THAPS: Automated Vulnerability Scanning of PHP Applications

Torben Jensen, Heine Pedersen, Mads Chr. Olesen, and René Rydhof Hansen

Department of Computer Science, Aalborg University, Denmark
{tjens10,hpede06}@student.aau.dk {mchro,rrh}@cs.aau.dk

Abstract. In this paper we describe the THAPS vulnerability scanner for PHP web applications. THAPS is based on symbolic execution of PHP with specialised support for scanning extensions and plug-ins of larger application frameworks. We further show how THAPS can integrate the results of dynamic analyses, generated by a customised web crawler, into the static analysis. This enables analysis of often used advanced dynamic features such as dynamic code load and reflection. To the best of our knowledge, THAPS is the first tool to apply this approach and the first tool with specific support for analysis of plug-ins.
In order to verify our approach, we have scanned 375 WordPress plug-ins and a commercial (monolithic) web application, resulting in 68 and 28 confirmed vulnerabilities respectively.

1 Introduction

In this paper we describe the THAPS vulnerability scanner for web applications written in PHP[1], a general-purpose scripting language specifically designed with web development in mind. The language is robust, quite "forgiving", and is available on a wide range of platforms, which makes it easy to get started with and well-suited for rapid prototyping and agile development. It is the language of choice for some of the most successful and well-known web-sites, with Facebook as the most obvious example, as well as for widely used web application frameworks such as Moodle, TYPO3, and WordPress.

Developing *secure* web applications is already notoriously difficult, as witnessed by the CWE top 25 list of vulnerabilities [3]. However, many of the features that make PHP a popular choice for web development, especially among novice programmers, also make it even harder to develop secure applications, e.g. the lack of variable declarations and strong typing. This is further exacerbated by the typical execution model of PHP, where scripts are essentially executed in the context of a web server, making even trivial programming bugs into potentially serious security vulnerabilities.

In the following we describe and discuss a newly developed vulnerability scanner, called THAPS, that automatically scans PHP source code for potential security vulnerabilities. The tool can be applied throughout the development of

[1] See http://php.net

an application and allows a developer to fix such bugs before the application is widely deployed.

THAPS is based on symbolic execution of PHP code. This enables the scanner to produce both precise and comprehensive vulnerability reports while retaining a relatively low rate of false positives. We believe this is an important trait in order to make the tool useful for a wide range of developers with particular emphasis on novice PHP programmers.

As mentioned above, a number of very popular web application frameworks are implemented in PHP, e.g., Moodle, TYPO3, and WordPress. Advanced frameworks like these are often designed around a (relatively) simple core whose functionality can be extended with plug-ins and/or extensions. Such an architecture can be quite problematic for a vulnerability scanner to handle since the core in such a framework typically makes heavy use of dynamic code loading, introspection, and reflection; all language features that are notoriosly hard to analyse statically. Furthermore, the individual extensions and plug-ins are often also hard to analyse statically, since they are likely to depend on functionality provided by the core.

To overcome these problems, THAPS implements specialised support for scanning extensions and plug-ins of larger application frameworks, through so-called *framework models*, as well as integration with dynamic analysis and a web crawler in order to facilitate analysis of advanced dynamic features of PHP, such as dynamic code load and reflection. To the best of our knowledge, THAPS is the first tool to implement this approach and to provide specific support for analysing plug-ins.

The remainder of the paper is structured as follows: In Section 2 we first give an overview of the general problem of writing secure PHP web applications and scanning for vulnerabilities in such applications. Section 3 discusses the fundamental symbolic execution engine of our THAPS tool as well as the specialised support for analysing plug-ins (Section 3.2). Integration with dynamic analysis and a customised web crawler is discussed in Section 4 while our tool is applied to real-world code and evaluated in Section 5. Sections 6 and 7 conclude with related work, conclusions, and future work.

2 Scanning for Security Vulnerabilities in PHP Code

In this section we briefly describe two of the most common security vulnerabilities found in web applications, SQL injection and cross site scripting (XSS), and further discuss some of the typical problems and challenges inherent in scanning PHP code, e.g., the use of dynamic code inclusion, and plug-in based application frameworks. In Section 3 we discuss how THAPS can be used to find such vulnerabilities.

We have chosen to focus on SQL injection and XSS vulnerabilities in this paper for two reasons: they are very dangerous and commonly found in web applications. That they are dangerous is evidenced by the fact that they are ranked number one and four respectively on the CWE/SANS “Top 25 Most

Dangerous Software Errors" list [3]. Here "dangerous" is taken to mean that the error will "frequently allow attackers to completely take over the software, steal data, or prevent the software from working at all". Since 25.7% vulnerabilities in the Common Vulnerabilities and Exposures[2] (CVE) database were classified as either SQL injections or XSS (as of April 24th 2012), it is clear that these vulnerabilities are commonly found in web applications.

Briefly, an SQL injection makes it possible to rewrite a database query made from a web application to a database, which could give rise to unexpected behavior and result in either data loss or bypassing security restrictions, e.g., by leaking user names and concomitant passwords. In comparison, an XSS vulnerability allows attackers to inject malicious code into the client part of the application and thereby change the behaviour of the client-side, potentially leaking authorisation and/or private information.

As mentioned in the introduction, THAPS is a symbolic execution engine that finds (potential) bugs essentially by simulating the execution of an application along all possible code paths and tracking whether any data from an untrusted *source*, i.e., any input to the application (typically controlled by the user or environment), can flow into a *sink*, i.e., code that uses said data, e.g., for output or as arguments in API calls. This approach is inspired by two well-known techniques: *taint tracking*, based on run-time monitoring, and *taint analysis*, based on static analysis, that have both been applied successfully to protect web applications against attacks such as SQL injection and XSS.

2.1 Dynamic Code Inclusion and Reflection in PHP

The main challenge in developing a PHP scanner is that the PHP language contains features that are well-known to be difficult to analyse statically, e.g., dynamic code inclusion and introspection/reflection. Below we illustrate the issues with PHP code examples. The first example shows a common idiom for including another file, at run-time, into the currently executing file:

```
1  include "pages/". $_GET["page"] .".php";
```

The intention with the above code is to include a page located in the `pages/` subfolder, which is determined by the query string sent to the server which, in this case, is stored in the `$_GET["page"]` variable. This construction trivially leads to a vulnerable web application, since an attacker could easily include files other than the intended by (manually) manipulating the `page` part of the query. Including other files (both at run-time and at compile time) is a commonly used strategy for making the development of PHP applications more modular.

The following example illustrates the use of reflection to let user input directly determine what function is called:

```
1  $func = "ajax_" . $_GET["action"];
2  $func("argument"); // or call_user_func($func, "argument");
```

[2] `http://cve.mitre.org`

A similar technique is used in WordPress to implement an event system which WordPress plug-ins (used to extend the WordPress core functionality) can use to register call-back function and to handle various AJAX calls. The problem with these call-back function is that static analysis cannot determine which function is actually called and hence it cannot perform a safe and precise analysis.

The third and final example, shown below, is an implementation of the register_globals feature of PHP which is used to initialise the environment with variables containing user input:

```
1 foreach (array_merge($_POST,$_GET) as $key => $value) {
2   $$key = $value;
3 }
```

This feature has been deactivated by default in recent versions of PHP. However, an old version[3] of Moodle implements functionality that can emulate the register_globals feature. This functionality takes each POST and GET key/-value pair and makes a variable with the name of the key and assigns the corresponding value to it. An example would be a request with ?a=value&b=other added to the query string, which would assign "value" to the $a variable and "other" to the $b variable. If the register_globals feature is enabled, a sound analysis has to assume the value is tainted, i.e., potentially contains dangerous or unsanitised user input, if it has not been encountered before in the analysis. The difference between the register_globals feature and this Moodle functionality is that the latter can be placed anywhere in the file and from that point the analysis has to ignore all previous assumptions about the involved variables.

2.2 Application Framework Plug-Ins

In addition to the challenges presented by the dynamic language features of PHP, discussed in the previous section, there are also challenges specific to scanning large applications such as the generic application frameworks that are often used in web development. We will briefly discuss these in the following.

Large systems like WordPress, TYPO3, and Moodle allow third party developers and users to extend the functionality of the system through a plug-in architecture, often realised through deep and ubiquitous use of the (problematic) dynamic language features discussed in the previous section. Since core application functionality is often provided only through plug-ins, potentially developed by external (untrusted) third parties, it is important that a vulnerability scanner can reliably scan external extensions and plug-ins.

In previous work [5] we tested a number of existing vulnerability scanners to see if and how well they could be used to scan plug-ins and identify vulnerabilities. In summary, the results showed that the tested tools would either attempt to analyse not only the plug-in itself, but also the underlying core framework and then mostly fail due the the size of the combined system; or perform an analysis

[3] Moodle v. 1.6.2 in lib/setup.php

of only the plug-in but generate imprecise results, both missing real vulnerabilities (false negatives) and reporting non-vulnerabilities (false positives), since key functionality and assumptions/guarantees provided by the underlying core was missing in the scanner.

As an example the WordPress core code base is very large: In version 3.2.1, with no plug-ins activated, 13.918 built-in and six user defined functions[4] are called just for rendering the front page. Even with a dynamic analysis the results are likely to be overly imprecise. With the assumption that the core system is safe, the developers of plug-ins and extensions only have to check their own work. This, however, can be a difficult task as the extensions might use functionality provided by the platform and hence the analysis needs to know about this in order to identify vulnerabilities caused by the platform.

3 Symbolic Execution and Vulnerability Scanning

In the following we give an overview of the THAPS vulnerability scanner and discuss how it can be used to scan for vulnerabilities as well as some of the design and implementation issues encountered.

3.1 Symbolic Execution

THAPS is fundamentally a taint analysis based on symbolic execution of PHP. It is in the symbolic execution engine that vulnerabilities are detected and where reports are generated, containing details of each (potential) vulnerability. In essence, symbolic execution simulates all the possible execution paths in the application. To identify vulnerabilities, the taint analysis identifies where user input is able to enter the application, called a *source*, and how it is propagated through the application. If the *tainted data* reaches critical points of the application where it is able to alter the outcome of the application it has reached a *sink*. Every time tainted data reaches a sink without being properly sanitised first, a vulnerability is reported. Sanitisation is a process in the application where the data is secured in such way that it cannot change the outcome unexpectedly.

The symbolic execution is performed by traversing the abstract syntax tree (AST) of the target application and simulating the effect of executing the code contained in a tree node (see below). The AST is generated by the PHP-Parser[5] library which also provides functionality to traverse and manipulate the generated tree structure.

To simulate all the possible outcomes the analysis might need to store several (many) values for the same variable because of assignments inside different code branches. Whenever there are multiple values, the simulation has to be performed on each of these. We store the values in a *variable storage*, which also records what branch of the code the value belongs to. Figure 1 shows how the storage structure would look like after an analysis of the following code:

[4] Functions that allow users to configure and customise their WordPress installation.
[5] `https://github.com/nikic/PHP-Parser`

```
1  if ($a==0) {
2    if ($b==1) { .. }
3    else { .. }
4  }
5  elseif ($b==0) { .. }
6  else { .. }
7  if ($c) {
8    $ui = $_POST["a"];
9  } else {
10   $ui = "test";
11 }
12 echo $ui;
```

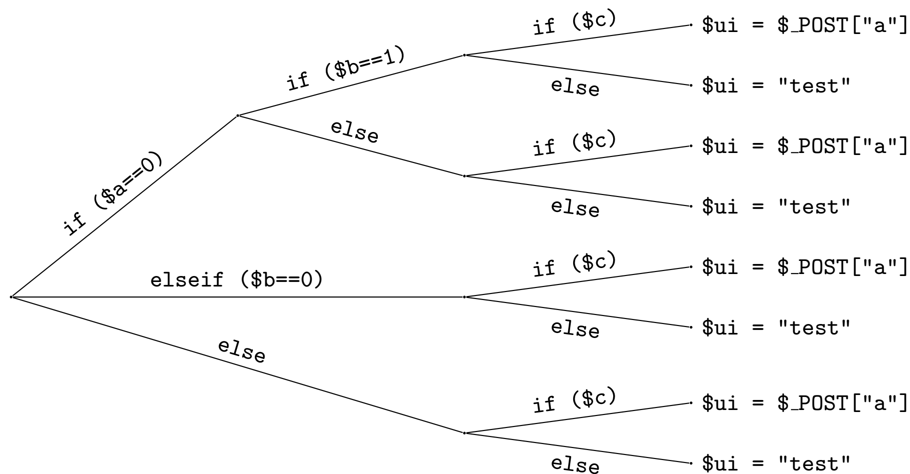

Fig. 1. Example of variable storage layout.

The analysis has been divided into the following steps:

Body analysis where the global scope and the bodies of methods and functions are analysed. The statements in the program are divided into three groups: control structures, assignments, and the rest.
The control structures include statements where the control branches based on a condition. Whenever such a statement is reached, the corresponding branches are recorded in the variable storage (see above) and the conditions leading to a given branch are added to a dependency stack and removed again once the branch has been analysed in this new, extended context.
Assignment statements update the knowledge about variables in the variable storage by traversing the right hand side of the assignment and adding the result to the variable storage.

Pattern Detection where certain idioms, or programming patterns, that must be handled in a special way are recognised and processed. These patterns are an easy and useful way to configure and adapt THAPS to the programming style of a particular application or project.

This has proven particularly useful for recognising application specific methods for sanitising input and thereby reducing the false positive count. As an example, consider the following code snippet that shows an example of avoiding SQL injection by escaping all GET and POST values in the beginning of the application:

```
foreach ($_GET as $k => $v) {
  $_GET[$k] = mysql_real_escape_string($v);
}
```

The above snippet could result in a (false positive) report if not handled specifically, because the number of user input variables is unknown and hence they would not be sanitised. When the pattern is recognised the analysis from this point on returns sanitised values if they are unknown in the variable storage.

The implementation is generalised so it is able to identify which variables that are sanitised and to which context the variables are sanitised, that is XSS or SQL injections.

Class processing where classes are analysed. The result of the analysis is a description of the class members and methods which is used when running the body analysis. The description tells if a method is potentially vulnerable. The analysis of the methods is a body analysis (see above) with initial knowledge of tainted data in class members and parameters of the methods, and the `return` node will also create vulnerabilities as returned values has to be tracked as well.

When the analysis of a method is finished any potential vulnerabilities are further analysed in order to determine the source of the vulnerability. Here the source means the origin of the user input, so if the flow path of a vulnerability originates from a parameter, that vulnerability is grouped as a "parameter vulnerable" (see below). A vulnerability can be added to multiple groups:

Always vulnerable: vulnerabilities that are valid as soon as the method is called, the source is within the method.

Parameter vulnerable: vulnerabilities that are present if the source is a parameter.

Property vulnerable: vulnerabilities that are present if the source is a property (of a class).

Return always/parameter/property: is where the method could return tainted data which is considered a return vulnerability allowing for determining if the data is tainted.

Property vulnerable parameter/property: not really vulnerabilities, but needed to describe if tainted data is transferred to an object's properties by the method.

- **Global vulnerable:** also not really vulnerabilities, but needed for the `global` keyword, simulating side effects of global variables.

Function processing where functions are analysed in the same way as methods are analysed in the class processing step (see above). It finds all functions (except built-in functions), makes an analysis of them, and stores a description of the functions similar to the one for methods from the class processing. The difference is that the descriptions only contain the *Always vulnerable*, *Parameter vulnerable*, *Return always/parameter*, and *Global vulnerable* information as functions do not relate to classes and hence do not have members to consider.

Inclusion resolving where the AST nodes representing file inclusion, i.e., an `include` statement, are replaced with the AST generated from the included file. This step will attempt to resolve which files are included at that exact point. It is however limited to simple inclusion, i.e., inclusion where the file is identified in the same line as the inclusion (see the code snippet below). File inclusion based on run-time values, called dynamic inclusion (see code snippet below for an example), is not supported in the symbolic execution engine, since it is very imprecise at best and impossible at worst (as it may even depend on code not present during analysis). Instead, information about dynamic file inclusion may be collected during the dynamic analysis phase of THAPS and then used in the symbolic execution to resolve (some) of the dynamic file inclusions. The dynamic analysis is described in more detail in Section 4.

```
1 // Supported
2 include "file.php";
3 include CONSTANT_DIR."file.php";
4
5 // Unsupported
6 include $file;
7 include $file.".php";
```

We do not go into further details with the symbolic execution engine here, instead we refer to the full report [2].

3.2 Framework Models

As discussed in Section 2.2, in order for a vulnerability scanner to work well on extensions and plug-ins for application frameworks, it is necessary for the scanner to have at least some information about the structure and workings of the underlying framework. This includes APIs, protocols for plug-ins, functionality provided by the framework etc.

As an example of the latter, functionality provided by the application framework, we consider the database wrapper provided by WordPress, which allows the developers to access the database without knowing about the credentials and authorisation. The wrapper is instantiated by the core of WordPress and

then used by extensions. To avoid analysing both the plug-in, the wrapper, and the core for each identified use of the wrapper functionality, we designed and implemented the notion of *framework models* in THAPS. A framework model can be seen as a template, or skeleton program, that superficially implements core APIs and functionality from the application framework. THAPS can then use these models as basis for analysing a plug-in in a more meaningful context. This not only increases the precision of the analysis, but is essential for analysis of plug-ins. Below we show a model of the database wrapper in WordPress:

```
1  class WPDB {
2    public function prepare($arg) { return $arg; }
3    public function get_results($arg) { mysql_query($arg); }
4    public function get_row($arg) { mysql_query($arg); }
5    public function get_col($arg) { mysql_query($arg); }
6    public function get_var($arg) { mysql_query($arg); }
7    public function escape($arg) { return
         mysql_real_escape_string($arg); }
8  }
9  $wpdb = new WPDB();
10 function esc_attr($a) { return htmlentities($a); }
11 foreach ($_POST as $key => $val) { $_POST[$key] =
       mysql_real_escape_string($val); }
12 foreach ($_GET as $key => $val) { $_GET[$key] =
       mysql_real_escape_string($val); }
13 foreach ($_REQUEST as $key => $val) { $_REQUEST[$key] =
       mysql_real_escape_string($val); }
```

When a plug-in is analysed the model is loaded and analysed before the actual analysis and the gathered information is used during the analysis of the plug-in.

Knowing about the platform provided functionality is, however, not enough to perform an analysis. Some platforms allow the extensions to have multiple entry points and hence the analysis needs to analyse all of these. One way the plug-ins are executed, besides being used as an entry point directly, is by registering its own (call-back) functions to either events/actions (used by WordPress) or hooks (used by TYPO3). This way a function will be called when this event is triggered by the core system. The function call `addAction("event","functionName")` from WordPress triggers the function `functionName` when `event` happens and hence the model analysis has to emulate these events. This has been accomplished by allowing the model developer to specify function names that attaches events and which argument that identifies the method.

It is, in general, hard to estimate the effort required to implement a model from scratch, e.g., for a new system or framework, since the development time depends on many factors such as the complexity of the system, how familiar the developer is with the system, and how precise the model has to be. However, in our experience a useful first version of a model can be developed within days if not hours. As an example, the first version of the above model for WordPress was developed in less than a day.Furthermore, models can be developed incre-

mentally, adding new API calls or events only as needed, e.g., when the false positive rate for a particular system becomes too high.

4 Dynamic Analysis and Web Crawling

To overcome some of the uncertainties of the static analysis that reduce the usefulness of the tool, especially dynamic file inclusion and the use of reflection, we have integrated a dynamic analysis phase in THAPS that can be configured to monitor specific dynamic elements. In this section we illustrate the kinds of problems dynamic analysis can help solve and discuss how the dynamic analysis works.

The first problem involves dynamic file inclusion. As discussed in a previous section, file inclusion determined by user input is a typical pattern found in PHP applications and, in particular, in application frameworks. Tutorials and forum posts regarding the pattern are easily found on the Internet[6][7][8]. An example of this pattern is shown below, which illustrates how an input string from the user is used to directly determine which file to include:

```
1 $page = $_GET["page"];
2 if (preg_match("#[a-z0-9]+#i", $page)) {
3    include "pages/" . $page . ".php";
4 }
```

The regular expression (line 2 above) ensures that input with special characters is disallowed, preventing an attacker from navigating the file tree. Since the user input is only available when the application is actually executing, the static analysis cannot possibly determine which file is included and thus cannot perform a sound analysis. In contrast, the dynamic analysis is able to determine (by straightforward run-time monitoring) the exact set of files included during *one particular* execution run of the target application.

A similar, but slightly different, problem is that some application architectures make it difficult to analyse only a part of the application, e.g. where user input is used to determine which files to include. This approach is shown below. Note that the user input is only used to determine the branch of the switch and not the file to include:

```
1 switch($_GET["page"]) {
2   case "products":
3     include "products.php";
4     break;
5   case "contact":
6     include "contact.php";
7     break;
```

[6] http://www.webdesign.org/web:programming/php/navigation-using-switch.11147.html
[7] http://www.digital-web.com/articles/easypeasy_php/
[8] http://www.codingforums.com/showthread.php?t=85482

```
8    case "about":
9      include "about.php";
10     break;
11  }
```

This has the disadvantage for a pure static analysis that the number of execution paths in the symbolic execution can get very large, and thus take very long time to scan. Dynamic analysis can reduce the number of included files, as, in the case above, only one of the files will be included for each scan.

A somewhat different problem, albeit with some of the same fundamental characteristics as the above file inclusions, is the use of reflection to perform function calls. Again, where the static analysis cannot determine which function is called and hence fails to do a correct analysis (or results in massive over-approximation), the dynamic analysis can trivially extract such information at run-time. This is illustrated by the code snippet below that static analysis would fail to analyse correctly:

```
1  $func = "ajax_".$_GET["action"];
2  $func();
```

Static analysis would be able to determine that the function call is to a function whose name starts with "ajax_", but if the application has more than one function with that prefix it cannot tell which one. Again the dynamic analysis is able to determine which function is actually called and collect this data for use in the static analysis.

4.1 Implementation of Dynamic Analysis

To gather the required data during execution we have written an extension for the Zend[9] Engine. The Zend Engine is the run-time system underlying and executing PHP applications: after PHP source code is first parsed, it is converted into opcodes, that are then interpreted by the Zend Engine. The engine allows extension developers to hook into the execution of opcodes and, e.g., run a specified function when a given opcode is encountered.

Our extension works by first establishing a connection to a running MongoDB[10] database and generating a unique id to identify the current analysis run. We then add opcode hooks on function calls and "include/eval opcodes" for each request and record function calls and which files are included. This information is stored in the MongoDB database, under the unique id of the current analysis run.

When the analysed web application returns a response, through the web server that controls it, our Zend extension ensures that the unique analysis id is included in the response to indicate where the run-time information is stored in the database. This information is used by the static tool, so it initially can fetch

[9] www.zend.com
[10] www.mongodb.org

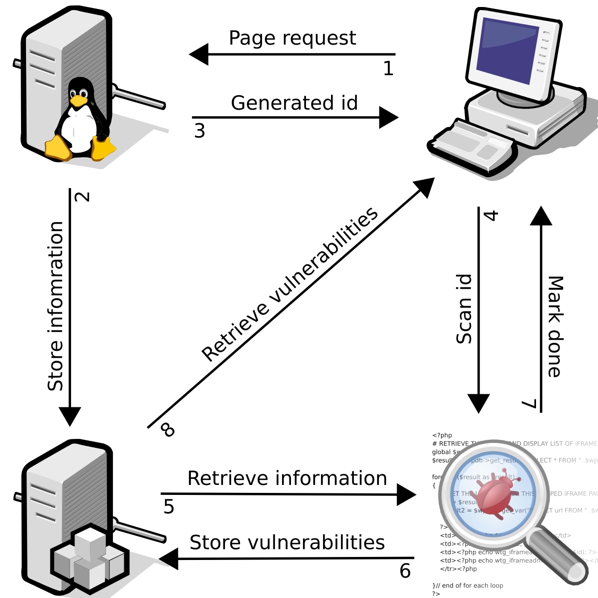

Fig. 2. Overview of the interaction between the dynamic analysis, and subsequent static analysis.

the required information to make a complete scan on the basis of information found by the PHP extension.

Once the dynamic analysis is done, the static analysis is performed with this additional information, as shown in Figure 2. The collected information is used to analyse a specific branch of file inclusions and dynamic function invocations. However, the static analysis will still consider all branches in `if` and `switch`es.

In the next section we discuss how a *web crawler* can be used to automate the process of generating requests to, and responses from, the web application under analysis in order to generate enough information for the dynamic analysis to be useful.

4.2 Crawling a Site

The task of performing a request to a page, collecting the analysis id from the header information, and running the static analysis with the collected id easily becomes a tedious task if the site has many pages. We have automated this process by developing a *web crawler*.

To perform a crawl only a few inputs are required: the URL of the site on a web server with our Zend extension installed, the path to the source code, and an optional cookie that provides authentication on the site. During the crawl any vulnerabilities identified, will be displayed immediately.

Instead of developing a crawler from scratch, we have chosen to extend Crawler4j[11] as it is an open source, multi-threaded, cross platform crawler that is simple to use and extend. Crawler4j consists of two parts: a controller to setup how the crawler should behave, and the actual crawler which should be extended to add custom functionality, such as what should happen when the crawler successfully have visited a page. Although Crawler4j is a comprehensive and working crawler, it was necessary to modify it in order to support and provide access to the special cookies and analysis id's generated by THAPS.

With the provided information the crawler analyses the applications in two ways: from the web application where it extracts all the links from the pages and follows them as long they are not to an external site, and by accessing the files found in the application's file system directory directly.

The first approach tests the application the way the developer intended the site to be used, and the second reveals vulnerabilities in unintended accessible files, files that are secure when included, but not when accessed directly, and other unintended uses of the application. Combining these two approaches gives a more complete crawl of the entire web application.

5 Results

In order to examine and verify the effectiveness of THAPS, we have tested it against a number of WordPress plug-ins (to test the Framework Models discussed in Section 3.2) and against a commercial web-application, developed by a newly established company that wishes to remain anonymous (to test the integration of the dynamic analysis and to test how the tool handles a large real-world monolithic application). In the following we shall refer to this application as "System S".

5.1 WordPress

WordPress was chosen because it is one of the most popular and widely used CMSs[12]. Furthermore, the WordPress web site hosts almost 20.000 different plug-ins, with some of the most popular plug-ins having been downloaded more than 10 million times[13]. Hence, a vulnerability found in any one of these can have widespread consequences.

Our test was conducted by analysing the 300 most popular (according to the ranking on the WordPress web-site) plug-ins along with 75 arbitrarily chosen plug-ins. In total THAPS reported 273 vulnerabilities in 29 plug-ins. Of these, 68 were confirmed by manually making exploits for at least one vulnerability for each file in each plug-in, and acknowledged by the WordPress security team[14].

---

[11] `http://code.google.com/p/crawler4j/`
[12] `http://w3techs.com/technologies/overview/content_management/all`
[13] `http://wordpress.org/extend/plugins/`
[14] See `http://packetstormsecurity.org/files/authors/9770` for a full list

Type	Found	Confirmed	Potential ①	FP	Unresolved ②
XSS	249	63	130	41	15
SQL injection	24	5	13	3	3
Total	**273**	**68**	**143**	**44**	**18**

① = The number of reported vulnerabilities that have not been examined.

② = Reported vulnerabilities that could be exploitable, but we were unable to confirm it with a reasonable exploit.

FP= False positives.

Table 1. Scan results for WordPress extensions.

The results are summarised in Table 1. As shown, THAPS identified both XSS and SQL injection vulnerabilities, with a modest rate of false positives. The model approach allowed us to analyse the plug-ins separately, without having to re-analyse the entire core WordPress codebase — something which no tool we know of can currently do.

Although the theoretical time-complexity of symbolic execution as implemented by THAPS is, at least, exponential in the size of the program, most of the plug-ins were analysed in a few minutes with only a few taking up to 20 minutes.

5.2 "System S"

For testing the dynamic analysis parts of THAPS, "System S" was chosen because it is a large, monolithic, real-world web application that has few entry points. Due to the size and structure of "System S", THAPS is unable to analyse it purely by symbolic execution. Instead dynamic analysis is performed (as described in Section 4) and the information gathered by the dynamic analysis is then used to guide the static analysis and avoid unused code paths.

The results of analysing "System S" are summarised in Table 2. As shown in the table, THAPS found 32 vulnerabilities in total, of which we confirmed 28 by generating exploits manually as well as confirming the vulnerabilities with the developers of "System S". Four of the reported vulnerabilities were unexploitable and thus marked as false positives. It follows from the results that the biggest problem in "System S" is to prevent attackers from performing SQL injection on the site. The system contains a lot of users but sign up is not open for public. Each user is a member of a group and some of the confirmed exploits only work when the user is a member of a certain group. Even though a valid user login is required to exploit some of the vulnerabilities, it was still possible to perform SQL injection on some of the publicly available pages.

6 Related Work

In this section we briefly survey related work, focusing on tools that are either static scanners or dynamic monitoring. In our previous work [5], we have per-

Type	**Found**	**Confirmed**	**False Positives**
XSS	7	7	0
SQL injection	25	21	4
Total	**32**	**28**	**4**

Table 2. Scan results for "System S".

formed a more comprehensive study and comparison of such tools. The results of this study have been used to guide the design and feature set of THAPS.

WebSSARI (now commercialised as CodeSecure) is able to perform a static analysis based on a tainting analysis [1]. The approach analyses a control flow graph generated from an abstract syntax tree and determines whether any path contains a source and a vulnerable sink without a proper sanitisation between. WebSSARI combines traditional secure information flow analysis with run-time monitoring into a sound analysis of web applications.

Xie and Aiken [6] presents an algorithm to improve the handling of dynamic features of PHP as well as large application during a static analysis. The algorithm uses symbolic execution on function bodies and keep descriptions of these parts for use when the functions are called. This technique is complementary to our approach and would likely enable THAPS to analyse bigger systems.

PHP Taint[15] is an implementation of dynamic taint analysis. It alters the Zend scripting engine to add additional information about all variables, and check this information when vulnerable functions are called. The current implementation lacks support for several language features and custom sanitisation like regular expressions, however, it allows for high precision detection of vulnerabilities.

RIPS[16] is one of the newest tools available. It analyses the code by performing taint analysis on a list of tokens generated by the built-in PHP tokeniser. RIPS is fast, but lacks precision in the analysis resulting in false positives, and support for object oriented code which limits the amount of applications it can analyse.

Fuzzing has proven to be an efficient technique to identify vulnerabilities in applications [4]. Wapiti[17] is a fuzzer targeted web applications combined with a database of known vulnerabilities. By combining these two methods Wapiti is able to find new vulnerabilities and ensure old ones have not been reproduced.

7 Conclusion

We have presented the THAPS vulnerability scanner for web applications implemented in PHP. We have shown how symbolic execution of PHP can be extended with two novel approaches: *framework models* and integration with

[15] https://wiki.php.net/rfc/taint
[16] http://sourceforge.net/projects/rips-scanner/
[17] http://www.ict-romulus.eu/web/wapiti/home

dynamic analysis. The former allows tools to scan individual plug-ins or extensions, separately from the (large) underlying core framework, and the latter enabling analysis of advanced dynamic features, such as dynamic code load and reflection. To the best of our knowledge, THAPS is the first tool to implement and combine these techniques in one tool.

The general approach of THAPS was validated by analysing 375 WordPress plug-ins and a (monolithic) commercial web application, resulting in 96 confirmed vulnerabilities in total.

7.1 Future Work

The two main strands of future work currently pursued are: improving the memory management of the symbolic execution and automatically generating exploits: The current variable storage, used in the symbolic execution, cannot handle the state space explosion resulting from large applications. One way to solve, or mitigate, this problem would be to merge similar branches instead of merely pruning identical branches.

As a part of verifying our results we created exploits for one of the vulnerabilities per file. These exploits were created almost exclusively from information contained in the THAPS vulnerability reports. The dependencies and flow path were enough in many cases to blindly create the exploits. We would like to investigate the possibility of automatically generating an initial exploit based on this information.

References

1. Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 40–52. ACM, 2004.
2. Torben Jensen and Heine Pedersen. THAPS—Analysis of PHP web applications. Master's thesis, Department of Computer Science, Aalborg University, Denmark, 2012. Forthcoming.
3. Bob Martin, Mason Browne, Alan Paller, and Dennis Kirby. 2011 CWE/SANS top 25 most dangerous software errors. Available at `http://cwe.mitre.org/top25/index.html`, September 2011. Last accessed June 10, 2012.
4. B. P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of unix utilities. *Comm. of the ACM*, 33(12):32, December 1990.
5. Heine Pedersen and Torben Jensen. A study of web application vulnerabilities and vulnerability detection tools. Project report (sw9), Department of Computer Science, Aalborg University, 2011.
6. Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*. USENIX, August 2006.