Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware: A Literature Survey

Peter Wrench

27th May 2013

Supervisor: Professor Barry Irwin

Department of Computer Science, Rhodes University

Abstract

The creation and proliferation of Remote Access Trojans (or web shells) capable of compromising web platforms has fuelled research into automated methods of dissecting and analysing these shells. In the past, such shells were ably detected using signature matching, a process that is currently unable to cope with the sheer volume and variety of web-based malware in circulation. This survey describes and evaluates some of the notable solutions that have been proposed to address the twin problems of code deobfuscation and dissection with the aim of identifying viable and automatable analysis techniques.

Contents

I Introduction 4

II PHP Overview 4

1 Language Features 5

2 Performance and Use 5

3 Security 6

4 Web Shells 6

III Code Obfuscation 7

1 Methods of Obfuscation 8

1.1 Layout Obfuscation 8

1.1.1 Format Modification 8

1.1.2 Identifier Name Modification 8

1.2 Data Obfuscation 9

1.2.1 Storage and Encoding Modification 9

1.2.2 Data Aggregation 9

1.2.3 Data Ordering 10

1.3 Control Obfuscation 10

1.3.1 Computation Modification 11

1.3.2 Code Aggregation 11

1.3.3 Code Ordering 12

2 Code Obfuscation and PHP 12

3 Deobfuscation Techniques 12

3.1 Pattern Matching 13

3.2 Program Slicing 13

3.3 Statistical Analysis 13

3.4 Partial Evaluation 14

4 Existing Deobfuscation Systems **14**
4.1 LOCO . . . 14
4.1.1 Features . . . 14
4.1.2 Limitations . . . 14
4.2 PHP Deobfuscation using the *evalhook* Module . . . 15
4.2.1 Features . . . 15
4.2.2 Limitations . . . 15

IV Code Dissection **15**

1 Dissection techniques **16**
1.1 Static Approaches . . . 16
1.1.1 Signature Matching . . . 16
1.1.2 Pattern Matching . . . 16
1.2 Dynamic Approaches . . . 16
1.2.1 API Hooking . . . 17
1.2.2 Sandboxes and Function Overriding . . . 17

2 Existing Code Dissection Systems **18**
2.1 Eureka . . . 18
2.1.1 Features . . . 18
2.1.2 Limitations . . . 18
2.2 CWSandbox . . . 18
2.2.1 Features . . . 19
2.2.2 Limitations . . . 19

V Conclusion **19**

Part I

Introduction

The deobfuscation and dissection of PHP-based malware is a non-trivial task with no well-defined solution. Many different techniques and approaches can be found in the literature, each with their own advantages and limitations. In an attempt to evaluate these approaches, this paper begins by providing an overview of the PHP language itself, including its notable features, performance relative to other languages, usefulness, inherent security characteristics and most particularly its role as the language of choice for developers of Remote Access Trojans and other malware. Part three introduces the concept of code obfuscation and the many methods of achieving it, before moving on to discuss techniques for reversing it and briefly exploring existing systems capable of automated code deobfuscation. The fourth part covers code dissection, the two main approaches that are often followed in pursuit of it, and the properties and uses of sandboxes before comparing two widely-used frameworks for analysis. In closing, the paper discusses the viability of PHP as an implementation language, the feasibility and ideal characteristics of an automated deobfuscation system, and finally the approach that should be followed when developing a complete system for the dissection of PHP-based malware.

Part II

PHP Overview

PHP (the recursive acronym for PHP: Hypertext Preprocessor) is a general purpose scripting language that is primarily used for the development and maintenance of dynamic web pages [2, 56]. First conceived in 1994 by Rasmus Lerdof [2], the power and ease of use of PHP has enabled it to become the world's most popular server-side scripting language by numbers [54]. It is able to transform static web pages with predefined content into pages capable of displaying dynamic content based on a set of parameters. Although originally developed as a purely interpreted language, multiple compilers have since been developed for PHP, allowing it to function as a platform for standalone applications [55, 46]. Since 2001, the reference releases of PHP have been issued and managed by The PHP Group [18].

1 Language Features

Much of the popularity of PHP can be attributed to its relatively shallow learning curve. Users familiar with the syntax of C++, C#, Java or Perl are able to gain an understanding of PHP with ease, as many of the basic programming constructs have been adapted from these C-style languages [2, 47]. As is the case with more recent derivatives of C, users need not concern themselves with memory or pointer management, both of which are dealt with by the PHP interpreter [30]. The documentation provided by the PHP group is concise and comprehensively describes the many built in functions that are included in the language's core distribution [48]. The simple syntax, recognisable programming constructs and thorough documentation combine to allow even novice programmers to become reasonably proficient in a short space of time.

PHP is compatible with a vast number of platforms, including all variants of UNIX, Windows, Solaris, OpenBSD and Mac OS X [2]. Although most commonly used in conjunction with the Apache web server, PHP also supports a variety of other servers, such as the Common Gateway Interface, Microsoft's Internet Information Services, Netscape iPlanet and Java servlet engines [49]. Its core libraries provide functionality for string manipulation, database and network connectivity, and file system support [18, 2, 55], giving PHP unparalleled flexibility in terms of deployment and operation.

As an open source language, PHP can be modified to suit the developer. In an effort to ensure stability and uniformity, however, reference implementations of the language are periodically released by The PHP Group [18]. This rapid development cycle ensures that bug fixes and additional functionality are readily available and has contributed directly to PHP's reputation as one of the most widely supported open source languages in circulation today [2, 39]. An abundance of code samples and programming resources exist on the Internet in addition to the standard documentation [50, 69, 57], and many extensions have been created and published by third party developers [51].

2 Performance and Use

PHP is most commonly deployed as part of the LAMP (Linux, Apache, MySQL and PHP/Perl/Python) stack [9]. It is a server-side scripting language in that the PHP code embedded in a page will be executed by the interpreter on the server before that page is served to the client [18]. This means that it is not possible for a client to know what PHP code has been executed - they are only able to see the result. The purpose of this preprocessing is to allow for the creation of dynamic pages that can be customised and served to clients on the fly [2].

When implemented as an interpreted language, studies have found that PHP is noticeably slower than compiled languages such as Java and C [65, 58]. However,

since version 4, PHP code has been compiled into bytecode that can then be executed by the Zend Engine, dramatically increasing efficiency and allowing PHP to outperform competitors written in other languages (such as Axis2 and the Java Servlets Package) [44, 12, 59]. Performance can be further enhanced by deploying commonly-used PHP scripts as executable files, eliminating the need to recompile them each time they are run [3].

At the time of writing, PHP was being used as the primary server-side scripting language by over 240 million websites, with its core module, mod_php, logging the most downloads of any Apache HTTP module [54]. Of the websites that disclosed their scripting language (several chose not to for security reasons), 79.8 percent were running some implementation of PHP, including popular sites such as Facebook, Baidu, Wikipedia and Wordpress [61].

3 Security

A study of the National Vulnerability Database performed in April 2013 found that approximately 30 percent of all reported vulnerabilities were related to PHP [16]. Although this figure might seem alarmingly high, it is important to note that most of these vulnerabilities are not vulnerabilities associated with the language itself, but are rather the result of poor programming practices employed by PHP developers. In 2008, for example, a mere 19 core PHP vulnerabilities were discovered, along with just four in the language's libraries [16]. These numbers represent a small percentage of the 2218 total vulnerabilities reported in the same year [16].

Apart from a lack of knowledge and caution on the part of PHP developers, the most plausible explanation for the large number of vulnerabilities involving PHP is that the language is specifically being targeted by hackers. Because of its popularity, any exploit targeting PHP can potentially be used to compromise a multitude of other systems running the same language implementation [16]. PHP bugs are thus highly sought after because of the high pay-off associated with their discovery. This mentality is clearly demonstrated in the recent spate of exploits targeting open source PHP-based Content Management Systems like phpBB, PostNuke, Mambo, Drupal and Joomla, the last of which has over 30 million registered users [31, 33].

4 Web Shells

The overwhelming popularity of PHP as a hosting platform [54, 45] has made it the language of choice for developers of Remote Access Trojans (or web shells) and other malicious software [13, 43]. Web shells are typically used to compromise web platforms by providing the attacker with basic remote access to the system, including file transfer, command execution, network reconnaissance

and database connectivity [25]. Once infected, compromised systems can be used to defraud users by hosting phishing sites, perform Distributed Denial of Service attacks, and serve as anonymous platforms for sending spam or other malfeasance [26].

The proliferation of such malware has become increasingly aggressive in recent years, with some monitoring institutes registering over 70 000 new threats every day [1, 24]. The sheer volume of software and the rate at which it is able to spread make traditional, static signature-matching infeasible as a method of detection [15, 35]. Previous research has found that automated and dynamic approaches capable of identifying malware based on its semantic behaviour in a sandbox environment fare much better against the many variations that are constantly being created [22, 15, 32, 10]. Furthermore, many malware tools disguise themselves by making extensive use of obfuscation techniques designed to frustrate any efforts to dissect or reverse engineer the code [28, 14]. Advanced code engineering can even cause malware to behave differently if it detects that it is not running on the system for which it was originally targeted [37].

Part III

Code Obfuscation

Code obfuscation is a program transformation intended to thwart reverse engineering attempts [34]. The resulting program should be functionally identical to the original, but may produce additional side effects in an attempt to disguise its true nature [8].

In their seminal work detailing the taxonomy of obfuscation transforms, Colberg et al. define a code obfuscator as a "potent transformation that preserves the observable behaviour of programs" [17]. The concept of "observable behaviour" is defined as behaviour that can be observed by the user, and deliberately excludes the distracting side effects mentioned above, provided that they are not discernible during normal execution. A transformation can be classified as potent if it produces code that is more complex than the original [34].

All methods of code obfuscation can be evaluated according to three metrics [8]:

- Potency - the extent to which the obfuscated code is able to confuse a human reader
- Resilience - the level of resistance to automated deobfuscation techniques
- Cost - the amount of overhead that is added to the program as a result of the transformation

Although primarily used by authors of legitimate software as a method of protecting technical secrets, code obfuscation is also employed by malware authors to hide their malicious code [27]. Reverse engineering obfuscated malware can be tedious, as the obfuscation process complicates the instruction sequences, disrupts the control flow and makes the algorithms difficult to understand [27]. Manual deobfuscation in particular is so time-consuming and error-prone that it is often not worth the effort [27].

1 Methods of Obfuscation

Although the number of code obfuscation methods is limited only by the creativity of the obfuscator, the common ones listed below fall neatly into the three categories of layout, data and control obfuscation [64]. Each category boasts methods of varying potency, and a powerful obfuscator should employ methods from each category to achieve a high level of obfuscation.

1.1 Layout Obfuscation

Perhaps the most trivial form of obfuscation, layout obfuscation is concerned with the modification of the formatting and naming information in a program [19].

1.1.1 Format Modification

The removal of formatting information such as line breaks and white space from source code is the most common method of obfuscation [17]. It can only be performed on programs written in languages that don't depend on formatting as a structural device and is of low potency, as it removes very little semantic content and is easily processed by automated deobfuscation systems [17]. This method is resilient to manual deobfuscation due to the decrease in code readability, however, and can be performed without adding any overhead to the original program [17].

1.1.2 Identifier Name Modification

The transformation or scrambling of meaningful variable names into arbitrary identifiers is another common method of obfuscation [19]. Like format modification, it does not affect the efficiency of the resulting program (it contributes no additional overhead) and fails to confound automated deobfuscation systems [17]. It is of a slightly higher potency, however, as variable names in unmodified form contain a wealth of semantic information that could be of use to a manual deobfuscator [19, 17].

1.2 Data Obfuscation

The obscuring of data structures in a program by modifying how they are stored, accessed, grouped and ordered is known as data obfuscation [17]. It is considered more powerful than layout obfuscation as it obscures the semantics of a program and is able to stymie some automated deobfuscation systems [37]. Programs written using object-oriented languages in particular store much of their semantic information in the form of data structures. Data obfuscation is thus of paramount importance when attempting to obscure code written in these languages [17].

1.2.1 Storage and Encoding Modification

Modifying the data storage characteristics of a program changes the way data structures are stored in memory [19]. Typical examples of this type of obfuscation include variable splitting (parts of a single variable stored in many different locations) and the conversion of static data (such as a string) to procedural data (such as a function that produces the same string at runtime). The former makes it difficult to discern the purpose of a variable (it could be a variable fragment with no individual value) and the latter removes static data that may contain information that could be used to aid in the reverse engineering process [64].

Modifying the data encoding characteristics of a program changes how stored data is interpreted [19]. Changing the encoding of a variable, for example, can make it more complex to reverse engineer, as is demonstrated in Figure 1 below:

```
int i=1;                          int i=11;
while (i < 1000) {    T           while (i<8003) {
  ··· A[i] ···;       ⟹            ··· A[(i-3)/8] ···;
  i++;                              i+=8;
}                                 }
```

Figure 1: Variable Encoding Example [17].

Before the transformation, it is clear that the loop will run exactly 999 times. After the transformation, some simple arithmetic is required to arrive at the same conclusion. Although the encoding in this example is rudimentary, more complex encodings will yield variables that are more resilient to reverse engineering [17].

1.2.2 Data Aggregation

Modifications to the way data is grouped in a program can also serve to obscure the data structures contained within it [19]. Three common examples of this type of obfuscation are listed below:

- Scalar variables such as integers can be merged into a single variable provided that the single variable is sufficiently large to accommodate the scalar variables with no loss in precision [17]. It is possible, for example, to store two 32-bit integers in one 64-bit integer, although this would then require major changes to how each variable is referenced in the rest of the program [17].
- Structures such as arrays can be merged, split, folded or flattened to increase their complexity [64]. These techniques all complicate access to the arrays and further remove them from the data they are intended to represent (flattening a two-dimensional array that was intended to represent a chess board, for example, will make it more difficult to extract this representation after the obfuscation process) [64].
- Class inheritance relationships can be complicated by splitting a single class into multiple classes or by introducing fake classes into the inheritance hierarchy [17]. The result of these operations is a class structure in which classes no longer represent complete entities and relationships are convoluted and illogical [17].

1.2.3 Data Ordering

When constructing a program, it is common practice to follow the principle of locality of reference and group data structures with the functions that are likely to modify them [36]. This fact can be used by deobfuscators to identify which data structures are related to various functions, making it simpler for them to reverse engineer the code . Reordering data structures removes this advantage and increases the complexity of the deobfuscation process. Simple techniques include reordering variables (this often includes making some local variables global to thwart locality analysis), reordering object methods and their parameters, and reordering elements within an array [64]. When data reordering is combined with data aggregation and storage and encoding modification, it becomes very difficult for a deobfuscator to correctly restore the program's data structures [17].

1.3 Control Obfuscation

Perhaps the most important characteristic of a program that needs to be obscured during the obfuscation process is the control flow [17]. Reverse engineering a program when the control flow and data structures are known is a trivial process - as has been discussed, other obfuscation methods such as layout modification are simple to overcome [17]. As was the case with the obfuscation of data, the aggregation and ordering of control flow statements are important and can be modified to increase the program's complexity and resilience [17].

1.3.1 Computation Modification

The modification of the computations involved in the determination of control flow (such as condition calculations in loops and predicate evaluation in if statements) is a powerful method of obfuscation, but it does introduce a significant amount of overhead into the resulting program [17]. Computation modification can be achieved in the following ways:

- Irrelevant code (i.e. code that has no impact on the control flow) can be inserted into a program to frustrate deobfuscators and make the reverse engineering process more time-consuming, as the deobfuscator has no way of knowing whether a section of code is irrelevant until it has processed it [19].
- Loop conditions can be made arbitrarily complex without affecting the number of iterations that will be performed [19]. If the loop was intended to run eight times, for example, the condition could be i $<$ 2 * (24 - 20) instead of i $<$ 8. Once again, this technique is of a low potency, as it serves only to make the deobfuscation process more lengthy [19].
- Dummy processes can be added to the program to distract reverse engineering attempts and code can be parallelised to complicate the control flow, making it more difficult to unravel [17]. The latter technique is considered one of the more powerful methods of obfuscation, as each parallel process increases the number of possible execution paths exponentially, greatly complicating and sometimes defeating the deobfuscation process altogether [17].

1.3.2 Code Aggregation

Much like data aggregation, code aggregation merges dissimilar blocks of code and separates similar blocks of code. Colberg et al. describe the twin goals of code aggregation as follows [17]:

1. Code that a programmer placed in a method (because it logically belonged together) should be scattered throughout the program
2. Code that has no logical relationship should be aggregated into a single method

Further obscuring of the abstractions usually employed by programmers can be achieved through the use of inline and outline methods [64]. Instead of abstracting commonly used code into a separate method, an obfuscator will include this code (as an inline method) wherever it is needed, effectively removing a semantically rich procedural abstraction that could be leveraged by a deobfuscator [64]. Outline methods, by contrast, abstract a section of code that is

not commonly used into a separate method, granting it an undeserved status as a procedural abstraction and potentially misleading any reverse engineering attempts [17].

1.3.3 Code Ordering

When writing code, programmers tend to organise expressions and statements in a logical manner that makes the program easy to read and understand [19]. Since the goal of obfuscation is to discourage understanding, it follows that the ordering of code should be as random as possible. This is trivial for structures such as methods in classes, but in some cases the ordering of statements cannot be entirely randomised because of the dependencies that exist between them (a variable declaration cannot be placed below an expression that includes that variable, for example). In these cases, a dependency analysis must be performed between the two statements before any form of code reordering is attempted [19]. Although reordering is not a powerful method of obfuscation when used in isolation, its effectiveness increases when combined with code aggregation and computation modification [19].

2 Code Obfuscation and PHP

As a procedural language with object-oriented features, PHP can be obfuscated using all of the methods detailed above [56]. In addition to this, the language contains several functions that directly support the protection/hiding of code and which are often combined to form the following obfuscation idiom [66]:

```
eval(gzinflate(base64_decode(str_rot13($str))))
```

To begin with, the string containing the malicious code is encrypted using the rot13 algorithm. It is then encoded in base64 (using base64_encode) before being compressed (using gz_inflate). At runtime, the process is reversed and the *eval* function is used to evaluate the resulting string as PHP code [48, 66].

Although seemingly complex, code obfuscated in this manner can easily be neutralised and analysed for potential backdoors. Replacing the *eval* function with an echo command will display the code instead of executing it, allowing the user to determine whether it is safe to run. This process can be automated using PHP's built in function overriding mechanisms [62], which are examined in more detail in Part IV, Section 1.2.2.

3 Deobfuscation Techniques

The obfuscation methods described in the previous sections are all designed to prevent code from being reverse engineered. Given enough time and resources,

however, a determined deobfuscator will always be able to restore the code to its original state. This is because perfect obfuscation is provably impossible, as is demonstrated by Barak et al. in their seminal paper "On the (Im)possibility of Obfuscating Programs" [5]. Colberg et al. concur, postulating that every method of code obfuscation simply "embeds a bogus program within a real program" and that an obfuscated program essentially consists of "a real program which performs a useful task and a bogus program that computes useless information" [17]. Bearing this in mind, it is useful to review the techniques that are widely employed by existing deobfuscation systems.

3.1 Pattern Matching

Sophisticated deobfuscation systems are able to construct databases of previously detected bogus code segments [64]. They can then compare fragments of an obfuscated piece of code with the patterns stored in the database and remove these fragments from the program before applying the other techniques described below [64]. The resultant decrease in the size of the program greatly increases the efficiency of the deobfuscator - the larger the database, the greater the increase in efficiency [64].

3.2 Program Slicing

Deobfuscators that employ program slicing techniques are able to split an obfuscated program into manageable units called slices that it can then evaluate both individually and in relation to other slices[17]. In this way, the system can avoid bogus code entirely and group similar code blocks together, reversing the efforts of the obfuscator and making the code more readable [17]. Advanced slicing systems are able to create chains of slices leading up to a target slice that represent the code blocks that were executed up to that point, even if said blocks were scattered throughout the program [17].

3.3 Statistical Analysis

Like pattern matching, statistical analysis aims to remove unimportant code, but it is able to do so without knowledge of previously discovered bogus segments [17]. Instead, the deobfuscator will repeatedly test an expression in an obfuscated program and record the results [38]. If the expression always returns the same value, then it is likely to belong to the meaningless part of the obfuscated code and can safely be substituted with the value itself or removed from the program altogether [17].

3.4 Partial Evaluation

A partial evaluator is a system capable of splitting a source program into a static segment and a dynamic segment [17]. The static segment consists of all the code that can be identified and computed by the evaluator prior to runtime [17]. This code can be considered to be unimportant in the sense that it produces no useful result and therefore corresponds to the spurious code blocks often introduced by code obfuscators. Once the static segment has been removed, the remaining dynamic segment represents the original program [17].

4 Existing Deobfuscation Systems

Several automatic tools exist online that are capable of deobfuscating PHP code [41, 4]. The source code for these tools is not available, however, and their features are not well documented or even disclosed, making them poor subjects to study. Instead, a brief summary of two generic deobfuscation systems is presented below, with a view to identifying features to replicate and pitfalls to avoid.

4.1 LOCO

LOCO is a interactive graphical environment in which a user can experiment with and observe the effects of both obfuscation and deobfuscation transformations [29].

4.1.1 Features

Based on a visualisation tool called Lancet and an obfuscation infrastructure called Diablo, LOCO is able to expose the control flow of a program and show the effects of any obfuscating or deobfuscating actions on it [29]. Users can choose either to execute and evaluate existing obfuscation/deobfuscation transformations or to develop and test transformations of their own. The environment's visualisation feature is particularly helpful when it comes to identifying flaws in deobfuscation transformations, as the user can step through the program and identify the effects of the transformation at any point in the code [29]. It also facilitates the manual deobfuscation of programs by allowing users to modify the source code and observe how each modification affects the flow of control [29].

4.1.2 Limitations

Although LOCO includes powerful transformation testing and visualisation features, it is more a tool for developing deobfuscation systems than a system in

itself. It lacks the ability to store and reuse code transformations, and its built in deobfuscation algorithms are designed to be extensible rather than comprehensive [29]. LOCO also functions at the assembler level, which gives it more flexibility but means that its algorithms cannot be adapted for use in deobfuscation systems that function at a higher level [29].

4.2 PHP Deobfuscation using the *evalhook* Module

In a study attempting to analyse exploitation behaviour on the web, Canali and Balzarotti [11] found it necessary to develop and implement an automated deobfuscator of PHP code.

4.2.1 Features

The system implemented by Canali and Balzarotti made use of the *evalhook* PHP extension, which attaches itself to all calls to dynamic code evaluation functions such as *eval [11].* This meant that any malicious code hidden in an *eval* construct could be monitored in real time [11]. The system was able to achieve a success rate of 24 percent, which is remarkable since it relied solely on one very specific deobfuscation technique [11]. It was also fully automatic, requiring no human intervention during the deobfuscation process [11].

4.2.2 Limitations

As an auxiliary system to the main project, the deobfuscator lacked several of the deobfuscation techniques discussed in Section 3 of Part III. As a result of this, it was unable to correctly deobfuscate scripts encoded with proprietary tools such as Zend Optimiser or ionCube PHP Encoder [11]. The incorporation of other techniques would increase the robustness of the system, as well as its success rate [11].

Part IV
Code Dissection

The process of analysing the behaviour of a computer program by examining its source code is known as code dissection or semantic analysis [7]. The main goal of the dissection process is to extract the primary features of the source program, and, in the case of malicious software, to neutralise and report on any undesirable actions [63]. Sophisticated anti-malware programs go beyond traditional signature matching techniques, employing advanced methods of detection such as sandboxing and behaviour analysis [60].

1 Dissection techniques

All code dissection techniques can be classified as being either static or dynamic in nature [7].

1.1 Static Approaches

Static analysis approaches attempt to examine code without running it [40]. Because of this, these approaches have the benefit of being immune to any potentially malicious side effects. The lack of runtime information such as variable values and execution traces does limit the scope of static approaches, but they are still useful for exposing the structure of code and comparing it to previously analysed samples [68].

1.1.1 Signature Matching

A software signature is a characteristic byte sequence that can be used to uniquely identify a piece of code [68]. Anti-malware solutions make use of static signatures to detect malicious programs by comparing the signature of an unknown program to a large database containing the signatures of all known malware - if the signatures match then the unknown program is flagged as suspicious. This kind of detection can easily be overcome by making trivial changes to the source code of a piece of malware and thereby modifying its signature [67].

1.1.2 Pattern Matching

Pattern matching is a generalised from of signature matching in which patterns and heuristics are used in place of signatures to analyse pieces of code [68]. This allows pattern matching systems to recognise and flag code that contains patterns that have been found in previously analysed malware samples, which, although an improvement on signature matching, is still insufficient to identify newly developed malware [68]. Patterns that are too general will lead to false positives (benign code that is incorrectly classified as malicious), whereas patterns that are too specific will suffer from the same restrictions faced by signature matching [68].

1.2 Dynamic Approaches

Dynamic approaches to analysis extract information about a program's functioning by monitoring it during execution [40]. These approaches examine how a program behaves and are best confined to a virtual environment such as a sandbox so as to minimise the exposure of the host system to infection [40].

1.2.1 API Hooking

API (Application Programming Interface) hooking is a technique used to intercept function calls between an application and an operating system's different APIs [23]. In the context of code dissection, API hooking is usually carried out to monitor the behaviour of a potentially malicious program [6]. This is achieved by altering the code at the start of the function that the program has requested access to before it actually accesses it and redirecting the request to your own injected code [6]. The request can then be examined to determine the exact behaviour exhibited by the program before it is directed back to the original function code [23].

The precision and volume of code required for correct API hooking mean that behaviour monitoring systems that make use of the technique are complex and time consuming to implement [6]. They are also virtually undetectable and thoroughly customisable (only functions relevant to behaviour analysis need be hooked) [6].

1.2.2 Sandboxes and Function Overriding

A sandbox is a restricted programming environment that is used to separate running programs [20]. Malicious code can safely be run in a sandbox without affecting the host system, making it an ideal platform for the observation of malware behaviour [21].

Later versions of PHP include the Runkit_Sandbox class that is capable of executing PHP code in a sandbox environment [52]. This class creates its own execution thread upon instantiation, defines a new scope and constructs a new program stack, effectively isolating any code that is run within it from other active processes [52]. Other options are also provided to further restrict the sandbox environment [52]:

- The safe_mode_include_dir option can be used to specify a single directory from which modules can be included in the sandbox
- The open_basedir option can be used to specify a single directory that can be accessed from within the sandbox
- The allow_url_fopen option can be set to false to prevent code in the sandbox from accessing content on the Internet
- The disable_functions and disable_classes options can be used to disable any functions and classes from being used inside the sandbox

Of particular interest to a developer of a code dissection system is the *runkit.internal* configuration directive that can be used to enable the ability to modify, remove or rename functions within the sandbox [53]. This can facilitate the dissection

of PHP code by providing the functionality to replace functions associated with code obfuscation (such as *eval*) with benign functions that merely report an attempt to execute a string of PHP code [53]. Network activity could be monitored in much the same way - calls to *url_fopen* could be replaced by an echo statement that prints out the URL that was requested by the code.

2 Existing Code Dissection Systems

Two slightly different code dissection systems are presented below: the first uses dynamic analysis and execution tracing and the second uses dynamic analysis and API hooking [38, 42].

2.1 Eureka

Designed by Sharif et al. in 2008, Eureka is a framework that aims to enable dynamic malware analysis [38, 48].

2.1.1 Features

Eureka is able to analyse malware by employing statistical analysis and execution tracing techniques [38]. These techniques allow the system to identify API calls (without resorting to traditional dynamic analysis approaches such as a sandbox) and even group these calls according to their functionality [38]. Execution tracing is performed by logging all system calls made by a process bearing the malware's program ID and statistical analysis is performed on the program's memory space to determine when it terminates and if it terminates correctly [38].

2.1.2 Limitations

Eureka is unable to track the execution of malware that only reveals part of its source code during an execution stage and then re-encrypts the code once it has been run [38]. It is also possible that a piece of malware capable of detecting API hooking could avoid certain system calls and thereby avoid setting off the triggers that drive the framework [38].

2.2 CWSandbox

CWSandbox is a generic malware analysis tool that boasts automatic, effective and accurate software analysis [42]. It is automated in the sense that it is able to produce detailed reports of malware activity with no user intervention

and effective in the sense that it is able to produce a comprehensive list of the detected features [42]. It is correct in the sense that no false positives are returned (i.e. all the logged activity was a result of the actions of the malware) [42].

2.2.1 Features

CWSandbox analyses malware dynamically in a sandbox environment [42]. Because of this, it is able to bypass the problems faced by static analysers when faced with obfuscated code, as it is concerned solely with the behaviour of the code at runtime [42]. As was the case with the Eureka framework, CWSandbox uses API hooking to determine malware behaviour [42]. The system is able to monitor all calls to the Windows API during execution and determine whether each call has originated from the malware or not [42].

2.2.2 Limitations

As a large-scale, commercial malware analysis system, CWSandbox is able to accurately dissect most malware instances [42]. The system can be bypassed, however, by making system calls directly to the kernel instead of via the Windows API [42]. Since the system is not able to monitor calls to the kernel, this malware activity would go unnoticed [42].

Part V

Conclusion

The paper began with a discussion on the merits of the PHP language. It was found to be a robust, fully-featured language that employs a simple, C-like syntax, making it easy to learn and develop in. As a language with a well-developed community, PHP enjoys regular updates and bug fixes and is endowed with a comprehensive set of documentation and example code. Although the language is associated with many security flaws, it was determined that these flaws generally occur as a result of poor programming practice on the part of PHP developers rather than core issues with the language itself.

Code obfuscation was introduced as an obstacle to automated code dissection. Various methods of obfuscation were presented and it was determined that a combination of these techniques greatly complicated the deobfuscation process. Techniques for reversing code obfuscation were then presented and it was found that even highly obfuscated code could be restored to its original state given enough time. Two existing deobfuscation systems were briefly introduced and

evaluated. LOCO, a graphical environment for observing the effects of obfuscating transforms, proved to be more a tool for developing a deobfuscation system than a system in itself. The second system made use of the *evalhook* module, employed only one deobfuscation technique, and was able to decode 24 percent of the scripts that it encountered.

The concept of code dissection was then introduced and discussed. The two main approaches to dissection - namely static and dynamic analysis - were compared, and it was found that dynamic analysis techniques fared better against new types of malware, but were more complex to implement. Two existing code dissection systems were also compared: the first, Eureka, was able to dissect most malware examples, but was stymied by code that only revealed part of its source during a given execution stage and then re-encrypted itself. CWSandbox was found to be a powerful commercial code analyser with only one observable flaw - it could not intercept system calls made directly to the kernel and was thus unable to dissect malware that behaved in this way.

After discussing the ease of use, security, performance, and feature set of PHP, it became clear that it would be a fitting host language for the implementation of a code dissection system. A review of the literature concerning code deobfuscation and dissection revealed that a dynamic analysis approach with a sandbox as its primary testing entity was the most viable solution. With its built in sandbox and a wide array of functions deliberately designed to facilitate the analysis of live code, PHP was chosen as the sensible implementation choice.

References

[1] Malware Statistics. Online, 2009. Available from: `http://www.av-test.org/en/statistics/malware/`.

[2] Argerich, L. *Professional PHP4*. Professional Series. Wrox Press, 2002.

[3] Atkinson, L., and Suraski, Z. *Core Php Programming*. Core series. PRENTICE HALL COMPUTER, 2004.

[4] Ballast Security. PHP Decoder. Online, June 2012. Available from: `https://www.ballastsecurity.net/php-decoder/`.

[5] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. On the (im) possibility of obfuscating programs. In *Advances in Cryptology-CRYPTO 2001* (2001), Springer, pp. 1–18.

[6] Berdajs, J., and Bosnic, Z. Extending applications using an advanced approach to dll injection and api hooking. *Software: Practice and Experience 40*, 7 (2010), 567–584.

[7] Binkley, D. Source code analysis: A road map. In *2007 Future of Software Engineering* (Washington, DC, USA, 2007), FOSE '07, IEEE Computer Society, pp. 104–119.

[8] Borello, J.-M., and Me, L. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology 4*, 3 (2008), 211–220.

[9] Bughin, J., Chui, M., and Johnson, B. The next step in open innovation. *The McKinsey Quarterly 4*, 6 (2008), 1–8.

[10] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.

[11] Canali, D., and Balzarotti, D. Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium* (San Diego, États-Unis, Feb. 2013), p. n/a.

[12] Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoel, W. Performance comparison of middleware architectures for generating dynamic web content. In *Middleware 2003*, M. Endler and D. Schmidt, Eds., vol. 2672 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 242–261.

[13] Cholakov, N. On some drawbacks of the php platform. In *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing* (New York, NY, USA, 2008), CompSysTech '08, ACM, pp. 12:II.7–12:2.

[14] Christodorescu, M., and Jha, S. Testing malware detectors. *SIGSOFT Softw. Eng. Notes 29*, 4 (July 2004), 34–44.

[15] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on* (May), pp. 32–46.

[16] Coelho, F. PHP-related vulnerabilities on the National Vulnerability Database. Online, April 2013. Available from: `http://www.coelho.net/php_cve.html`.

[17] Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[18] Doyle, M. *Beginning PHP 5.3*. Wiley, 2011.

[19] Ertaul, L., and Venkatesh, S. Jhide-a tool kit for code obfuscation. In *8th IASTED International Conference on Software Engineering and Applications (SEA 2004)* (2004), pp. 133–138.

[20] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (Berkeley, CA, USA, 1996), SSYM'96, USENIX Association, pp. 1–1.

[21] Gong, L., Mueller, M., and Prafullch, H. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1997), pp. 103–112.

[22] Hyung Chan Kim, Daisuke Inoue, M. E. Y. T. K. N. Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation. Online, 2009. Available from: `http://jwis2009.nsysu.edu.tw/location/paper/Toward%20Generic%20Unpacking%20Techniques%20for%20Malware%20Analysis%20with%20Quantification%20of%20Code%20Revelation.pdf`.

[23] Ivanov, I. Api hooking revealed. *The Code Project* (2002).

[24] Kaspersky, E. Number of the Month: 70K per day. Online, October 2011. Available from: `http://eugene.kaspersky.com/2011/10/28/number-of-the-month-70k-per-day/`.

[25] Kazanciyan, R. Old Web Shells, New Tricks. Online, December 2012. Available from: https://www.owasp.org/images/c/c3/ASDC12-Old_Webshells_New_Tricks_How_Persistent_Threats_haverevived_an_old_idea_and_how_you_can_detect_them.pdf.

[26] Landesman, M. Malware Revolution: A Change in Target. Online, March 2007. Available from: http://technet.microsoft.com/en-us/library/cc512596.aspx.

[27] Laspe, E. An Automated Approach to the Identification and Removal of Code Obfuscation. Online, September 2008. Available from: http://www.blackhat.com/presentations/bh-usa-08/Laspe_Raber/BH_US_08_Laspe_Raber_Deobfuscator.pdf.

[28] Li, J., Xu, J., Xu, M., Zhao, H., and Zheng, N. Malware obfuscation measuring via evolutionary similarity. In *Future Information Networks, 2009. ICFIN 2009. First International Conference on* (Oct.), pp. 197–200.

[29] Madou, M., Van Put, L., and De Bosschere, K. Loco: an interactive code (de)obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 2006), PEPM '06, ACM, pp. 140–144.

[30] McLaughlin, B. *PHP & MySQL: The Missing Manual*. Missing Manual. O'Reilly Media, Incorporated, 2012.

[31] Miller, R. PHP Apps A Growing Target for Hackers. Online, January 2006. Available from: http://news.netcraft.com/archives/2006/01/31/php_apps_a_growing_target_for_hackers.html.

[32] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (Dec.), pp. 421–430.

[33] Open Source Matters. What is Joomla? Online, January 2013. Available from: http://www.joomla.org/about-joomla.html.

[34] Preda, M., and Giacobazzi, R. Semantic-based code obfuscation by abstract interpretation. In *Automata, Languages and Programming*, L. Caires, G. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., vol. 3580 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 1325–1336.

[35] Preda, M. D., Christodorescu, M., Jha, S., and Debray, S. A semantics-based approach to malware detection. *SIGPLAN Not. 42*, 1 (Jan. 2007), 377–388.

[36] Rogers, A., and Pingali, K. *Process decomposition through locality of reference*, vol. 24. ACM, 1989.

[37] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Impeding malware analysis using conditional code obfuscation, 2009.

[38] Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., and Lee, W. Eureka: A framework for enabling static malware analysis. In *Computer Security - ESORICS 2008*, S. Jajodia and J. Lopez, Eds., vol. 5283 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 481–500.

[39] Sklar, D. *Learning PHP 5*. O'Reilly Media, 2008.

[40] Spasojevic, B. Using optimization algorithms for malware deobfuscation.

[41] Sucuri Labs. PHP Decoder. Online, January 2012. Available from: `http://ddecode.com/phpdecoder/`.

[42] Sunbelt Software. CWSandbox Service. Online, May 2013. Available from: `https://mwanalysis.org/?site=1&page=about`.

[43] Sunner, M. The rise of targeted trojans. *Network Security 2007*, 12 (2007), 4 – 7.

[44] Suzumura, T., Trent, S., Tatsubori, M., Tozawa, A., and Onodera, T. Performance comparison of web service engines in php, java and c. In *Web Services, 2008. ICWS '08. IEEE International Conference on* (2008), pp. 385–392.

[45] Tatroe, K. *Programming Php*. Oreilly & Associates Inc, 2005.

[46] Tatsubori, M., Tozawa, A., Suzumura, T., Trent, S., and Onodera, T. Evaluation of a just-in-time compiler retrofitted for php. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 121–132.

[47] The PHP Group. Basic Syntax. Online, May 2013. Available from: `http://php.net/manual/en/language.basic-syntax.php`.

[48] The PHP Group. Function Reference. Online, May 2013. Available from: `http://www.php.net/manual/en/funcref.php`.

[49] The PHP Group. Installation and Configuration. Online, May 2013. Available from: `http://www.php.net/manual/en/install.php`.

[50] The PHP Group. PEAR - PHP Extension and Application Repository. Online, 2013. Available from: `http://pear.php.net/`.

[51] The PHP Group. PECL. Online, January 2013. Available from: `http://pecl.php.net/`.

[52] The PHP Group. Runkit Sandbox. Online, May 2013. Available from: `http://php.net/manual/en/runkit.sandbox.php`.

[53] The PHP Group. Runtime Configuration. Online, May 2013. Available from: http://php.net/manual/en/runkit.configuration.php.

[54] The PHP Group. Usage Stats for January 2013. Online, May 2013. Available from: http://php.net/usage.php.

[55] The PHP Group. What Can PHP Do? Online, May 2013. Available from: http://www.php.net/manual/en/intro-whatcando.php.

[56] The PHP Group. What Is PHP? Online, May 2013. Available from: http://www.php.net/manual/en/intro-whatis.php.

[57] The Resource Index Online Network. The PHP Resource Index. Online, January 2005. Available from: http://php.resourceindex.com/.

[58] Titchkosky, L., Arlitt, M., and Williamson, C. A performance comparison of dynamic web technologies. *SIGMETRICS Perform. Eval. Rev. 31*, 3 (Dec. 2003), 2–11.

[59] Trent, S., Tatsubori, M., Suzumura, T., Tozawa, A., and Onodera, T. Performance comparison of php and jsp as server-side scripting languages. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2008), Middleware '08, Springer-Verlag New York, Inc., pp. 164–182.

[60] Wagener, G., State, R., and Dulaunoy, A. Malware behaviour analysis. *Journal in Computer Virology 4*, 4 (2008), 279–287.

[61] Web Technology Surveys. Usage statistics and market share of PHP for websites. Online, May 2013. Available from: http://w3techs.com/technologies/details/pl-php/all/all.

[62] Welling, L., and Thomson, L. *PHP and MySQL Web development*. Sams Publishing, 2003.

[63] Willems, C., Holz, T., and Freiling, F. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE 5*, 2 (2007), 32–39.

[64] Wroblewski, G. General method of program code obfuscation, 2002.

[65] Wu, A., Wang, H., and Wilkins, D. Performance comparison of alternative solutions for web-to-database applications–. In *Proceedings of the Southern Conference on Computing* (2000), Citeseer, pp. 26–28.

[66] Wysopal, C., Eng, C., and Shields, T. Static detection of application backdoors. *Datenschutz und Datensicherheit - DuD 34*, 3 (2010), 149–155.

[67] Zaremski, A. M., and Wing, J. M. *Signature matching: A key to reuse*, vol. 18. ACM, 1993.

[68] Zaremski, A. M., and Wing, J. M. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM) 4*, 2 (1995), 146–170.

[69] Zend Technologies. The PHP Company. Online, February 2013. Available from: `http://www.zend.com/en/resources/`.