040	4.4	4	906	2
mail-subscribe-list v2.0.9	Mail	59,553	4.6	1	144	
mathjax-latex v1.1	Content management	6,075	4.6	1	404	
montezuma v1.1.7	Webmaster	513,246	4.5	7	502	
newsletter v3.2.7	Newsletter	815,291	4.6	3	1,288	1
occasions v1.0.4	Content management	3,334	3.0	2	755	
paypal-digital-goods-monetization-powered-by-cleeng v2.2.13	e-commerce	6,807	3.7	3	522	
qtranslate v2.5.34	Localization	978,012	3.7	9	4,337	
securimage-wp v3.2.7	Captcha	3,158	2.3	2	997	
simply-poll v1.4.1	Poll	20,795	3.2	6	847	
social-media-widget v4.0.1	Social media	1,181,508	4.1	1	1077	
syntaxhighlighter v3.1.5	Content management	310,490	4.4	1	1257	
top-10 v1.9.2	Content management	126,923	4.3	5	891	
trafficanalyzer v3.3.2	Webmaster	14,676	3.4	22	4,684	5
underconstruction v1.08	Content management	473,311	4.9	3	754	1
user-role-editor v3.12	User Management	901,598	4.2	2	545	
videojs-html5-video-player-for-wordpress v3.2.3	Video	105,561	4.3	2	388	
wordpress-simple-paypal-shopping-cart v3.5	e-commerce	343,249	4.3	1	16	
wp-photo-album-plus v5.0.2	Image	852,578	4.3	31	12,339	
wp-symposium v13.02	Social media	115,112	4.0	44	23,173	
wp125 v1.4.9	Advertising	435,757	4.2	4	547	
xili-language v2.8.4	Localization	107,070	4.2	3	5,090	
	Total	**11,006,607**		**225**	**86,343**	**43**
	Average	**314,474.5**	**4.1**	**6.4**	**2,466.9**	**8.6**

LOC – Lines of Code

diverse set. In order to not bias the study, we did not study the vulnerability proneness of these plugins beforehand.

The list of the 35 WordPress plugins selected is presented in TABLE II. Overall, these plugins have been downloaded over 11 million times and have an average rating of 4.1 out of 5 stars among WordPress users. These plugins are implemented in a total of 225 PHP files (that were analyzed) and they have more than 86 thousand lines of code. The list of plugins is varied, in what regards the function they execute, the size of the code, the complexity, and the number of downloads.

IV. PHP STATIC ANALYSIS

In this section we detail the concept of static analysis for security (in particular, taint analysis), introduce the RIPS and phpSAFE tools used in our study, and discuss some of the known limitations of this type of security tools.

To perform the static analysis, the tool receives a source code file as input (Figure 2). Next it adds the other files that are referred by the original source file as included files. With this data, it builds a syntax tree model representing the code, which is analyzed based on a body of knowledge about security vulnerabilities. The list of vulnerabilities discovered is finally presented to the user.

To obtain the syntax tree model for the specific case of PHP applications there is a handy PHP function called `token_get_all`. It parses a PHP file and builds an array with the PHP tokens [14]. Based on this function there is an API that simplifies the static code analysis by building a syntax tree more complete and, to some extent, easier to work with [17]. However, both RIPS and phpSAFE still use directly the `token_get_all` to build the parse tree model. The reason is that they need to perform a series of low-level manipulations very fast, which makes the higher level API less interesting to use.

A. Taint analysis for vulnerability detection

The typical approach to find input validation bugs is using taint analysis [4][35][6][28], which is also a feature of some languages, like Perl and Ruby. In fact, it first appeared in Perl, allowing the detection of attacks at runtime by running the Perl scripts with `taintperl` instead of `perl` [29]. Taint analysis is well suited for the input validation class of vulnerabilities, like XSS and SQLi, as it follows the path of input variables, which are also the input vectors of the attacks that exploit these vulnerabilities.

In taint analysis the data that comes from an uncontrolled

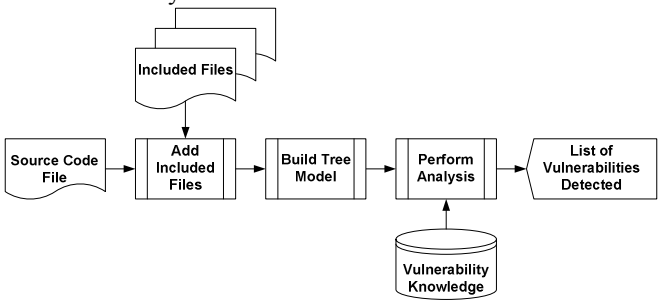

Figure 2. Static analysis tool block diagram, adapted from [35]

environment (i.e., input data that may be malicious) is tainted. When a tainted variable is used by the application in some sensitive way an attack becomes possible. In a conservative approach every input that comes directly from the outside is considered to come from an uncontrolled and unsafe environment; whether it is from a user input, a file, the database, or the return of a function that is not implemented locally in the code (in the configuration of the static code analyzer tool). This tainted data may propagate to other program variables, making them also tainted. The variables may, however, become untainted through a set of operations that depend on the nature of the data and how they are going to be used. That is, this untainting process is dependent on the vulnerability type.

To clarify the idea, let's consider an example of different ways to sanitize (untaint) data in the context of multiple types of vulnerabilities. Assume then an input variable that is tainted. If this variable is going to be used in a MySQL database query we may want to make it safe against SQLi attacks and thus use the PHP function `mysql_real_escape_string`. This sanitizes the variable for SQLi but not for other kind of vulnerabilities, like XSS. On the other hand, if that variable is to be displayed in the web browser, we may want to make that variable safe for XSS attacks by using the PHP function `htmlentities`. This sanitizes the variable for XSS but not for other kind of vulnerabilities, like SQLi. There are, however, situations where the sanitization process makes the variable untainted for all (or several) vulnerabilities at once. For example, if the tainted variable is supposed to store an integer value, we have to take into consideration that PHP is a weakly typed programming language so it does not enforce the type checking of the variables. The sanitization function in this case may be the `intval` function, which allows untainting the variable for both XSS and SQLi, since it guarantees that only a numeric value can be stored in the variable (if a text is inputted it returns 0, which is also a number).

B. RIPS static code analyzer tool

RIPS is a static source code analyzer for vulnerabilities in PHP scripts, developed by Johannes Dahse [20]. RIPS is a well-known PHP free source code analysis tool that has been around since May 2010 and is being continuously improved. It was developed in PHP and has a comprehensive user interface. Its vulnerability detection mechanism is complemented by an integrated code audit framework that allows further manual analysis. These features make it very easy to explore the path of the tainted variables. RIPS is able to detect XSS, SQLi, HTTP Response Splitting, Code Execution, File Inclusion, File Disclosure, File Manipulation, Command Execution, XPath Injection, LDAP Injection, Header Injection, and Possible Flow Control vulnerabilities. However, the current version of RIPS, which is 0.54, does not support Object Oriented Programming (OOP).

RIPS includes three main configuration files devoted to the PHP tokens, which are used during the static analysis. Although RIPS is deployed with a default configuration, it is in these files that the user can configure the entry points where the tainted variables are generated, how they can be considered untainted and how their use may turn into a potential vulnerability.

a) `sources.php`*:* contains the inputs of variables that are considered as potentially unsecure. This is composed by PHP variables (like `$_GET` and `$HTTP_COOKIE_VARS`), server parameters (like `HTTP_ACCEPT` and `PHP_SELF`), PHP file functions (like `fgets` and `fread`), and database manipulation functions (like `mysql_fetch_array` and `mysql_fetch_row`).

b) `securing.php`*:* defines the PHP functions that can be used to untaint the variables. These functions change the values of their parameters so that they become safe to use when they are returned to the caller. Some of these functions are generic (like `intval` and `md5`), while others are specific to a given vulnerability (like `htmlentities` for XSS and `mysql_real_escape_string` for SQLi, as explained previously).

c) `sinks.php`*:* contains the PHP functions or language constructs that may be exploited by an attacker. They are specific to a given vulnerability and are affected by variables manipulated to take advantage of that vulnerability (like `echo` for XSS and `mysql_query` for SQLi).

C. phpSAFE static code analyzer tool

phpSAFE is a static source code analyzer for XSS and SQLi vulnerabilities in PHP scripts, currently under development by the authors of this paper [23]. It is a follow-up of a project proposed by Automattic [3] in an effort to improve the security of WordPress plugins, as well as PHP based web applications. The interface of phpSAFE is very simple and not as complete as the RIPS one. The tool was designed to facilitate the integration with other projects as a PHP class that can be included to search for vulnerabilities. phpSAFE performs a number of tasks: obtains vulnerable variables, output variables and other variables; functions used by the target code, other PHP files included or required, the complete list of tokens (the tree model of the PHP files) and finally debug information. This data can be very useful in helping security practitioners trace back the path of the tainted variables since they entered the system.

The configuration of phpSAFE, as far as the input, the output, and the sanitization of the variables, is based on the data contained on the RIPS configuration files. However, phpSAFE also includes specific configurations for the analysis of WordPress plugins (using data source and securing functions gathered from various resources, like [19] and [14]). Furthermore, phpSAFE has the advantage of being prepared to deal with OOP, unlike RIPS. This is a major highlight because many of the current CMS platforms with a very large community of users are developed using OOP (TABLE I). Implementing OOP in a static code analyzer is not a trivial matter, because of the complexity needed to deal with the multiple ways variables and methods can be used, the scope of variables, arrays, object instantiation, etc.

D. Known limitations of static analysis tools

A key limitation of static analysis tools has to do with the difficulty in implementing the analysis of dynamic code. For

example, in PHP this may come in the form of dynamic inclusions, such as when the name of the file to be included is stored in a variable, or the variable name is itself stored in another variable, for example:

```
$include_file=$_POST[ 'file' ];
require_once($include_file);
```

Moreover, existing tools are not ready to parse Javascript embedded code. They are also not prepared for logical or architectural vulnerabilities, whose detection is very difficult to automate. In fact, this is currently mainly addressed by manual analysis, usually with help of tools, like the Microsoft Threat Modeling Tool [25].

In addition to PHP specific functions, web application plugins rely on functions and variables provided by the core application to get data and to manipulate variables. For example, in the case of WordPress, an input variable may come from the `$wpdb->get_results` method, it may be sanitized using the `esc_html` function and may be used in the output through the `$wpdb->query` method [15][10].

As RIPS is a generic PHP static code analyzer, it is not configured by default to deal with specific application plugins. Therefore, running step b of the second phase of the experimental process, the target analysis, (see Section III.A) requires adding WordPress related functions to the RIPS `sources.php` and `securing.php` configuration files. On the other hand, phpSAFE is by default configured for WordPress.

V. Results and Discussion

This section presents and discusses the results of our study within the perspective of the two key questions we are addressing. First we analyze the performance of the static analysis tools. Afterwards we analyze the impact of plugins in the application security through a detailed analysis of the vulnerabilities found by both tools. Finally, we discuss the lessons learned and the limitations of the study.

A. Tool detection effectiveness

The effectiveness of the tools is assessed in two steps: the first consists of running each static code analyzer with base configurations for generic XSS and SQLi vulnerabilities, and the second consists of tuning those tools for the specific context of WordPress plugins. The aim is to understand whether the effectiveness of the vulnerability detection in plugins improves (or not) by having the tools configured for the specificities of the core application.

It is well known that automated security tools typically originate a high number of false alarms (false positives), so we performed a manual verification of all the results provided by both RIPS and phpSAFE, in both steps of the second phase of the experimental process (see Section III.A). This important process, which is very labor intensive and time consuming, allowed us to classify 139 of the 495 situations pointed by the tools as not being real vulnerabilities (28.1% false positives).

Due to the unfeasibly large amount of work, we did not manually analyze all the target source code files looking for vulnerabilities missed by the tools. Therefore, to obtain the coverage data we combined all the true vulnerabilities found by both tools, despite knowing that there could be some left undetected. This will give an optimistic view of the coverage rates of the tools, which is a best effort result that may be improved using more tools and a thorough manual review of the code using well established procedures, like the ESA Guide for Independent Software Verification and Validation [8] or the 1012-2004 IEEE Standard for Software Verification and Validation [19]. The overall results regarding detection coverage when the tools were configured for the WordPress are depicted in TABLE III. As can be seen, phpSAFE presents better overall results than RIPS.

During the experiments we observed that phpSAFE was unable to analyze five files (2% of the files) from the set of plugins due to an implementation defect of the current version related to an infinite loop. This problem happens when a file has many included files and becomes too complex. To compare the detection capabilities of both tools we discarded these five files. From a sample analysis of the discarded vulnerabilities (as reported by RIPS), we verified that most of them where already detected when the included files were analyzed by the tools individually, which minimizes the impact of not considering such vulnerabilities.

During the study, only phpSAFE was able to detect SQLi vulnerabilities (i.e., RIPS did not report any vulnerabilities of this type). It detected 10 vulnerabilities, from which 2 were considered false positives after the manual analysis (see TABLE IV). As SQLi data represents as little as 2% of all the vulnerabilities found, in the following subsections we focus on the detection of XSS data.

a) Generic analysis

As mentioned before, we first configured the two tools for generic PHP vulnerabilities, by leaving the RIPS default setup untouched and removing the WordPress configuration features from phpSAFE. The results of running the tools with such configuration are summarized in TABLE V. As shown, the two tools detected correctly a similar number of vulnerabilities, although RIPS is not prepared for OOP and some plugins are developed in OOP (43 source code files). In fact, the presence of OOP in some of the target files may mislead the RIPS detection mechanism. This may justify the reason why this tool presents a false positives rate higher than phpSAFE.

Another important question that should be analyzed is the overlap of the output of the tools: do the tools detect the same vulnerabilities or should we use several tools to im-

TABLE III. Tools Overall Vulnerability Detection

Vulnerabilities	Total	RIPS	phpSAFE
#	356	135	316
%	100%	37.9%	88.8%

TABLE IV. SQLi detection

	Vulnerability detection	False positives	Vulnerability detection accuracy
RIPS	0	0	-
phpSAFE	8	2	80%

TABLE V. Tools configured for generic XSS detection

	Vulnerability detection	False positives	Vulnerability detection accuracy
RIPS	135	81	62.5%
phpSAFE	131	46	74.0%

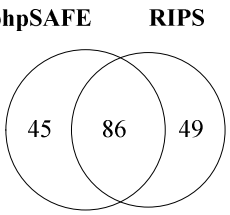

Figure 3. phpSAFE and RIPS configured for generic XSS detection

prove the global coverage? This is addressed in Figure 3 that shows a Venn diagram where the radius of each circle is proportional to the number of vulnerabilities, providing a comparative visual image of the coverage of each tool. Overall the tools detected 180 distinct vulnerabilities. In the diagram we can see a fair amount of the same vulnerabilities detected by both tools, which are represented by the intersection of the circles. There are around half of such vulnerabilities (86 out of 180). Each tool detected many vulnerabilities that the other did not (45 and 49, respectively for phpSAFE and RIPS), confirming the well-known idea that there is no silver bullet, as also stated by other studies [21].

b) Targeted analysis

As a second step we tuned both tools to include the analysis of WordPress related variable input sanitization. The results are summarized in TABLE VI. As shown, RIPS detected the same vulnerabilities as in the first step (i.e., without tuning). This is due to the fact that WordPress is built using OOP, so the input and sanitization functions are based on class methods and they were used in the new configuration. Since RIPS is not prepared for OOP it is natural that it may not be able to use the new set of configuration parameters properly. On the other hand, as phpSAFE is prepared for OOP, there is a huge improvement in the results. In fact, it was able to detect correctly more than twice the number of vulnerabilities. Furthermore, the overlap of vulnerabilities reported by the two tools, presented in the Venn diagram of Figure 4, shows that phpSAFE was even able to detect six more vulnerabilities that RIPS also found in the first step of the experimental process. We found that these situations have to do with OOP methods that, although RIPS is not prepared to deal with, were pointed as vulnerabilities. On the other side, phpSAFE was able to detect them only when configured to deal with WordPress specificities.

B. Plugin impact analysis

From the aggregated results of the test of the 35 WordPress plugins by the two static code analysis tools, we obtained 348 XSS and 8 SQLi true vulnerabilities. To the best of our knowledge, these were previously unknown vulnerabilities that are being disclosed to the public for the first time in this paper. Due to false positives, they were confirmed by a subsequent manual analysis. This data is shown in TABLE VII and represents an average of more than 10

TABLE VI. TOOLS CONFIGURED FOR WORDPRESS: XSS DETECTION

	Vulnerability detection	False positives	Vulnerability detection accuracy
RIPS	135	81	62.5%
phpSAFE	305	63	82.9%

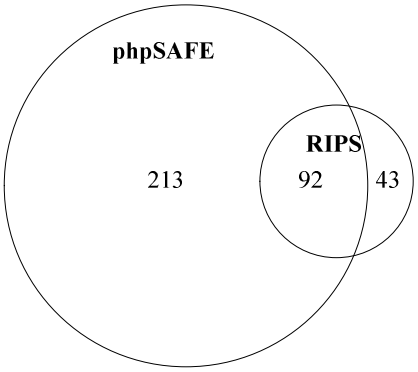

Figure 4. phpSAFE and RIPS adapted for WordPress XSS detection

vulnerabilities per plugin. The vulnerabilities were found in 20 plugins, which is 57% of all the plugins analyzed. On average, each vulnerable plugin has 17.8 vulnerabilities, which corresponds to one vulnerability for each 207 lines of code.

PHP is a weak typed programming language, which means that any variable may contain data from any type. Although this gives a lot of flexibility, it also poses serious security problems when not used properly. Almost half (41%) of the vulnerable variables are used to store numeric values, but they have no restriction about what values they may store. This is inline with findings from other studies that found 45% of numeric variables in the vulnerability fixes done by developers [23]. These variables open the door for attackers to use them to freely carry the exploitation text. In a SQLi attack the attacker may change the query and in a XSS he may change the structure of the page. These situations are usually easier to exploit than a text variable, because numbers are not usually enclosed by quotes or double quotes, which need to be overcome by the attacker. The ease of exploitation of numeric variables may also explain why they are extensively targeted by attackers [30]. An interesting aspect is that they are very easy to fix: in PHP this can be done using `$var = intval($var);` or `$var = (int)$var;` after `$var` received the value from the outside.

We also found that the number of distinct variables that are vulnerable is only 127, which averages a variable reuse of 2.8 times. The largest number of times a single variable name was used in the plugins was 24 times. The reuse of variables in consecutive exploitation by hackers is also a common situation [22]. In those cases the developer frequently protects one instance and forgets the others, allowing the attacker to use a similar exploit to take advantage of the use of the same variable elsewhere. The variable reuse also means that a single fix at the origin of the variable is likely to solve several vulnerabilities at once. Since there was only a few SQLi, all the reutilization occurred in XSS and converting the values that come from the outside to HTML entities (for example, convert `>` to `>`) as soon as possible is a best practice that should be followed by practitioners.

To better understand the source of the vulnerabilities, we made an extensive analysis on how such data is stored in the variables and used by the plugins. We observed that the sources of data might be classified in the following types, regarding the apparent ease of exploitation:

TABLE VII. PLUGIN VULNERABILITY ANALYSIS

WordPress Plugin	Vulnerability		Numeric Variable	Distinct variables	Origin of the Vulnerable Input							Indirect Output
	XSS	SQLi			POST	GET	POST/GET/ COOKIE	DB	File	Function	Array	
calendar v1.3.2	24		10	8				24				1
contextual-related-posts v1.8.6	2			2	2							
digg-digg v5.3.4	6			4	3	3						3
easy-adsense-lite v6.06	2			1		2						
events-manager v5.3.8	30		2	14	1	16	6	7				19
feedweb v1.8.8	9	1	3	5	1	6		2		1		3
funcaptcha v0.3.7	8			2		8						
jaspreetchahals-coupons-lite v2.1	32		14	4	1			31				14
login-with-ajax v3.0.4	2			2		2						
mail-subscribe-list v2.0.9	5	1	2	3	1	2		3				
newsletter v3.2.7	2	1	1	2		2		1				2
paypal-digital-goods-monetization-powered-by-cleeng v2.2.13	6	2	5	7			6	2				5
qtranslate v2.5.34	26	1	5	16	5	11	5	1	1	5	1	9
securimage-wp v3.2.7	2			1		2						
trafficanalyzer v3.3.2	3	2		3		1	2	2				
underconstruction v1.08	1			1		1						
videojs-html5-video-player-for-wordpress v3.2.3	1			1								
wp-photo-album-plus v5.0.2	64		44	16	2	28		34	1			4
wp-symposium v13.02	102		54	30	3	12	1	86				6
wp125 v1.4.9	21		7	5	3			18				3
Total	**348**	**8**	**147**	**127**	**22**	**96**	**20**	**211**	**2**	**6**	**1**	**69**
Average	**17.4**	**0.4**	**7.35**	**6.35**	**1.1**	**4.8**	**1**	**10.55**	**0.1**	**0.3**	**0.05**	**3.45**

1. **Likely to be directly manipulated by users, through POST, GET or COOKIES**. These types of vulnerabilities are usually targeted by occasional hackers and script kiddies, but are also massively exploited [22]. From a cost/benefit perspective of the attacker the question is: why try to find a vulnerability difficult to exploit when there are so many low hanging fruits out there? We found 138 of such vulnerabilities, all of them XSS, which represent 39% of the vulnerabilities detected. It was a surprise to find so many of these vulnerabilities, since they are very easy to spot by the developer (and the attacker), specifically those where the input is used directly as output, without passing through any other variable. These vulnerabilities are easy to exploit because the attacker only has to manipulate the form fields in the application interface, the URL, use specific tools like proxies (e.g., Paros, WebScarab), web browser plugins (e.g., Cookie Manager, Cookie Editor) or even frameworks like the Metaspoit. Moreover, all of these tools (and many others) can be found organized, integrated and ready to be used in Linux distributions like Kali. To illustrate this type, let's consider the following example adapted from wp-symposium plugin:

```
<input type="hidden" name="redirect_to" value="<?php echo $_GET['u']; ?>" />
```

We can observe the direct use of the input, without any sanitization, clearly showing a XSS vulnerability. To successfully exploit it, before the payload, the attacker only has to close the double quote and the input tag with `">playload`.

2. **Indirectly manipulated by users, but they have an easy access to them, like the database**. Database values are easily altered by the interface of the application that was specifically developed for that function. Many times what is stored in the database is only free of SQLi (using prepared statements, for example) so it cannot alter the query executed; but it is often forgotten that the information stored will eventually be used by other parts of the application and that this may also be exploited. One common case is XSS, where the SQLi filters have no effect on what can be used to change the structure of the web page. This kind of exploitation has the ability to be permanent and to affect many users of the application because every time a page uses the information stored, the attack is executed and that user becomes another victim. We found 211 of such cases, which represent 59% of our data. This value is not larger because sometimes the database query only returns a number (an ID, a count or a sum), which cannot be used to exploit (at least from a non convoluted way). Looking at the code, it was rare to find a plugin developed with concerns about the quality of the information that comes from the database. In complex systems with many users, built around COTS that come from different places, that have no previous validation and anyone can freely use, we cannot have a

guarantee that what is stored in the database is clean and safe to use. For such reasons, the output from the database should be treated as potentially evil, as any direct input. To illustrate this type, let's consider the following example adapted from wp-photo-album-plus plugin:

```
$image = $wpdb->get_var(
$wpdb->prepare("SELECT %s FROM ..."));
echo stripslashes($image);
```

We can observe concerns about preventing SQLi, but not about XSS, since the `stripslashes` does not prevent this type of problems.

3. **Unlikely to be easily manipulated, like operating system files, the core application or plugin functions and other variables like arrays**. Although these situations may be less prone to attacks due to the increased difficulty in taking advantage of them, we have seen advanced attacks using them, specially file manipulation [22]. Also, previous considerations about the quality of the plugins should advise developers to not trust the output of functions that come from the outside. Even when not intended to cause harm, sometimes they do, like the case of two security plugins that allowed attacks to be performed due to bugs in their code [22]. To illustrate this type, let's consider the following example adapted from qtranslate plugin:

```
$res = fgets($fp, 128);
echo $res;
```

We can observe the direct use of the content of the file, without any sanitization, clearly showing a XSS vulnerability. To successfully exploit it, the attacker has to be able to either change the content of the file or change the chosen file to one he controls.

During code review, some situations may mislead the reviewer into thinking the variable is safe when it is not. This may occur because of the complexity of the code, the difficulty on following the flux of the variable, or because the variable is passed as argument of functions and the final variable seems to be completely different from the original. One such situation that may give the code reviewer a false sense of security occurs when the variable seems to disappear inside a function when it is one of the arguments (that may also be defined in other source code files), when its value is just assigned to another variable returned by the function. From our data, we accounted for 69 of such situations and we displayed them in the column Indirect Output in TABLE VII.

Taking into consideration the entry point of the attack we can argue that many vulnerabilities may not be exploited if the attacker is unable to manipulate the files, the return of WordPress functions or the database. That is, if other protection mechanisms in place prevent these actions from occurring, the vulnerabilities would be instead "just" code that did not follow best coding practices, but without presenting a direct real danger. This can, however, change dramatically if the previous assumptions do not hold, either by a new bug discovered, a change in the architecture or by finding a new way of exploitation. These are some of the reasons that justify the need to follow the defense-in-depth paradigm [32] and fix all the code constructs that may originate a vulnerability, either with a known exploitation path or not.

C. Lessons learned

Although the number of downloads of the plugins analyzed is quite high (more than 11 million), the results presented cannot be easily generalized, as the set of plugins may not be representative of the collection of plugins developed for all the existing web applications. They may not even be representative of WordPress plugins, at least quantitatively. There are, however, qualitative reasons to believe that key security problems in wide spread web applications are due to plugins and not to the core web application, and this is also shown by other studies [5][22]. In fact, one major conclusion is that web plugin developers need to improve the security of their plugins, by following best practices for security.

Another important aspect is that many large web applications are developed using OOP (see TABLE I for the Top 5) and there should be more PHP static code analyzers prepared for it. This is confirmed by the observation that a tool not prepared for OOP, as expected, does not perform well in an OOP environment, leaving many vulnerabilities undetected. Also, we see the need to use several detection tools in order to increase the overall coverage of the vulnerabilities, which is again in line with other studies [21]. In practice, current PHP static analysis tools need to be improved and, besides targeting generic PHP vulnerabilities, they should be adapted (or adaptable) to the major core web application platforms.

We are aware that even the aggregate results of both tools do not cover all the vulnerabilities thoroughly. Although this best effort may not deliver a perfect solution for all the security problems, major core web applications should enforce a detailed security check on the plugins available. Even if the static code analysis tools available are not good enough to assure that a plugin is free of vulnerabilities, the security of those plugins can be hugely improved.

As a result of our work we sent reports on the vulnerabilities we identified to the plugin developers and many of them were analyzed, according to their responses. Furthermore, as a consequence of our findings, some (or all) of the vulnerabilities were already fixed. In fact, from the feedback received, we can confirm that 14 vulnerabilities from the plugins Mail-subscribe-list and Funcaptcha were quickly fixed and new versions of the plugins are already available (although many others may also have been fixed without our knowledge). On its own, this may improve the security of over 71 thousand installations (value obtained from the downloads of the plugins) and many more users of these blogs. As a collateral outcome, we should add as a recommendation for administrators of the various WordPress sites, the need to update the plugins as soon as the updates are released. There are lots of security fixes in the various versions of the plugins and attackers may also abuse this information to do a differential analysis to find the vulnerabilities that were fixed and to develop exploits for them, which is indeed a common practice.

VI. Conclusion & Future Work

In this paper we analyzed the security vulnerabilities of 35 WordPress plugins using two static analysis tools: RIPS and phpSAFE. More than 350 XSS and SQLi previously

unknown vulnerabilities were detected and 14 quickly fixed thanks to this work. From the vulnerabilities detected, 138 can be considered as very easy to exploit as they are directly related to user inputs. This confirms that plugins are a potential source for security problems even in the context of well tested and widely used web applications, like WordPress.

Results also show that the effectiveness of static analysis tools needs to be improved, both in terms of coverage and false positives. Furthermore, when possible the tools should be tuned for the specific context of the plugins and the core web application being tested and not only regarding generic programing language constructs. This can provide more than the double of detection rate. Due to the high prevalence of vulnerabilities, the security of plugins should be enforced and implemented by the developers and the core application providers by performing static analysis before releasing the plugins to the public. Web site administrators should also update the plugins as soon as new releases are deployed.

Future work includes the analysis of plugins of other common CMS and the development of a repository of CMS aware list of functions that can be used to configure the static code analyzers for their plugins.

REFERENCES

[1] A. Wiesmann, M. Curphey, A. van der Stock, and R. Stirbei, "A Guide to Building Secure Web Applications and Web Services", V2.0.1, OWASP Foundation, 2005

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, "Basic concepts and taxonomy of dependable and secure computing", IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004

[3] Automattic, http://automattic.com/, visited in November 2013

[4] B. Chess, J. West, "Secure Programming with Static Analysis", Addison-Wesley Professional, 2007

[5] Checkmarx, "The Security State of WordPress' Top 50 Plugins", June 2013

[6] D. Ceara, "Detecting Software Vulnerabilities Static Taint Analysis" Vérimag - Distributed and Complex System Group, Polytechnic University of Bucharest, September 2009

[7] D. Stuttard, and M. Pinto, "The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws", Wiley, 2007

[8] ESA, "ESA Guide for Independent Software Verification & Validation", ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/ecss/ESAISVVGuideIssue2.029dec2008.pdf, visited in November 2013

[9] http://blog.sucuri.net/2013/04/wordpress-plugin-social-media-widget.html, visited in November 2013

[10] http://codex.wordpress.org/Data_Validation, visited in November 2013

[11] http://community.websense.com/blogs/securitylabs/archive/2012/03/02/mass-injection-of-wordpress-sites.aspx, visited in November 2013

[12] http://en.wordpress.com/stats/, visited in November 2013

[13] http://seclists.org/fulldisclosure/2012/Dec/242, visited in November 2013

[14] http://us3.php.net/manual/en, visited in November 2013

[15] http://wp.tutsplus.com/tutorials/creative-coding/data-sanitization-and-validation-with-wordpress/, visited in November 2013

[16] http://www.darkreading.com/database/hackers-timthumb-their-noses-at-vulnerab/231902162, visited in November 2013

[17] https://github.com/nikic/PHP-Parser, visited in November 2013

[18] IBM Global Technology Services, "IBM Internet Security Systems X-Force® 2010 Trend & Risk Report", IBM Corp., 2011

[19] IEEE Computer Society, "1012-2004 - IEEE Standard for Software Verification and Validation", June 2005

[20] J. Dahse, November 2013, "RIPS", http://rips-scanner.sourceforge.net/

[21] J. Fonseca, M. Vieira, H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks", Pacific Rim International Symposium on Dependable Computing, December 2007

[22] J. Fonseca, M. Vieira, H. Madeira, "The Web Attacker Perspective – A Field Study", IEEE 21st International Symposium on Software Reliability Engineering, November 2010

[23] J. Fonseca, M. Vieira, "Mapping Software Faults with Web Security Vulnerabilities", IEEE/IFIP Int. Conference on Dependable Systems and Networks, June 2008

[24] J. Fonseca, November 2013, "phpSAFE", https://github.com/JoseCarlosFonseca/phpSAFE

[25] K. Ivan, "Software vulnerability analysis", PhD Thesis, Purdue University, 1998

[26] M. Howard, D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2003

[27] M. Howard, D. LeBlanc, and J. Viega, "19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them", McGraw-Hill Osborne Media, 2005

[28] N. Jovanovic, C. Kruegel, E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities", IEEE symposium on security and privacy, pp. 258-263, 2006

[29] N. Patwardhan, E. Siever, S. Spainhour, "Perl in a Nutshell", Second Edition, O'Reilly, ISBN 0-596-00241-6, December 1998

[30] N. Sam, www.owasp.org/images/7/7d/Advanced_Topics_on_SQL_Injection_Protection.ppt, 2006

[31] Netcraft, "Web Server Survey", http://news.netcraft.com, visited in November 2013

[32] NSA, "Defense in depth", http://www.nsa.gov/ia/_files/support/defenseindepth.pdf, 2004

[33] NTA, March, 2011, http://www.nta-monitor.com/posts/2011/03/01-tests_show_rise_in_number_of_vulnerabilities_affecting_web_applications_with_sql_injection_and_xss_most_common_flaws.html, visited in May 2013

[34] OWASP Foundation, "OWASP top 10", July 2010

[35] R. Zhang, S. Huang, Z. Qi, H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery" Computers & Mathematics with Applications Journal, Vol. 63 Issue 2, pp. 469-480, January 2012

[36] S. Christey, R. Martin, "Vulnerability Type Distributions in CVE", Mitre report, May, 2007

[37] S. Neuhaus, T. Zimmermann, "Security Trend Analysis with CVE Topic Models", International Symposium on Software Reliability Engineering, pp. 111-120, 2010

[38] w3techs, http://w3techs.com/technologies/overview/content_management/all/, visited in November 2013

[39] w3techs, http://w3techs.com/technologies/overview/programming_language/all, visited in November 2013

[40] Walden, J., Doyle, M., Welch, G., Whelan, M., "Security of Open Source Web Applications" International Symposium on Empirical Software Engineering and Measurement, 2009