Aalborg University

Department of Computer Science

Title:

THAPS: Detection of Web Application Vulnerabilities

Topic:

Security in Web Applications

Project Period:

February 1st - June 7th, 2012

Project Group: sw101f12

Torben Jensen

Heine Pedersen

Supervisors:

René Rydhof Hansen

Mads Christian Olesen

Number of appendices: 3

Total number of pages: 41

Number of pages in report: 30

Number of reports printed: 5

Abstract:

This report presents THAPS, a vulnerability scanning tool for PHP web applications. The tool explores two new ways of analyzing web applications by extending the traditional static analysis with a model analysis, and by combining the static analysis with a dynamic analysis.

The extended static analysis allows the tool to analyze the extensions of modular systems, such as WordPress and TYPO3, without having to analyze the core system.

The combined approach allows for analyzing custom built application with few entry points. The problem with these types of applications is that they cannot be modeled and analyzed in bits, and they are too large to analyze in a single analysis. Using the combination approach the tool can split the code to analyze in bits and still give results. This also allows for analyzing newly added features to these systems as well.

The result of the project is 30 new confirmed vulnerabilities, 29 in WordPress modules and one in a core TYPO3 extension. Additionally it has been used to identify 33 vulnerabilities in a newly established company's web application.

Summary

This project presents the newly developed THAPS, a vulnerability scanning tool for PHP web applications. The tool explores two new ways of analyzing web applications by extending the traditional static analysis with a model analysis, and by combining the static analysis with a dynamic analysis.

The extended static analysis allows the tool to analyze the extensions of modular systems, such as WordPress and TYPO3, without having to analyze the core system. The benefit of this is a more precise analysis because any insecurities of a real analysis of the core system is eliminated.

The static analysis uses symbolic execution on which a taint analysis is performed. Symbolic execution simulates all possible outcomes of an application, and the taint analysis tracks user input through these executions, as malicious user input might change the behavior of the application in an unwanted manner. To prevent this from happening the user input has to be validated or changed before it is used it will not do any harm.

The combined approach allows for analyzing custom built application with few entry points. The problem with these types of applications is that they cannot be modeled and analyzed in bits, and they are too large to analyze in a single analysis. Using the combination approach the tool can split the code to analyze in bits and still give results. This also allows for analyzing newly added features to these systems as well.

The result of the project is that THAPS identified 30 new confirmed vulnerabilities in publicly available systems. The vulnerabilities were distributed as 29 in WordPress modules and one in a core TYPO3 extension. Additionally THAPS has been used to identify 33 vulnerabilities in a newly established company's web application.

Preface

This report substantiates the result of Software Engineering group sw101f12's 10^{th} semester project at the Department of Computer Science, Aalborg University. The report is documentation for the master's thesis made in the period from 1^{st} of February until 7^{th} of June 2012.

The topic of this semester project is "Security in Web Applications" and the goal is to develop a new tool based on experience gained in the previous semester.

The reader is expected to have understanding of programming corresponding to a student that has completed the 9^{th} semester of Software Engineering.

Unless otherwise noted, this report uses the following conventions:

- Cites and references to sources will be denoted by square brackets containing a number where each reference corresponds to an entry in the bibliography on page 33.
- Abbreviations will be represented in their extended form the first time they appear.
- When a person is mentioned as *he* in the report, it refers to *he/she*.

Throughout the report, the following typographical conventions will be used:

- References to classes, variables and functions in code listings are made in `monospace` font.
- Omitted unrelated code is shown as "..." in the code examples.
- Lines broken down in two are denoted by a ↵.

The code examples in the report are not expected to compile out of context.

Appendices are located at the end of the report. The source code for the software project is included on the attached CD-ROM on page 41.

Contents

1 Introduction

In 2011 web vulnerabilities was responsible for security breaches at many high-profile organizations such as Sony Pictures, PBS.com, and mysql.com [8, 9, 40]. Given that the web applications can be reached from anywhere security plays an important role, because if an attacker is able to exploit a vulnerability he might gain access to confidential data. Hence it is critical to web applications that they are secure so that data is only accessible by the intended people.

According the Common Weakness Enumeration (CWE) [3] SQL injection is the most dangerous software error. A SQL injection makes it possible to rewrite a database query, which could give unexpected behavior and result in either data loss or bypassing security restrictions. Another vulnerability is Cross Site Scripting (XSS) which is the fourth most dangerous but the most prevalent vulnerability in web applications. XSS is a vulnerability that allows attackers to inject malicious code that can change the behavior of the application to the users. Possible attacks would be to steal users' identities or redirect to another page. More details about the vulnerabilities can be found in our study [1].

SQL injection and XSS are common vulnerabilities in web applications. As of April 24th 2012, 25.7% of the vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database [2] were classified as either SQL injections or XSS.

One of the reasons that vulnerabilities arise in web applications is that web programming languages does not provide automatic protection and leaves this responsibility to the developers. Some languages and frameworks provide optional protection but it requires that it is used correctly. The focus for this project is on finding vulnerabilities in PHP, one of the most used languages [7], as PHP does not provide this functionality. Also many large systems such as Facebook, WordPress, TYPO3, and Moodle, use PHP, which makes it a natural choice.

To help the developers avoid vulnerabilities in their applications tools have been made that can identify them. The identification process is, however, extensive, which makes it difficult to create a tool with perfect results. The next sections clarifies why this is the case.

1.1 Existing Tools

We have earlier tested 13 existing tools [1] that perform either static analysis or dynamic analysis. A static analysis is one where the application is analyzed without being executed and a dynamic analysis is one where the application is executed and tried exploited. The tools did not perform as expected as many of the known vulnerabilities were not found by the tools.

The expectations to the tools were with regard to the soundness and completeness of them. A sound tool is one that report actual vulnerabilities and not problems that are not problems, so-called false positives. The completeness describes if the tool find all vulnerabilities in the application or not. If the tool does not find a vulnerability it is called a false negative.

The tools were tested on four different systems, where two of them are widely used modular systems, namely WordPress and Moodle. Another system was custom-made by a newly established company where no CMS was used. In this report this system will be referred to as DoubtfulSystem. The last system was a test script written by ourselves which contained a number of vulnerabilities. All these systems are further described in Chapter 3.

Of the tested tools we found that the static analysis tool RIPS was one of the best tools available. It found several of the known vulnerabilities, but it had its limitations on bigger systems such as DoubtfulSystem and WordPress. Some of the limitations are due to missing support for object oriented code, and the issues regarding PHP described in Section 1.2.

Another of the best tools was PHP Taint which provides a dynamic taint analysis. A taint analysis is when user input is tracked through the code statements, from when it enters the application, the sources, to the vulnerable parts of the application, the sensitive sinks. A vulnerability is reported when the user input reaches a sink without being properly sanitized, that means that malicious user input is removed. The problem with PHP Taint is that it will only give results for the executed code and not the other code, and it does not give results for alternative paths through the code. Using a crawler to visit all the pages of the site would not solve this problem as code might not be reached at any point, but can still be vulnerable.

1.2 Dynamic Code Load

PHP is a difficult language to analyze. One reason is that PHP uses plenty of dynamic code load: including files with code, calling functions, and accessing variables to name a few. This makes it difficult to perform a pure static analysis of the application if the user input is used as a part of the dynamic code load, as the analysis would have to guess values for the user input.

Some code examples with dynamic code load that could cause problems to a static analysis of PHP is demonstrated below.

The first example in Source Code 1.1 exemplifies how other files can be included into the current file at runtime. The intention with the code is to include a page located in the *pages* subfolder, which is determined by the query string sent to the server, and in this case the `$_GET["page"]` variable is used in a query string. This construction could easily lead to a vulnerable web application, as an attacker could easily include other files than intended. It is very common to include other files when developing PHP applications, and in this example, there is no way to determine which file to include when doing static analysis.

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

Source Code 1.1: File inclusion based on user input.

The second example in Source Code 1.2 shows how user input can be used to make function calls. A similar technique is used in WordPress to implement an event system which the modules can use to register function calls, and to handle various AJAX calls. The problem with these function calls is that the static analysis cannot determine which function is actually called and hence it cannot perform a safe and precise analysis because this will add imprecision to the knowledge of variables.

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

Source Code 1.2: Dynamic function call which name begins with ajax_.

The third and final example in Source Code 1.3 is an implementation of the `register_globals` feature of PHP which initializes the environment with variables containing user input. This feature has been deactivated by default in the newer versions, however, an older version of Moodle has an implementation that emulates the `register_globals` feature, a construction much like the example. The code takes each POST and GET key/value pair and makes a variable with the name of the key and assigns the 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 the analysis has to assume the value is tainted if it has not been encountered before in the analysis. The difference from the `register_globals` feature and this code is that this code can be placed anywhere in the file and from that point the analysis has to ignore all previous knowledge it may have had.

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

Source Code 1.3: Emulate PHP's `register_globals` functionality by creating new reflected variables.

The amount of dynamic code load used even in small PHP applications makes it difficult to reason about bigger systems like WordPress, as errors from the unknown user input might accumulate and give erroneous output. In addition the application might have only one or few entry points, the intended points to start execution of the application by the developer, resulting in an analysis of the whole application. This could possibly take long time depending on the application, and if the developer has only changed or added a single functionality checking for vulnerabilities could be tiresome. Additionally the new functionality could be in the form of an extension/plugin and the developer of such would not be interested in the security of the core system as this should have been tested separately.

1.3 The Problem

We learned in our study [1] that no single analysis approach for detecting vulnerabilities in PHP applications performs perfectly when analyzing different types of applications. Additionally we believe that if a tool return many false positives the real problems disappear in the crowd and the developer becomes annoyed of validating them.

The problem with false positives especially occurred in the larger applications we tested [1], see Table 3.1 on page 22 for the results. Our work has been on detecting vulnerabilities in these systems with a limited amount of false positives. In this project we combine three approaches in order to make a more precise detection of vulnerabilities.

We use symbolic execution as our static analysis approach on which we perform a taint analysis which can receive additional information regarding the target in order to improve the analysis. The static analysis is also able to analyze applications that use a restricted subset of dynamic code loading without receiving additional information.

A dynamic analysis provides information at runtime about file inclusions and function calls. This helps the static analysis to determine the output of dynamic code load based on user input. Additionally this information is used to limit analysis of humongous applications with few entry points when an analysis is performed of newly added functionality.

To cope with extendable systems like WordPress without having to analyze the core it is possible to use what we call a model analysis. A model analysis is specific for the system in question where the vulnerable functionality of the system is identified and used to analyze extensions. Examples of such functionality include: database layer objects, custom sanitization functions, and how to trigger the potential vulnerable parts of the extension.

To summarize we want to:

- Reduce the number of false positives in larger applications.
- Improve the analysis of extensions of modular systems.
- Allow specific analysis of newly added features in applications with few entry points.

2 Approach

Our solution to the problems identified in Chapter 1 is to allow the analysis of an application to receive and use information from different sources. We have created a tool that performs a static analysis, a Zend extension that collects information at runtime, and a crawler that is able to automate the analysis of an application. A Zend extension is a plugin to the Zend Engine, the scripting engine of PHP written in C, which allow developers to add high performance functionality such as drivers to databases and other heavy computational operations, but it does also allow to change the behavior of the core.

The static analysis is able to do an analysis of applications on its own, but it is also able to include information gained from the extension or, as described later, from a model in order to improve the analysis. The extension performs a dynamic analysis which collects information during execution to reduce the amount of ambiguities during the static analysis. The crawler helps automate the analysis. An application can be analyzed in two ways: from the file system where each file is accessed directly through the web server and if it succeeds it is analyzed, and from the web site where the links of the site are identified and then analyzes the entry points with the gained dynamic information.

The data flow of THAPS when the crawler is used to analyze a web application is shown in Figure 2.1. It would be possible to extend the tool with additional information sources before performing the static analysis if new ideas arise. The intention was to make THAPS easy to extend so new functionality could append new data to the database. Therefore the document database MongoDB has been chosen, as it is easy to append arbitrary data to an entry.

The following sections will describe the different parts of the analysis in more detail.

2.1 Static Analysis

The advantage of doing a static analysis is that all aspects of an application can be throughly analyzed. In some cases vulnerabilities can arise when a specific piece of code is mistakenly accessible, because the developer failed to follow the flow path. Furthermore applications can be vulnerable if accessed in a way the developer did not intend. A static analysis can help discover these vulnerabilities.

The static analysis part of THAPS is where all vulnerabilities are detected and is where a report is generated that contains details of each vulnerability. The static analysis is a symbolic execution upon which a taint analysis is performed. Symbolic execution is able to reason

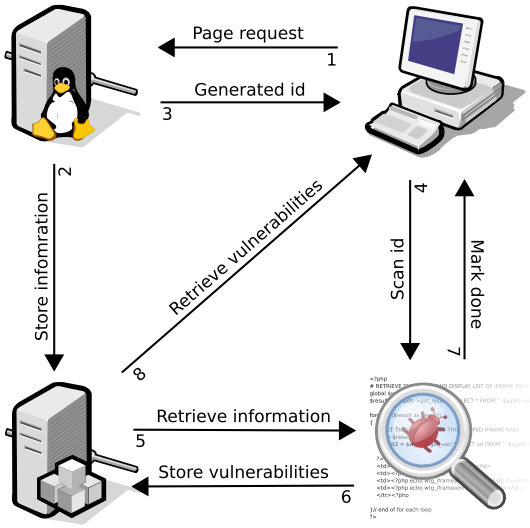

Figure 2.1: The data flow of THAPS. The crawler request a page from the web server, where the extension collects and saves data about the request in the database. It returns the page with an identifier to the data which the static analysis uses to fetch the information. The static analysis identifies vulnerabilities and saves a report of them in the database and when it is done, the crawler presents the report.

about/simulate all the possible outcomes of the application. To identify the vulnerabilities the taint analysis identifies where user input is able to enter the application, a source, and how it is propagated through the application. If this user input, also called tainted data, reaches critical points of the application where it is able to alter the outcome of the application it has reached a vulnerable sink, or just sink. Every time tainted data reaches a sink without being properly sanitized first, a vulnerability is reported. Sanitization is a process in the application where the data is secured in such way that it cannot change the outcome unexpectedly.

To simulate all the possible outcomes the analysis might need to store several values for the same variable because of assignments inside code branches. Whenever there is multiple values the simulation has to be performed on each of these we store the values in a *variable storage*, which also stores the branch of the code the value is stored in. Figure 2.2 shows how the storage structure would look like after an analysis of the code in Source Code 2.1.

To generate an abstract syntax tree (AST) an existing library is used, namely the PHP-Parser[1] library, which is written in PHP. It is possible to use the provided functionality to traverse the tree, manipulate it, and perform the analysis required for each node. A node of the tree represents a part of the source code and describes some functionality. A node might contain subnodes as well, i.e. an assignment node has a variable node and an expression node, where the variable node describes where to assign the value determined from the value of the expression node.

[1] https://github.com/nikic/PHP-Parser

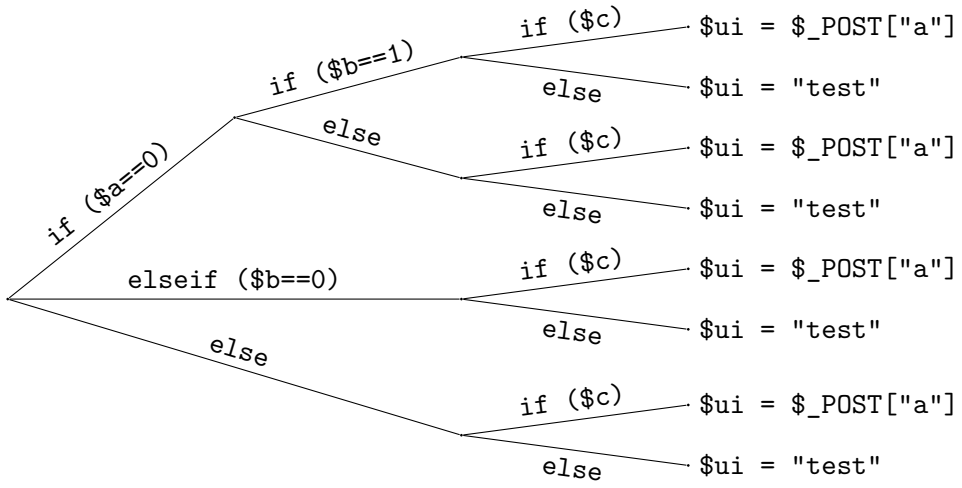

Figure 2.2: Variable storage layout after analysis of the code in Source Code 2.1.

```
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;
```

Source Code 2.1: Example code to demonstrate the analysis.

The analysis has been divided into the following steps:

Body analysis where the global scope and the body of methods and functions will be analyzed, and the present vulnerabilities are identified.

Class identification where all user defined classes are identified with their members and methods.

Function identification where all user defined functions are identified and analyzed.

Inclusion resolving where the inclusion nodes are replaced by the AST generated from the included file. This step tries to resolve which files is included at that exact point.

Each of the steps will be described in depth in the following sections.

2.1.1 Body Analysis

The body analysis traverses the nodes of the body and simulates the effect of them. The nodes have been divided into three groups: control structures, assignments, and the rest.

The control structures include the nodes where the code branches: the if-then-else, and switch-case nodes. Whenever such a node is reached the corresponding branches are created in the variable storage. Every time a new branch: `if`, `elseif`, `else`, or `case` is entered, the conditions of the branch are added to a dependency stack and removed again when left. The dependency stack describes the branches of the variable storage that is currently valid to simulate and to describe vulnerabilities better.

The assignment nodes update the knowledge about variables in the variable storage by traversing the right side of the assignment, namely the expression. This is exemplified by the code in Source Code 2.2. First it finds the possible branches by passing the dependencies, that identifies which `if-then-else` / `case` it currently is in. Afterwards it does an evaluation of the right side (`$node->expr`) of the assignment and saves the value on the current branch.

```
1  $taintVisitor = new BodyVisitor;
2  ...
3  // Identify the branches of the variable scope
4  $varValueConfigurations = ↵
       $this->vScope->getVariableValueConfigurations($this->dependencies);
5  foreach ($varValueConfigurations as $varValueConfiguration) {
6    ...
7    // Setup the traverser and traverse
8    $taintVisitor->setVScope($varValueConfiguration);
9    $taintTraverser->traverse(array($node->expr));
10   // save the new value in the configuration
11   $taintInfo = $taintVisitor->getTaint();
12   ...
13 }
```

Source Code 2.2: Analysis of an assignment node.

The rest of the nodes covers function calls, variable values, concatenations etc. These nodes are used to calculate the possible variable values and detect vulnerabilities. All vulnerabilities are due to some specific function calls and hence the logic of describing them is located in the handling of those. A vulnerability is described by the line where the vulnerable function call is performed and the type of the vulnerability, a flow path describing how the vulnerable data reaches the function call, and which conditions that are required to reach this vulnerability. To avoid multiple reports about the same vulnerable function call, the vulnerabilities are grouped so a report can have several flow path and dependency pairs.

2.1.1.1 Pattern Detection

To overcome a problem that we experienced during our study of the static tools we have added detection of a pattern that gave false positives in the reports.

To understand why this is required Source Code 2.3 shows an example of one of the ways people overcome SQL injection easily by escaping all GET and POST values in the beginning of the application.

```
1  foreach ($_GET as $k => $v) {
2    $_GET[$k] = mysql_real_escape_string($v);
3  }
```

Source Code 2.3: Sanitizing all GET input variables with regard to SQL injections.

In our implementation this code would rise a problem if not detected, because the number of user input variables is unknown and hence they would not be sanitized. When the pattern is recognized the analysis from this point on returns sanitized values if they are unknown in the variable storage.

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

This approach gives false positives if the developer assigns values to the arrays before reaching the `foreach` loop, but a solution to this is omitted because we find it a bad practice to perform changes to the user input array when using a "sanitizing everything at once" approach.

The pattern detection could also be implemented to handle the `register_globals` functionality. If the pattern is recognized the variable storage should be instructed to return a tainted value if the variable is unknown. However, the side effects of this has to be considered with regard to false positives.

2.1.2 Class Identification

This step analyses the user defined classes in the AST. 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 potential vulnerable.

The analysis of the methods is a body analysis 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. Source Code 2.4 shows the principle of the initialization of initial knowledge.

```
1  private function parseMethod($className, ClassMethodDescription ↵
       $method) {
2    ...
3    $vars = new VariableStorage;
4    // Initialize the pre-known information
5    foreach ($properties as $propNr => $property) {
6      $vars->setVariableValue(new VariableValue(true), "this", ↵
         $property->name);
```

```
7       $props[$propNr] = "this->" . $property->name;
8     }
9     ... // Do the same for parameters
```

Source Code 2.4: Initialization of the variable storage before a body analysis is performed on a method.

When the analysis of a method is finished each of the found vulnerabilities are further analyzed in order to determine the source of the vulnerability. The source means where the user input origins, so if the flow path of a vulnerability origins from a parameter, that vulnerability is grouped under that parameter. A vulnerability can be split into multiple groups. The groups are described below and Source Code 2.5 shows the code that determines and groups the vulnerabilities and Source Code 2.6 shows how the side effects are identified.

```
1   $methodVulnerabilities = $bodyTraverser->traverse($stmts);
2   if (count($methodVulnerabilities) > 0) {
3     foreach ($methodVulnerabilities as $nr => $vuln) {
4       $vulnParamName = vulnOriginsFromVar($vuln->flowpath); // Locate ↵
            the source
5       if (in_array($vulnParamName,$params)) { // parameter
6         if ($vuln->return) { ... } // Return vuln
7         else { ... } // in method vuln
8       } else if (in_array($vulnParamName,$props)) { // property
9         ... // same as above
10      } else { // always
11        ... // same as above
12      }
13    }
14  }
```

Source Code 2.5: Part of the implementation of how vulnerabilities are grouped.

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

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

Property vulnerable is vulnerabilities that is present if the source is a property of the 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 Similar to the previous group, but with regard to the `global` keyword, simulating side effects of global variables.

When a method is called all of these groups of vulnerabilities are considered because a method can be in several groups.

```
... // Find variable configurations after analysis
// Check if any properties are updated.
foreach ($properties as $propNr => $property) {
  foreach ($variableConfigurations as $variableConfiguration) {
    $taintInfo = ↵
        $variableConfiguration->getVariableValue("this",$property->name);
    if ($taintInfo !== null && $taintInfo instanceof VariableValue) {
      $vulnParamName = vulnOriginsFromVar($taintInfo->flowpath);
      if (($key = array_search($vulnParamName,$params)) !== false) {
        // updated with parameter data
      } else {
        // updated with user input directly
      }
    }
  }
}
... // same for global
```

Source Code 2.6: Part of the implementation of identification of side effects of methods.

2.1.3 Function Identification

This step does the same as the Class Identification step but with regard to functions instead of methods. It finds all user defined functions and makes an analysis of them and stores a description of the functions similar to the one of the methods from the class identification. The difference is that the descriptions does only contain the *Always vulnerable*, *Parameter vulnerable*, *Return always/parameter*, and *Global vulnerable* information as functions does not relate to classes and hence does not have class members to consider.

2.1.4 Inclusion Resolving

The step of inclusion resolving is replacing the include nodes of the AST with a new AST constructed from the source code of the target. This step, however, has been limited to straight forward inclusion, that is inclusions where the file is identified at the same line as the inclusion.

The reason that dynamic inclusion is not supported is that is it difficult and imprecise to determine which files that actually are included statically, so this is only supported when using the dynamic analysis. Additionally we believe that it is a bad practice that possibly could end in unforeseen vulnerabilities when the developer is unaware of which files that possible could be included. Source Code 2.7 shows the supported and some unsupported ways to include files.

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

Source Code 2.7: Examples of different file inclusions.

2.2 Model Analysis

An observation was made during the study. Large systems, like WordPress and Moodle, allow external developers to extend the functionality of the system through a plugin platform. If a developer of a plugin want to identify vulnerabilities in it, the tools tried to analyze the whole system and failed, or did an analysis but missed vulnerabilities due to missing information about the core system.

The code base of WordPress is very large. In version 3.2.1, with no plugins activated, 13.918 built-in and 6 user defined functions were called just for rendering the front page. Even with a dynamic analysis this will cause imprecision and the results with regard to the extensions would suffer from this. With the assumption that the core system is safe, the developers of these 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.

An example of such provided functionality is WordPress' database wrapper which allows the developers to access the database without knowing about the credentials. This wrapper is instantiated by the core of WordPress and then used by the extensions. To overcome the problem of analyzing the core each time we manually identified the functionality of WordPress and wrote a model of it, as seen in Source Code 2.8, which contained the functionality in its simplest vulnerable way.

```
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); }
```

Source Code 2.8: The WordPress model used to identify vulnerabilities in WordPress extensions.

When a plugin is analyzed the model is loaded and analyzed before the actual analysis and the gained information is used during the analysis of the plugin.

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 analyze all of these. One way the plugins are executed, besides being used as an entry point directly, is by registering its own 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/function.

Other plugin systems, like TYPO3, uses the structure of the plugin as entry point. A TYPO3 plugin can be a class description with a main method, and when the module is loaded this main method is executed. The model system allows the model developer to specify which classes that need to be initialized and have its methods called. Before the actual body analysis is performed the AST is extended with nodes equivalent to the ones parsed from the example code in Source Code 2.9.

```
1  $THAPS_0 = new PluginClass;
2  $THAPS_0->method1();
3  $THAPS_0->method2();
```

Source Code 2.9: Generated code to ensure that all methods of a TYPO3 extension is analyzed.

The model system does also need to handle if the extensions is including files from the core system. This is a problem because it would result in an analysis of the core which is undesirable. We have solved this problem by enabling the model developer to define from where file inclusions should be allowed. In the WordPress file structure this would be the directory where the extension is located.

2.3 Dynamic Analysis

To overcome some of the uncertainties of the static analysis that reduce the usefulness of the tool a dynamic analysis of the application can be used. Where the static analysis have trouble

determining which file to include because it is based on user input, the dynamic analysis is able determine it precisely as the application is executed. The same problem exists with regard to reflected function calls. The static analysis cannot determine which function is called and hence fails to do a correct analysis, but the dynamic analysis has this information.

The following sections covers the problems further and shows the implementation of how the information is acquired.

2.3.1 File Inclusions

File inclusion determined by user input is a typical pattern found in PHP applications. Tutorials and forum posts regarding the pattern are easily found on the Internet [4–6]. One example is shown in Source Code 2.10, which is an example of where an input string from the user is used to include a file.

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

Source Code 2.10: Load a file determined by user input.

The regular expression ensures that no input with special characters will be included allowing the attacker to navigate the file system. As the include is ambiguous the static analysis cannot determine which file to include and it ignores the include node. However, if this scenario is evaluated at runtime, the exact file(s) to include can be determined and this information could be parsed on to the static analysis.

2.3.2 Function Calls

The same is the case with reflected function calls. Source Code 2.11 shows a piece of code that static analysis would fail to analyze correctly. It is able to determine that the function call is to a function which name starts with “ajax_”, but if the application has as little as 2 different functions that satisfy this it cannot tell which one to use.

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

Source Code 2.11: Dynamic function call based on user input.

Again the dynamic analysis is able to determine which function is actually called and collect this data for use in the static analysis.

2.3.3 Abstact Syntax Tree Explosion

Another problem with the pure static analysis is that some application architectures makes it difficult to analysis only a part of the application, e.g. where user input is used to determine which files to include. This approach is shown in Source Code 2.12, 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;
8    case "about":
9      include "about.php";
10     break;
11 }
```

Source Code 2.12: Load a specific file determined by user input.

This has the disadvantage in a pure static analysis that the AST often get very large, thus it takes a very long time to scan. Using the dynamic analysis will reduce the number of included files, as in the case above, only one of the file will be included.

2.3.4 Data Gathering

To gather the required data during execution we have developed a Zend extension. When executing PHP, the source code is first parsed and converted into opcodes, which is then interpreted by the Zend Engine. The engine allows extension developers to hook into these opcodes and run a function when such an opcode arise.

Additionally it is possible to hook into events and perform actions when those are triggered. Some of these event are *Module Initialization*, *Request Initialization*, and *Module Shutdown*. The *Request Initialization* event is used to distinguish the requests apart, as separated data is required for them to be useful to the analysis.

Source Code 2.13 shows how the opcode hooks are added on function calls and include/eval opcodes for each request. The dynamic analysis is started if the THAPS cookie is set to `ENABLED`, but it is also possible to start the dynamic analysis from the PHP code by calling the `thaps_enable()` function.

```
1  PHP_RINIT_FUNCTION(thaps) {
2    char *cookie;
3    /* Retrieve the THAPS cookie variable */
4    cookie = get_request_variable(THAPS_TS_CC "_COOKIE", ↵
         INI_STR("thaps.cookie"));
```

```
5
6     /* Lets add some hooks.. :-) */
7     zend_set_user_opcode_handler(ZEND_INCLUDE_OR_EVAL, ↵
          include_or_eval_handler);
8     zend_set_user_opcode_handler(ZEND_DO_FCALL, fcall_handler);
9     zend_set_user_opcode_handler(ZEND_DO_FCALL_BY_NAME, fcall_handler);
10
11    if (cookie != NULL && strncmp(cookie, "ENABLED", 7) == 0) {
12      dbug(THAPS_TS_CC "THAPS cookie (%s) found - trace enabled", ↵
            INI_STR("thaps.cookie"));
13      init_thaps(THAPS_TS_C);
14    }
15    return SUCCESS;
16  }
```

Source Code 2.13: PHP extension function that is executed for each user request.

The `init_thaps()` function establishes a connection to a MongoDB database and generates an unique id to identify the current request, which is appended to header of the response. This is shown in Source Code 2.14.

```
1   void init_thaps(THAPS_TS_D) {
2     bson_oid_t b_oid;
3     char request_id[25];
4     ...
5     /* Generate a new oid and append it to the bson to build */
6     bson_oid_gen(&b_oid);
7     bson_oid_to_string(&b_oid, request_id);
8     ...
9     set_header(THAPS_TS_CC "X-THAPS-RequestId", request_id);
10    ...
11  }
```

Source Code 2.14: Generation of the database id which is sent back to the client by appending it to the HTTP header.

Then when a client sends a HTTP request with the THAPS cookie, the web server makes a response containing the id where to find the runtime information in the database. A MongoDB id is returned in the header in the form of `X-THAPS-RequestID: 4f9668ee172d204f00000001`. This information is used by the static tool, so it initially can fetch the required information to make a complete scan on the basis of information found by the PHP extension.

Source Code 2.15 shows how the data about included files is gathered. The code determines if the dynamic analysis has to be performed and if that is the case, it identifies `include` and `require` statements. When such statement is identified the filename and path is extracted

from the `execute_data` struct and saved in the database along with the file and line information about the include statement.

```
1  int include_or_eval_handler(ZEND_OPCODE_HANDLER_ARGS) {
2    /* Only do this if THAPS is enabled */
3    if (!THAPS_G(thaps_enabled))
4    return ZEND_USER_OPCODE_DISPATCH;
5    ...
6    // Do not support eval()'s
7    if (opline->op2.u.constant.value.lval == ZEND_EVAL) {
8      dbug(THAPS_TS_CC "eval() detected. I'm not touching that!");
9    } else {
10     // We have an include/require request
11     inc_file = thaps_get_zval(THAPS_TS_CC &opline->op1, execute_data);
12     // Check if we can access the file
13     if (zend_stream_open(inc_file->value.str.val, &file_handle ↵
           TSRMLS_CC) == SUCCESS) {
14       ...
15       char *c = thaps_itoa(THAPS_G(c_includes)++);
16       bson_append_start_object( &THAPS_G(bson_includes), c );
17       bson_append_string( &THAPS_G(bson_includes), "current", ↵
             execute_data->op_array->filename );
18       bson_append_int( &THAPS_G(bson_includes), "line", ↵
             get_opline(execute_data) );
19       bson_append_string( &THAPS_G(bson_includes), "included", ↵
             file_handle.opened_path );
20       bson_append_finish_object( &THAPS_G(bson_includes) );
21       // Don't be dirty, lets clean up
22       free(c);
23       zend_file_handle_dtor(&file_handle TSRMLS_CC);
24     } else {
25       ...
26     }
27   }
28   return ZEND_USER_OPCODE_DISPATCH;
29 }
```

Source Code 2.15: Handling include opcodes.

When the dynamic analysis is done the static analysis is performed with this additional information. This means that the static analysis has been altered to have two modes: one where it trust the data blindly and relies on them, and one where it tries to resolve function calls and includes itself.

2.3.5 Crawling a Site

The task of performing a request to a page, collecting the 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. Because it is a trivial task we have automated this process by developing a crawler.

To perform a crawl only a few input is required: the URL of the site on a web server with the dynamic analysis installed, the path to the source code, and an optional cookie that provides authentication on the site. During the crawl the found vulnerabilities will be displayed immediately and if the request has vulnerabilities in common they will be grouped.

We have chosen to use Crawler4j 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.

With the provided information the crawler analyzes 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 from the application directly via the web server.

The first approach test the application the way the developer intended the site to be used. The second approach reveal vulnerabilities in unintended use of the application, because files might be secure when included but not when accessed directly. Combining these two approaches gives a more complete crawl of the entire web application.

Source Code 2.16 shows a part of the controller, where the crawler is configured to send cookies along with each request. The THAPS cookie is sent contemporary with the user submitted cookies from a GUI. The controller is also responsible for specifying which URLs to visit, setting the number of threads to use, etc.

```
1  CrawlConfig config = new CrawlConfig();
2  ...
3  config.setCookie("THAPS", "ENABLED");
4  String cookieField = GUI.getInstance().getCookies();
5  if (!cookieField.equals("")) {
6      try {
7          String[] cookies = cookieField.split(";");
8          for (String cookie : cookies) {
9              String[] cparts = cookie.split("=");
10             config.setCookie(cparts[0], cparts[1]);
11         }
12     } catch (Exception e) {
13         GUI.getInstance().warning("Cookie", "Invalid cookie format ↵
               entered");
14     }
15 }
```

Source Code 2.16: Configuration of the controller where cookies are added to the crawler.

Even though Crawler4j is a complete crawler, it lacked modification of the header information sent and received for each request. The PHP extension required the THAPS cookie to be sent, so Crawler4j needed to be extended to support cookies. Furthermore the received header information was not directly accessible for us, so Crawler4j had to be extended to support this feature as well.

In Source Code 2.17 the controller is instantiated. First the crawler visits the user submitted URL and then starts crawling. The `start` method is blocking for further execution until the crawler is finished visiting all pages. Our own `Controller` class is started as a thread, and thus the rest of the application does not suffer from the crawling process. When the crawler is done visiting links the crawler is started yet another time. This time it crawls the unlinked, but potential accessible pages from the document root. In line 11 the files of the user specified web application document root is retrieved.

```
1  public class Controller implements Runnable {
2      ...
3      /* Crawl linked sites */
4      controller = new CrawlController(config, pageFetcher, ↵
           robotstxtServer);
5      controller.addSeed(GUI.getInstance().getURL());
6      controller.start(Crawler.class, threads);
7
8      /* Crawl docroot files */
9      if (!aborted) {
10         controller = new CrawlController(config, pageFetcher, ↵
               robotstxtServer);
11         ArrayList<String> files = new FileScanner().getFiles();
12         for (int i = 0; i < files.size(); i++)
13             controller.addSeed(files.get(i));
14         Crawler.pubCrawl();
15         controller.start(Crawler.class, threads);
16     }
17 }
```

Source Code 2.17: Initialization of the crawler.

As mentioned earlier, to use Crawler4j it is required to extend its `WebCrawler` class, and Source Code 2.18 shows how it is extended.

```
1  public class Crawler extends WebCrawler {
2      @Override
3      public boolean shouldVisit(WebURL url) {
4          ...
5      }
6      @Override
7      public void visit(Page page) {
8          String requestId = null;
```

```
9           String url = page.getWebURL().getURL();
10          GUI.getInstance().updateStatus("Successfully visited " + url);
11          for (Header h : page.getHeaders()) {
12              if (h.getName().equals("X-THAPS-RequestId")) {
13                  requestId = h.getValue();
14                  break;
15              }
16          }
17          if (requestId != null)
18              execThaps(requestId);
19      }
20  }
```

Source Code 2.18: The `visit` method is called when a new crawled page is ready to be processed. Each time the THAPS id is extracted the static tool knows where to find the required information.

The `shouldVisit` method is called before crawling a page. Here a filter discards common patterns in the requested page, such as MPEG4 videos, RAR files, etc. Furthermore it ensures that the crawler does not visit external sites. The `visit` method is called when a page has been successfully crawled. Here the MongoDB id is found in the HTTP header and used when calling the static tool in the `execThaps` method.

The developers of DoubtfulSystem is not comfortable in using command line tools, so on top of the crawler implementation, a GUI is built in order to increase the usability. Here the Standard Widget Toolkit (SWT) is used. A screenshot of the GUI of the crawler is located in Appendix C

3 Results

The goal with THAPS was to find vulnerabilities in larger systems with a lower number of false positives. To make an assessment of this, THAPS has been tested the same way as the tools from our study. Additionally to test the gain of the model based analysis, which should improve the identification of vulnerabilities in modular systems, several extensions to both WordPress and TYPO3 have been analyzed.

To test the dynamic approach DoubtfulSystem was analyzed, because it makes no use of a modular system, and is too large to analyze in a single run.

3.1 Comparison

To compare THAPS with the tools from the study it has to be tested the same way as them. The test consists of an analysis of four systems: TestApp, DoubtfulSystem, WordPress, and Moodle. TestApp is a script developed by us, made to test the basic functionality of the tool. DoubtfulSystem, made by an anonymous company established by two former students at Aalborg University with less experience in web development, is a system not built on top of a framework. WordPress and Moodle are two publicly available application which are widely used.

Table 3.1 is a copy of the results from the study [1] where the results of THAPS have been filled in. The table lists how many vulnerabilities the tools reported, how many we confirmed, and the number of false positives. Note that the results for WordPress does not include the model approach results, because the other tools were not tested on these extensions.

As seen in the results THAPS does find vulnerabilities in the same systems as the other tools, however, the number of false positives is considerable lower. The only tools performing better is the dynamic tools PHPIDS and PHP Taint as they are reacting to the actual values. As neither of them can be used before deployment the web application will suffer a performance hit as they have to be executed simultaneously. Moodle is the only system in which THAPS does not find vulnerabilities, however, none of the tools but PHPIDS and PHP Taint, were able to identify any either.

3.2 Model Approach

THAPS performs well in the test cases that the other tools were tested against, but one of the strengths of the tool is the models of core systems used to identify vulnerabilities in extensions to the cores. Two of the systems from the test cases, namely WordPress and Moodle,

		TestApp			WordPress			Moodle			DoubtfulSystem		
		FV	**VV**	**FP**	**FV**	**VV**	**FP**	**FV**	**VV**	**FP**	**FV**	**VV**	**FP**
D	PHP Taint	4	4	0	1[①]	1[①]	0[①]	1[①]	1[①]	0[①]	1[①]	1[①]	0[①]
S	Yasca	0	0	0	3	0	3	3	0	3	11	10	1
D	Metasploit Pro	2	2	0		[②]		0	0	0	0	0	0
S	PHP-Sat	6	3	3	3	0	3	10	0	10		[②]	
D	PHPIDS	4	4	0	1[①]	1[①]	0[①]	1[①]	1[①]	0[①]	0[①]	0[①]	0[①]
D	Wapiti	6	4	2	27	0	27	0	0	0	67	15 **?**	0 **?**
S	RIPS	4	4	0	144	1 **?**	14 **?**	21	0 **?**	3 **?**	79	10 **?**	5 **?**
S	CodeSecure	4	4	0		[③]			[③]		0	0	0
D	Secubat	1	1	0	0	0	0	0	0	0	0	0	0
S	Pixy	4	4	0		[②]			[②]			[②]	
D	N-Stalker	1	1	0	0	0	0	0	0	0	0	0	0
D	Acunetix	1	1	0	0	0	0	1	0	1	0	0	0
D	Skipfish	2	2	0	0	0	0	5	0	5	0	0	0
D+S	THAPS	4	4	0	1	1	0	0	0	0	32	28	4

FV = Found vulnerabilities.
VV = Verified vulnerabilties.
FP = False positives.
D = Dynamic tool.
S = Static tool.
?= Partial result of 15 entries (if available).
[①]= Was not tested thorough.
[②]= Failed to scan application.
[③]= Not tested due to missing license.

Table 3.1: Summary of all tested tools. Copied from the study [1] and extended with THAPS.

allow the developers to create extensions. WordPress provides an API which creates a clear cut between the extensions and the core system allowing us to model the core. Moodle on the other hand does not provide this clear cut and creating a model of it is difficult, thus testing using the model approach on Moodle is unsuitable. TYPO3 has been analyzed instead because it provides a clear cut through it's extension API and is still very different in the structure of its extensions compared to WordPress. To show the gain of the model approach we have tested it on several extensions for these two systems and the results is described in the following sections.

3.2.1 WordPress

WordPress is the most used CMS [10] and hosts almost 20.000 different plugins on the WordPress web site. Some of the most popular extensions have been downloaded more than 10 million times [41] and hence if a vulnerability is found in one of them it might result in severe damage.

The 300 most popular and an additional 75 arbitrary plugins were analyzed with THAPS. In total THAPS reported 273 vulnerabilities in 29 of the analyzed plugins. Of these 68 of them was confirmed by manually making exploits for at least one vulnerability for each file in each plugin [11–39].

The results are shown in Table 3.2.

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 potitives.

Table 3.2: Scan results for WordPress extensions.

For a complete list of the extensions with found and confirmed vulnerabilities see Table A.1.

Some of the plugin vulnerabilities were more critical than others. It turned out that many vulnerabilities only were exploitable if the user was logged into the site, and this could make it harder to perform the actual exploit. Other vulnerabilities were more critical though, where XSS was the main vulnerability type. One critical vulnerability worth mentioning was a plugin that was vulnerable to XSS, where it was possible to inject XSS so all visitors of the site could be attacked.

3.2.2 TYPO3

To verify that the model approach works for other systems than WordPress the same procedure was used on TYPO3. TYPO3 uses a MVC approach for its extensions and because of this the model was a bit more complex, however, a subset of TYPO3 was modeled and used to analyze the 100 most popular extensions. Table 3.3 shows the results.

Type	Found	Confirmed	False Positives
XSS	4	1	3
SQL injection	0	0	0
Total	**4**	**1**	**3**

Table 3.3: Scan results for TYPO3 extensions.

THAPS found four vulnerabilities, however, only one of them could be confirmed. A XSS was present in one of the core extensions, *rtehtmlarea*, which is installed along with TYPO3 itself.

No SQL injection vulnerabilities were found. This might be because TYPO3 offers a more robust API for communicating with the database than WordPress. We could, however, find SQL injection vulnerabilities detected in earlier versions of vulnerable extensions.

In retrospect, the model could be more complete, however, it required some modifications to THAPS. Furthermore a couple of the false positives for these extensions was due to the fact that THAPS does not have support for all internal PHP functions.

3.3 Dynamic Approach

The dynamic approach of THAPS had two purposes. Firstly it should help the static tool to determine the result of dynamic code loads, and secondly it should help reducing the code to be analyzed of applications with few entry points. As described DoubtfulSystem is an application with few entry points and the static analysis of THAPS cannot analyze it due to the memory limit of PHP. DoubtfulSystem has therefore been analyzed using the dynamic approach and the results are listed in Table 3.4.

The tool found 32 vulnerabilities in total of which 28 were confirmed by generating exploits manually, and four of them were unexploitable and thus they were marked as false positives. As seen in the table, the biggest problem 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 member of a group and some of the confirmed exploits only works when the user is 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 public available pages.

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

Table 3.4: Scan results for DoubtfulSystem.

As seen in the table the tool performed well against DoubtfulSystem the number of false positives is low and it found several vulnerabilities.

3.3.1 Revised Code Base

The report from the first scan was given to the developers and they returned a revised version on which a new analysis was performed. The results from the revised code base are listed in Table 3.5.

As seen in the table the found vulnerability number has increased. Some of the confirmed vulnerabilities are the same as in the old code base but new ones have also emerged. This is where we believe that the dynamic approach also could help the developers. Limiting the analysis to the part that is vulnerable would reduce the time required to perform the analysis, and hence the tool possibly could be used along with the development.

Type	Found	Confirmed	False Positives
XSS	11	11	0
SQL injection	26	22	4
Total	**37**	**33**	**4**

Table 3.5: Scan results for DoubtfulSystem in the revised code base.

3.4 Test Case Script

As a minor side result THAPS has been tested against the test case script from our study containing various scenarios of vulnerabilities. This test were performed as a measure of the tools capabilities. All test cases for the script are found in Appendix B.

Table 3.6 shows the tested tools and which case they successfully passed. As seen in the table, THAPS successfully passed all tests but one, namely the Regular expression test case. This test case modifies the input string and prints a sanitized string back to the user. Because we do not trust any regular expression functionality and because our focus did not involve regular expression sanitization, the result for this case was a false positive.

		Unknown tainted variable	Conditional sanitizing	Wrong escaping for XSS	Wrong escaping for SQL	Silenced SQL query	Hidden field SQL query	Tainted function returns	Unreachable code	Object-oriented code	Regular expression	Sanitizing source
D	PHP Taint		✓	✓	✓	✓	✓	✓		✓		✓
D	Wapiti		✓	✓	✓	✓		✓		✓	✓	✓
S	CodeSecure	✓	✓	✓	✓	✓	✓	✓	✓			
S	RIPS	✓	✓	✓	✓	✓	✓	✓	✓			
D + S	THAPS	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓

D = Dynamic tool.

S = Static tool.

Table 3.6: Summary of what vulnerability test cases the tools successfully passed. Copied from the study [1] and extended with THAPS.

The test cases should not be seen as a measurement of the quality of the tool, it is solely used to compare the strengths and weaknesses in the most simple form. Each tool has its strengths and weaknesses and all these are elaborated in our study.

4 Conclusion

This report documents the development of THAPS, a vulnerability scanner for PHP web applications. The reason for developing a new tool is on the basis of our study, where we concluded that many of the existing tools did not perform a precise analysis. This imprecision implies that the tools either had no results or many unusable results due to many false positives, rendering the tools less helpful.

The focus of THAPS has been to improve three aspects of detecting vulnerabilities. Firstly the number of false positives in larger applications should be reduced. This has been accomplished by performing a taint analysis used on a symbolic execution of the application, which explores all possible outcomes of an application. As of the results the number of false positives given by THAPS compared to the existing tools is reduced.

Secondly the analysis of extensions of modular systems should be improved. To accomplish this, an analysis is performed on a model of the core system instead of the actual core. The gain is that the possible imprecision of an analysis of the core is reduced. The result of this approach is the discovery of 30 unknown vulnerabilities in extensions to WordPress and TYPO3.

Thirdly specific analysis of newly added features in applications with few entry points should be allowed. Gaining information from a dynamic analysis allows the tool to scan only relevant code regarding the feature. This approach has been used to analyze a newly established company's application. THAPS identified 37 vulnerabilities whereof 33 was confirmed by creating an exploit.

Based on the discovery of unknown vulnerabilities and the limited amount of false positives, we consider the goals of the project have been reached.

THAPS - THAPS Has Awesome Protection Skills!

5 Future Work

Even though the results from our techniques have shown that they work, more work is required to make it even better. The following sections will describe some of the considerations of improvements that we have had during the development.

5.1 New Variable Storage

The current variable storage might explode in branches if the code is written in specific ways such that the clean up code is unable to improve the performance. The result is an analysis that might take very long time or even run out of memory. The same problem exists for large applications.

Currently there is implemented a small improvement where the tree is pruned each time a if-then-else block is left by comparing the subnodes' variables and if they are all equal, they are pruned.

An idea to improve the storage would be to merge similar branches such that the amount of branches is lowered instead of just pruning when all are equal.

5.2 Side effects

The current version of the tool has a very limited support for side effects of methods and functions. There is limited support for global variables, and objects parsed as parameters that are changed within the function or method, are not supported either.

The tool should be extended to support side effects better, either by making the descriptions of the functions and methods better or by analyzing them each time they are called. Analyzing them each time does, however, require the analysis to avoid cyclic functions calls like recursion.

5.3 Automatic Model Generation

Currently the models have to be written by hand, meaning that the writer has to identify the parts of the core system that the extensions are using. This can be a difficult or comprehensive task to perform, so a method to either fully or semi-automatic generate the models from an analysis of the core system and save them for future use, would be preferred. This would give

the information of variables, functions and classes that is available, but the variable storage would be need to be cleaned such that irrelevant information is left out.

This approach can be accomplished in at least two ways. One would be to serialize the information and unserialize it before use. This approach is however difficult to maintain as even the smallest changes to the core would require a new model. Another approach would be to store all the information in the storage in such a way that a pretty printer could write a new model in PHP that would use the current way of loading models. The second approach would require a rewrite of the tool but if a new variable storage is made the model system should be taken into account when designed.

5.4 Exploit Generation

As a part of confirming the results we manually created exploits. To create these exploits the information given by the reports was used. The dependencies and flow path were enough in many cases to blindly create the exploits. The tool should be changed such that it suggests an initial exploit based on these information.

Additionally the information from the dynamic analysis could be used to improve the generating of exploits. An example is a WordPress plugin which uses the provided AJAX functionality to execute the vulnerable code, meaning that another entry point than the actual file had to be used. This information is available from the dynamic analysis.

5.5 Limit False Positives

Even though THAPS did not report an immense amount of false positives, it did report a couple because the use of custom sanitization on the user submitted data. In such cases it is hard to determine if the data is correctly sanitized during the symbolic execution analysis, and currently THAPS does not have any support for this scenario. Using exploit generation and the crawler, it could be possible to decrease the number of false positives, by executing the exploit and check if the exploit was performed successfully.

Bibliography

[1] Heine Pedersen and Torben Jensen. A Study of Web Application Vulnerabilities and Vulnerability Detection Tools. Aalborg University, 2011.

[2] Common Vulnerabilities and Exposures.
http://cve.mitre.org, April 2012.

[3] 2011 CWE/SANS Top 25 Most Dangerous Software Errors.
http://cwe.mitre.org/top25, April 2012.

[4] Web Design Library.
http://www.webdesign.org/web:programming/php/navigation-using-switch.11147.html, Marts 2007.

[5] Digital Web Magazine.
http://www.digital-web.com/articles/easypeasy_php/, Marts 2005.

[6] Codingforums.com: Include switch question.
http://www.codingforums.com/showthread.php?t=85482, Marts 2005.

[7] Programming Language Popularity.
http://langpop.com/, October 2011.

[8] MySQL.com Database Compromised By Blind SQL Injection.
http://techie-buzz.com/tech-news/mysql-com-database-compromised-sql-injection.html, March 2011.

[9] The Hacker News: PBS (Public Broadcasting Service) & Writerspace Hacked Again by Warv0x (AKA Kaihoe).
http://thehackernews.com/2011/06/pbs-public-broadcasting-service.html, June 2011.

[10] W3Techs: Usage of content management systems for websites.
http://w3techs.com/technologies/overview/content_management/all, April 2012.

[11] WordPress Track That Stat 1.0.8 Cross Site Scripting.
http://packetstormsecurity.org/files/112722/, May 2012.

[12] WordPress WP Forum Server 1.7.3 SQL Injection / Cross Site Scripting.
http://packetstormsecurity.org/files/112703/, May 2012.

[13] WordPress GD Star Rating 1.9.16 Cross Site Scripting.
http://packetstormsecurity.org/files/112702/, May 2012.

[14] WordPress iFrame Admin Pages 0.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112701/, May 2012.

[15] WordPress PDF And Print Button Joliprint 1.3.0 Cross Site Scripting.
http://packetstormsecurity.org/files/112700/, May 2012.

[16] WordPress Leaflet 0.0.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112699/, May 2012.

[17] WordPress LeagueManager 3.7 Cross Site Scripting.
http://packetstormsecurity.org/files/112698/, May 2012.

[18] WordPress Media Categories 1.1.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112697/, May 2012.

[19] WordPress Mingle Forum 1.0.33 Cross Site Scripting.
http://packetstormsecurity.org/files/112696/, May 2012.

[20] WordPress Network Publisher 5.0.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112695/, May 2012.

[21] WordPress Newsletter Manager 1.0 Cross Site Scripting.
http://packetstormsecurity.org/files/112694/, May 2012.

[22] WordPress 2 Click Social Media Buttons 0.32.2 Cross Site Scripting.
http://packetstormsecurity.org/files/112711/, May 2012.

[23] WordPress Pretty Link Lite 1.5.2 Cross Site Scripting.
http://packetstormsecurity.org/files/112693/, May 2012.

[24] WordPress SABRE 1.2.0 Cross Site Scripting.
http://packetstormsecurity.org/files/112692/, May 2012.

[25] WordPress Share And Follow 1.80.3 Cross Site Scripting.
http://packetstormsecurity.org/files/112691/, May 2012.

[26] WordPress Sharebar 1.2.1 SQL Injection / Cross Site Scripting.
http://packetstormsecurity.org/files/112690/, May 2012.

[27] WordPress Soundcloud Is Gold 2.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112689/, May 2012.

[28] WordPress Subscribe2 8.0 Cross Site Scripting.
http://packetstormsecurity.org/files/112688/, May 2012.

[29] WordPress WP Easy Gallery 1.7 Cross Site Scripting.
http://packetstormsecurity.org/files/112687/, May 2012.

[30] WordPress WP-Statistics 2.2.4 Cross Site Scripting.
http://packetstormsecurity.org/files/112686/, May 2012.

[31] WordPress WP Survey And Quiz Tool 2.9.2 Cross Site Scripting.
http://packetstormsecurity.org/files/112685/, May 2012.

[32] WordPress Zingiri Web Shop 2.3.5 Cross Site Scripting.
http://packetstormsecurity.org/files/112684/, May 2012.

[33] WordPress CataBlog 1.6 Cross Site Scripting.
http://packetstormsecurity.org/files/112710/, May 2012.

[34] WordPress CodeStyling Localization 1.99.16 Cross Site Scripting.
http://packetstormsecurity.org/files/112709/, May 2012.

[35] WordPress Download Manager 2.2.2 Cross Site Scripting.
http://packetstormsecurity.org/files/112708/, May 2012.

[36] WordPress Download Monitor 3.3.5.4 Cross Site Scripting.
http://packetstormsecurity.org/files/112707/, May 2012.

[37] WordPress Dynamic Widgets 1.5.1 Cross Site Scripting.
http://packetstormsecurity.org/files/112706/, May 2012.

[38] WordPress EZPZ One Click Backup 12.03.10 Cross Site Scripting.
http://packetstormsecurity.org/files/112705/, May 2012.

[39] WordPress GRAND Flash Album Gallery 1.71 Cross Site Scripting.
http://packetstormsecurity.org/files/112704/, May 2012.

[40] Sony Pictures Website Hacked, 1 Million Accounts Exposed.
http://mashable.com/2011/06/02/sony-pictures-hacked/, June 2011.

[41] WordPress.org: Plugin Directory.
http://wordpress.org/extend/plugins/, April 2012.

A WordPress Plugin Results

The 29 vulnerable plugins are listed in the table on the next page. *Found* implies the number of found vulnerabilities according to THAPS, and *Confirmed* implies how many exploits we made in order to confirm a vulnerability. *Reported Files* implies how many files THAPS reported as vulnerable for the plugin.

In the most cases the number of reported files and the number of confirmed vulnerabilities are the same, as we manually generated at least one exploit for each reported file. Some of the plugins contains several confirmed vulnerabilities for each file, because the file contains different types of vulnerabilities, namely SQL injection and XSS and thus an exploit for each type has been generated.

Rank implies the popularity rank when searching for the most popular plugins on the WordPress plugin page. The rank, however, was made after all vulnerabilities was reported to the WordPress team, and because they decided to take down the plugins from their site until the vulnerabilities have been fixed, the rank does not imply the same rank as before contacting the team. The dash means that the plugin was not yet available because the authors did not fix them before the list was assembled on May 7th 2012.

Rank	Plugin	Version	Found	Con-firmed	Reported Files
819	2 Click Social Media Buttons	0.32.2	2	2	2
507	CataBlog	1.6	9	4	3
270	CodeStyling Localization	1.99.16	32	1	1
260	Download Manager	2.2.2	2	2	1
107	WP Download Monitor	3.3.5.4	3	3	1
139	Dynamic Widgets	1.5.1	4	1	1
-	EZPZ One Click Backup	12.03.10	3	1	1
57	GRAND Flash Album Gallery	1.71	2	2	2
337	WP Forum Server	1.7.3	6	3	3
79	GD Star Rating	1.9.16	2	2	1
-	iFrame Admin Pages	0.1	5	1	1
1135	PDF & Print Button Joliprint	1.3.0	3	2	2
-	Leaflet Maps Marker	0.0.1	9	2	2
-	LeagueManager	3.7	11	2	2
1780	Media Categories	1.1.1	4	2	2
72	Mingle Forum	1.0.33	4	3	3
228	Network Publisher	5.0.1	1	1	1
774	Newsletter Manager	1.0	50	6	6
74	Pretty Link Lite	1.5.2	13	4	4
304	SABRE	1.2.0	1	1	1
-	Share and Follow	1.80.3	2	1	1
112	Sharebar	1.2.1	14	2	1
226	Soundcloud is Gold	2.1	4	2	2
71	Subscribe2	8.0	3	3	3
255	Track That Stat	1.0.8	1	1	1
295	WP Easy Gallery	1.7	1	1	1
225	WP-Statistics	2.2.4	2	2	1
485	WP Survey And Quiz Tool	2.9.2	5	5	5
1138	Zingiri Web Shop	2.3.5	13	6	4
		Total	**211**	**68**	59

Table A.1: Results of scanned WordPress plugins.

B Test Cases

The following test cases are recalled from our study. It describes test cases written by ourselves in order to find properties for the tested tools. In this project the test cases are used to compare THAPS with the tools tested in our study for simple vulnerabilities.

Unknown tainted variable

Since the black box tools does not know about the source code, this test is basically a trap no black box tools are able to find. The test simply checks if a specific user variable contains data, but there are no references to this variable on the website, thus the test is called *Unknown tainted variable*. The variable is printed back to the user without being sanitized.

Conditional sanitizing

This tests sanitizes tainted user input when several conditions are met, but if none of these are met, the user input is not sanitized and is written back to the user with tainted content.

Wrong escaping for XSS

In this test the `escapeshellcmd()` function is used when printing the user input instead of the proper `htmlentities()` function. This test tries to trick the static white box tools to think that the user input is properly sanitized, as the user data is not used as a shell command.

Wrong escaping for SQL

Like the above test, this test escapes one field correctly, but another field is sanitized using the improper `htmlentities()` function which paves the way for a SQL vulnerability.

Silenced SQL query

Some tools detect SQL injections by identifying an error message in the response. To get those error messages it submits input that could change the SQL statement such that it becomes invalid. PHP's error control operator can silence warnings and errors, so no messages occur on unexpected behavior. This operator is used in this test.

Hidden field SQL query

Forms typically contains several input fields and some of them can be of the type `hidden`, which is not a visible field on the rendered page but is yet another field that

gets posted when submitting the form. This field can change value as all other form fields. This test contains a hidden field which is not sanitized after being posted and thus an SQL injection is possible.

Tainted function returns

To test the traceability of the white box tools, this test executes a function with tainted input data and returns the same data. The tools should recognize if the input data is sanitized when evaluating the returned result.

Unreachable code

This test is made to trick the static white box tools. Unreachable code cannot be executed by dynamic tools, but the static tools might mark it as a vulnerability. This is not actually the case as this is not a vulnerability, however, small modifications in the source code can make the code executable, thus making it a vulnerability.

Object-oriented code

As many of all the tested tools failed on object-oriented code, a test was created with objects containing tainted user input. This test verifies if the tools are able to track the tainted variables on object-oriented code.

Regular expression

Custom sanitization routines are difficult to evaluate. This test replaces all characters that are not numbers and not in the alphabet with an empty string. Static tools can have difficulties detecting if the output of the regular expression is properly sanitized, whereas dynamic tools can evaluate the output at runtime.

Sanitizing source

This final test sanitizes all user input data for XSS before actually using them. All `$_GET` variables are assigned a new sanitized value, which is done in order to trick the static white box tools, as they might not detect that all these variables have been properly sanitized before use.

C Screenshot of Crawler

Screenshot of THAPS scanning the test case script are shown in Figure C.1 on the following page. The URL and the location of the local document root is entered, if any. The cookie value(s) must be entered if any additional cookies should be supplied when crawling. The cookie list is separated with the ; character, and lastly a click on the "Go!" button initializes the crawling procedure.

The results are shown in an embedded browser window. It shows the request that triggered the static scan, and the static tool populates the database with the vulnerability, the flowpath, and its dependencies.

The 'Save as HTML' button makes it possible to save the report as a HTML file on the local machine.

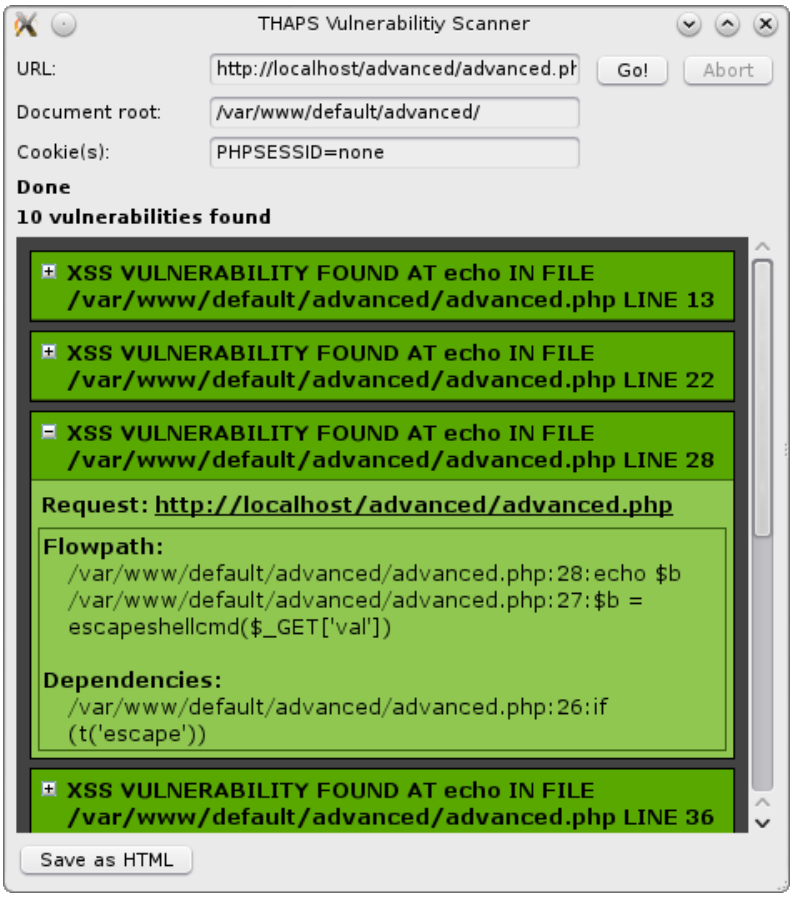

Figure C.1: Screenshot of THAPS crawler.

This page is left blank for the purpose of containing the attached CD-ROM.