BOREL Enzo

Free University of Brussels

PROJ-H-415

PHP Webshells detector

supervised by M. Thibault Debatty

May 2018

Statement and Goal

A web shell is a script (generally written in PHP), uploaded by a hacker on a server, giving him/her the ability to take a complete control over the server. Actual detectors, as PHP Shell Detector, use signatures in order to detect web shells, and are therefore efficient only against well-known shells. The goal of this project is to develop a web shell detector based on a fuzzy logic mechanism, in order to detect new threats too.

An unrestricted file upload could give an attacker the ability to put on a server his/her own malicious files in order to compromise it. Even if he/she is not supposed to have high privileges, having the ability to execute system commands could be really harmful: defacing, source code leak, or even worse, privileges escalation. However, checking the dangerouness of a file only at the uploading phase could be insufficient (e.g. a two-step upload : encoded payload + decoder). It's then better to be able to scan all files at any time. Because of PHP flexibility, new web shells appear regularly, and well-known ones can be easily obfuscated, that's the reason why we propose here a new detection tool based on several different mechanisms, since each mechanism could not be always reliable nor accurate by itself.

Structure of the system

Our system is based on a set on modules, each one analyzing a different aspect of a given file. Each module computes an indicator of harmfulness between 0 (harmless) and 1 (surely harmful), and a final decision is taken based on these indicators. Except for modules based on signatures, indicators are computed based on reference values, obtained by testing the system on Wordpress files. We computed all scores, selected only meaningful values (without zeros, and by removing the 10% minimal and maximal values), and kept extrema. Then, if the score of the analyzed file is out of range, its score becomes 0 or 1.

1. Entropy

The intuition behind the notion of entropy could be explained by viewing the information as the unexpectedness of a signal [1]. It makes sense since an unexpected signal could be meaningful, whereas an expected one could be overlooked among the large number of similar signals. To compute it, the first thing to do is to find the relative frequency of each character (f_i), and the entropy of the file is then equal to

$$-\sum_{i=0}^{n} f_i \times \log_2(f_i)$$

Since taking a logarithm in this range of values leads to a negative value, the result is inverted. By computing the entropy of Wordpress files, we set the two extrema to 4.027 and 5.518.

2. Dangerous routines

PHP is a powerful language, which could be very dangerous in case of misuse. The language offers a bunch of dangerous routines, and the detector tries to find them. Three tests are performed:

- external program execution (`exec`, `passthru`, `system` , backtick operator, etc.)

- anonymous routines, passed as callback parameters
- variable functions, such as :
  ```
  $var = "phpinfo";
  $var();
  ```

3. Obfuscation

To bypass a basic detection, an attacker will often try to obfuscate the code, whereas a harmless code doesn't need it. This detector will try to determine if a code has been obfuscated or not. Three factors are taken into account: non-ASCII characters, usage of decoding routines (`base64_decode`, `gzuncompress`, etc.), and the longest string size.

4. Signatures

Detection based on signatures is a well-known technique, quite useful, but only regarding known malicious files. A signature is based on a portion of the file (often the malicious part), which can be used to identify it, or a similar one. The database we used comes from the project PHP Webshell Detector [2]. However, if the web shell has been modified, or is unknown, detection based on signatures will probably fail. Because PHP is a very flexible language, modify or create a new web shell, and bypass detection based on regular expressions or signatures is not so difficult, that the reason why this detection mechanism is not sufficient by itself.

5. Fuzzy hashing

This detector is based the paper "Identifying almost identical files using context-triggered piecewise hashing" written by Jesse Kornblum [3]. As the title suggests, the idea behind fuzzy hashing is to compute similar values for similar files. Indeed, two similar files have common sequences of bits, and the longer they are, the closer the hashes will be.

First, the analyzer applies the Spamsum algorithm on the file to scan, and then computes the Levenshtein distance between the hash just obtained and each hash in the database. This distance is equal to the minimal number of single-digit manipulations (insertion, deletion, or replacement) required to transform a string to another. If the distance is small, it probably means that the scanned file is an altered version of a well-known web shell. All hashes in the database were computed by removing all useless white spaces and comments, and the detector does the same for each scanned file. Indeed, it ensures that useless information will not affect the distance computation.

Usage

The detector can be integrated in a project as a Composer library [4], or used as a standalone tool.

1. The dependency can be added to the project thanks to the following command

```
$ composer require rucd/webshell-detector
```

And in the PHP code:

```
require_once "vendor/autoload.php";
use RUCD\WebshellDetector\Detector;
```

```
$detector = new Detector();
echo $detector->analyzeFile("file_to_scan.php");
```

2. The executable PHAR can be downloaded from the Releases pages [5]. The tool proposes two commands, the first one allows to scan a directory (and sub-directories)

```
$ webshell-detector.phar analyze:directory <dir>
```

whereas the second one is for single files

```
$ webshell-detector.phar analyze:file <file>
```

It's also possible to define the sensitivity to harmfulness by setting a different threshold between 0 and 1, the default value being 0.4

```
$ webshell-detector.phar analyze:directory -t <threshold> <dir>
```

Project continuation

To continue the project, one should first clone the repository from Github and install dependencies with `composer`:

```
$ git clone https://github.com/RUCD/webshell-detector.git
$ composer install
```

The code is in the folder `src`, where each file named `[...]Analyzer.php` implements the interface `Analyzer`. The interface defines the routine `analyze`, taking as parameter the string to analyze, and returning a value between 0 and 1. The `Detector` loads all modules, and launches the main analysis. The command line tool is finally handled by `Main`, and the two `Command` files. The folder `res` contains three text files, used by signature analyzers. Each analyzer has its own test file in the folder `tests`. They can be manually performed with the command

```
$ ./vendor/bin/phpunit <file>
```

which has its configuration in `phpunit.xml`. Once a `push` is done, all tests are automatically performed using Travis-CI, configured in `.travis.yml`.

Finally, regarding the command line tool, it uses the library Symfony Console [6]. The executable PHAR can then be built with

```
$ ./vendor/bin/box build
```

and would be located in `bin`.

[1] http://www.onlamp.com/pub/a/php/2005/01/06/entropy.html
[2] http://www.emposha.com/security/php-shell-detector-web-shell-detection-tool.html
[3] http://dfrws.org/sites/default/files/session-files/paper-identifying_almost_identical_files_using_context_triggered_piecewise_hashing.pdf
[4] https://packagist.org/packages/rucd/webshell-detector
[5] https://github.com/RUCD/webshell-detector/releases
[6] https://symfony.com/doc/3.4/components/console.html