GuruWS: A Hybrid Platform for Detecting Malicious Web Shells and Web Application Vulnerabilities

Van-Giap Le, Huu-Tung Nguyen, Duy-Phuc Pham, Van-On Phung, and Ngoc-Hoa Nguyen(✉)

VNU University of Engineering and Technology, Hanoi, Vietnam
{giaplv_57,tungnh_57,duyphuc,onphungvan,hoa.nguyen}@vnu.edu.vn

Abstract. Web application/service is now omnipresent but its security risks, such as malware and vulnerabilities, are indeed underestimated. In this paper, we propose a protective, extensible and hybrid platform, named GuruWS, for automatically detecting both web application vulnerabilities and malicious web shells. Based on the original PHP vulnerability scanner THAPS, we propose E-THAPS which implements a novel detection mechanism, an improved SQL injection, Cross-site Scripting and vulnerability detection capabilities. For malicious web shell detection, taint analysis and pattern matching methods are chosen to be implemented in GuruWS. A number of extensive experiments are carried out to prove the outstanding performance of our proposed platform in comparison with several existing solutions in detecting either web application vulnerabilities or malicious web shells.

Keywords: White-box penetration testing
Web application vulnerability · Web shell · Taint analysis
Pattern matching · SQLi detection · XSS detection · YARA rules

1 Introduction

According to Internet Live Stats up to May 2017 [1], there is an enormous amount of websites being attacked everyday, causing both direct and significant impact on nearly 3.64 billion Internet users. Even with support from security specialists, they continue having troubles due to the complexity of penetration procedures and the vast amount of testing cases in both penetration testing and code reviewing. As a result, the number of hacked websites per day is increasing: from 25.000 hacked websites per day on April 2015 to 56.470 hacked websites per day on May 2017.

Several popular approaches for securing web applications [4] have been investigated, for example safe web development [3], implementing intrusion detection and protection systems, code reviewing, and web application firewalls. Petukhov et al. [10] presented an efficient way for securing web applications by searching and eliminating vulnerabilities therein. In fact, an attack campaign is temporary.

However, attackers might upload their backdoors to that system for persistence, as they can come back to interact and steal information anytime without exploiting any vulnerability. This situation leads to serious consequences [5] since these backdoors are *Web shells*, and they allow to remotely control files, databases and execute commands. They are not only flexible but also countless.

Indeed, lacking of secure programming awareness and of ability to discover both malicious web shells and web vulnerabilities from web developers are main root causes. These current issues in web application security raise a demand for one solution which allows web developers and security penetration testers to detect security-related problems in the easiest way. In this research, we propose **GuruWS**: a hybrid platform for automatically detecting both web application vulnerabilities and malicious web shells using white-box testing approach. Moreover, we focus on web applications written in the PHP language because the popular usage of PHP in server-side programming languages – about 82.6% of all the websites [16].

This paper is an extended version from our previous work [2] in which we improve and clarify more details about our platform and related works. Additional experiments and functional improvements to **GuruWS** have been made at the time writing this paper as well.

The rest of this paper is organized as follows: In Sect. 2, we revise some basic principles, literature research and related work in vulnerabilities scanning and taint analysis. In Sect. 3, we describe our hybrid and extensible platform for automatically detecting both web application vulnerabilities (based on E-THAPS) and malicious web shells (using taint-analysis, pattern matching and YARA rules). In Sect. 4, we present our experiment results, evaluate our work and provide benchmarks. The last section is dedicated to some conclusions and future work.

2 Background and Related Work

In this section, we first describe the concept of detecting web application security risks (resulting from either malicious files or vulnerabilities). Then, two main techniques used in our platform are presented before summarizing some related works.

2.1 Detection of Web Application Security Risks

In general, the security risks of web applications are typically composed of threats that exploit the vulnerabilities and malicious codes inside those applications. Actually, there are two main approaches to hunt for malicious web shells and vulnerabilities in PHP applications [6]:

- White-box testing: An attempt to track down defective and vulnerable chunks of code by analysing application source code. This operation is often integrated into the development process by creating add-on tools for common development environments.

– Black-box testing: The source code is not directly examined. Instead, special input test cases are generated and sent to the application. Then, the results returned by the application are analysed for unexpected behaviour that indicates errors and potential vulnerabilities.

In general, security researchers prefer using black-box vulnerability and malware scanners to find threats in Web applications. These tools are operated by launching attacks against an application using the fuzzing technique [15]. Although often being valuable components when testing the security of a website, they are both time and resource consuming because of fuzzing limitations.

So far, white-box testing has not broadly used for finding security threats in web applications. The main reason is detection capability limitations of white-box analysis tools, in particular due to heterogeneous programming environments and the complexity of applications involving database operations, business logic, and user interface components [6].

However, white-box testing provides an overall picture and better code coverage than black-box analysis does. In some cases, security risks arise when a specific piece of code is mistakenly accessible because the developer failed to follow the expected path flows. Furthermore, applications can be vulnerable if being accessed in an unexpected way that developers does not intend [7].

To better understand the detection of vulnerabilities and malicious codes in web applications, we will discuss two basic techniques, i.e. symbolic execution and taint analysis, in next section.

2.2 Symbolic Execution and Taint Analysis

It is irrefutable that symbolic execution has a significant potential for programmatically detecting wide-ranging classes of security flaws in modern software due to its efficiency. In the past decade, the topic of symbolic execution attracted many security researchers and had been the backbone of almost every security conference around the world. In general, symbolic execution denotes the process of analyzing what inputs cause each part of a program to execute.

In symbolic execution, an interpreter follows the code flow, assigning symbolic values to input rather than obtaining actual input which is generated during normal execution of the program [14]. In an ideal situation, it can simulate all possible outcomes of the application. Consider a piece of code written in PHP as below, which receives user input and trigger the `fail` function if the input value is equal to `"not good!"`:

```
<?php
  $x = $_GET['input']; $y = $x . " good!";
  if ($y == "not good!") {   fail();  }
  else {  echo "good news!";  }
?>
```

During normal execution, this program will retrieve user input (e.g., `"oh"`) and assign it to the `$x` variable. Execution would then proceed with the string concatenation (`$y = "oh good!"`) and routes to the false branch, which print `"good news!"`.

During symbolic execution, the aforementioned program processes a symbolic value (for example: λ) and assigns it to the variable `$x`. The program would then proceed with the concatenation and assign λ . `"good!"` to `$y`. When reaching the `if` statement, it will check λ. `"good!"` with `"not good!"`. λ could take any value at this point of the program; therefore, symbolic execution can proceed along both branches, by ”forking” two different paths. Each path assigns a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is λ. `"good!"` is equal to `"not good!"` for the `True` branch and λ. `"good!"` is not equal to `"not good!"` for the `False` branch. Both paths can be symbolically executed independently.

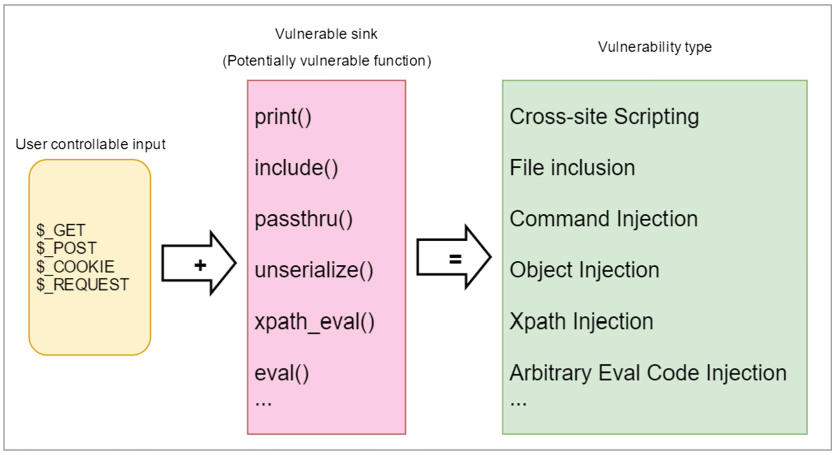

Fig. 1. Concept of taint-style vulnerabilities in PHP

To identify vulnerabilities, a taint analysis is performed after the symbolic execution process. Taint analysis attempts to identify which variables have been “tainted” with user controllable input and how they are propagated through the application. If user input reaches a critical point of the application such as a vulnerable sink, where it is able to alter the outcome of the application, without being properly sanitised, thereafter it will be flagged as a potential vulnerability [7,19]. Specifically, a web application security vulnerability can be denoted when user input data (tainted data) is not appropriately sanitised before being used in the critical operations of the dynamic execution. These kinds of flaws is defined as taint-style vulnerabilities. It is important to notice that any user controllable data must be considered as tainted one. For every vulnerability type, there exists different sanitization routines as well as vulnerable sinks which are potentially vulnerable functions (PVF) that execute critical operations and should only be called within trusted data [8]. Figure 1 is an illustration of taint-style vulnerabilities in the PHP programming language:

2.3 Related Work

Web application penetration testing has been extensively studied in literature however the existing industrial solutions are still struggling with web shell detection and white-box penetration testing. In this section, we briefly introduce some related solutions that we did research on and have been evaluated in the **GuruWS** implementation process:

Web Shell Detector [12] is a Python tool which can detect web shells. It is a good solution for easily using, developing and customizing. However, the web shell pattern set in web Shell Detector's database is outdated and very limited. Moreover, it is not able to detect simple and tiny web shells as well as self-written web shells, due to the lack of a taint analysis mechanism.

NeoPI is a Python script which uses statistical techniques to detect obfuscated and encrypted content within source code. Its approach is based on the recursive scanning and ranking of all files in the base directory [9]. This solution requires manual investigation to validate whether it is a web shell or not.

RIPS[1] is a PHP application with the ability to find vulnerabilities in PHP applications using statistical analysis [11]. By tokenizing and parsing all source code files, RIPS can transform PHP source code into a program model, then use taint analysis to detect sensitive potentially vulnerable functions that can be tainted by user input during the program flow. As in our practical evaluation, RIPS scans very fast, yet, the false positive rate is still high. Some of the limitations are due to missing support for object-oriented code [7,8].

PHP-Parser[2] is a parser for PHP programming language, in which it is used for static analysis, manipulation of code and basically any other application dealing with programmatically coding [18]. This parser has the ability to produce an abstract syntax tree (AST)[3], also known as a node tree, from PHP code. From the acquired AST, it is possible to traverse the syntax tree, manipulate it, and perform necessarily analysis for each node. Additionally, PHP-Parser supports in converting a syntax tree back to PHP code, which can be used in code preprocessing.

sqlmap is an open source penetration testing tool that automates the processes of detecting and exploiting SQL injection flaws and taking over database servers [24]. This tool uses black-box testing techniques and supports various database management systems (DBMS) including MySQL, Oracle, PostgreSQL, Microsoft SQL Server, etc.

THAPS [7] is a vulnerability scanner for web applications written in PHP. It uses symbolic execution as its static analysis approach on which it performs a taint analysis in generated AST to detect web application vulnerabilities. Based on our experiment, THAPS shows lower false positive rate compared to RIPS. However, it is only able to detect SQL injection and XSS vulnerabilities. That motivates us to extend this scanner in order to improve the performance of web application vulnerabilities detection.

[1] http://sourceforge.net/projects/rips-scanner/.
[2] https://github.com/nikic/PHP-Parser.
[3] https://en.wikipedia.org/wiki/Abstract_syntax_tree.

THAPS uses the PHP-Parser library, a parser for the PHP programming language, to generate an AST. Then it performs a taint analysis in the created AST by using the provided functionality of PHP-Parser to traverse the syntax tree, manipulate it, and perform the predefined analysis for each node. A node of the tree represents a part of the source code and describes some certain functionality.

Basic workflow of THAPS is presented in the following diagram (Fig. 2):

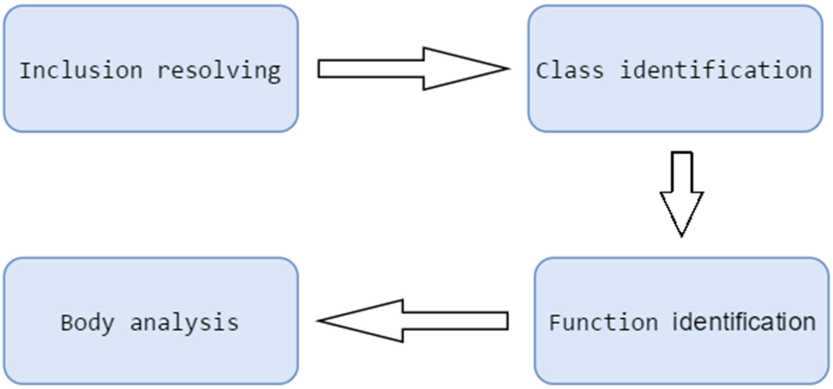

Fig. 2. Basic THAPS workflow

- **Inclusion resolving** is a module in which the inclusion nodes are replaced by the AST generated from the included file.
- **Class identification** module is responsible for identifying all user-defined classes. Moreover, all class members and methods will be also identified and used for further processing.
- **Function identification** is where all user-defined functions are being identified and analysed.
- **Body analysis** is a module in which the global scope, the body of methods and functions are analysed for potential vulnerabilities.

3 Hybrid Platform of GuruWS

In this section, we first propose the motivation of GuruWS, then we will detail our approach to design our proposed platform. Finally, we describe our contributions related to the detection of both malicious web shells and vulnerabilities.

3.1 Motivation

As mentioned in previous sections, web application vulnerability and malicious web code scanners operate independently and do not work together on the same platform. Moreover, existing solutions on scanning vulnerabilities and web shells in web application still have some limitation as described in Sect. 2.3. These reasons motivate us to propose a platform that allows both vulnerabilities and malicious code detection to avoid security risks in web applications.

In order to analyse PHP based web applications to identify the security risks, we propose a hybrid platform named **GuruWS**. It aims to detect both vulnerabilities and malwares in PHP applications by using the white-box testing techniques. And that is why GuruWS is called the hybrid platform.

3.2 Platform Design

The **GuruWS** platform architecture is described in Fig. 3.

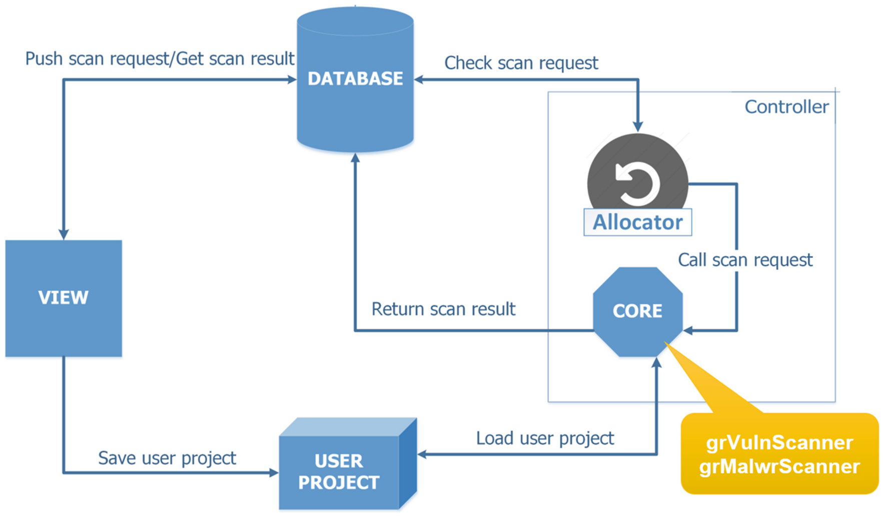

Fig. 3. GuruWS platform architecture

In this platform design:

- **View** is a web front-end service which receives compressed web application source code from the user; then validates, extracts the compressed source to *UserProject* and pushes scanning requests to the *Database*. After final scan results are available in the *Database*, the web service will also retrieves and displays them to the user
- **Database** assumes the role to store scanning requests and scanning reports
- **UserProject** module keeps user projects which are the input resource for *Core* module
- **Allocator** works as a background service, continuously retrieving scanning requests from the *Database*. If there is any available scanning request, it will be sent to the *Core* module to start scanning; the results will be pushed to the *Database* after the scanning processes are done
- **Core** consists of *grVulnScanner* and *grMalwrScanner* modules. Each module executes simultaneously as independent services:

- **grMalwrScanner** is an extensible scanner, in which its main objective is malicious PHP scripts detection. More details of this module are clarified in Sect. 3.3
- **grVulnScanner** is an extensible scanner for white-box web application vulnerability detection. More details of this module are clarified in Sect. 3.4.

The basic work-flow of can be described as following:

1. First, a user uploads his compressed web application project to **GuruWS**'s web UI. After being safely sanitised and extracted to the specified location, a scanning request will be pushed into scanning queue in the *Database*. This queue contains a list of projects which must be scanned by the *Core* module
2. The *Allocator* gets scanning requests from the scanning queue in the *Database* and then calls the *Core* module for scanning on demand of the request
3. After taking user extracted project as an input, the *Core* module scans this data and pushes final results to the *Database*
4. **GuruWS**'s web UI, which regularly checks for the scan progress, then will display scan results immediately after they are stored in the database.

Our platform has the ability to scan several projects at the same time. To achieve this feature, using multi-threading techniques while carefully avoiding race conditions is needed. By distributing the server resources into multiple threads with one separated SQL connection per thread, the race condition is automatically handled by the DBMS. The number of multi-thread projects at the same time can be customized, otherwise we set a default value of 10 scanning processes simultaneous projects due to our computing resources.

The reason that we have to implement the *Allocator* module and wait for the results instead of showing them immediately is because **grVulnScanner** and **grMalwrScanner** take a small amount of time to scan the whole project.

In the next section, we will describe in more detailed methods used in the *Core* module in order to detect the malicious web application files and vulnerabilities.

3.3 grMalwrScanner: Malicious Web Application File Detection

The goal of the **grMalwrScanner** module is to support developers, webmasters and security specialists in order to detect malicious files in their web application project. In their perspective, they will proactively disclose the source codes for answering two questions: (i) *Does their application contain malicious files?* and (ii) *If malicious files exist, where do these files locate in their application?*

To answer those questions, our first step is to detect web shells. Because of their flexibility, we decided to use taint analysis method [19]. Additionally, for other complex scripts, there are many common web shells which were protected by encoding themselves and challenge the taint analysis methods. There is a limited amount of popular web shells, thus we propose another method using pattern matching consisting of a set of strings and a boolean expression which determines its logic.

One key idea in our work is to improve and support all available approaches which are relevant to the corresponding type of web shells. Besides, a common post-exploitation technique, which intruders usually use, is to embed fraud advertisement scripts or malicious HTML frames. Fortunately, **grMalwrScanner** has special modus operandi to detect adware and its variants.

3.3.1 Taint Analysis for Detecting Web Shells

Taint analysis method is performed as following: The code is divided into tokens using a lexical analysis process to make it easier to manipulate and perform post analysis. Thereafter, **grMalwrScanner** analyses the token list of each file once in which it passes through the token list and identifies important tokens by name. Thus, potential dangerous functions (PDFs) are determined, then all significant arguments of these functions will be traced back to their 'source', that includes:

– *Environmental input:* get_headers(), get_browser(), etc.
– *User input:* $_GET, $_POST, $_COOKIE and $_FILES as well as other $_SERVER and $_ENV variables.
– *Server arguments:* HTTP_ACCEPT, HTTP_KEEP_ALIVE, etc.
– *File operations:* fgets(), dlob(), readdir() and so on.
– *Database handlers:* mysql_fetch_array(), mysql_fetch_object(), etc.

In the taint analysis method, we proposed some principles in **grMalwrScanner**'s as follow:

– The source is always marked as tainted.
– The string created from tainted variables is also marked as tainted.
– With a function (not belonging to secure functions or PDFs), if it has any tainted input arguments, its return value will be marked as tainted.
– With every function in the PDF list, there will be a set of corresponding securing functions. Hence, when significant arguments of the PDF are traced back, any argument passed through a securing function will make an untainted return value even though this is a tainted variable.

3.3.2 Pattern Matching for Detecting Web Shells

Regarding the taint analysis approach, **grMalwrScanner** only focuses on PHP web shells detection. We assume a certain file could be a relevant malicious sample only if the two following conditions are met. First, the sample must be written in PHP. Second, it has to be observed that it is working properly and shows some apparent PHP backdoor behavior.

The pattern set is based on powerful YARA rules [17] because of their flexibility and simplicity. Consisting both sub-parts of web shells and particular tricky patterns, our pattern set is very efficient in detecting in-the-wild web shells; this will be evaluated in the experiment section of this paper.

grMalwrScanner is developed to be a collaborative platform which combines the power of not only analysis tools but also social interactions between security analysts. Therefore, a novel feature, which allows analysts to write their own ruleset or refer to a social ruleset, will be helpful.

Consider a common lightweight PHP backdoor as below:

```
<?php  eval($_GET['c']); ?>
```

This is a very simple and effective backdoor regardless of its small size, which receives and executes an attacker's commands through the HTTP GET `c` parameter, and allows attackers to perform remote code execution. Furthermore, consider another tiny backdoor which receives remote commands through HTTP headers:

```
<?php system($_SERVER['HTTP_USER_AGENT'])?>
```

Fortunately, all of these samples above can be detected using our YARA rules as demonstrated below:

```
rule GuruWS_LightWeightBackdoor
{
  strings:
    $ = /\<\?(php)?(.*|\n)system\(\$(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)eval\(\$(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)passthru\(\$(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)shell_exec\(\$(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)exec\(\$(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)fread\(popen\(\$\_(.*)\]\, \'r\'\)\,\$\_(.*)\]\)/
    $ = /\<\?(php)?(.*|\n)pcntl_exec\(\'(.*)'\,array\(\'\-c\'\,\$\_(.*)\]\)\)/
    $ = /\<\?(php)?(.*|\n)preg_replace\(\'(.*)\'\,\$\_(.*)\],(.*)/
    $ = /\<\?(php)?(.*|\n)call_user_func_array\(\$\_(.*)\]\, array\(\$\_(.*)\]\)\)/
    $ = /\<\?(php)?(.*|\n)assert\(\$\_(.*)\]\)/
    $ = /\<\?\=\@\$(.*)\(\$(.*)/
    $ = /\<\?(php)?\$(.*)\=str_replace\((.*)\)\;\@\$(.*)\(\$\_(.*)\]\)/
    $ = /\<\?(php)?\$(.*)\;\@\$(.*)\(\$\_(.*)\]\)/
    $ = /\<\?(php)?\$x\=strrev\(\"(.*)\"\)\;echo \@\$(.*)\(\$\_(.*)\]\)/

  condition:
     any of them
}
```

For instance, adware and its variants can be observed, based on the unique pattern of their obfuscated code.

```
<?php
  echo('<html></html><script>var _0x5264 = ["\x3c\x73\x63\x72
    \x69\x70\x74\x20\x74\x79\x70\x65\x3d\x22\x74\x65\x78\x74\
    x2f\x6a\x61\x76\x61\x73\x63\x72\x69\x70\x74\x22\x20\x73\
    x72\x63\x3d\x22\x2f\x2f\x61\x64\x77\x61\x72\x65\x2e\x63\
    x6f\x6d\x22\x3e\x3c\x2f\x73\x63\x72\x69\x70\x74\x3e", "\
    x77\x72\x69\x74\x65", "\x3c\x73\x63\x72\x69\x70\x74\x20\
    x73\x72\x63\x3d\x22\x2f\x2f\x61\x64\x77\x61\x72\x65\x2e\
    x63\x6f\x6d\x22\x3e\x3c\x2f\x73\x63\x72\x69\x70\x74\x3e
    "];
  document[_0x5264[1]](_0x5264[0]);
  document[_0x5264[1]](_0x5264[2]);</script>');
?>
```

Basically the PHP above is to fetch obfuscated JavaScript in hex format from external host *adware.com*, to confuse and complicate the analysis process. Analysts can customize their default ruleset for their own purpose as simple as the YARA rule below:

```
rule adware001
{
  strings:
    $a = /var [\_0x\d(a-f)(A-F)]* = \[["'](\[\\\x\d(A-F)(a-f)]*)
    ["']/i
    $a = /document\[[\_0x\d(a-f)(A-F)\[\]]*\]\([\_0x\d(a-f)(A-F)
    \[\]]*\)/i
  condition:
    any of them
}
```

For analyst customization, we provide a set of information such as lists of malicious Web application hashes, blacklisted patterns, common YARA rules etc. Moreover, analysts are able to interact and contribute to the social rulesets using the explicit API calls.

Besides, we analyzed and created another web shell statistical analysis module for **GuruWS** in PHP. Our module processes source code and returns a ranking table which is based on the probability of being a web shell.

The calculation is performed on these factors: Entropy, Language IC, Longest Word Test, Signature Potential Test and Signature Malicious Test. In detail, in the Entropy calculation, grMalwrScanner removes spaces then calculates a percentage (p) of each character, then the result will be $-p * \log_2 p$. Result from Language IC calculation will be $s/(t * (t + 1))$, s is the sum of pairs of the number of appearances of two adjacent character in ASCII map, t is the total of the appearance of all character. To calculate the Longest Word, grMalwrScanner ranks these files based on the length of its longest word. To calculate the

Signature Potential Test, grMalwrScanner counts the number of appearances of following malicious strings in the input file.

```
('/(eval(|fileput_contents|base64_decode|python_eval|exec(|
    passthru|popen|proc_open|pcntl|assert(|system(|shell)/i')
```

grMalwrScanner do the same with the following string to calculate Signature Mailicous Test.

```
'/(@\$[]=|\$=@\$_GET|\$[+""]=)/i'
```

We built this module as an optional feature for security experts.

3.4 grVulnScanner: Web Application Vulnerabilities Detection

The **grVulnScanner** module takes the role of web application vulnerabilities detection. For that purpose, we extended and improved the latest version of THAPS engine as described in the Sect. 2.3. The reason we chose THAPS to extend comes from the fact that it has lower false positive rate compared to RIPS and capability to detect SQL injection as well as XSS vulnerabilities. Our extended version of THAPS is named as **E-THAPS** (Enhanced THAPS). E-THAPS is totally superior to the original one, in which it is implemented with new vulnerability detection mechanisms and various improvements:

– Having abilities to detect further web application security flaws namely: Command Injection [20], Object Injection [21], File Inclusion [22], Arbitrary Eval Code Injection [23]. Details are denoted in Sect. 3.4.1
– Being equipped with an improved SQL injection vulnerability detection mechanism. Details are denoted in Sect. 3.4.2
– Being equipped with an improved Cross-Site Scripting vulnerability detection mechanism. Details are denoted in Sect. 3.4.3
– Other improvement to achieve better performance. Details are denoted in Sect. 3.4.4.

3.4.1 New Detection Mechanism

The intended purpose of THAPS is only XSS and SQL injection vulnerabilities detection, hence indeed it is not trivial to improve its vulnerabilities detection capability. Our contributed detection improvement is briefly illustrated in Fig. 4. Additionally, we implemented this new mechanism supporting object-oriented programming flaws detection.

Our contributed detection mechanism particularly consists of the following features:

– PHPParser_Node_Expr_FuncCall handles built-in function nodes which defines how a vulnerability should be detected with new vulnerable sinks and corresponding securing functions.

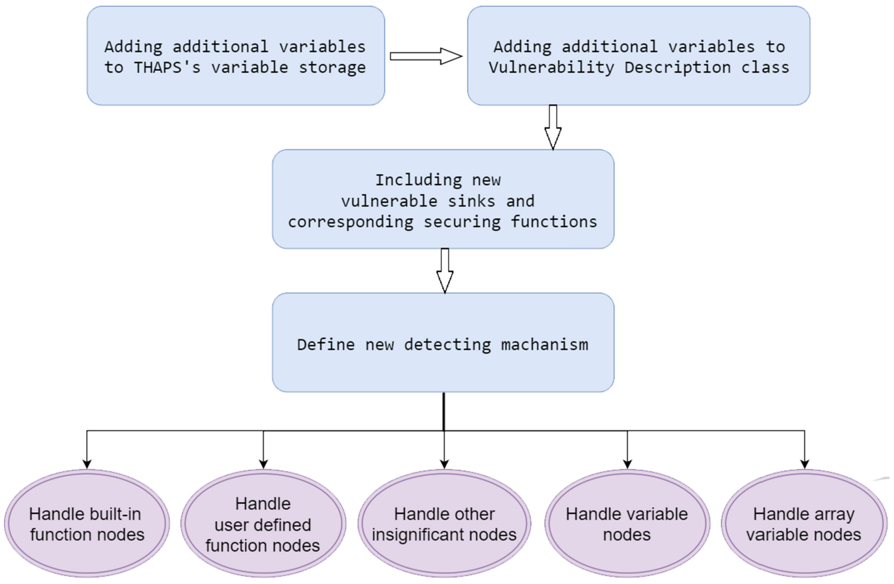
Adding additional variables to THAPS's variable storage
Adding additional variables to Vulnerability Description class
Including new vulnerable sinks and corresponding securing functions
Define new detecting machanism
Handle built-in function nodes
Handle user defined function nodes
Handle other insignificant nodes
Handle variable nodes
Handle array variable nodes

Fig. 4. Process of adding new detecting mechanism

– PHPParser_Node_Expr_MethodCall handles user-defined function nodes, including new variables in the variable storage and all user defined function types.
– PHPParser_Node_Expr_Variable handles variable nodes, considering whether variable comes from user input (e.g: GET, POST, etc.), and marks it as tainted for all new variables in the variable storage.
– PHPParser_Node_Expr_ArrayDimFetch handles array variable nodes, considering whether an array variable comes from user input, and marks every element inside as tainted for new variables in the variable storage.
– Other modifications to other nodes:
  • Conditional nodes (PHPParser_Node_Stmt:If—Elseif—Else)
  • Looping nodes (PHPParser_Node_Stmt:For—While—Foreach)
  • Logical nodes (PHPParser_Node_Expr_Logical:Or—And—Xor)
  • Casting nodes (PHPParser_Node_Expr_Cast:Bool—Int—Double)
  • Concat nodes (PHPParser_Node_Expr_Concat)
  • Assigning concat nodes (PHPParser_Node_Expr_AssignConcat)
  • Scalar Encapsed nodes (PHPParser_Node_Scalar_Encapsed)
  • Assigning plus node (PHPParser_Node_Expr_AssignPlus)
  • Static function (in class) nodes (PHPParser_Node_Expr_StaticCall)
  • Return nodes (PHPParser_Node_Stmt_Return)

The newly contributed vulnerable sinks and corresponding secure functions are listed below:

- **Command Injection**

```
$SINKS = array('exec', 'expect_popen', 'passthru',
    'popen', 'proc_open', 'shell_exec', 'system');
$SEC_FUNC = array('escapeshellarg', 'escapeshellcmd');
```

- **File Inclusion**

```
$SINKS = array('include', 'include_once',
'parsekit_compile_file', 'require', 'require_once',
    'set_include_path', 'virtual');
$SEC_FUNC = array('basename', 'dirname', 'pathinfo');
```

- **Object Injection**

```
$SINKS = array('unserialize', 'yaml_parse');
$SEC_FUNC = array();
```

- **XPATH injection**

```
$SINKS = array('xpath_eval, 'xpath_eval_expression',
    'xptr_eval');
$SEC_FUNC = array('addslashes');
```

- **Arbitrary Eval Code Injection**

```
$SINKS = array('assert', 'eval');
$SEC_FUNC = array();
```

3.4.2 Improved SQLi Detection Mechanism

In the wild, THAPS has some limitations in analyzing projects using modern built-in SQL functions such as prepared statement. Therefore, MySQL-Improved (**MySQLi**) potential vulnerable function were added to the SQLi detection mechanism. There are some further functions which were contributed in the securing functions set for SQL injection:

- **mysqli_real_escape_string(mysqli $link, string $escapestr):** escapes special characters in a string for use in an SQL statement.
- **mysqli_escape_string(mysqli $link, string $escapestr):** alias of the mysqli_real_escape_string() function.

Additionally, **mysqli_query()** function was added to SQL sink because along with tainted arguments, it can lead to SQL injection vulnerabilities.

These simple, yet efficient implementations significantly improved the ability to detect SQL injection vulnerabilities of THAPS. This statement will be proved in the Experiment section.

3.4.3 Improved XSS Detection Mechanism

THAPS' XSS sinks are defined via PHP-Parser nodes:

- PHPParser_Node_Stmt_Echo: represents the **echo()** function.
- PHPParser_Node_Expr_Print: represents the **print()** function.
- PHPParser_Node_Expr_Exit: represents the **exit()** function.

Besides, there are other different functions that can likely be XSS sinks such as **print_r()**, **die()**, **printf()**, **vprintf()**, **trigger_error()** and **user_error()**.

By adding these functions to THAPS's XSS sinks, the XSS detecting capacity of THAPS significantly improved. There were some difficulties when making this inclusion since the arguments' positions in each function is not the same. In the next release version, E-THAPS should be more flexible on adding functions to sinks without considering the variable arguments position.

Furthermore, the **highlight_string()** function was added to the securing functions set for XSS. This implementation reduces the false positive rate of THAPS while scanning for XSS vulnerabilities.

3.4.4 Other Improvements

- **Update secure functions in conditional statements:** This secure function set contains a list of functions used in conditional statements (if/elseif/else) to ensure that passed argument is safe for all sinks.
  There were some further functions added to this set: is_bool(), is_null(), is_finite() and is_infinite().
- **Update secure functions for every vulnerability:** This secure function set contains a list of function names which return untainted values with all passed tainted/untainted arguments. These untainted values are safe with all vulnerable sinks. 39 further functions are included in this set, e.g. *strftime()*, *md5_file()*, *sha1_file()*, etc.
- **Update functions for insecure strings:** This function set contains a list of functions that return tainted value with all passing tainted/untainted arguments. These tainted values are presumably unsafe with all vulnerable sinks. 23 further functions included to this set in total, including: *gzdecode*, *hex2bin()*, *recode()*, *gzinflate()*, etc.

This implementation allows the detection capabilities of **E-THAPS** to become more efficient than THAPS as explained in Sect. 4.

4 Experiment and Evaluation

4.1 Environment Setup and Results

The experiment with **GuruWS** platform was taken on our computing system–Intel Xeon E5-2630L with 2 GB memory. This server runs CentOS 6.7 (2.6.32-642.15.1.el6.x86_64), Python 2.7.10, YARA 3.4.0 [4], PHP 5.6.12, Apache 2.2.15 and MySQL 5.6.35.

[4] https://github.com/plusvic/yara/releases/latest.

GuruWS platform is now published for public use[5]. The source codes and datasets of **GuruWS** are freely accessible[6]. In order to validate **GuruWS**'s competency, we separately evaluated its two core modules **grMalwrScanner** and **grVulnScanner**.

4.2 Evaluation of grMalwrScanner

4.2.1 Evaluation Method

To evaluate our method of malicious Web application files detection to other prevalent anti-virus products, we use the online service of VirusTotal[7] reports via its public API. In this service, it allows to interact with a lot of popular anti-virus engines for analysing and detecting malicious files, including viruses, worms, ... Thus, we built an evaluation system which is illustrated in Fig. 5.

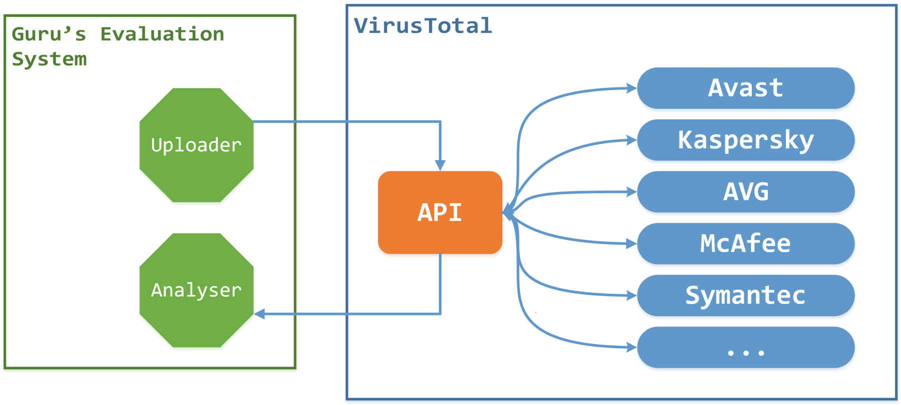

Fig. 5. grMalwrScanner's evaluation system model

In this system, **Uploader** is a script module which gathers web shells from web shell test set, then sends every web shell to the VirusTotal system. Each web shell will have a *scan_id* belonging to VirusTotal's response. **Uploader** keeps these *scan_id* and waits for the scanning progresses of VirusTotal. After the scanning progress is finished, **Analyser** uses the *scan_id* to retrieve the corresponding result containing a list of anti-viruses and their answer when scanning these web shells in JSON format.

To evaluate the ability of **grMalwrScanner**, we will use two different test sets: one contains malicious PHP web shells and one is a collection of clean, benign PHP codes. We will observe the true positive (TP), false negative (FN)

[5] http://112.137.130.56/guruws/.
[6] https://github.com/giaplv57/GuruWebScanner/.
[7] https://virustotal.com/.

and False Positive (FP) scores, then compute the Recall (sensitivity, or true positive rate - TPR), Fall-out (or Fall Positive Rate - FPR), F1-score and Precision with the following formulas [13]:

$$Recall = TPR = \frac{TP}{TP + FN} \quad \textit{Fall-out} = FPR = \frac{FN}{FN + TP}$$

$$F1\textit{-score} = \frac{2TP}{2TP + FP + FN} \quad Precision = \frac{TP}{TP + FP}$$

4.2.2 Datasets

For web shells detection, from reliable sources[8], we collected a wide range of web shells in various types and programming language such as ASP, PHP, Perl, Python etc. Then, we only select PHP web shells and divide this dataset into two parts with the ratio of 7:3 as the rule of thumb [25]. The larger part will be used to construct the training dataset, and the rest, named the testing dataset, will be used to measure the efficiency of both **grMalwrScanner** and other similar scanners. After removing all the irrelevant files in both datasets, the web shells are distributed as follow:

– Number of web shells in training dataset: 1507 (71%)
– Number of web shells in testing dataset: 616 (29%)

In addition, to evaluate the accuracy of **grMalwrScanner** for the benign web applications, we collected 150 clean PHP codes from the official site of WordPress plugins[9]. This clean dataset contains 3,843 benign PHP scripts in total.

4.2.3 Evaluation Results

By using the malicious testing dataset of 616 web shells, Table 1 shows the final results we obtained from **grMalwrScanner** and 58 others scanners:

Note that, in this table, to save the space, we do not show the FP due of its value 0. Apparently the differences of both Recall and F1-score rate from **grMalwrScanner** compared to other scanners although they were tested on the same test set. **grMalwrScanner** achieved an outstanding precision rate (85.56%) and F1-score (0.92), much higher than the rest.

In the case of benign dataset, there is zero FN in **grMalwrScanner**'s untainted files detection (that means its FPR is 0%). This result is the same for the other scanners, except BKAV scanner. The latter produced 18 FN cases for the benign dataset. Thus, the BKAV scanner has its FPR rate 0.47%.

[8] https://sourceforge.net/p/laudanum/code/25/tree/, Github: /tennc/webshell, /shiqiaomu/webshell-collector, /tdifg/WebShell, /BlackArch/webshells, /JohnTroony/other-webshells, /lhlsec/webshell, /fuzzdb-project/fuzzdb, /JohnTroony/php-webshells.

[9] https://wordpress.org/plugins/browse/popular/.

Table 1. Number of detected web shells, F1-score and precision

Scanner	TP	FN	Recall	F1-score	Precision
grMalwrScanner	**527**	**89**	**85.56%**	**0.92**	**85.55%**
Web Shell Detector	407	209	66.07%	0.8	66.07%
AhnLab-V3	320	296	51.95%	0.68	51.95%
Ikarus	284	332	46.10%	0.63	46.10%
AegisLab	278	338	45.13%	0.62	45.13%
GData	273	343	44.32%	0.61	44.32%
Qihoo-360	269	347	43.67%	0.61	43.67%
Avast	266	350	43.18%	0.6	43.18%
Symantec	246	370	39.94%	0.57	39.94%
ALYac	242	374	39.29%	0.56	39.29%
Tencent	239	377	38.80%	0.56	38.80%
ESET-NOD32	235	381	38.15%	0.55	38.15%
Avira	234	382	37.99%	0.55	37.99%
TrendMicro-HouseCall	229	387	37.18%	0.54	37.18%
Baid	221	395	35.88%	0.53	35.88%
Arcabit	220	396	35.71%	0.53	35.71%
MicroWorld-eScan	219	397	35.55%	0.52	35.55%
BitDefender	219	397	35.55%	0.52	35.55%
NANO-Antivirus	218	398	35.39%	0.52	35.39%
Comodo	217	399	35.23%	0.52	35.23%
Ad-Aware	217	399	35.23%	0.52	35.23%
F-Secure	215	401	34.90%	0.52	34.90%
ZoneAlarm	213	403	34.58%	0.51	34.58%
Kaspersky	213	403	34.58%	0.51	34.58%
Emsisoft	213	403	34.58%	0.51	34.58%
TrendMicro	195	421	31.66%	0.48	31.66%
AVG	195	421	31.66%	0.48	31.66%
Sophos	172	444	27.92%	0.44	27.92%
Fortinet	169	447	27.44%	0.43	27.44%
ClamAV	166	450	26.95%	0.42	26.95%
VBA32	142	474	23.05%	0.37	23.05%
DrWeb	140	476	22.73%	0.37	22.73%
McAfee-GW-Edition	134	482	21.75%	0.36	21.75%
Cyren	127	489	20.62%	0.34	20.62%
Bkav	126	490	20.45%	0.33	20.45%
McAfee	125	491	20.29%	0.34	20.29%

(*continued*)

Table 1. (*continued*)

Scanner	TP	FN	Recall	F1-score	Precision
Microsoft	121	495	19.64%	0.33	19.64%
AVware	117	499	18.99%	0.32	18.99%
CAT-QuickHeal	115	501	18.67%	0.31	18.67%
CMC	97	519	15.75%	0.27	15.75%
Rising	96	520	15.58%	0.27	15.58%
ViRobot	95	521	15.42%	0.27	15.42%
F-Prot	95	521	15.42%	0.27	15.42%
VIPRE	74	542	12.01%	0.21	12.01%
Jiangmin	65	551	10.55%	0.19	10.55%
TotalDefense	54	562	8.77%	0.16	8.77%
Yandex	52	564	8.44%	0.16	8.44%
Panda	33	583	5.36%	0.1	5.36%
Antiy-AVL	27	589	4.38%	0.08	4.38%
K7GW	24	592	3.90%	0.08	3.90%
K7AntiVirus	24	592	3.90%	0.08	3.90%
TheHacker	18	598	2.92%	0.06	2.92%
Webroot	10	606	1.62%	0.03	1.62%
Zillya	8	608	1.30%	0.03	1.30%
Kingsoft	4	612	0.65%	0.01	0.65%
nProtect	1	615	0.16%	0	0.16%
Zoner	0	616	0.00%	-	0.00%
SUPERAntiSpyware	0	616	0.00%	-	0.00%
Malwarebytes	0	616	0.00%	-	0.00%

4.3 Evaluation of grVulnScanner

4.3.1 Evaluation Method

To justify the performance of **grVulnScanner** with the E-THAPS scanner, we created a test set including vulnerable code in PHP and do evaluate it with other similar scanner like original THAPS and RIPS v0.55.

For evaluation, we observed a number of TP and FP cases in every test case scanning process. A TP case is defined when the PVF leads to a vulnerability and the scanner reports a vulnerability. Meanwhile, when the PVF does not lead to a vulnerability and the scanner still reports a vulnerability, we consider it as an FP case. F1-score and Recall will be also used to measure the performance of scanners.

The total number of vulnerabilities (TNOV) of every test case is also depicted as well. TNOV is computed by the sum of TP and FN: $TNOV = TP + FN$. Besides, the execution time of these white-box scanners is not significant. Hence, we decided not to mention in the evaluation result.

4.3.2 Datasets

We construct a test set to compare the performance of 3 scanners, including 16 test cases that were built by us. All of them are vulnerable PHP scripts and they cover classical cases of various vulnerability types: *Command Injection, Object Injection, File Inclusion, Arbitrary Eval Code Injection, XSS, SQL Injection*. Some of them also include object-oriented code. Additionally, the testing set contains a series of web application challenges which belong to the BKAV WhiteHat Contests[10] including:

- Web challenge in BKAV WhiteHat Contest 7, belongs to a monthly Whitehat contest of Bkav
- Web challenge in BKAV WhiteHat Contest 8, belongs to a monthly Whitehat contest of Bkav
- Web challenge in BKAV WhiteHat Contest 10, belongs to a monthly Whitehat contest of Bkav
- WhiteHat Grand Prix 2014 scanning result, belongs to an annual Whitehat contest organised by Bkav

These web challenges trustworthy simulate real vulnerable web applications in the wild. All of our 20 test cases are available and freely accessible from GitHub (See footnote 6). Note that in these test cases, there are totally 52 vunerabilites.

To verify the FP rate of grVulnScanner, we collect 4 featured plugins of WordPress[11] (named Akismet Anti-Spam v4.01, WP Super Cache v1.5.8, Theme Check v.20160523.1 and bbPress v.2.5.14). Based on the WPVulnDB[12] and CVE Detail[13], these plugins are considered benign and have no vulnerability. Thus, these plugins constitute the clean/benign test set of 262 PHP files.

4.3.3 Evaluation Result

By using our 20 test cases, we performed also deep experiments to detect web application vulnerabilities by using 3 scanners: E-THAPS (our scanner), THAPS and RIPS. The results we obtained are summarized in Table 2 and listed in detail in Table 3. According to Table 2, E-THAPS has the highest TPR as well as F1-score: 0.893 compared with 0.377 (THAPS) and 0.804 (RIPS). Note that in Table 2, FN is calculated from TNOV subtracted by TP.

[10] https://ctftime.org/ctf/112.
[11] https://wordpress.org/plugins/browse/featured/.
[12] https://wpvulndb.com/plugins.
[13] https://www.cvedetails.com/vulnerability-list/vendor_id-2337/product_id-4096/.

Table 2. Overall results in comparison

Using scanner	TP	FP	FN	Recall/TPR	F1-score
RIPS	41	9	11	78.85%	0.804
THAPS	13	**4**	39	25.00%	0.377
E-THAPS	**46**	5	**6**	**88.46%**	**0.893**

Table 3. Detailed performance comparison of THAPS, E-THAPS and RIPS

Input		THAPS		E-THAPS		RIPS	
Test name	TNOV	TP	FP	TP	FP	TP	FP
Test case 1	2	0	0	2	0	2	0
Test case 2	1	0	1	1	1	1	1
Test case 3	2	0	0	2	0	2	0
Test case 4	1	0	0	1	0	1	0
Test case 5	2	0	0	2	0	2	0
Test case 6	1	0	0	0	0	0	1
Test case 7	2	1	0	2	0	1	0
Test case 8	1	0	1	0	1	0	1
Test case 9	2	0	0	2	0	2	0
Test case 10	1	1	0	1	0	1	1
Test case 11	2	2	0	2	0	0	0
Test case 12	1	0	0	1	0	1	0
Test case 13	2	0	0	2	0	2	0
Test case 14	2	2	0	2	0	2	0
Test case 15	2	0	0	2	0	2	0
Test case 16	2	2	0	2	0	1	0
White-hat 7	3	0	0	2	0	3	0
White-hat 8	4	1	2	4	3	4	3
White-hat 10	4	2	0	3	0	3	1
Grand Prix 2014	15	2	0	13	0	12	1
Total	52	13	**4**	**46**	5	41	9

Following are the detailed results and explanation after we examine 3 scanners to BKAV WhiteHat Contests web applications:

– **Web challenge in BKAV WhiteHat Contest 7**

Table 4. WhiteHat Contest 7 scanning result

Using scanner	Objection injection	File inclusion	Possible flow control
RIPS	1	1	1
THAPS	0	0	0
E-THAPS	1	1	0

THAPS demonstrated the lowest performance since it was unable to detect any vulnerability. RIPS showed the best performance with 3 detected flaws while E-THAPS detected Objection Injection flaw, File Inclusion flaw and unable to detect the Possible Flow Control flaw. The ability to detect Possible Flow Control has not been implemented in E-THAPS yet. Since this feature will be carefully considered since it can lead to FP rate rising in E-THAPS (Table 4).

– **BKAV WhiteHat Contest 8 Web challenge**

Table 5. WhiteHat Contest 8 scanning result

Using scanner	Cross site scripting		SQL injection		File inclusion	
	TP	FP	TP	FP	TP	FP
RIPS	1	2	2	1	1	0
THAPS	1	2	0	0	0	0
E-THAPS	1	2	2	1	1	0

In this test, it is evident that E-THAPS and RIPS are performing equally, however both are better than THAPS in detecting SQL injection and File Inclusion flaws. There also appeared some FP cases in these vulnerabilities as the table shows (Table 5).

– **Web challenge in BKAV WhiteHat Contest 10**

Table 6. WhiteHat Contest 10 scanning result

Using scanner	Cross site scripting		SQL injection	
	TP	FP	TP	FP
RIPS	2	0	1	1
THAPS	2	0	0	0
E-THAPS	2	0	1	0

This time, E-THAPS showed the best performance with 3 identified vulnerabilities: 2 XSS and 1 SQL injection. Additionally, in the detected SQL injection flaw, E-THAPS, unlike RIPS, did not make any False Positive case (Table 6).

- **WhiteHat Grand Prix 2014 web challenge**: This web challenge is based on the attack-defense category of this contest, and it is a practical web application (Table 7).

Table 7. WhiteHat Grand Prix 2014 scanning result

Using scanner	Cross site scripting		SQL injection	
	TP	FP	TP	FP
RIPS	1	1	11	0
THAPS	2	0	0	0
E-THAPS	2	0	11	0

E-THAPS, again showed the best performance with the highest TP and lowest FP rates in both XSS and SQL injection vulnerabilities. It is evident that E-THAPS can detect SQL injection flaws much more efficient than the others.

To validate deeply the FP rate of our E-THAPS engine, we perform the last experiment with the benign test set of 262 PHP collected from 4 featured WordPress plugins (as mentioned above). As figures shown in Table 8, THAPS and E-THAPS result 0 flaw in all vulnerability types, while RIPS shows considerable number in every vulnerability types and all of them are False Positive cases. It is obvious to see that the FP rate of THAPS and E-THAPS in this test is 0%, RIPS got 100%.

Table 8. Scanning results with WordPress featured plugins

Scanner	File disclosure			Cross site scripting			File manipulation		
	Found	Confirmed	FP	Found	Confirmed	FP	Found	Confirmed	FP
RIPS	5	0	5	23	0	23	32	0	32
THAPS	0	0	0	0	0	0	0	0	0
E-THAPS	0	0	0	0	0	0	0	0	0

5 Conclusion and Future Work

In this paper, we presented an extensible and hybrid platform, **GuruWS**, which allows us to automatically detect both vulnerabilities and malicious web shells. **GuruWS** is composed of two core modules named **grMalwrScanner** and **grVulnScanner**.

For detecting malicious files in **grMalwrScanner**, our work is based on taint analysis and pattern matching techniques. By using the taint analysis, we first identify the PDFs, then all significant arguments of these functions will be traced back to their 'source'. We proposed also some principles in order to hunt the web shells during the taint analysis. For the pattern matching technique, we used the powerful Yara rules, added our new rules to detect both web shells and adware. For validating our method, we collected a wide range of 2123 web shells (1507 + 616) and used VirusTotal to evaluate **grMalwrScanner** with 58 others popular scanners. Our evaluation experiment shows that **grVulnScanner** is able to achieve an outstanding precision (85.55%) and F1-score (0.92), much higher than other similar scanners.

Regarding the detection of web application vulnerabilities, we proposed a number of significant improvements in E-THAPS to a state of the art in modern web static vulnerabilities analysis. It applies advanced detection improvements to original THAPS: accurate SQLi flaws, XSS sinks, securing functions in conditional statements, revised all vulnerable functions, and declared a set of functions that create insecure strings. E-THAPS showed the best performance with the highest found and confirmed result for security flaws, XSS and SQLi vulnerabilities.

Furthermore, we consider developing **grVulnScanner** to become a gray-box scanner and able to detect possible flow control. For the malicious web application file detection, improved and optimize web shell pattern sets, strong enough rules for detecting stealthy web shells will be considered to be implemented in **grMalwrScanner**. The combination of these two solutions outstandingly eases web penetration procedure, and the whole module is an open source. We believe that in the future, with our adopted social ruleset and web shell statistic analysis modules, **GuruWS** will be a step forward for both the academia and the industry.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve this paper.

This work is partially supported by the national research project No. KC.01/16-20, granted by the Ministry of Science and Technology of Vietnam (MOST).

References

1. Internet Live Stats. http://www.internetlivestats.com/. Accessed 21 May 2017
2. Le, V.-G., Nguyen, H.-T., Lu, D.-N., Nguyen, N.-H.: A solution for automatically malicious web shell and web application vulnerability detection. In: Nguyen, N.-T., Manolopoulos, Y., Iliadis, L., Trawiński, B. (eds.) ICCCI 2016. LNCS (LNAI), vol. 9875, pp. 367–378. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45243-2_34
3. Mazumder, M., Braje, T.: Safe client/server web development with Haskell. In: 2016 IEEE Cybersecurity Development (SecDev), p. 150 (2016)
4. Bherde, G.P., Pund, M.A.: Recent attack prevention techniques in web service applications. In: International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), pp. 1174–1180 (2016)
5. Khari, M., Sangwan, P., Vaishali: Web-application attacks: a survey. In: 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, pp. 2187–2191 (2016)
6. Kals, S., Kirda, E., Kruegel, C., Jovanovich, N.: SecuBat: a web vulnerability scanner. In: 15th International Conference on World Wide Web, pp. 247–256 (2006)

7. Jensen, T., Pedersen, H., Olesen, M.C., Hansen, R.R.: THAPS: automated vulnerability scanning of PHP applications. In: Jøsang, A., Carlsson, B. (eds.) NordSec 2012. LNCS, vol. 7617, pp. 31–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34210-3_3
8. Dahse, J.: RIPS - a static source code analyser for vulnerabilities in PHP scripts. In: Seminar Work at Chair for Network and Data Security (2010)
9. Sasi, R.: Web backdoors - attack, evasion and detection. In: C0C0N Sec Conference (2011)
10. Petukhov, A., Dmitry, K.: Detecting security vulnerabilities in Web applications using dynamic analysis with penetration testing. In: OWASP Application Security Conference. Computing Systems Lab, Department of Computer Science, Moscow State University (2008)
11. Dahse, J., Holz, T.: Static detection of second-order vulnerabilities in web applications. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 989–1003 (2014)
12. Starov, O., Dahse, J., Ahmad, S., Holz, T., Nikiforakis, N.: No honor among thieves: a large-scale analysis of malicious web shells. In: 25th International Conference on World Wide Web, pp. 1021–1032 (2016)
13. Le, H.H., Nguyen, N.H., Nguyen, T.T.: Exploiting GPU for large scale fingerprint identification. In: Nguyen, N.T., Trawiński, B., Fujita, H., Hong, T.-P. (eds.) ACIIDS 2016. LNCS (LNAI), vol. 9621, pp. 688–697. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49381-6_66
14. Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., Yang, Z.: Dependence guided symbolic execution. IEEE Trans. Softw. Eng. **43**(3), 252–271 (2017)
15. Bhme, M., Paul, S.: A probabilistic analysis of the efficiency of automated software testing. IEEE Trans. Softw. Eng. **42**(4), 345–360 (2016)
16. Web Technology Surveys. http://w3techs.com/technologies/overview/programming_language/all/. Accessed 21 May 2017
17. YARA - The pattern matching swiss knife for malware researchers. http://virustotal.github.io/yara/. Accessed 10 May 2017
18. Popov, N.: PHP-parser introduction. https://github.com/nikic/PHP-Parser/blob/master/doc/0_Introduction.markdown. Accessed 15 Apr 2016
19. The Open Web Application Security Project. Static Code Analysis. https://www.owasp.org/index.php/Static_Code_Analysis. Accessed 22 May 2017
20. The Open Web Application Security Project. Attack Category: Command Injection. https://www.owasp.org/index.php/Command_Injection. Accessed 18 May 2017
21. The Open Web Application Security Project. Attack Category: PHP Object Injection. https://www.owasp.org/index.php/PHP_Object_Injection. Accessed 18 May 2017
22. The Open Web Application Security Project. Testing for Local File Inclusion. https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion. Accessed 18 May 2017
23. The Open Web Application Security Project. Attack Category: Direct Dynamic Code Evaluation (‘Eval Injection’). https://www.owasp.org/index.php/Direct_Dynamic_Code_Evaluation_('Eval_Injection'). Accessed 18 May 2017
24. Bernardo Damele, A.G., Stampar, M.: SQLMap - automatic SQL injection and database takeover tool. http://www.sqlmap.org/. Accessed 12 May 2017
25. Deng, W., Liu, Q., Cheng, H., Qin, Z.: A malware detection framework based on Kolmogorov complexity. J. Comput. Inf. Syst. **7**, 2687–2694 (2011)