A SANDBOX-BASED APPROACH TO THE DEOBFUSCATION AND DISSECTION OF PHP-BASED MALWARE

P. Wrench* **and B. Irwin**†

* *Department of Computer Science, Rhodes University, P.O. Box 94, Grahamstown 6140, Email: pete.wrench@gmail.com*

† *Department of Computer Science, Rhodes University, P.O. Box 94, Grahamstown 6140, Email: b.irwin@campus.ru.ac.za*

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. Current malware tools disguise themselves by making 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 paper presents a sandbox-based environment that aims to accurately mimic a vulnerable host and is capable of semi-automatic semantic dissection and syntactic deobfuscation of PHP code.

Key words: Code deobfuscation, Sandboxing, Reverse engineering

1. INTRODUCTION

The overwhelming popularity of PHP as a hosting platform [1] has made it the language of choice for developers of Remote Access Trojans (RATs) and other malicious software [2]. 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. 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 [3].

The proliferation of such malware has become increasingly aggressive in recent years, with some monitoring institutes registering over 70 000 new threats every day [4]. 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 [5,6]. 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 [5, 7]. 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 [8]. 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 [9]. 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. The novel technique of performing deobfuscation based on the identification and reversal of common obfuscation idioms proved highly effective in revealing hidden code. Although not included in the scope of this project, this could act as a stepping stone for the identification of PHP-based malware on production servers.

This paper expands substantially on a work previously published [10] in the proceedings of the 13th Annual Information Security South Africa Conference held in 2014. The two most substantial extensions included the reworking of the Design and Implementation section to provide more detail on how the two major components were configured, and the inclusion of additional testing outcomes in the Results section. The Summary section was also updated to reflect these two changes, and to provide a more in-depth analysis of the new results.

2. PAPER STRUCTURE

The remainder of the paper commences with an overview of the PHP language and its relevant features in Section III, along with an outline of a typical web shell and its common capabilities. The concept of code obfuscation is also introduced, with particular emphasis on how it is typically achieved in PHP. Several alternate deobfuscation techniques are briefly discussed, along with dynamic approaches to code analysis such sandboxing. Section IV details how the system was designed and implemented, outlining the structure and functionality of the two main components (namely the decoder and the sandbox). The results obtained during system testing are presented in Section V. The work concludes in Sections VI and VII, which provide a summary of the results and present ideas for future work and improvement.

3. BACKGROUND AND PREVIOUS WORK

The deobfuscation and dissection of PHP-based malware is a non-trivial task with no well-defined generalised

Based on: "Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware", by Peter Wrench and Barry Irwin which appeared in the Proceedings of Information Security South African (ISSA) 2014, Johannesburg, 13 & 14 August 2014. © 2014 IEEE

solution. Many different techniques and approaches can be found in the literature, each with their own advantages and limitations [8,11,12,13,14]. In an attempt to evaluate these approaches, this section provides an overview of PHP, a description of the structure and capabilities of typical web shells, and an overview of both code obfuscation and dissection techniques.

3.1 PHP Overview

PHP (the recursive acronym for PHP: Hypertext Preprocessor) is a general purpose scripting language that is primarily used for the development and maintenance of dynamic web pages. First conceived in 1994 by Rasmus Lerdof [15], the power and ease of use of PHP has enabled it to become the world's most popular server-side scripting language by numbers. Using PHP, it is possible to transform traditional 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. Since 2001, the reference releases of PHP have been issued and managed by The PHP Group [16].

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 [15, 17]. 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 [18]. 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 [19]. 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 [15]. 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 [15]. Its core libraries provide functionality for string manipulation, database and network connectivity, and file system support [15, 16], giving PHP unparalleled flexibility in terms of deployment and operation.

As an open source language, PHP can be modified to suit the developer. In an effort to ensure stability and uniformity, however, reference implementations of the language are periodically released by The PHP Group [16]. 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 [15, 20]. An abundance of code samples and programming resources exist on the Internet in addition to the standard documentation [21, 22, 23], and many extensions have been created and published by third party developers [24].

Performance and Use:

PHP is most commonly deployed as part of the LAMP (Linux, Apache, MySQL and PHP/Perl/Python) stack [25]. 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 [16]. 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 [15].

When implemented as an interpreted language, studies have found that PHP is noticeably slower than compiled languages such as Java and C [26, 27]. 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) [28, 29, 30]. 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 [31].

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 [32]. 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 [33].

Security:

A study of the United States National Vulnerability Database performed in April 2013 found that approximately 30% of all reported vulnerabilities were related to PHP [34]. 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 [34]. These numbers represent a small percentage of the 2218 total vulnerabilities reported in the same year [34].

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 [34]. 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 [35, 36].

3.2 Web Shells

Remote Access Trojans (or web shells) are small scripts designed to be uploaded onto production servers. 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 [37]. Any server that allows a client to upload files (usually via the HTTP POST method or compromised FTP) is vulnerable to infection by malicious web shells.

In addition to basic remote administration capabilities, most web shells include a host of other features, such as access to the local file system, keystroke logging, registry editing, and packet sniffing capabilities [3].

The motive behind the use of web shells to compromise servers is usually financial gain. Compromised servers can either be monetised directly (by selling access to compromised servers to a third party), or indirectly, by using the servers to facilitate fraudulent activities. Common uses include the establishment of phising sites, piracy servers, malware download sites, and spam sites [3, 37].

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 fully-featured RAT when it is less likely to get noticed.

3.3 Code Obfuscation

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

In their seminal work detailing the taxonomy of obfuscation transforms, Collberg *et al.* [38] 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.

All methods of code obfuscation can be evaluated according to three metrics [38]:

- 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.

Although the number of code obfuscation methods is limited only by the creativity of the obfuscator, the ones listed in the sections below fall neatly into the three categories of layout, data and control obfuscation [39]. Each category boasts methods of varying potency, and a powerful obfuscator should employ methods from each category to achieve a high level of obfuscation.

3.4 Code Obfuscation and PHP

As a interpretedl language with object-oriented features, PHP can be obfuscated using all of the methods detailed above. 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:

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

The string containing the malicious code is encoded in base64 before being compressed. At runtime, the process is reversed. The resulting code is executed through the use of the `eval()` function.

The ubiquitous nature of idioms such as these means that they can be used as a means of detecting obfuscated code. Although seemingly complex, code obfuscated in this manner can easily be neutralised and analysed for potential backdoors. Replacing the `eval()` function with an echo command will display the code instead of executing it, allowing the user to determine whether it is safe to run. This process can be automated using PHP's built-in function overriding mechanisms.

3.5 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.* [40] in their seminal paper "On the (Im)possibility of Obfuscating Programs". Collberg *et al.* [38] 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:

- Pattern matching – the detection and removal of known bogus code segments
- Program slicing – the decomposition of a program into manageable units that can then be evaluated individually
- Statistical analysis – the replacement of expressions that are discovered to always produce the same value with that value
- Partial evaluation – the removal of the static part of the program so as to evaluate just the remaining dynamic expressions

3.6 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 [41]. 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 [42]. Sophisticated anti-malware programs go beyond traditional signature matching techniques, employing advanced methods of detection such as sandboxing and behaviour analysis [43].

3.7 Static Dissection Techniques

Static analysis approaches attempt to examine code without running it [44]. 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 [45].

Signature Matching:

A software signature is a characteristic byte sequence that can be used to uniquely identify a piece of code [45]. 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 [45].

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 [45]. 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 [45]. 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 [45].

3.8 Dynamic Dissection Techniques

Dynamic approaches to analysis extract information about a program's functioning by monitoring it during execution [44]. 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 [44].

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 [46]. In the context of code dissection, API hooking is usually carried out to monitor the behaviour of a potentially malicious program [47]. 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 [47]. 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 [46].

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 [47]. They are also virtually undetectable and thoroughly customisable (only functions relevant to behaviour analysis need be hooked) [47].

Sandboxes and Function Overriding:

A sandbox is a restricted programming environment that is used to separate running programs [48]. 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 [49].

PHP's Runkit extension contains the Runkit Sandbox class, which is capable of executing PHP code in a sandbox environment [50]. 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 [50]. Other options are also provided to further restrict the sandbox environment [50]:

- 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 [50]. 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 [50]. 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.

4. 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 4.2. The design and implementation of the decoder responsible for code normalisation and deobfuscation is presented in Section 4.5 and the next stage of the analytical process, the sandbox capable of dynamic shell analysis, is described in Section 4.6.

4.1 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.

4.2 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 [50]. 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. 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 is used to run an Ubuntu host in a virtual machine environment. 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 4.6), there is 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, it proved more expedient to backup snapshots of 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 SHA-256 [51]. 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 1.

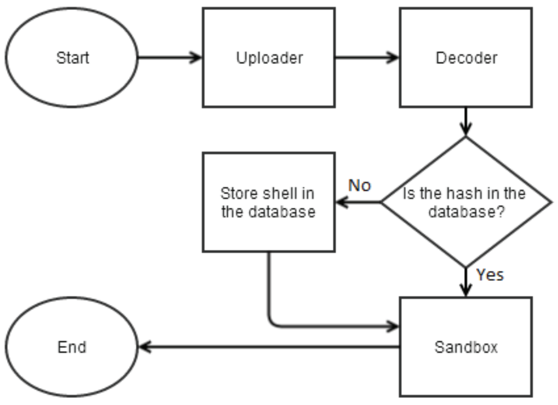

Figure 1: The path of a web shell though the system

4.3 *Web Server Sandbox*

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 [52]. 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 [53, 9]. 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 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 4.6 for full details of how the sandbox was configured).

Choice of Apache:

As the world's most popular HTTP server, Apache is used to power over half of all websites on the Internet [54]. 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 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 4.6.

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.

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 [55]. 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`

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 [55]. 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.

4.4 PHP Configuration

As was the case with Apache, PHP was compiled from source and installed in the wwwroot 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. Listing 1 depicts an obfuscation idiom that includes a call to the aforementioned function, where the string in brackets represents the obfuscated code.

```php
<?php
    eval(gzinflate(base64_decode("4+VKK8n...")));
?>
```

Listing 1: A common obfuscation idiom

`--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 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.

4.5 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 [56].

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 4.5. It makes use of two core supporting functions, `processEvals()` and `processPregReplace()`.

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. Some code normalisation (or pretty printing) is also performed on the output of the deobfuscation process in an attempt to transform it into a more readable form.

```
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 2: Psuedo-code for the `decode()` function

Decode():

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. Furthermore, `eval()` is often used in conjunction with auxiliary string manipulation and compression functions in an attempt to further obfuscate the actual code.

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, 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 2.

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.

ProcessEvals():

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 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. [57]

```
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
    END WHILE
END
```

Listing 3: Psuedo-code for the `processEvals()` function

Listing 3 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 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.

ProcessPregReplace():

The `preg_replace()` function is used to perform a regular expression search and replace in PHP [17]. 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.

Listing 4 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 they were attempting to obfuscate. In much the same way as the `processEvals()` function, string processing techniques are used to extract the

```
BEGIN
    WHILE there is still a preg_replace
        Find the starting position
        Find the end position
        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
    END WHILE
END
```

Listing 4: Psuedo-code for the `processPregReplace()` function

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

Listing 5: Call to the `preg_replace_all()` function

`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.

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 5.

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 1.

Table 1: 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_]+)/'`

```
<?php

...
//Update server
$updateurl = "http://emp3ror.com/N3tshell//update/";
//Sources server
$sourcesurl = "http://emp3ror.com/N3tshell/";
...

?>
```

Listing 6: Extract from c99.php showing a reference to an update server

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 variants derived from the original c99 shell, will attempt to update themselves via an update server if given the opportunity (see Listing 6). Large resources are also often stored on remote servers and accessed at runtime to minimise shell size [43]. A list of these servers could potentially be 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 7.3 for more details).

4.6 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 [42]. 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 discussed later in this section.

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. Finally, shell execution and call logging is performed to complete the process.

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 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.

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, 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.

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.

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 (i.e. prevent activity such as network and file system access).

```
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 7: Pseudo-code for the `redefineFunctions()` function

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.

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 7.

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. 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.

```
//Output handler for the sandbox

//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("/.../temp/output.txt",
            $arr[$i]."\n", FILE_APPEND);
    }
}
return '';
```

Listing 8: Output handler for the Runkit Sandbox object

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 Listing 8.

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 9.

```
//--------------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 9: Example of a function redefinition

5. RESULTS

Throughout the development of the shell analysis system 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. The smaller unit tests were based on sections of real shell code, but were adapted to clearly demonstrate the specific capabilities of the system.

In addition to the adapted unit tests, several active and fully-featured web shells were used as inputs to the system in order to assess its performance in a live production environment. These shells were sourced from a comprehensive web malware collection maintained by Insecurety Research [58], 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.

5.1 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.

5.2 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 10.

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

Listing 10: Single-level `eval()` with a base64-encoded argument

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

Listing 11: Expected decoder output with the script in Listing 10 as input

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 11.

The actual output produced by the decoder component matched the expected output exactly.

5.3 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 12 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.

The decoder was expected to detect all of these functions and apply them to the string, leaving only the decoded string shown in Listing 13. The actual output produced by the decoder component matched the expected output exactly. 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 4.5, all with the same degree of success.

```
<?php
    eval(gzinflate(base64_decode(str_rot13('GIKK
        PhmVSslK+7V2LJg+S3Lrv...')))));
?>
```

Listing 12: Extract of a single-level `eval()` with multiple auxiliary functions

```
<?php
    h5('http://mycompanyeye.com/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 13: Extract of the expected decoder output with the script in Listing 12 as input

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

Listing 14: Single-level `preg_replace()` with explicit string arguments

5.4 Single-level Preg_Replace()

The single-level `preg_replace()` test was very similar to the single-level `eval()` test in Section 5.2, 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 4.5, 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 14.

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 15). The actual output produced by the decoder component matched the expected output exactly.

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()`

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

Listing 15: Expected decoder output with the script in Listing 14 as input

```
<?php
    preg_replace("/.+/e","\x65...",".");
?>
```

Listing 16: Extract of a simple `preg_replace()` statement

```
<?php
    eval(gzinflate(base64_decode('TVCuzIFfy...')));
?>
```

Listing 17: Extract of an `eval()` construct encapsulating the `preg_replace()` statement in Listing 16

construct was still identified and correctly removed from the script, 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. This extension is discussed further in Section VII as a possible addition to the system in the future.

5.5 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 5.3 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 `preg_replace()` is shown in Listing 16, and the `preg_replace()` encapsulated in the `eval()` is shown in Listing 17.

The decoder was expected to remove both layers of obfuscation and replace them with the script from Section 5.3. The actual output showed that the decoder was able to handle the layered obfuscated construct, and is shown in Listing 18.

5.6 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 part of a final and more comprehensive set of tests, a

```
<?php
    h5('http://mycompanyeye.com/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 18: Extract of the actual decoder output with the script in Listing 16 as input18

```
eval(gzinflate(base64_decode('FJ3HcqPsFkUVA...')));
```

Listing 19: Extract of the outermost obfuscation layer

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 19.

The decoder correctly produced the output shown in Listing 20. An analysis of the output found that all `eval()` and `preg_replace()` constructs had been correctly removed from the input script.

5.7 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

```
<?php
    if(!function_exists("getmicrotime"))
    {
        functiongetmicrotime(){list($usec,$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 20: Extract of the decoder output with the script in Listing 19 as input

```
ıp
 echo getlastmod();
 echo "\n";
 echo getlastmod_new();
```

ing 21: Script calling an overridden function and the esponding copied function

```
ibox Ouput:

2402952
2402952
```

ing 22: Sandbox output and results with the script in ing 21 as input

:uted in the sandbox to determine its feasibility as a for code dissection.

Function Copy

first step during function redefinition is the copying ıe original function to a new function so that it can overridden without losing its functionality. The end lt of this process should be the existence of two :tions, one with the original function name that has ı overridden to echo log information when it is called, a new function that contains the logic of the original :tion. This outcome was tested by utilising a script that ; both the overridden function and the copied function, ; shown in Listing 21. The function used in this test ıe `getlastmod()` function, which simply returns a ıber denoting the date of the last modification of the ent file [59].

as expected that both of the function calls would be :essful and would return identical results. The output ıe sandbox is shown in Listing 22.

ın be seen that both functions were run successfully, logic of the original function was preserved, and the :ridden function was able to call the copied function to plete its task before logging the call.

Overriding and Classification of System Functions

:tions in the sandbox are overridden to report rmation about the name of the function and where it called. The type of vulnerability that they represent ıld also be recorded. To test this, a script containing ə functions (one each from the Command Execution, rmation Disclosure, and Code Execution classes of :tions described in Section 4.6) was constructed and t to the sandbox. This script is shown in Listing 23.

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

Listing 23: Script calling three exploitable functions

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 24.

```
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 24: Sandbox results with the script in Listing 23 as input

5.10 *Full Shell Test - 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 25.

In order to get the shell to run, 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 26.

The testing of the sandbox proved to be far more complex and unpredictable. Shells containing malformed CSS and JavaScript failed to run at all, and modifications had to be made to some shells to ensure that certain functions were

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

Listing 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 26: Sandbox results and output with the script in Listing 25 as input

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.

6. SUMMARY

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 5.1 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 5.7).

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 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.

7. FUTURE WORK

7.1 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.

7.2 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.

7.3 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 [60, 61] 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 [62].
- The construction of a taxonomy tracing the evolution of popular web shells such as c99, r57, b374k and barc0de [63] 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.* [64].

REFERENCES

[1] K. Tatroe, *Programming PHP*. O'Reilly & Associates Inc, 2005.

[2] N. Cholakov, "On some drawbacks of the PHP platform," in *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, ser. CompSysTech '08. New York, NY, USA: ACM, 2008, pp. 12:II.7–12:2. [Online]. Available: http://doi.acm.org/10.1145/1500879.1500894

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

[4] E. Kaspersky. (2011, October) Number of the Month: 70K per day. Kaspersky Labs. Accessed on 1 March 2013. [Online]. Available: http://eugene.kaspersky.com/2011/10/28/number-of-the-month-70k-per-day/

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

[6] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *SIGPLAN Notices*, vol. 42, no. 1, pp. 377–388, January 2007. [Online]. Available: http://doi.acm.org/10.1145/1190215.1190270

[7] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Twenty-Third Annual Computer Security Applications Conference*, December 2007, pp. 421–430.

[8] M. Christodorescu and S. Jha, "Testing malware detectors," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 34–44, Jul. 2004. [Online]. Available: http://doi.acm.org/10.1145/1013886.1007518

[9] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in *NDSS*, 2008.

[10] P. Wrench and B. Irwin, "Towards a sandbox for the deobfuscation and dissection of php malware," in *Information Security for South Africa (ISSA), 2014*, Aug 2014, pp. 1–8.

[11] H. C. Kim, D. Inoue, and M. Eto. (2009) Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation. Accessed on 1 March 2013. [Online]. Available: http://jwis2009.nsysu.edu.tw/location/paper/Toward%20Generic%20Unpacking%20Techniques%20for%20Malware%20Analysis%20with%20Quantification%20of%20Code%20Revelation.pdf

[12] E. Laspe. (2008, September) An Automated Approach to the Identification and Removal of Code Obfuscation. Riverside Research Institute. Accessed on 26 May 2013. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-08/Laspe_Raber/BH_US_08_Laspe_Raber_Deobfuscator.pdf

[13] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," in *Computer Security - ESORICS 2008*, ser. Lecture Notes in Computer Science, S. Jajodia and J. Lopez, Eds. Springer Berlin Heidelberg, 2008, vol. 5283, pp. 481–500. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88313-5_31

[14] M. Madou, L. Van Put, and K. De Bosschere, "LOCO: an interactive code (de)obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '06. New York, NY, USA: ACM, 2006, pp. 140–144. [Online]. Available: http://doi.acm.org/10.1145/1111542.1111566

[15] L. Argerich, *Professional PHP4*, ser. Professional Series. Wrox Press, 2002. [Online]. Available: http://books.google.co.za/books?id=gcD3NX92fucC

[16] M. Doyle, *Beginning PHP 5.3*. Wiley, 2011. [Online]. Available: http://books.google.co.za/books?id=1TcK2bIJlZIC

[17] The PHP Group. (2013, May) Basic Syntax. Accessed on 22 May 2013. [Online]. Available: http://php.net/manual/en/language.basic-syntax.php

[18] B. McLaughlin, *PHP & MySQL*, ser. Missing Manual. O'Reilly Media, Incorporated, 2012. [Online]. Available: http://books.google.co.za/books?id=39s5PElSmg8C

[19] The PHP Group. (2013, May) Function Reference. Accessed on 22 May 2013. [Online]. Available: http://www.php.net/manual/en/funcref.php

[20] D. Sklar, *Learning PHP 5*. O'Reilly Media, 2008. [Online]. Available: http://books.google.co.za/books?id=PVvmMRSGzFEC

[21] The Resource Index Online Network. (2005, January) The PHP Resource Index. Accessed on 24 May 2013. [Online]. Available: http://php.resourceindex.com/

[22] The PHP Group. (2013, May) PEAR - PHP Extension and Application Repository. Accessed on 24 May 2013. [Online]. Available: http://pear.php.net/

[23] Zend Technologies. (2013, February) The PHP Company. Accessed on 24 May 2013. [Online]. Available: http://www.zend.com/en/resources/

[24] The PHP Group. (2013, May) PECL. Accessed on 24 May 2013. [Online]. Available: http://pecl.php.net/

[25] J. Bughin, M. Chui, and B. Johnson, "The next step in open innovation," *The McKinsey Quarterly*, vol. 4, no. 6, pp. 1–8, 2008.

[26] A. Wu, H. Wang, and D. Wilkins, "Performance Comparison of Alternative Solutions For Web-To-Database Applications," in *Proceedings of the Southern Conference on Computing*, 2000, pp. 26–28.

[27] L. Titchkosky, M. Arlitt, and C. Williamson, "A performance comparison of dynamic Web technologies," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 3, pp. 2–11, Dec. 2003. [Online]. Available: http://doi.acm.org/10.1145/974036.974037

[28] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content," in *Middleware 2003*, ser. Lecture Notes in Computer Science, M. Endler and D. Schmidt, Eds. Springer Berlin Heidelberg, 2003, vol. 2672, pp. 242–261.

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

[30] S. Trent, M. Tatsubori, T. Suzumura, A. Tozawa, and T. Onodera, "Performance comparison of PHP and JSP as server-side scripting languages," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '08. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 164–182. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496950.1496961

[31] L. Atkinson and Z. Suraski, *Core PHP Programming*, ser. Core series. Prentice Hall Computer, 2004. [Online]. Available: http://books.google.co.za/books?id=e7D-mITABmEC

[32] The PHP Group. (2013, May) Usage Stats for January 2013. Accessed on 21 May 2013. [Online]. Available: http://php.net/usage.php

[33] Web Technology Surveys. (2013, May) Usage statistics and market share of PHP for websites. Accessed on 24 May 2013. [Online]. Available: http://w3techs.com/technologies/details/pl-php/all/all

[34] F. Coelho. (2013, April) PHP-related vulnerabilities on the National Vulnerability Database. Accessed on 25 May 2013. [Online]. Available: http://www.coelho.net/php-cve.html

[35] R. Miller. (2006, January) PHP Apps A Growing Target for Hackers. Accessed on 25 May 2013. [Online]. Available: http://news.netcraft.com/archives/2006/01/31/php_apps_a_growing_target_for_hackers.html

[36] Open Source Matters. (2013, January) What is Joomla? Accessed on 25 May 2013. [Online]. Available: http://www.joomla.org/about-joomla.html

[37] R. Kazanciyan. (2012, December) Old Web Shells, New Tricks. Mandiant. [Online]. Available: 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

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

[39] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *In ACM Conference on Computer and Communications Security*. ACM Press, 2003, pp. 290–299.

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

[41] D. Binkley, "Source Code Analysis: A Road Map," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 104–119. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.27

[42] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *Security & Privacy, IEEE*, vol. 5, no. 2, pp. 32–39, 2007.

[43] G. Wagener, R. State, and A. Dulaunoy, "Malware behaviour analysis," *Journal in Computer Virology*, vol. 4, no. 4, pp. 279–287, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11416-007-0074-9

[44] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith, "Software transformations to improve malware detection," *Journal in Computer Virology*, vol. 3, no. 4, pp. 253–265, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11416-007-0059-8

[45] A. M. Zaremski and J. M. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 2, pp. 146–170, 1995.

[46] H.-M. Sun, Y.-H. Lin, and M.-F. Wu, "API Monitoring System for Defeating Worms and Exploits in MS-Windows System," in *Information Security and Privacy*, ser. Lecture Notes in Computer Science, L. Batten and R. Safavi-Naini, Eds. Springer Berlin Heidelberg, 2006, vol. 4058, pp. 159–170. [Online]. Available: http://dx.doi.org/10.1007/11780656_14

[47] J. Berdajs and Z. Bosnic, "Extending applications using an advanced approach to DLL injection and API hooking," *Software: Practice and Experience*, vol. 40, no. 7, pp. 567–584, 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.973

[48] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the Wily Hacker," in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96. Berkeley, CA, USA: USENIX Association, 1996, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267569.1267570

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

[50] The PHP Group. (2013, May) Runkit Sandbox. Accessed on 27 May 2013. [Online]. Available: http://php.net/manual/en/runkit.sandbox.php

[51] W. Dai. (2009, March) Crypto++ 5.6.0 Benchmarks. Accessed on 26 October 2013. [Online]. Available: http://www.cryptopp.com/benchmarks.html

[52] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40–52.

[53] K. Borders, A. Prakash, and M. Zielinski, "Spector: automatically analyzing shell code," in *Twenty-Third Annual Computer Security Applications Conference*, 2007, pp. 501–514.

[54] NetCraft. (2013, June) June 2013 Web Server Survey. Accessed on 9 October 2013. [Online]. Available: http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html

[55] The Apache Software Foundation. (2013, February) Dynamic Shared Object (DSO) Support. Accessed on 10 October 2013. [Online]. Available: http://httpd.apache.org/docs/2.2/dso.html

[56] M. Preda and R. Giacobazzi, "Semantic-Based Code Obfuscation by Abstract Interpretation," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, L. Caires, G. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Springer Berlin Heidelberg, 2005, vol. 3580, pp. 1325–1336. [Online]. Available: http://dx.doi.org/10.1007/11523468_107

[57] The PHP Group. (2013, May) Eval. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/function.eval.php

[58] Insecurety Research. (2013, June) Web Malware Collection. Accessed on 26 October 2013. [Online]. Available: http://insecurety.net/?p=96

[59] The PHP Group. (2013, May) Get Last Mod. Accessed on 24 October 2013. [Online]. Available: http://php.net/manual/en/function.getlastmod.php

[60] A. Walenstein and A. Lakhotia, "The Software Similarity Problem in Malware Analysis," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/964

[61] A. Gupta, P. Kuppili, A. Akella, and P. Barford, "An empirical study of malware evolution," in *Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International*, 2009, pp. 1–10.

[62] J. Kornblum. (2013, July) Context Triggered Piecewise Hashes. Accessed on 26 October 2013. [Online]. Available: http://ssdeep.sourceforge.net/

[63] T. Moore and R. Clayton, "Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, R. Dingledine and P. Golle, Eds. Springer Berlin Heidelberg, 2009, vol. 5628, pp. 256–272. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03549-4_16

[64] J. Li, J. Xu, M. Xu, H. Zhao, and N. Zheng, "Malware obfuscation measuring via evolutionary similarity," in *First International Conference on Future Information Networks*, 2009, pp. 197–200.