PHPWander: A Static Vulnerability Analysis Tool for PHP

Pavel Jurásek

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

PHPWander: A Static Vulnerability Analysis Tool for PHP

Pavel Jurásek

© 2018 Pavel Jurásek

PHPWander: A Static Vulnerability Analysis Tool for PHP

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

PHPWander: A Static Vulnerability Analysis Tool for PHP

Pavel Jurásek

16th May 2018

Contents

Abstract

PHP is a leading server-side scripting language for developing dynamic web sites. Given the prevalence, PHP applications have become the common targets of attacks. One cannot rely on the programmers alone to deliver a vulnerability-free code. Automated tools can help discovering these vulnerabilities. We present PHPWander, a static vulnerability analysis tool for PHP written in PHP. As modern PHP applications are written in object-oriented manner, the tool is able to process object-oriented code as well.

Chapter 1

Introduction

The world of website and web application development is still dominated by the PHP language. Prevalent number of websites are written in PHP [23] [14]. Thanks to its widespreadness and a steep learning curve, PHP is often the first scripting language newcomers to the IT industry get their hands on to. Novices can learn from vast amount of articles, courses, blogposts, Reddit and StackOverflow threads. Unfortunately, many of the articles (for the sake of simplicity or due to the incompetence of an author) omit security measures from the code. It may take some time to get a grasp of security practices in PHP. But even professionals and senior developers cannot guarantee that their code is 100% vulnerability-free.

Coding standards enforcement tools, static analysis tools for programming error detection and test execution tools are standard part of software development and deployment process across every programming language even in small teams. Even such a simple thing as variable highlighting in an IDE is possible thanks to static analysis. We believe that checking for security vulnerabilities should be part of this process as well.

This thesis contributes by development of the PHPWander: a flow-sensitive, interprocedural, and context-sensitive data flow analysis tool for static taint analysis of PHP source code. The tool is open-sourced under MIT[1] license. The tool supports object-oriented features of PHP as modern PHP frameworks and applications are mostly written in object-oriented manner. Language constructs used in procedural and functional programming paradigms are basically also used in OOP and so our tool is no strictly limited to the OOP code.

Why we need such tools is well illustrated by number of security fixes in many popular PHP frameworks and applications. In WordPress, the most widespread CMS, and its plugins there have been found over 10 000 vulnerabilities so far[2]. Hundreds of vulnerabilities were reported for Joomla

[1] https://opensource.org/licenses/MIT
[2] https://wpvulndb.com

CMS[3] and Drupal CMS[4] on CVE Details website. Joomla and Drupal are number two and number three in popularity of PHP content management systems [5]. CVE is a database of disclosed vulnerabilities giving each vulnerability an identification number, description and references. Severe vulnerabilities were also found in popular Roundcube webmail client and phpMyAdmin tool for database administration.

We provide a background to vulnerabilities related to a PHP development, to the PHP language specifics, and to a static analysis. Second part of the thesis describes a development process of the tool. Last part is devoted to an evaluation and a future work is discussed.

[3] https://www.cvedetails.com/vendor/3496/Joomla.html
[4] https://www.cvedetails.com/vendor/1367/Drupal.html
[5] https://websitesetup.org/popular-cms/

Part I

Background

Chapter 2

Security vulnerabilities

The Open Web Application Security Project (OWASP) publishes an annual report of the most critical web application security risks. The report doesn't take into account a particular technology and therefore it is universally applicable to all programming languages used for web application development. We will describe common security vulnerabilities, highlight implications of the vulnerabilities and show common exploitation use cases and how to fix such vulnerabilities. We will refer to OWASP Top 10 Project[1].

2.1 Injections

Listing 2.1: SQL injection

```
$id = $_GET['id'];
$query = "SELECT * FROM user WHERE id = " . $id;
$mysqli->query($query);
```

The first category of vulnerabilities are injections. These can occur when untrusted, user-provided input is passed to an interpreter without proper sanitization. Sanitization is a process of modification of an input to ensure its validity. Listing 2.1 gives an example of SQL injection, where input controlled by an user is passed to an SQL query as is. An adversary can change the HTTP `id` parameter and ultimately change the executed query. For example, another table's contents can be queried using `UNION` statement. A code in given example can be fixed by using a prepared statement as given in Listing 2.2. Prepared statements separate query building from data used in that query. Injection vulnerabilities can also be found in other query or expression languages, LDAP queries, XML parsers, and operating system commands [9].

The impact of injection ranges from violation of confidentiality (attacker can obtain data he wasn't supposed to access) to full takeover of the server.

[1] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Listing 2.2: SQL injection sanitization

```
$query = "SELECT * FROM user WHERE id = ?";
$stmt = $mysqli->prepare($query);
$stmt->bind_param('i', $_GET['id']);
$stmt->execute();
```

2.2 Cross-site scripting (XSS)

Cross-site scripting can occur when user supplied data is used as part of HTML output without proper sanitization. Receiving such a tampered HTML, a browser may execute an illegitimate JavaScript code. The code can carry out any action on an user's behalf. XSS can be DOM based, stored or reflected. Listing 2.3 shows a reflected XSS: the user's browser sends a request to an url and the request parameter (e.g. the request url contains `?text=<script>alert(1);</script>`) value is used in the output. The generated HTML output would be `<p><script>alert(1);</script></p>` and the browser would open an alert window. The user input must be passed through the `htmlspecialchars` built-in PHP function in order to transform all special HTML characters into corresponding safe HTML entities. The user must be tricked into opening such an url with a crafted XSS payload.

Listing 2.3: Reflected XSS

```
<p><?php echo $_GET['text']; ?></p>
```

Some browsers provide basic protection against XSS and don't execute portions of content which is also present in the url[2]. Developers can and are also hereby encouraged to enforce this behaviour by sending the "X-XXS-Protection: 1; mode=block" HTTP header. This header is not the primary protection for XSS, but rather a last resort if other protections fail.

A payload of stored XSS is at first persisted in a storage (it may be a database, a file, or any other storage) and then retrieved in the subsequent requests. The payload is part of generated HTML output and therefore the built-in browser protection wouldn't be triggered. The straightforward way to discover stored XSS is to distrust any output received from these storages and report any use of such data in a generated response. The code in Listing 2.4 should be sanitized the same way as described in the reflected XSS.

Listing 2.4: Stored XSS

```
$article = db_get_article(...);
<p><?php echo $article->getTitle(); ?></p>
```

[2]fun fact: the author of this paper abused this browser behaviour to prevent websites from loading scripts from legitimate advertising companies that would be responsible for loading advertisements into the page

Table 2.1: Context aware escaping in PHP, source [10] (czech)

HTML	`htmlspecialchars($s, ENT_QUOTES)`
XML, XHTML	`preg_replace('#[\x00-\x08\x0B\x0C\x0E-\x1F]+#', '', $s)` and then the same escaping as in HTML
SQL	depends on used DB system
JavaScript, JSON	`json_encode((string) $s)`, then HTML encoding if used in HTML attributes (e.g. `onclick`)
CSS	`addcslashes($s, "\x00..\x2C./:;<=>?@[\\]^‘{|}~")`, then HTML escaping if used in `<style>` tag or `style` attribute
URL	`rawurlencode($s)`

The challenging part of the protection against XSS is to determine a context the data should be escaped in. The data need to be escaped differently for different contexts as every context has its own set of characters with a special meaning. Table 2.1 gives an overview of proper escaping in PHP.

The impact of XSS ranges from simple web page defacement, stealing the user's session data, a bypass of CSRF[3] protection to abusing client's resources, e.g. for mining of crypto currencies.

2.3 Command injection

PHP provides functions for calling operating system commands. In Listing 2.5, we can see that user input is not validated in any way. Passing `.;cat /etc/passwd` as an argument would list current directory, but a content of the `/etc/passwd` file would be given out too.

Listing 2.5: Command execution

```
system('ls '. $_GET['dir']);
```

The capabilities of command injections are limited to the system privileges given to a user owning a web-server or a PHP process.

Arguments passed to these functions must be properly escaped. There are `escapeshellarg($arg)` and `escapeshellcmd($cmd)` functions for this purpose.

2.4 Code injection

[3]Cross-site request forgery: attacker's crafted page can make user's browser send a request to another page without her knowledge. The simplest case is poll rigging, but impacts are usually far more severe.

Listing 2.6: Command execution

```
eval('$var = '. $_GET['value'] .';');
```

In contrast with command injections, code injections are limited to execution of a code of PHP language (or the code of a given language in general). Example exploitations are authentication mechanism bypass or command execution. Given a scenario where an user authentication is controlled by a `$logged` variable, accessing the url `file.php?value=5; $logged = true` makes the interpreter assign number 5 to a variable `$var` and set the variable `$logged` to `true`, effectively overwriting any value passed to the variable `$logged` before.

Code injections allow an attacker to modify a control flow of the executed code or perform a command injection.

Static analysis can be helpful with discovering improper usage of `eval` function. In general, `eval` should not be called with user-provided data.

2.5 Path traversal

A path traversal vulnerability often occurs in a file download functionality or in a dynamic file inclusion based on request parameters. Listing 2.7 shows how some web applications might load the dynamic content. A legitimate request URL might look like `/index.php?page=contact.php`, but a malicious request may be crafted in similar fashion `/index.php?page=../config.yaml` in order to expose the application configuration.

Listing 2.7: Path traversal

```
include_once __DIR__ . '/pages/' . $_GET['page'];
```

Execution of the path traversal attack enables an attacker to access files outside of a web root directory. This usually leads to a disclosure of sensitive data.

Whitelisting is the most effective way of mitigating this vulnerability: only a predefined set of files can be accessed.

2.6 Other vulnerabilities

There are groups of vulnerabilities that are particularly hard or even impossible to detect using static analysis techniques. Vulnerabilities caused by a lack of code, such as access control bypasses [21] and CSRF are inherently omitted by a general-purpose static analysis tools.

Chapter 3

PHP Language

This chapter describes some of the specifics of the PHP language and possible implications on static taint analysis.

3.1 History and description

The history of PHP dates back to 1994, and it took 4 more years to release PHP version 3 that "resembles PHP as it exists today" [5]. PHP (PHP: Hypertext preprocessor) is an open source, multi-paradigm, dynamically typed interpreted scripting language suitable for server-side scripting and command line utilities. Its underlying implementation is written in C and runs on all major systems. PHP also supports most of the web servers and runs as module or CGI is a standardised protocol for communication between web server and a program that generates dynamic content. Current major version is PHP7, active support for PHP5 ended on 19 January 2017 and security support ends on 31 December 2018 [6].

3.2 Typing

In PHP, variables are not declared with explicit types. A variable type is given by a value assigned to it and its type can changed during a runtime. PHP specifies the following types: **bool**ean, **int**eger, **float** and **string** are scalar types, **array**, **object**, **callable** and **iterable** are compound types and **resource** and **NULL** are special types. PHP5 supports type hinting for class names, interfaces, arrays, and callables in function and method signatures [22]. PHP7 introduced scalar type declarations (and also return type declaration) with two modes of operation: either coercive or strict mode. Coercive mode automatically casts improper types to the one declared. In strict mode, **TypeError** is thrown when wrong type is encountered. Strict typing must be enabled on a per-file basis with `declare(strict_types=1);` at the beginning of the file.

Before the introduction of PHP7, the first attempts to support both static and dynamic typing (so called gradual type system) were introduced by

Table 3.1: Predefined variables, source [7]

$GLOBALS	All variables available in global scope
$_SERVER	Server environment information
$_GET	HTTP GET variables
$_POST	HTTP POST variables
$_FILES	HTTP File Upload variables
$_REQUEST	HTTP Request variables
$_SESSION	Session variables
$_ENV	Environment variables
$_COOKIE	HTTP Cookies
$php_errormsg	The previous error message
$HTTP_RAW_POST_DATA	Raw POST data
$http_response_header	HTTP response headers
$argc	The number of arguments passed to script
$argv	Array of arguments passed to script

PHP-like programming language Hack[1]. Developed by Facebook, Hack is a PHP dialect for the HipHop Virtual Machine (HHVM). HHVM is an alternative virtual machine using just-in-time (JIT) compilation.

We can see the transition from loose typing to a more mature type system. Even though variables can still change types on the fly, an introduction of scalar type hints and coercive and strict mode lead to a more predictable and statically analysable code. Code without scalar type hints may cause unexpected behavior such as implicit type casting.

3.3 Predefined variables

Every executed PHP script has a number of predefined variables depending on the environment they run in, e.g. a command line script has `$argc` and `$argv` variables while a script executed from a web server has completely different set of predefined variables. There are special web server, environment, and user input array variables. These are unique in that they are global, that is, they are available in every context. These so called superglobals are of a particular interest to us because they are sources of a potentially malicious data. Some variables are only dynamically created when a particular function is called.

3.4 Object-oriented aspects in PHP

PHP supports a full object-oriented programming model, providing developers with all the power this paradigm has to offer. Overridden methods are always virtual, that is, method to be called is determined at runtime based on an actual type of a given object. Classes can be organised

[1] http://hacklang.org/

into namespaces. This is especially useful when a programmer uses 3rd party libraries because it helps avoid naming conflicts.

As of PHP version 5.4, classes can make use of traits for code reuse. Stateful traits are a way to share common behaviour for otherwise completely unrelated classes and avoid diamond problem at the same time [3].

Classes can implement a number of so-called magic methods that define special behaviour. These are, for example, `__call`, `__callStatic`, `__get`, `__set`, `__isset`, and `__unset` for special treatment of calling undefined and undefined static methods or accessing and setting undefined variables respectively. Constructor `__construct` and destructor `__destruct` are also magic methods.

Objects in PHP also support a serialization to string and reversely a deserialization. Behaviour of the serialization can be specified by implementing the `__sleep` and `__wakeup` magic methods or implementing the `Serializable` interface.

3.5 Class autoloading

It it a good practice to create separate files for every named class definition. Manually including every single file would be tedious. PHP offers the `spl_autoload_register` function that can be used to register autoloaders to automatically load classes and interfaces. Unfortunately PHP executes the autoloaded file which may cause undesired side-effects such as printing an output or even an early termination of static analysis tools. Therefore it is a good practice not to mix class and function declarations with other executable code.

Chapter 4

Static analysis

A static program analysis is a program analysis that is performed on a source code without actual execution of that code. Static analysis is a mere approximation to an actual run-time behaviour. Static analysis is basically a trade-off between a precision and a time consumption.

Source code must be first tokenized and transformed into an AST, a CFG, or any other intermediate representation that is suitable for an analysis.

Use cases of static analysis range from bug discovery, dead code detection, compiler optimisation, to detect security related issues.

4.1 Data flow analysis

Data flow analysis is a compiler and program analysis technique used for optimisation, taint analysis, and other use-cases. Kildall published his data flow analysis for compiler optimisation in 1973 [12]. The program is to be represented by a graph in which every statement is a node having edge to every information which is influenced by the statement [15].

A dynamic data flow analysis performs the data flow analysis on run-time information during an application execution [19]. This however means that the execution path is dependent on concrete input values provided to the program. In order to reach multiple execution branches and achieve high code coverage, the dynamic analysis must be run with a multitude of data sets.

Static analysis, on the other hand, can analyse all possible execution paths and track control-flow dependencies (i.e. set of statements that must have been reached before a particular statement is reached [17]).

Dynamic and static analysis techniques should be considered complementary. A static analysis can work as a pre-stage for a dynamic analysis,

where a static analyser identifies control-flow dependencies that could potentially lead to a vulnerability and thus makes the dynamic analysis more effective [4] [25].

4.2 Taint analysis

Taint analysis tracks information flow between sources and sinks [19]. Taint analysis is a branch of data flow analysis where inputs (sources) - data obtained as user input, from files, databases, web services or by any other means are marked as tainted (low integrity data). If such tainted variable reaches output (sink) without proper sanitization, the analyser yields a warning. Sanitization is a necessary requirement for data to be considered as high integrity data and to be safe to use [2].

4.2.1 Tainting method

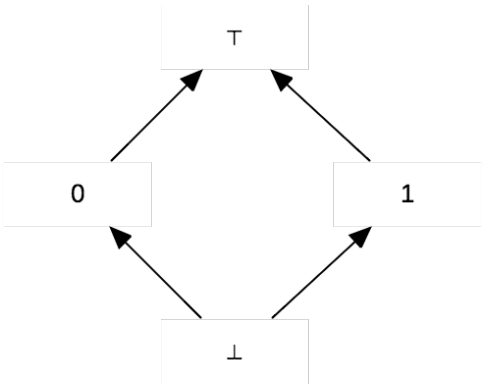


Figure 4.1: Lattice of a taint

For capturing a taint of a variable, we use a lattice structure proposed by [15]. Moreover we use a trivial labelling of tainted and untainted variables with a context they are unsafe and safe to use in. This categorisation is discussed later in Section 5.2. Figure 4.1 shows a schematic drawing of the lattice where symbols $\bot$, 0, 1 and $\top$ stand for "unknown", "untainted", "tainted" and "both", respectively.

4.2.2 Control flow graphs

Control flow graph is a graph representation of a source code where each node of the graph is a basic block and directed edges represent jumps in the control flow. A basic block consists of a linear list of statements [1], it has one entry point and one exit point.

4.3 Data flow sensitivities

As mentioned before, the exact runtime behaviour cannot be determined by a static analyser. The trade-off between speed and precision may be decided based on the phase of the development cycle it is used in. During the development lifecycle, programmers usually need fast response so that his/her workflow doesn't get interrupted. On the other hand, once the code is to be released, programmers expect profound analysis of the code. Therefore, tools can be run with a higher level of precision and the analysis doesn't need to be as fast, but a reasonable time frame still has to be maintained.

The running time of a static analyser can be adjusted by omitting complicated language constructs, by implementation optimisations, and also by adjusting the level of sensitivity of the analysis.

4.3.1 Flow sensitivity

Listing 4.1: Tainted variable $a reaches sensitive sink

```
1 $a = 'No leak';
2 $a = $_GET['a'];
3 echo $a;
```

Listing 4.2: Secure variable $a reaches sensitive sink

```
1 $a = $_GET['a'];
2 $a = 'No leak';
3 echo $a;
```

Listings 4.1 and Listing 4.2 differ in lines 2 and 3 being swapped and result in different call graphs. Flow sensitive analysis distinguishes these changes and correctly mark only the code in Listing 4.1 as vulnerable.

Data flow analysis is flow sensitive by nature as we analyse the code statement by statement in order they occur in a CFG or in other intermediate representation.

Flow insensitive analysis does not consider order of statements in a basic block. A possibility to analyse all statements in parallel and randomly is traded for less precise analysis results. Codes in both Listing 4.1 and Listing 4.2 would be marked as vulnerable.

4.3.2 Context sensitivity

Listing 4.3: Two different calls to a sensitive sink

```
1 function id(string $s): string
2 {
3         return $s;
4 }
5
6 $a = id('No leak');
7 echo $a;
8 $a = id($_GET['a']);
9 echo $a;
```

An approach to context sensitivity is a good example of a trade-off between resulting precision of an analysis and its complexity. The complexity of context sensitive analysis stems from the need of tracking interprocedural information: tracking control flow of function calls and returns from these calls, parameter passing (whether they are passed by value or by reference), aliasing and the scoping problem [20].

Context-insensitive analysis doesn't keep track of a calling context of a sensitive sink. It combines all flow information from all calls into a single flow that is used for the analysis. This behaviour would result in both `echo` calls in Listing 4.3 being marked as vulnerable. As multiple values are passed to the `echo` call and analyser didn't keep track of the context, it needs to consider all calls as the "worst case" scenario, that is, if at least one input value is marked as tainted, then mark all the calls to the sensitive sink as tainted.

Conversely, context-sensitive analysis is able to distinguish what is passed to a sensitive sink and produces more precise analysis. In Listing 4.3 we can see that a static string, passed through the `id` function on line 6 and then `echo`ed to the output on line 7, poses no threat. In contrast, fetching user input on line 8 and `echo`ing it on line 9 can lead to an XSS in client's browser.

4.4 SSA form

SSA form is an intermediate representation of a source code with a property that every variable is assigned to exactly once. This is achieved by renaming variables such that each variable name occurs only once on the left-hand side of an assignment statement. Listing 4.4 and Listing 4.5 illustrate such variable renaming. This representation however introduces a complication in join points of different flow branches. To solve this problem, a ϕ function is used in places where we couldn't pinpoint exact variable "version" [24] [18]. A code corresponding to the phi function usage and a simplified graph excerpt are shown in Listing 4.6 and in Figure 4.2.

Listing 4.4: Code before transformation into SSA

```
$a = 1;
$b = $a;
$a = $a + 3;
$c = $a + $b;
```

Listing 4.5: Code after transformation into SSA

```
$a1 = 1;
$b1 = $a1;
$a2 = $a1 + 3;
$c1 = $a2 + $b1;
```

Listing 4.6: Code requiring a phi function

```
$a = 3;
if (rand(0, 1) === 1) {
        $a = 5;
}
$b = func($a);
```

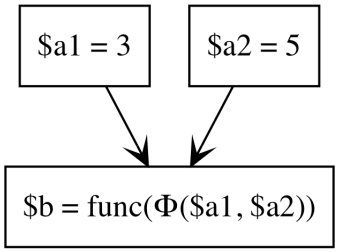


Figure 4.2: SSA join point

4.5 Static analysis tools for PHP

We will give a few examples of tools providing a static analysis of PHP source code. They can either be used for improving code quality and readability or for more complex purposes such as our tool for security checking.

4.5.1 Code improvement tools

As PHP is an interpreted language, there is no compiler that would perform validation of a source code. Every syntactical or semantical error is discovered only when an interpreter tries to interpret a file where the error occurs. Some errors can also be syntactically correct, but don't exhibit intended behaviour, e.g. strict comparison of values with mismatch types always resulting in `false` or conversely a type juggling causing a bypass

of a validation rule. Such erroneous code can unwittingly be deployed to a production environment.

PHPStan[1] is one of the tools that try to substitute a missing compiler. PHPStan provides multiple run levels that provide more and more strict analysis. It checks for common mistakes (such as passing wrong number of parameters to a function call), type incompatibility, use of undefined variables, and more. Framework or project specific rules can be easily added via extensions.

PHP-CS-Fixer and PHP_CodeSniffer are tools that validate one's code against a set of predefined rules of coding standards. These tools aim to maintain uniformity of the formatting of source code.

4.5.2 Security checking tools

Another group of static analysis tools are performing a security checking. Every users' code eventually interacts with the surrounding environment via a database connection, a filesystem, remote system or a standard I/O communication. These tools look for such an interaction and check that no tainted (i.e. user provided) data reach these functions without proper sanitization.

We provide a list of currently publicly available security analysis tools and show a comparison of these tools in Chapter 6.

[1] https://github.com/phpstan/phpstan

Part II

Development

Chapter 5

Development phase

5.1 Architecture

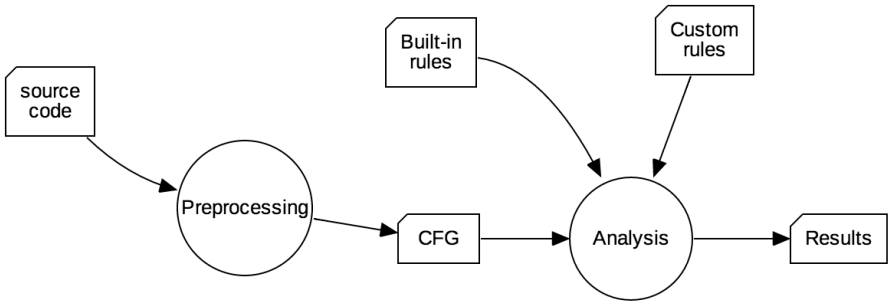


Figure 5.1: Architecture of the project

Figure 5.2 shows an high-level overview of PHPWander: source code is transformed into an intermediate representation that is used in the analysis. Processed node is checked against a set of predefined.

Architecturally, our tool is similar to PHPStan. We also reused some parts of the tool, for instance `Type` namespace for tracking types of variables, and classes for files manipulation.

5.2 Design decisions

5.2.1 Configuration

Potentially dangerous data sources, sinks, securing and tainting functions are defined in configuration files. Configuration is taken from the RIPS (introduced in Chapter 6) [8] and modified according to use in our tool.

Sources are grouped according to the origin of untrusted data. Sanitizing functions are categorized according to a type of vulnerability they mitigate.

These category labels are used to mark variables that are safe to use in context they are sanitized for. For other contexts, variable is still treated as tainted. As an example, variable sanitized againts SQL injection still may cause XSS vulnerability. General sanitizing and tainting functions that sanitize or conversely taint strings are configured.

A formal description of sinks is a slight modification of the one provided by [13]. Sinks are triples <*n*, *a*, *s*>, where *n* is the name of a sink. The sink may be a function or a method. Example sink names are `eval` for a function, `mysqli->query` for the non-static method `query` called on the object of type `mysqli` and `UserClass::sink` for static a method. *a* is a list of numbers that denote what arguments of given sink are to be checked. Number 0 denotes that an output of a sink is tainted if at least one of passed arguments is tainted. *s* is a list of securing functions that prevent given sink from being exploited.

Configuration is done via Neon[1] configuration files. Neon syntax is similar to YAML data serialization language, but it is simpler and faster to compile. An excerpt of how sinks and sanitizers are configured in neon syntax follows:

Listing 5.1: Configuration of sinks

```
xss: # category
        # printf is checked
        # [0] - all parameters are checked
        # \%sanitizingXss\% - reference to
        # section of sanitizing functions in
        # another configuration file
        printf: [[0], \%sanitizingXss\%]
commandExecution:
        # [1] - first parameter is checked
        exec: [[1], \%sanitizingSystem\%]
sqli:
        # call to method query on object
        # of type mysqli is checked
        'mysqli->query': [[1], \%sql\%]
```

Listing 5.2: Configuration of sanitizers

```
sanitizingXss:
        - htmlentities
        - htmlspecialchars
sanitizingSystem:
        - escapeshellarg
        - escapeshellcmd
```

[1] https://ne-on.org/

5.2.2 Software foundations

Software used for building of our tool is influenced by PHPStan: we use parts of the Symfony[2] framework and parts of the Nette[3] framework.

nikic/php-parser

This library[4] parses a source code and converts it into an abstract syntax tree (AST) and supports manipulation of the source code. Node tree traversal and pretty printing (conversion of AST back to source code) is offered out-of-the-box. The library is a major option for PHP source code parsing as almost 400 other tools or libraries depend on it.

ircmaxell/php-cfg

The essential library we use is ircmaxell/php-cfg. It transforms source code into a control flow graph in static single assignment form. SSA representation is advantageous for a reason: compared to an abstract syntax tree, the representation is cleaned from particular structure of the code, but keeps its semantics. This makes traversing and manipulation much easier. For instance, we don't need to look for assign statements in other statements such as conditions and function calls, because we know that all variables are declared beforehand.

Using ircmaxell/php-cfg resulted in two contributions to this library: a simple bug fix and a cooperation on upgrading its dependency on nikic/php-parser to a higher version.

5.2.3 Implementation details

`Analyser` class manages the whole process: it accepts an array of files to be analysed, the file transformation is handed over to ircmaxell
php-cfg package, and constructed CFG is passed to Node scope resolver. `Analyser` also passes currently processed node through a rule set for given node type and collects errors generated by the rule set.

`Node scope resolver` does the processing of individual nodes. Control flow nodes are handled by recursive calls. We keep track of the analysed blocks in order to prevent infinite recursion. We try to evaluate a file inclusion argument to make the analysis as precise as possible, but there are cases when the file cannot be determined.

State of the analysis is captured in `Scope`. It keeps record of current variable taints, a position within the graph, dependencies (statements that need to be reached) and other information necessary for the analysis.

[2] https://symfony.com
[3] https://nette.org
[4] https://github.com/nikic/php-parser

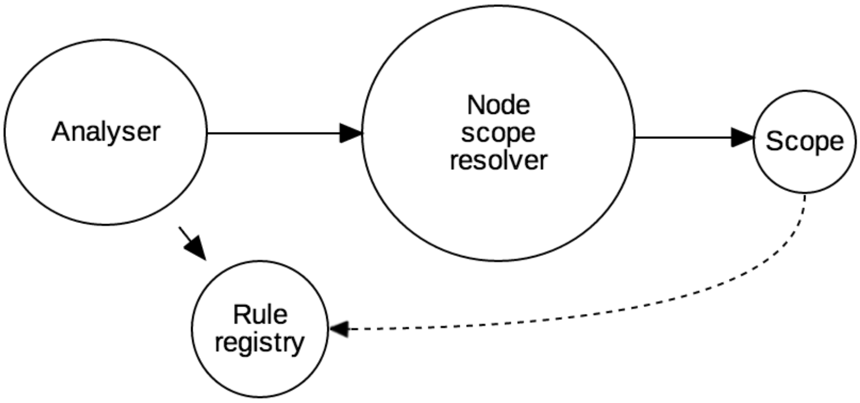
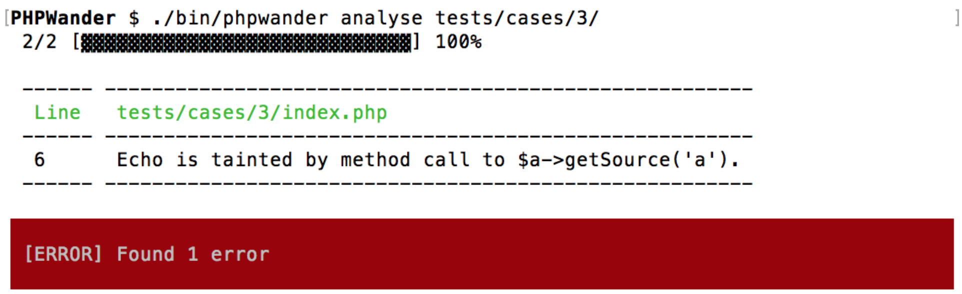


Figure 5.2: Closer look at analysis process

5.2.4 Result output

Result of the analysis is interpreted via the PHPStan formatter in the command line mode. PHPWander does not have a web interface at the time of writing of this thesis, but it is planned to be implemented (see Chapter 7). Example result output can be seen in Figure 5.3.



```
PHPWander $ ./bin/phpwander analyse tests/cases/3/
 2/2 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

 ------ -------------------------------------------------------
  Line   tests/cases/3/index.php
 ------ -------------------------------------------------------
  6      Echo is tainted by method call to $a->getSource('a').
 ------ -------------------------------------------------------

 [ERROR] Found 1 error
```

Figure 5.3: PHPWander CLI output

5.3 Risk analysis

False positives may be reported due to an imprecision of the analysis. All reported vulnerabilities should be manually checked and potential false positives can be simply ignored by custom configuration, and they should be reported on project's page so that PHPWander can be improved.

Part III

Evaluation

Chapter 6

Evaluation

To evaluate PHPWander, we selected three static analysis tools written in PHP: RIPS scanner, Phortress, and Progpilot. All tools are tested against custom testing code snippets and against the Damn Vulnerable Web Application (DVWA)[1], the web application built for penetration testing and for educational purposes.

6.1 Competing tools

6.1.1 RIPS scanner

The RIPS scanner[2] is a tool based on results brought by another tool, Pixy [11]. Released in 2010 by Johannes Dahse, RIPS scanner is written in PHP and reportedly offers partial support for object-oriented code. The source code is tokenized, processed and analysed. SQL injections, XSS, code injection and other vulnerabilities are checked. Public development of this project was discontinued and currently is offered commercially on https://ripstech.com.

6.1.2 Phortress

Phortress[3] is a command line tool that can handle object-oriented code. It uses the nikic/php-parser library (see Section 5.2.2) for transformation of the source code to the abstract syntax tree. The last contribution to the project was in November 2014 and therefore it cannot deal with features introduced in newer versions of PHP.

6.1.3 Progpilot

Progpilot[4] is also a command line tool with capability to analyse object-oriented source code. The configuration is similar to both RIPS and

[1] https://github.com/ethicalhack3r/DVWA

[2] http://php-security.org/2010/05/24/mops-submission-09-rips-a-static-source-code-analyser-for-vulnerabilities-in-php-scripts/index.html

[3] https://github.com/lowjoel/phortress

[4] https://github.com/designsecurity/progpilot

PHPWander. The tool performs analysis on constructed control flow graph. Output of the tool is in JSON format and therefore it is quite hard to manually review produced results.

6.2 Code snippets analysis results

We run presented tools on 16 simple code snippets (link[5]). These should show capabilities of how selected tools handle branching, loops and object-oriented code. Table 6.1 shows number of vulnerabilities reported by those tools.

We can see that Phortress is was unable to analyse more than a half of these test cases. Therefore we exclude this tool from further evaluation.

6.2.1 Comparison

We will describe a few outstanding differences in analysis results and explain their causes.

There are four potential vulnerabilities in code snippet 0: two sensitive `file_get_contents` function calls with tainted arguments and two XSS vulnerabilities. RIPS reported only the first two vulnerabilities, Progpilot also reported the call to `file_get_contents` with argument sanitized by the `basename` function call. This vulnerability is questionable as it indeed prevents directory traversal, but it still may cause infinite recursion in case when the call is "locked" to the same directory the analysed file is located in.

Code snippet number 1 contains a dynamic function call on a tainted variable and was correctly reported by RIPS and by PHPWander.

Code snippets number 3, 10 and 13 test capabilities of the tools to handle object-oriented code. Only PHPWander was able to handle track tainted data wrapped into a class instance.

In code snippet number 4 RIPS and Progpilot omitted a possibility that fetched row from a database contains malicious payload and that it could cause an XSS on line number 15.

We incorrectly propagate sanitization of a variable for one context while keeping it tainted for another context in code snippet number 6. Progpilot also reports a call to `include` with a sanitized variable, but the explanation is the same as in the code snippet 0.

[5] https://github.com/PavelJurasek/PHPWander/tree/master/tests/cases

Table 6.1: Number of vulnerabilities found by tools in code snippets

Snippet no.	RIPS	Phortress	Progpilot	**PHPWander**
0	2	e	5	**4**
1	3	3	2	**3**
2	3	2	3	**3**
3	0	2	0	**1**
4	2	e	2	**3**
5	1	e	1	**1**
6	1	e	2	**1**
7	1	e	1	**1**
8	1	e	0	**1**
9	1	e	0	**2**
10	0	2	0	**1**
11	1	1	0	**1**
12	1	e	0	**3**
13	0	1	0	**1**
14	0	0	1	**1**
15	1	e	1	**1**
sum	18	11	18	**28**

e - error, tool couldn't process the code

Cases 8 and 9 show that Progpilot is having troubles analysing loops while PHPWander is reporting a potential vulnerability in do-while loop twice, which is coming from the nature of this statement. We simplify approximation of the loops by checking whether there is a tainted value in the iterated array and eventually treat the whole array as tainted.

Case 12 show that only PHPWander handles static method calls, while RIPS is only able to track a taint on static properties.

6.2.2 Evaluation

We can see that PHPWander was able to detect more vulnerabilities than other tools. PHPWander performs better when analysing object-oriented code, which was the objective of our effort. We still need to improve propagation of a taint when it comes to santization against one category of vulnerability while keeping tainted for other categories of vulnerabilities. It is fair to mention that these test cases were constructed in order to test capabilities of PHPWander and therefore they do not form a representative and fair sample of vulnerabilities. We propose a neutral test set in Chapter 7 as a future work;

6.3 DVWA analysis results

PHPWander, RIPS and Progpilot were run on latest DVWA version available on it's Github repository's master branch. Tools were set up such

Table 6.2: Number of vulnerabilities found by tools in DVWA

Tool	Reported vulnerabilities	FP	FN	Relevant reports
PHPWander	28	7	5	21
Progpilot	33	12	2	21
RIPS	30	7	4	23

FP - false positive
FN - false negative
Relevant reports - reported vulnerabilities without false positives

that external libraries were not scanned. Number of vulnerabilities these tools found were 40, 30, 49 and respectively. PHPWander and Progpilot has reported some vulnerabilities multiple times. The number of reported vulnerabilities after duplicates removal are 28, 30 and 33. Table 6.2 shows results in more detail.

PHPWander was unable to identify file inclusion vulnerabilities found in `vulnerabilities/fi/index.php` file and in related files `low.php`, `medium.php`, and `high.php`.

Another missed vulnerabilities are SQL injections in `sqli/source/medium.php` and `sqli_blind/source/medium.php`. User-provided value was passed through a sanitizing function, but its usage in SQL query is not enclosed in quotes and therefore the query can be extended with e.g. `UNION` statement.

6.3.1 Evaluation

Based on the results, we can see that PHPWander is able to compete with other available tools. This comparison revealed a few weak spots we should address in further development, though. We will discuss those in Chapter 7.

Every tool was able to identify that all vulnerabilities present in the DVWA eventually boil down to a function call of `dvwaHtmlEcho` in the `dvwaPage.inc.php` file. None of the tools found a CSRF vulnerability. CSRF vulnerability is especially hard to discover by static analysis as static analysis techniques look for patterns in the source code to determine the presence of a vulnerability [16] while it is often caused by a lack code that would prevent this vulnerability.

Chapter 7

Conclusion and future work

We have developed the PHPWander, a tool for static analysis of PHP source code. Besides the ability to analyse procedural source code, it is also able to **analyse object-oriented code**. The tool is interprocedural, flow-sensitive and context-sensitive. The evaluation has shown that our tool is competitive, but needs to improve a few weak spots.

During the evaluation process we have found that development of a similar tool - Progpilot - had been started started approximately five months later after PHPWander. Both tools are architecturally different, but both tools work based on a CFG traversal. PHPWander reported more vulnerabilities in the code snippets tests and reported the same number of relevant vulnerabilities in the DVWA.

For a more relevant comparison, tools should be tested against a larger vulnerability test suite, such as stivalet/PHP-Vulnerability-test-suite. Test suite created for the needs of this thesis is too small to provide a solid confirmation of the tool's vesatility.

7.1 Future work

Even though PHPWander is able to analyse short scripts and simple applications, it is still not ready for production use. We provide a roadmap for further development.

7.1.1 Detailed vulnerability report

Currently supported CLI mode provides enough information for addressing a reported vulnerability, but refactoring of how the report messages are constructed enables a better flexibility and is a prerequisite for a web interface implementation. All concrete implementations of the abstract class `AbstractRule` now returns an array of strings that describe a particular vulnerability. These strings should be replaced with a class that could capture the information in a structured way that could be formatted in many ways.

7.1.2 Web interface

While a command line interface is useful for an incorporation of a tool into a continuous integration environment, it is more comfortable to use a web interface when doing a manual analysis of an application. The web interface should provide a more structured way of results interpretation, e.g. displaying dependencies of a vulnerability that need to be met in order to trigger the vulnerability.

7.1.3 Early termination

`die` and `exit` are built-in functions used for termination of the script execution. Also a `return` statement can be treated as a terminator of a current block while there might be unreachable statements after the return statement. An example is given in Listing 7.1: an `echo` statement on line 3 will never be reached.

Some frameworks implement a function or a method call that throws an exception as a way to terminate a request. PHPWander should offer functionality to mark such a function or method as early-terminating.

The functionality of terminating analysis of current block should be added to PHPWander.

Listing 7.1: Example of unreachable statement

```
function earlyTermination() {
        return true;
        echo $_GET['a'];
}
```

7.1.4 Templating engines support

Modern PHP frameworks separate a business logic, a view and a control between the two. So called templating engines are often used for HTML generation. Ability to analyse templates generated by templating engines such as Twig, Smarty or Latte would be beneficial for users of PHPWander.

7.1.5 Reflection without side-effects

Parser reflection[1] and Better reflection[2] are libraries built upon nikic/php-parser (mentioned in 5.2.2). They provide reflections constructed from a source code without a need to have given class loaded into a memory. This is especially useful to avoid undesired side-effects as described in section 3.5.

[1] https://github.com/goaop/parser-reflection
[2] https://github.com/roave/betterreflection

Bibliography

[1] Frances E. Allen. 'Control Flow Analysis'. In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.acm.org/10.1145/390013.808479.

[2] Sepehr Amir-Mohammadian and Christian Skalka. 'In-Depth Enforcement of Dynamic Integrity Taint Analysis'. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. PLAS '16. Vienna, Austria: ACM, 2016, pp. 43–56. ISBN: 978-1-4503-4574-3. DOI: 10.1145/2993600.2993610. URL: http://doi.acm.org/10.1145/2993600.2993610.

[3] Alexandre Bergel et al. 'Stateful traits and their formalization'. In: *Computer Languages, Systems and Structures* 34.2 (2008). Best Papers 2006 International Smalltalk Conference, pp. 83–108. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2007.05.003. URL: http://www.sciencedirect.com/science/article/pii/S1477842407000140.

[4] Eric Bodden et al. 'Information Flow Analysis for Go'. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 431–445. ISBN: 978-3-319-47166-2.

[5] PHP community. URL: https://secure.php.net/manual/en/history.php.php#history.php3 (visited on 09/02/2017).

[6] PHP community. URL: https://secure.php.net/supported-versions.php (visited on 07/02/2017).

[7] PHP community. URL: https://secure.php.net/manual/en/reserved.variables.php (visited on 13/02/2017).

[8] Johannes Dahse. 'RIPS - A static source code analyser for vulnerabilities in PHP scripts'. In: *The Month of PHP Security* (2010).

[9] OWASP Foundation. *OWASP Top 10 Project*. Tech. rep. OWASP Foundation, 2017. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[10] David Grudl. *Escaping - the ultimate manual*. URL: https://phpfashion.com/escapovani-definitivni-prirucka.

[11] N. Jovanovic, C. Kruegel and E. Kirda. 'Pixy: a static analysis tool for detecting Web application vulnerabilities'. In: *2006 IEEE Symposium on Security and Privacy (SP '06)*. May 2006, pp. 257–263.

[12] Gary A. Kildall. 'A Unified Approach to Global Program Optimization'. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

[13] V. Benjamin Livshits and Monica S. Lam. 'Finding Security Vulnerabilities in Java Applications with Static Analysis'. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1251398.1251416.

[14] BuiltWith Pty Ltd. *Framework Usage Statistics*. 2018. URL: https://trends.builtwith.com/framework (visited on 17/05/2018).

[15] Flemming Nielson, Hanne R. Nielson and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[16] Giancarlo Pellegrino et al. 'Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs'. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 1757–1771. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3133959. URL: http://doi.acm.org/10.1145/3133956.3133959.

[17] P. Puhr-Westerheide. 'Graphs of Data Flow Dependencies'. In: *IFAC Proceedings Volumes* 12.1 (1979). IFAC Workshop on Safety of Computer Control Systems, Stuttgart, Germany, 16-18 May, pp. 117–127. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)65880-4. URL: http://www.sciencedirect.com/science/article/pii/S1474667017658804.

[18] B. K. Rosen, M. N. Wegman and F. K. Zadeck. 'Global Value Numbers and Redundant Computations'. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 12–27. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73562. URL: http://doi.acm.org/10.1145/73560.73562.

[19] Edward J. Schwartz, Thanassis Avgerinos and David Brumley. 'All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)'. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.26. URL: http://dx.doi.org/10.1109/SP.2010.26.

[20] Mario Südholt and Christoph Steigner. 'On interprocedural data flow analysis for object oriented languages'. In: *Compiler Construction*. Ed. by Uwe Kastens and Peter Pfahler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 156–162. ISBN: 978-3-540-47335-0.

[21] Fangqi Sun, Liang Xu and Zhendong Su. 'Static Detection of Access Control Vulnerabilities in Web Applications'. In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=2028067.2028078.

[22] *Types - Introduction*. URL: https://secure.php.net/manual/en/language.types.intro.php (visited on 20/03/2018).

[23] W3Techs. *Usage of server-side programming languages for websites*. 2017. URL: https://w3techs.com/technologies/overview/programming_language/all (visited on 18/01/2017).

[24] Mark N. Wegman and F. Kenneth Zadeck. 'Constant Propagation with Conditional Branches'. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.

[25] Ruoyu Zhang et al. 'Static program analysis assisted dynamic taint tracking for software vulnerability discovery'. In: *Computers and Mathematics with Applications* 63.2 (2012). Advances in context, cognitive, and secure computing, pp. 469–480. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2011.08.001. URL: http://www.sciencedirect.com/science/article/pii/S089812211100664X.

Listings

List of Figures

List of Tables

List of Abbreviations

AST	Abstract syntax tree
CFG	Control flow graph
CGI	Common Gateway Interface
CMS	Content Management System
CVE	Common Vulnerabilities and Exposures
DOM	Document Object Model
IDE	Integrated Development Environment
OOP	Object-Oriented Programming
SSA	Static single assignment
XSS	Cross-site Scripting