Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware

Peter M. Wrench
Department of Computer Science
Rhodes University, P.O. Box 94, Grahamstown 6140
Email: g10w1139@campus.ru.ac.za

Barry V. W. Irwin
Department of Computer Science
Rhodes University, P.O. Box 94, Grahamstown 6140
Email: b.irwin@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.**

***Index Terms*—Code deobfuscation, Sandboxing, Reverse engineering**

I. 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.

II. Paper Structure

This paper begins 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 static 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. Finally, Section VI presents ideas for future work and improvement.

III. Background and Previous Work

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 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.

A. 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 [10], 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 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 [11].

978-1-4799-3383-9/14/$31.00 ©2014 IEEE

B. 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 [12]. 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 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.

C. 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.* [13] 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 [13]:

- 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.

D. Code Obfuscation and PHP

As a procedural 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.

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.

E. 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.* [14] in their seminal paper "On the (Im)possibility of Obfuscating Programs". Collberg *et al.* [13] 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

F. 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 [15]. 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. Sophisticated anti-malware programs go beyond traditional signature matching techniques, employing advanced methods of detection such as sandboxing and behaviour analysis [16].

All code dissection techniques can be classified as being either static or dynamic in nature [15]. Static analysis approaches attempt to examine code without running it. 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. Dynamic approaches to analysis extract information about a program's functioning by monitoring it during execution. 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.

IV. 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 IV-B. The design and implementation of the decoder responsible for code normalisation and deobfuscation is presented in Section IV-C and the next stage of the analytical process, the sandbox capable of dynamic shell analysis, is described in Section IV-D.

A. 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.

B. 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 [17]. 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 IV-D2), 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 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 [18]. 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.

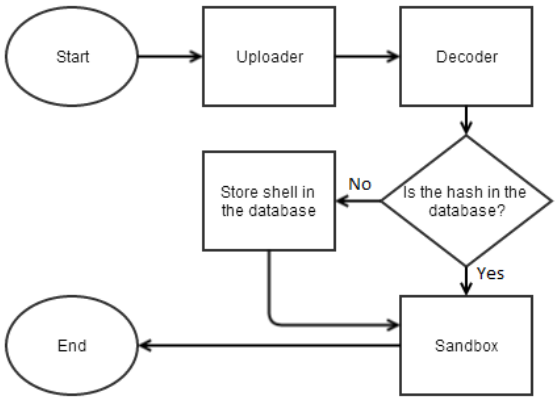

Figure 1. The path of a web shell though the system

C. 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 [19].

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 IV-C1. It makes use of two core supporting functions, `processEvals()` and `processPregReplace()`, the details of which are provided in Sections IV-C2 and IV-C3 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. 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.

1) 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 1.

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.

```
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 1. Psuedo-code for the `decode()` function

```
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 2. Psuedo-code for the `processEvals()` function

2) 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. [20]

Listing 2 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

```
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 3. Psuedo-code for the `processPregReplace()` function

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.

3) ProcessPregReplace(): The `preg_replace()` function is used to perform a regular expression search and replace in PHP [21]. 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 3 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 `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.

D. 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 [22]. 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 IV-D2.

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 IV-D3. Finally, the shell execution and call logging that is performed after execution is detailed in Section IV-D4.

1) 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.

2) 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 such activity as network and file system access). 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.

3) 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.

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.

4) 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.

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.

V. 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.

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.

```
<?php
    echo "Hello";
    eval(base64_decode("ZWNobyAiRJ5ZSI7"));
?>
```

Listing 4. Single-level `eval()` with a base64-encoded argument

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

Listing 5. Expected decoder output with the script in Listing 4 as input

A. 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.

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.

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 5. The decoder output matched this expected output exactly.

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

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

```
<?php
    exec("whoami");
    echo getlastmod();
?>
```

Listing 6. Script calling three exploitable functions

```
Sandbox Results:

Line 1 - Potentially malicious call to:
Command_Execution function "exec"

Line 2 - Potentially malicious call to:
Information_Disclosure function "getlastmod"
```

Listing 7. Sandbox results with the script in Listing 6 as input

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.

B. 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.

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 IV-D3) was constructed and input to the sandbox. This script is shown in Listing 6.

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 7.

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 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.

VI. 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 V-A 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 V-B).

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.

VII. Future Work

A. 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.

B. 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.

C. 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 [23], [24] 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 [25].
- The construction of a taxonomy tracing the evolution of popular web shells such as c99, r57, b374k and barc0de [26] 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.* [27].

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

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

[12] 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

[13] 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.

[14] 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.

[15] 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

[16] 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

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

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

[19] 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

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

[21] ——. (2013, May) Preg Replace. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/function.preg-replace.php

[22] 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.

[23] 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

[24] 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.

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

[26] 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

[27] 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.