2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

phpSAFE: A Security Analysis Tool for OOP Web Application Plugins

Paulo Nunes, José Fonseca
CISUC, University of Coimbra
UDI, Polytechnic Institute of Guarda
Portugal
pnunes@ipg.pt, josefonseca@ipg.pt

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

***Abstract*—There is nowadays an increasing pressure to develop complex web applications at a fast pace. The vast majority is built using frameworks based on third-party server-side plugins that allow developers to easily add new features. However, as many plugin developers have limited programming skills, there is a spread of security vulnerabilities related to their use. Best practices advise the use of systematic code review for assure security, but free tools do not support OOP, which is how most web applications are currently developed. To address this problem we propose phpSAFE, a static code analyzer that identifies vulnerabilities in PHP plugins developed using OOP. We evaluate phpSAFE against two well-known tools using 35 plugins for a widely used CMS. Results show that phpSAFE clearly outperforms other tools, and that plugins are being shipped with a considerable number of vulnerabilities, which tends to increase over time.**

Keywords—Static analysis; web application plugins; security; vulnerabilities

I. INTRODUCTION

The number and importance of web applications is growing exponentially fast in a world where web browsers are ubiquitous and used by everyone. Nowadays, almost everything is available, displayed, discussed, shared, processed or traded on the web and the demand for building new web applications is enormous. In fact, millions of people rely on web applications, either for work or leisure, accessing and managing sensitive information like financial and personal data.

The demand for developing web applications with increased complexity in very tight time constraints is making a huge pressure to build more at a lower cost. Many web applications are built on top of Content Management Systems (CMS) frameworks that can be easily deployed and customized to meet the requirements of a myriad of different scenarios, like personal web sites, blogs, social networks, webmail, banking, e-commerce, etc. To cope with this diversity, most CMS-based applications can be extended and configured with third-party server-side plugins provided by multiple developers.

The number of CMS applications is huge and there are thousands of plugins provided by third-party developers. In practice, developers with unknown agendas and uncontrolled software programming skills implement these plugins, which ultimately leads to suspicious trust levels. The problem becomes even worse as core CMS providers do not run quality assurance procedures on the third-party plugins they use, even for those made available directly on untrusted web sites. The only guarantee is related with the comments and ratings of the plugins by their end users and the number of downloads. This is clearly not enough and the reality is that no one can fully trust on the security of the plugins. Not surprisingly, many of these plugins can be exploited by malicious minds, which is the case in many situations [1].

Source code review is a key activity to improve the security of a software product. This can be confirmed by its relevance in the most important secure software development lifecycles, like Microsoft SDL, CLASP and Software Security Touchpoints [2]. However, source code review is a resource intensive task that is only feasible if supported by automated tools. There are several tools that can be used, but the vast majority of plugin developers cannot afford the expensive commercial source code analyzers. Although they can use free tools, like RIPS or Pixy, a key limitation of these tools is the absence of capabilities to analyze Object Oriented Programming (OOP) code, which is nowadays largely used for developing CMS applications [3]. Another drawback is the lack of knowledge on the CMS framework for which the plugin is being developed. Indeed, when analyzing the plugin, such tools are not aware of input and output vectors and of filtering functions included in the API of the CMS framework. These limitations lead to vulnerabilities being left undetected and at the same time to the generation of many false alarms.

In this paper we present phpSAFE, a source code analyzer for PHP based plugins able to detect Cross Site Scripting (XSS) and SQL Injection (SQLi) vulnerabilities. phpSAFE is a follow-up of a project whose development was requested by Automattic, the developer of WordPress [4], with the goal of improving the security of a number of plugins. The tool was developed from the ground up with OOP and plugin security in mind, thus including OOP concepts like objects, properties and methods. To evaluate phpSAFE we compared its ability to detect plugin vulnerabilities with two well-known free tools, RIPS and Pixy. As target CMS framework we selected WordPress because it accounts for about 23% of the all the web sites [4] and has a market share of approximately 60% among all CMS [5]. From the vast collection of plugins available we selected 35 with different sizes and complexities. As we also wanted to study how the tools behave regarding the evolution of plugins over time, we considered two versions of each plugin: one from 2012 and another from 2014.

Results show that phphSAFE is able to detect more vulnerabilities than the other tools, with fewer false alarms. We also observed that both phpSAFE and RIPS deal well with the evolution of plugin code. A key observation is that plugins that are currently being used in thousands of WordPress installations have dangerous XSS and SQLi vulnerabilities and this number is increasing over the years. In fact, we discovered more than 580 vulnerabilities in the plugins analyzed, many of them very easy to detect and exploit. We also verified that developers did not fix many vulnerabilities even after knowing them for more than

PEst-OE/EGE/UI4056/2014 – 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

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.16

one year.

The outline of this paper is as follows. The next section introduces background concepts and relevant related work. Section III presents the phpSAFE tool. Section IV details the methodology used to evaluate phpSAFE and Section V discusses the results. Finally, Section VI concludes the paper.

II. Background and Related Work

XSS and SQLi are two of the most common vulnerabilities found in web applications and are also widely exploited by hackers and the organized crime [6][7]. A XSS attack consists of the malicious injection of HTML and/or other scripting code (frequently Javascript) in a vulnerable web page. 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. Both XSS and SQLi 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, etc.

Static code analysis is a technique that consists in inspecting the source code of software to detect defects. These defects may be flaws that prevent fulfilling the software specification, but they may also be related to security problems. In fact, static analysis is considered by many as the most efficient way to automatically locate vulnerabilities in software developed for the web [8][9][10][11]. However, static analysis has some limitations regarding the precision of the results, when compared to other techniques, like dynamic analysis. In fact, it suffers from the conceptual limitation of *undecidability* that does not affect dynamic analysis [12]. Besides, precise static techniques are computationally expensive, which may be a drawback when analyzing large applications or a large volume of files. Moreover, static analysis requires access to the source code while dynamic analysis assesses the software as a black box. On the other hand, dynamic analysis may leave a large percentage of the code untested due to the difficulty in viewing all possible decision flows and boundary conditions. Static code analysis uncovers security problems by looking at the source code without executing it, so it may achieve 100% code coverage, being able to analyze all the possible execution paths. In practice, it is able to find problems in parts of the code that dynamic analysis may never reach.

Static analysis uses the concept of taint data to locate vulnerabilities [10][13]. Taint data starts with variables that come from an uncontrolled environment (the source), which can be maliciously manipulated from the outside [13]. When a tainted variable is used by the software in some sensitive way (the sink) an attack becomes possible. The data flow between sources and sinks is modeled and analyzed using several program analysis techniques, such as inter-procedural and context-sensitive data flow analysis [12]. During the process, tainted data may propagate to other program variables, making them also tainted. Also, tainted variables may become untainted using a variable validation process that is dependent on the variable type and on the vulnerability type being prevented.

Performing static analysis requires building and analyzing a Control Flow Graph (CFG) of the execution of the program. This is achieved by applying techniques like:

- *Context sensitive data flow analysis* – the analysis takes into account all possible paths of execution of the program by considering all conditional jumps.
- *Inter-procedural or global analysis* – tainted data can reach a function from its parameters, user input variables, other functions, and from global variables, so it depends on the global state of the program. The analysis also verifies if the function is able to sanitize the tainted data.
- *Intra-procedural or local analysis* – this follows the same procedure as the inter-procedural analysis, but it only processes the inside of the function, without considering the context from which the function is called.
- *Functions summaries* – a function is parsed only once. The summary of this analysis is reused in subsequent calls to determine the effects on the context of the calling code.
- *Whole-program analysis* – a function is parsed every time it is called. A way to perform this is to replace each function call by the function body (inline function), which results in a huge program. Consequently, this method requires a lot of memory and processing power.

Several static analysis tools have been developed to detect vulnerabilities in PHP code. Jovanovic et all developed Pixy, a Java tool to detect XSS and SQLi vulnerabilities [10]. Pixy uses a flow-sensitive, inter-procedural and context-sensitive data flow analysis to determinate if user data reaches sensitive sinks without being fully sanitized. It performs precise alias and literal analysis to refine the taint process and improve the precision of the detection, but it does not parse Object Oriented constructs. Pixy is a command line tool and provides a text-based report of the vulnerabilities offering several verbosity levels.

Dahse and Holz developed RIPS, a tool based on the specificities of the PHP language that performs a comprehensive analysis and simulation of built-in language features, such as PHP functions, taking into account only the called arguments that have to be traced [8]. It also includes information about user input variables, sensitive skins, sanitization functions, secure an unsecure PHP built-in functions, and other PHP features. Furthermore, RIPS performs a context-sensitive string analysis based on the current markup context, source type, and PHP configuration. RIPS is based on the abstract syntax tree of the PHP script and performs intra- and inter-procedural analysis to create the respective control flow graph, which consists of linked basic blocks and branches according to conditional program flow analysis. RIPS is able to perform backward-directed taint analysis for 20 different types of vulnerabilities, including XSS and SQLi. However, the tool does not parse PHP objects, consequently it misses encapsulated vulnerabilities in modern OOP based web applications and plugins [15].

Huang e el., the pioneers of static analysis, developed a tool called WebSSARI [9]. It uses a lattice of security levels to track the taintedness of variables through the program. It was made unavailable in 2006 and further advances were implemented in the commercial tool CodeSecure. PHP-SAT is an open-source static analysis tool that uses intra-procedural data flow analysis, developed by Bouwers written in Stratego/XT, however there is no stable release and its development ceased in 2007 [16]. Fortify 360 is a commercial tool that works by compiling program code to a generalized intermediate language and building a model from that. It allows detecting more than 300 categories of vulnerabilities in 17 programming languages, but has a high licensing cost.

III. THE PHPSAFE TOOL

phpSAFE is a static code analyzer for detecting XSS and SQLi vulnerabilities in PHP plugins, including the ones developed using OOP [18]. The only requirement to run phpSAFE is a local web server with the PHP interpreter enabled and a web browser. phpSAFE has a web interface that allows the end-user to specify search and output options, and performs vulnerability scanning in PHP applications and plugins. The output of the analysis is presented in a web page that helps reviewing the results, including the vulnerable variables, the entry point of the vulnerability in the source code PHP file, the flow of the vulnerable data from variable to variable, etc.

phpSAFE is prepared to be easily integrated with the software development process of other PHP projects. For example, the use of phpSAFE can be part of the software development lifecycle of a company, it can be used to automate the process of analyzing a large quantity of PHP scripts residing in different locations, it can be tuned to produce and store the results in other formats or distribute them over the network, etc. This integration ability is easily achieved by including phpSAFE in a PHP project, like an API. Since phpSAFE is developed in OOP, its functions become accessible through the instantiation of a single PHP class called *PHP-SAFE*, which receives as input the PHP file to be analyzed and delivers the results in the properties of the object instantiated from the *PHP-SAFE* class.

phpSAFE source code analysis is based on four stages: 1) configuration, 2) model construction, 3) analysis, and 4) results processing, as illustrated in Fig. 1.

A. Configuration stage

During this stage, phpSAFE loads the configuration data, containing the list of vulnerabilities (currently XSS and SQLi) correlated with the PHP language functions, and the target CMS framework specific functions that may have an effect in these vulnerabilities. In the configuration of phpSAFE, these functions are organized in four main sections: the potentially malicious sources (the entry point of the attack), the sanitization and filtering functions used by the target application to prevent attacks, revert functions (that revert the actions of the sanitization and filtering functions, therefore allowing the attack), and sensitive output functions (where the attack manifests itself). It is by comparing this data with the actual code of the target PHP file that phpSAFE is able to locate the vulnerabilities.

phpSAFE is deployed with a default configuration that is ready for detecting generic XSS and SQLi vulnerabilities, as well as for plugins for the WordPress framework. This solution, out-of-the-box, has the advantage of allowing the immediate use of the tool to analyze PHP code, either from applications or plugins without requiring further configuration. However, this ability can be easily extended to other CMSs, by adding their input, filtering and sink functions to the configuration files. phpSAFE configuration data is as follows:

a) class-vulnerable-input.php: contains the potentially unsecure input vectors. The sources can be PHP user input variables (e.g. $_GET, $_POST), input functions like PHP file functions (e.g. *file_get_contents*), database manipulations functions (e.g. *mysql_query*), or WordPress functions (e.g. *$wpdb->get_results*).

b) class-vulnerable-filter.php: contains the sanitization functions that can be used to protect (untaint) the variables (e.g. *intval*, *htmlentities*, *mysql_escape_string*) and the functions that revert those protections (e.g. *stripslashes*).

c) class-vulnerable_output.php: contains the PHP functions or language constructs that may be exploited by an attacker. Each entry is specific to a given vulnerability type, so it is affected by variables manipulated to take advantage of that vulnerability (like *echo* for XSS and *mysql_query* for SQLi).

The generic XSS and SQLi functions in these configuration files are based on the default configurations of the RIPS tool [8]. Additionally, phpSAFE configuration files contain WordPress specific functions and class methods related with XSS and SQLi, although data for other CMSs can be easily added to the configuration. In fact, this is what it takes for phpSAFE to be able to analyze plugins from other CMSs.

B. Model construction stage

In this stage, phpSAFE performs a lexical and semantic analysis based on the Abstract Syntax Tree (AST) of the PHP source code. The AST is obtained using the PHP function *token_get_all* that splits the PHP code into tokens. Each token can be an array with three items or a string: a) the array has the token identifier, the value of the token and the line number of the PHP script (e.g. [310, $_POST, 11]); b) the string represents a code semantics (e.g. ";"). The function *token_name* is used to get the token's name (e.g. "T_VARIABLE").

phpSAFE builds an AST for each PHP file being analyzed. Then it cleans the AST by removing comments and extra whitespaces. As the PHP file can include other PHP files recursively, all of them must be analyzed in order to obtain the complete AST. To speed up the analysis and the ability to cope with plugin code, phpSAFE collects from the AST information about all user-defined functions and their parameters, all the called functions, among other relevant data. This allows, for example, obtaining the list of plugin functions that are not called from the code of the plugin. However these functions should be parsed anyway, as they may be directly called from the main application. This ability to analyze all the functions, even those not called from within the plugin, is a very important aspect of security tools targeting plugin code.

C. Analysis stage

The objective of this stage is to follow the flow of the tainted variables from the moment they enter the application/plugin until they reach the output. While the input is any GET, POST, COOKIE, database values, files, etc., the output may be the display of the variable in a web page, the storage of the variable in an OS file or the database, etc. During this process, the tainted variable may contaminate recursively other variables that should

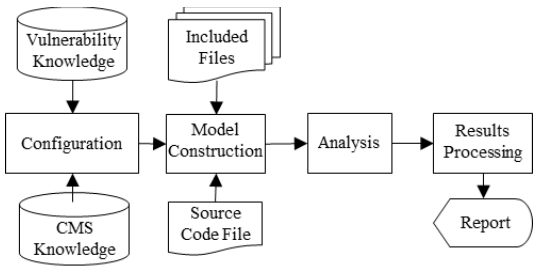

Fig. 1. phpSAFE architecture.

also be followed until they are finally outputted. On the other side, the malicious content of the variable may also be removed or neutralized, preventing its exploitation.

The data flow history of each variable is stored in the multi-dimensional associative array *parser_variables*. This array contains everything needed to allow phpSAFE to perform the taint analysis, like the variable[1] name, source file name and line number, the dependencies from other variables, if it is an input or output variable, the filter functions applied, etc.

To gather the data needed to fill the *parser_variables* array, phpSAFE follows the flow of the tainted variables. The tool parses all the AST files previously created and makes decisions based on code constructs like conditionals, loops, assignments, expressions, function or method calls, function or method returns, etc., which is done by following the path of the code, usually starting from the "main function". Furthermore, phpSAFE is able to parse plugins that do not have a "main function" or include functions that are never called directly from the plugin code. To reach 100% code coverage, all the functions should be analyzed, even those that are never called. To address this, phpSAFE starts by executing an inter-procedural parsing of the functions that are not called from the source code of the plugin. Then it performs the inter-procedural analysis starting from the "main function" and follow the program flow from there. This way, every piece of code of the plugin is analyzed.

The intra-procedural parsing goes through every token of the AST and parses it according to its nature. The most important tokens are described in the following paragraphs.

T_VARIABLE, T_GLOBAL. If a variable declaration is found, then its properties are extracted from the AST and stored into the *parser_variables* array, taking into account the data obtained from the configuration files. The variable can be an input, output or just a regular variable, may be tainted or untainted, and may depend on other variables (taking into account the scope), etc. The variable may derive its properties from the last time it was used in the same scope of the source code. An important situation occurs when a tainted variable is being merged with HTML code: this may represent a XSS vulnerability, so it is classified as a vulnerable output variable.

Equal symbol "=". The variable on the left side of the equal symbol depends on all variables and functions present in the expression on the right side of the equal symbol. This expression is parsed recursively, adding new variables and dependencies to the *parser_variables* array. The properties of the new variables may affect the classification and the dependencies of the variable on the left side, including the taint state.

Call of a PHP or CMS framework built-in function call. This refers to calling functions whose source code is from the outside of the plugin, either from PHP or from the CMS framework API. If a call to an output function (e.g. *echo*, *print*) is identified, then the function is checked for tainted variables that could cause a vulnerability. This is performed by adding to the *parser_variables* all the variables in the arguments as output variables. These are tainted according to the relation of the function with the vulnerability type (XSS or SQLi). If a call to a filtering function is found, then a new variable with the same name of the function is added to the *parser_variables*, corresponding to the return of the filtering function. This variable is classified as untainted and as being filtered by the function.

Call of a plugin user-defined function. If the user-defined function of the plugin has not been parsed yet, then it is parsed. The parsing begins by connecting the called arguments to the function's parameters. This is done by adding a variable to the *parser_variables* and a dependency to the corresponding argument. Then the parameters of the function are linked to the local variables propagating the effects of the arguments to inside the function. Afterwards, the function is parsed as any other block of code. Functions that are called recursively are parsed only once to avoid endless loops. The same happens when there is a call to a function that has already been analyzed. To optimize the speed and memory consumption of the analysis, every function is analyzed only the first time it is called, taking into account the context (parameters, global variables, scope, etc.) of the call. The data flow of the variables of this analysis is used to process future calls. Whenever the function is called, this data flow is added to the *parser_variables*, which is updated based on the calling arguments.

T_RETURN. When an expression is returned from the user-defined function, a new variable, with the same name of the function, is added to the *parser_variables*. This variable is updated with the properties of the resulting expression.

Conditional jumps (T_IF, T_ELSE, T_ELSEIF, T_SWITCH) and Loops (T_FOR, T_WHILE, T_DO, T_FOREACH). Conditions and loops do not change the data flow. Only the values of the variables involved are processed and updated. Also, the blocks of code are parsed normally.

T_UNSET. Unsetting a PHP variable means destroying it, therefore, the properties of the variable are updated as untainted and marked as non-vulnerable.

D. Results processing stage

One of the objectives of source code security analysis is the identification of vulnerabilities so they can be fixed. phpSAFE provides several invaluable resources to help in this task. Some of these resources are related to the variables (vulnerable variables, output variables and all the other variables), functions, PHP files included, tokens (the complete AST) and debug information. This data can be very useful in helping security practitioners to trace back the path of the tainted variables to the point they entered the system and locate the best place to fix the vulnerabilities found.

E. Supporting Object Oriented PHP

Since version 5.0, PHP implements several object oriented features like classes, objects, properties, methods, inheritance and override of methods. To cope with OOP is a very important matter, since all the plugins analyzed access OOP, even if they are developed using procedural programming. This happens because WordPress is developed using OOP, and plugins need to use the methods and attributes of existing WordPress objects. Moreover, some of these methods retrieve data from likely to be untrusted sources. All OOP vulnerabilities we found are, indeed, related with WordPress objects and method calls. phpSAFE detects these vulnerabilities because it is able to locate the calls to

[1] Since phpSAFE deals with OOP files, it parses variables and properties, functions and methods. To avoid unnecessary repetitions, whenever possible, we are going to refer to them in the text generically as variables and functions.

the WordPress methods defined in the configuration files, which the other tools are unable to do.

During the parsing of each token, phpSAFE is able to distinguish between variables and properties, functions and methods, and act accordingly. For properties and methods it obtains the full name by adding the name of the object to which they belong through a backward search in the AST (by following the T_OBJECT_OPERATOR and T_DOUBLE_COLON tokens). Each property is then parsed as a variable. The call to a method, including object creation with the PHP *new* construct, is parsed as a function by locating the source code of the called method inside the class.

The following code shows an example adapted from the *mail-subscribe-list 2.1.1* plugin, now fixed thanks to our work.

```
$results = $wpdb->get_results("SELECT * FROM ".
$wpdb->prefix."sml");
foreach($results as $row)
  echo '<tr> <td>'.$row->sml_name.'</td></tr>';
```

The call of the method *$wpdb->get_results* retrieves a set of database rows and stores them in the *$results* array. The *foreach* extracts each data row into the *$row* object, and finally the *echo* outputs the property *sml_name* without any filtering or sanitization. Failing to detect the method *$wpdb->get_results* prevents finding this vulnerability. The database data comes from other subscribers and it is not sanitized when stored. Consequently, any subscriber can inject malicious code into the database. When a victim visits the page of the mail subscribe list, the malicious code is injected in his web browser, executing the attack (which we confirmed in a experiment).

IV. Evaluation Methodology

This section presents the experiments conducted to evaluate the phpSAFE tool and thus understand its strengths and weaknesses. Given the current web scenario with many plugin based web applications developed with OOP and PHP and the existence of other static code analyzer tools, there are three main questions that we are addressing in this evaluation:

1. *How does phpSAFE performance compares with other free static analysis tools when analyzing open source plugins for an OOP developed web application, considering the most common and widely exploited vulnerabilities?*
2. *How does phpSAFE cope with the evolution of plugin code and vulnerabilities over a two-year-period of time, by looking at different versions of the same plugin?*
3. *Are plugin developers taking into consideration the vulnerability disclosure results in subsequent versions of the plugins, even for vulnerabilities easy to spot and exploit?*

A. Metrics analyzed

A static analysis tool should detect correctly all the existing vulnerabilities and be silent when there are no vulnerabilities left, which is very hard to achieve. The evaluation of this class of binary classification tools is normally based on the following set of metrics: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). Obviously, the best tool presents the highest values on the TP and TN and the lowest values on FP and FN. Since there are no perfect tools, a combination of these metrics should be used to characterize static analysis tools. In this work we considered the following:

1. *Precision*: represents the exactness of the classification of vulnerabilities, calculated by applying the following formula: `TP/(TP+FP)`. A high Precision means that the vulnerabilities found are likely to be true vulnerabilites.
2. *Recall*: represents the probability of the detection tool to classify existing vulnerabilities as vulnerabilities, calculated using the formula: `TP/(TP+FN)`. A high Recall means the tool leaves very few vulnerabilities undetected.
3. *F-score*: represents the harmonic mean of Precision and Recall. In practice, it represents a balance between both metrics: `2*(Precision*Recall)/(Precision+Recall)`.

Besides these important metrics that relate with the ability to detect vulnerabilities, the outcome of other metrics, like responsiveness and robustness, may also be determinant for the selection of a tool [19]. Responsiveness is related with the CPU time taken to analyze a target file. Robustness is the ability to finish the analysis and produce a result. In fact, it is very important that a tool is able to analyze any given file and deliver the results in due time using a reasonable amount of resources.

B. Procedure

To address the research questions presented we defined an experimental procedure based on five steps:

1) Selection of a widely deployed OOP web application with many open source plugins available. WordPress is developed in PHP and is the most widely used CMS [5], supporting the creation of web sites like TED, NBC, CNN, The New York Times, Forbes, eBay, Best Buy, Sony, TechCrunch, UPS, CBS Radio, etc. There are millions of WordPress sites, and they account for 23% of the web [4].

2) Gathering current and two-years-old versions of the web application plugins (from 2012 and 2014). We selected a set of 35 WorPress plugins, which is a reasonable number that allows both the execution of the experiments (including a manual verification of all the vulnerabilities reported by the static analysis tools), and that could be representative enough to obtain meaningful results. The reasoning behind the selection of the plugins was to include a very diverse set, as discussed in [3]. The need for the two versions of each plugin is justified by the goal to study if the tools are able to maintain their performance as the code of the plugins evolves and also to understand how the security of the plugins evolves over time. The older plugin versions should have the vulnerabilities reported to their developers to understand what actions were taken to mitigate them. In fact, the 2012 version of the plugins selected were analyzed in 2013 and the vulnerabilities found were communicated to the developers by then [3].

3) Selection of well-known free security static analysis tools for web applications for comparison purposes. RIPS and Pixy are two of the most referenced PHP static analysis tools, although they are not ready for OOP analysis. They also have been subject of several scientific publications [14], [7], [3]. RIPS has been developed over the past years until 2014, but Pixy has not been updated since 2007.

4) Execution of phpSAFE and the other tools to search for vulnerabilities in the collection of plugins. Different tools have different ways of being executed, as they have diverse features, configurations and user interfaces. Whenever possible we automated the execution of each tool considering the following aspects (to prevent bias, besides the options below, we executed the tools with their default configuration):

- phpSAFE provides a class that can be easily integrated in other PHP projects, so we developed a PHP script to automate the process. It executes phpSAFE for all the plugin files and collects the results into a single log file.
- RIPS has to be interacted through its rich web interface to search for vulnerabilities one file at a time. It was configured with verbosity level "show secured +1,2" (for all the origins of the malicious input) and vulnerability type "SQL Injection and Cross-Site Scripting" (since it has many other vulnerability types).
- Pixy is executed through a command line interface so we developed a shell script to automate the process. By configuring Pixy flags it is able to store all the results as a set of log files. We also activated the "–A" flag that enables the tool to cope with the PHP reference operator "&" (ex. $variable1=& $variable2).

5) Analysis of the results. As each tool delivers the results in a specific format we normalized and merged all of them into a single repository. The vulnerabilities reported by the tools were manually verified by a security expert (a computer science PhD student) looking for misclassification issues, which was a labor intensive and time consuming task. This is however, an important quality assurance aspect, given that it is well known that automated security tools typically originate a high number of false alarms. The total number of vulnerabilities detected by the tools and confirmed manually are those that we consider as the real set of vulnerabilities the plugin have. Due to the unfeasibly large amount of work, a manual analysis of the plugins looking for the vulnerabilities missed by the tools was not performed. Therefore, the data about the number of vulnerabilities detected and missed by the tools gives an optimistic view of the performance of the tools. This is a best effort result that may be improved using more tools, for example. Each vulnerability confirmed by the expert was also further analyzed in order to obtain more information about its characteristics, like input vectors, vulnerable variable domain and if it is present in both versions of the plugins.

V. PHPSAFE EVALUATION

This section presents the results of the experiments, starting with an overall analysis and then going into the details.

A. Overall analysis

Table I depicts the global results obtained by executing phpSAFE, RIPS and Pixy with all plugins. For each tool there are two columns: one for the most recent version of the plugins (V. 2014) and another for the older version (V. 2012). The table presents the values of the chosen metrics: TP, FN, Precision, Recall and F-score. The FN is needed to calculate the Recall and this implies knowing all the vulnerabilities present in the plugins. As mentioned before, we did not search the plugins for all the possible vulnerabilities using all possible means, like a thorough manual code review and using commercial tools. Therefore we considered as the FN of one tool the vulnerabilities that it did not detect but were detected by the other tools. So, the value of the Recall metric is also optimistic.

Of the 35 plugins analyzed, 19 are developed in OOP. phpSAFE found 151 vulnerabilities related to the use of WordPress objects in 10 plugins of the 2012 version, and 179 vulnerabilities in 7 plugins of the 2014 version. RIPS and Pixy were not able to detect any vulnerability of this kind.

TABLE I. VULNERABILITIES OF 2012 AND 2014 PLUGIN VERSIONS

		phpSAFE		RIPS		Pixy	
		V. 2012	*V. 2014*	*V. 2012*	*V. 2014*	*V. 2012*	*V. 2014*
XSS	**True Positives**	307	374	134	288	50	20
	False Positives	63	57	79	47	185	197
	Precision	83%	87%	63%	86%	21%	9%
	Recall	85%	68%	37%	53%	13%	4%
	F-Score	84%	76%	47%	65%	16%	5%
SQLi	**True Positives**	8	9	0	0	0	0
	False Positives	2	5	0	1	0	0
	Precision	80%	64%	-	0%	-	-
	Recall	100%	100%	0%	0%	0%	0%
	F-score	89%	78%	-	-	-	-
Global	**True Positives**	315	387	134	304	50	20
	False Positives	65	62	79	79	187	208
	Precision	83%	86%	63%	79%	21%	9%
	Recall	80%	66%	34%	52%	13%	3%
	F-score	81%	75%	44%	63%	16%	5%

Results show that phpSAFE is the tool that detects more vulnerabilities (True Positives). It is followed by RIPS and Pixy, which has the lowest detection value. The detection rate trend ranking is also followed by the other metrics (Precision, Recall and F-Score), for which phpSAFE also has the highest values followed by RIPS and Pixy. This means that phpSAFE is able to detect vulnerabilities more exactly (Precision) and that it leaves less vulnerabilities undetected (Recall) than the other tools. phpSAFE also has the best F-Score among the tools. These results show that phpSAFE should be the tool chosen for all situations, from business-critical applications to the less critical scenarios. However, it still leaves vulnerabilities undetected, so other assurance activities should also be used.

This ranking of phpSAFE, RIPS and Pixy holds also true when considering 2012 and 2014 versions of the plugins. One of the reasons for the detection performance of phpSAFE is its ability to cope with OOP and its out-of-the-box configuration for WordPress plugins. Comparing the two versions of the plugins, it seems that phpSAFE and RIPS are up-to-date with current programming practices, as they detected more vulnerabilities in recent plugins. Conversely, Pixy detected less, maybe due to the lack of upgrades since 2007.

phpSAFE was the only tool able to detect correctly SQLi vulnerabilities, which is an odd result, since both RIPS and Pixy are also capable of detecting SQLi. This may be due to the small number of SQLi in the plugins analyzed, which may not have triggered the detection mechanism of these tools. An interesting result is the 115% increase in XSS detection by RIPS from the 2012 to the 2014 version. This occurred because RIPS was able to detect vulnerabilities in some files of the 2014 versions that phpSAFE was unable to parse because these files had many includes and required a lot of memory.

In spite of the poor results, Pixy still found a considerable number of vulnerabilities not detected by the other tools. It is also the only tool that was concerned about the *register_globals = 1* PHP directive. In fact, half of the vulnerabilities it found were due to this directive. However, nowadays the PHP default configuration is much safer than it was back in 2007 when Pixy

was last updated. Critical configuration variables are now deprecated, not available or they have safer default values, like the *register_globals*, which is currently deactivated by default in the PHP configuration file. This may justify why the other tools did not detected this vulnerability type.

During the analysis process we also observed that although phpSAFE and RIPS are able to detect vulnerabilities in functions that are not called from the plugin code, Pixy is unable to do so. This may be common in plugins because some functions are to be called from the main application, so this is a feature that all tools prepared for analyzing plugins should have.

B. Vulnerability detection overlap

Since no tool presents a 100% Recall, using more tools should allow detecting more vulnerabilities. Conversely, it is also likely that different tools may report some common vulnerabilities. In fact, we found such situations during the manual verification of the vulnerabilities. They are depicted in Fig. 2 that shows a Venn diagram having the radius of the circles proportional to the number of vulnerabilities, providing a comparative visual image of the coverage of each tool.

Combining the results of all tools we detected 394 distinct vulnerabilities in 2012 versions and 586 in 2014 versions. This is an increase of 51% in just two years. In the diagram we can see that, although some vulnerabilities were reported by several tools (represented by the intersection of the circles), different tools also detected many different vulnerabilities. This confirms the well-known idea that there is no silver bullet to solve all security problems [20]. Furthermore, during the manual verification, additional vulnerabilities were found (represented by an empty circle in the figure). As this was not done systematically we do not have this data accurately defined. However, the fact that there are vulnerabilities that were not detected by either one of the tools reinforces the need for performing other types of vulnerability detection analysis, besides using automated tools. Many researchers and practitioners also advise the use of other security practices like security training, manual code reviewing, black-box testing, etc. [2].

C. Root cause of the vulnerabilities

To better understand the root cause of the vulnerabilities present in the most recent version of the plugins, we made an extensive analysis on how the malicious data reaches the code of the plugins. This was done by following the reverse path of the tainted data, from the vulnerable variable until it reaches the boundary of entrance of the plugin. We observed that the sources of data might be classified into three types, regarding the apparent ease of exploitation (see also Table II):

1. *Likely to be directly manipulated by attackers, through POST, GET or COOKIES.* These types of vulnerabilities can be easily discovered and exploited by occasional hackers and script kiddies, so they are likely to be massively exploited [17]. There are 211 of such vulnerabilities, representing 36% of all the vulnerabilities found. We consider this a huge value, due to the security danger of mass exploitation and because they are also usually quite easily spotted by plugin developers. To illustrate this type, consider the following example adapted from *wp-symposium* plugin:

```
'<p>Created '.$_POST[ 'img_path' ].'.</p>';/>
```

2. *Indirectly manipulated by attackers, but that can be easily accessed by them, like the database.* These vulnerabilities may be prevented if the manipulation of database values is properly protected. However, quite often developers do not care with parameterized queries to prevent SQLi (among other good practices). Other times they prevent SQLi, but they forget blended attacks where the malicious string is stored in the database, but is use to exploit other vulnerability types, like XSS. Although this may be more difficult to exploit than the previous one, its effect is usually more devastating, because it is persistent and may affect many users at once. For these reasons, it may be the preferred attack vector for professional hackers and the organized crime. There are 363 (62%) of such cases in the plugins analyzed, meaning that most developers do have a high confidence on what is coming from the database. Consider the example adapted from *wp-photo-album-plus* plugin:

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

3. *Unlikely to be easily manipulated by attackers, like operating system files, the core CMS framework or plugin functions and other variables like arrays.* These situations may be less prone to attacks due to the increased difficulty in taking control over the resources needed to exploit the vulnerability. It may even need a chained exploitation of other vulnerabilities, which may require a high level of expertise to be successful. We have, however, seen advanced attacks using them, specially file manipulation [17]. We only found 11 (1.8%) of such vulnerabilities in our dataset. Consider the following example adapted from *qtranslate* plugin:

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

Like many other scripting languages, PHP is weak typed, so variables are not restricted to a single data type (e.g., one variable may store interchangeably an integer or a string). This flexibility also poses serious security problems when not used properly. From the vulnerable variables of the plugins, 39% are meant to store numeric values, but they do not have any check to restrict what values they may store. This is inline with other studies that found 45% of numeric variables in the vulnerability fixes done by developers [2]. These numeric variables are usually easier to exploit than text variables, because numbers are not enclosed by quotes or double quotes when used in the code (that have to be bypassed to exploit the variable). This may explain why integer variables are widely exploited [17].

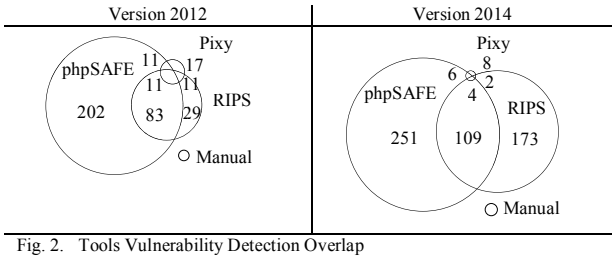

Fig. 2. Tools Vulnerability Detection Overlap

TABLE II. MALICIOUS INPUT VECTOR TYPE

Input Vectors	Version 2012	Version 2014	Both versions
POST	22	43	11
GET	96	111	36
POST/GET/COOKIE	24	57	19
DB	211	363	162
File/Function/Array	41	11	4

D. Inertia in fixing vulnerabilities

One of the quality assurance activities that should be done while maintaining software during its lifecycle is fixing bugs, giving priority to those that are more critical, like security issues. The vulnerabilities found in the 2012 version of the plugins were initially disclosed to the developers in November 2013 [3]. In the present study we analyzed which of the vulnerabilities found in the 2014 version were among the ones previously disclosed in the 2012 version. We found that 249 (42%) of the vulnerabilities discovered in the 2014 version are among the ones discovered and disclosed to the developers more that one year ago. From those, 59 (24%) are very easy to exploit (through GET, POST or COOKIE manipulation). This is a disturbing result that should raise the awareness of plugin developers, of maintainers of the CMS frameworks, of the site administrators, and of the end users.

E. Responsiveness and robustness of the tools

All tests were performed on an Intel Core i5 2.8 GHz with 8 GB of memory, running OS X Yosemite 10.10.1. The CPU time that each tool took to analyze all the 35 plugins is showed in table III. The time values are an average of five runs. RIPS needed 0.8 seconds per K Lines Of Code (LOC) to analyze the 2012 version of the plugins and phpSAFE took about 0.2 seconds per KLOC, but missed the analysis of one file. To analyze the 2014 version of the plugins phpSAFE took 1.0 second and RIPS took approximately the same time. Thus, we are positive that phpSAFE and RIPS should scale to larger files.

Overall the 2012 version of the plugins had 266 files analyzed with a total of 89,560 LOC, and the 2014 version had 356 files with 180,801 LOC. RIPS succeeded in completing the analysis of all files, while phpSAFE was unable to analyze one file in the 2012 version and three files in the 2014 version. Pixy failed to complete the analysis on 32 files. Moreover, Pixy raised one error message in the 2012 versions and 37 in the 2014 versions, probably because it is an old tool and does not recognize OOP code.

VI. Conclusion and Future Work

In this paper we presented phpSAFE, a source code vulnerability analyzer that is able to detect both XSS and SQLi vulnerabilities in plugins of applications developed in PHP with OOP. There are other free tools to search for vulnerabilities in PHP code, like RIPS and Pixy, but they are neither ready for OOP nor for analyzing plugins. As WordPress applications are so common, we evaluated phpSAFE, RIPS and Pixy with a set of 35 WordPress plugins, according to Precision, Recall and F-score metrics. Due to its novel features, phpSAFE outperformed the other tools. The experiments also showed that using many tools allows increasing the number of different vulnerabilities detected, showing that there is room for improvement.

We used two versions of the 35 plugins to analyze how the tools cope with the evolution of the code. phpSAFE and RIPS did not show a relevant change in their detection performance, but Pixy had a significant decrease, possibly due to its lack of updates since 2007. Also, we were able to notice a 50% increase in the number of vulnerabilities in just two years. A more critical observation is that 40% of all the vulnerabilities found in the updated plugins were already present in the older version, even for those vulnerabilities that were disclosed to the developers more than one year ago.

Future work includes the improvement of phpSAFE, mainly regarding performance, memory consumption and vulnerability coverage, along with the analysis of other CMS applications like Drupal or Joomla. We also intend to study the evolution of plugin security and plugin updates over time by enabling historic data in phpSAFE. Developers may use it for approving third-party plugins before allowing their integration.

TABLE III. Detection Time of All Plugins In Seconds

phpSAFE		**RIPS**		**Pixy**	
Ver. 2012	*Ver. 2014*	*Ver. 2012*	*Ver. 2014*	*Ver. 2012*	*Ver. 2014*
17.87	180.91	69.42	178.46	49.57	106.54

References

[1] Khan, Huda and Shah, Deven and Risk "Webapps Security With RIPS", International Conference on Electrical, Electronics Engineering 9th December 2012, Bhopal, ISBN: 978-93-82208-47-1

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

[3] José Fonseca, Marco Vieira, "A Practical Experience on the Impact of Plugins in Web Security", The 33rd IEEE Symposium on Reliable Distributed Systems (SRDS 2014), Nara, Japan, October 6-9, 2014

[4] Automattic, http://automattic.com/, visited in March 2015

[5] w3techs, http://w3techs.com/technologies/overview/content_management/all/, visited in November 2014

[6] N. Nostro, A. Ceccarelli, A. Bondavalli, and F. Brancati, "Insider threat assessment: A model-based methodology," SIGOPS Oper. Syst. Rev., vol. 48, no. 2, pp. 3–12, Dec. 2014.

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

[8] J. Dahse e T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, USA, 2014.

[9] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, S. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," Proc. 13th International Conference on World Wide Web, NY, USA, 2004.

[10] 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

[11] G. Wassermann, Z. Su, "Static detection of cross-site scripting vulnerabilities", 30th Int. Conference on Software Engineering, 2008

[12] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, et al., "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications", IEEE Symposium on Security and Privacy, SP 2008

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

[14] V. B. Livshits e M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis", Usenix Security, 2005.

[15] Vogt, Philipp, et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis." NDSS. 2007

[16] Eric Bouwers. Analyzing PHP, "An Introduction to PHP-Sat". Technical report, 2006.

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

[18] J. Fonseca, November 2014, "phpSAFE", https://github.com/JoseCarlos-Fonseca/phpSAFE

[19] N. L. de Poel, "Automated security review of php web applications with static code analysis", Master's thesis, vol. 5, 2010.

[20] J. Fonseca, M. Vieira, chapter "A Survey on Secure Software Development Lifecycles", Khalid Buragga, Noor Zaman (Eds.), "Software Development Techniques for Constructive Information Systems Design", ISBN: 9781466636798, IGI Global, 2013.