User Guide
PHP Vulnerability Suite

Authored by

Guillaume Pighi
Xavier Marchal
Jonathan Retterer

Summary

I. Introduction 3

II. Our aim 3

III. How to use our application 3

III.A. Generation of samples 3

III.B. Messages 4

III.C. Complexity generation 4

III.D. Output files structure 5

III.E. The manifest 6

IV. Adding new samples or flaws to the generator 7

IV.A. XML files structure 7

IV.B. Creating a new flaw's generator 9

References 10

I. Introduction

This tool was developed by a project team of Telecom Nancy [1] composed of Guillaume Pighi, Jonathan Retterer and Xavier Marchal with the help of Bertrand Stivalet and Aurélien Delaitre from the SAMATE [2] team of NIST [3].

This software is based on searches we've done on the most common flaws. In order to understand how they are working and how to secure them, we studied the OWASP (Open Web Application Security Project) top 10 [4] and CWE (Common Weakness Enumeration) [5].

The OWASP top 10 is a really popular reference about web application security. We can find in this document the 10 most common and dangerous flaws of web applications. For each vulnerability the dangers are explained, how an attacker can exploit it and how we can secure it. But this document is not standardized and there is not exhaustive information about each derivative of each flaw. That's why we also used the CWE documents in order to classify properly the samples generated and to get more information about each flaw.

II. Our aim

With this tool we want to permit the generation of a reference database for the most common flaws. With this one everybody should be able to see which are the good and bad use when you want to develop a web application. But this database can have an other user for companies which are developing static analyzer : with all the samples generated, they will be able to cover a large horizon of possible implementation and securement of a flaw.

III. How to use our application

III.A. Generation of samples

In order to use our tool, you need to install python 3.4 or later. This tool is only usable with a CLI (Command Line Interface), there is no graphical interface. First of all, we show you an example of command in order to use it (you have to be in the folder where the generator is stored) :

```
Computer:Desktop User$ cd generator
Computer:generator User$ python core.py -f Injection
```

As you can see above, we have used our generator in order to generate Injections Flaws from the OWASP top 10. In order to give you an overview of all the features offered by our tool, you can find below all the options you can use :

- -f : flaw to generate (flaw1,flaw2,…)

XSS	: Cross-site Scripting
IDOR	: Insecure Direct Object Reference
Injection	: Injection (SQL, LDAP, XPATH, OS Command, Code)
URF	: URL Redirects and Forwards

SM	: Security Misconfiguration
SDE	: Sensitive Data Exposure

- -c : generate samples by cwe (cwe1,cwe2,…)

78	Command OS Injection
79	XSS
89	SQL Injection
90	LDAP Injection
91	XPath Injection
95	Code Injection
98	File Injection
209	Information Exposure Through an Error Message
311	Missing Encryption of Sensitive Data
327	Use of a Broken or Risky Cryptographic Algorithm
601	URL Redirection to Untrusted Site
862	Insecure Direct Object References

- -h : show the usage of the tool

III.B. Messages

If a mistake is done in the syntax, a message given the syntax for using our application will be displayed.

When the tool is generating samples, a message containing the number of samples generated, the percentage of secured / unsecured files will be displayed for each vulnerability of the OWASP top 10.

III.C. Complexity generation

In order to generate all the samples, our generator base itself on three kind of XML files :

- Input : The input is the method in order to collect the data.
- Sanitize : The sanitize is used to clean the input from malicious content that can be given by the user, it can be suitable or not for request that we want to do.
- Construction : The construction is how we are constructing the request, for example, we can do a simple concatenation or use the interpretation of the variable.

Our generator base his generation on an XML file (output.xml) explaining how he has to construct the files with the samples from the three XML files. Here is the classic use of this one :

```xml
<?xml version="1.0"?>
<program>
    <input/>
    <sanitize/>
    <construction/>
</program>
```

With this configuration, each sample is constructed with the concatenation of 1 input, 1 sanitize and 1 construction. But we can do a lot of other things with this file : we surround each sample with a decorator which will describe in what we will include the sample (it can be used for sanitize, input and construction). If we want to insert the sanitize in a class, we can do it by modifying output.xml to :

```
<?xml version="1.0"?>
<program>
    <input/>
    <complexity type="class">
         <sanitize/>
    </complexity>
    <construction/>
</program>
```

And we can construct it recursively, for example we can write that all sanitize have to be in a class which will be in a separate file. You can find below the output.xml file that illustrate this example :

```
<?xml version="1.0"?>
<program>
    <input/>
    <complexity type="file">
        <complexity type="class">
            <sanitize/>
        </complexity>
    </complexity>
    <construction/>
</program>
```

These examples were showing you some complexity that are supported by our generator, but there other possibilities, here is the complete list of the complexities which are implemented on our generator :

• class :	<complexity type="class"> </complexity>
• for :	<complexity type="loop" kind="for"> </complexity>
• while :	<complexity type="loop" kind="while"> </complexity>
• if :	<complexity type="if"> </complexity>
• file :	<complexity type="file"> </complexity>
• function :	<complexity type="class"> </complexity>

III.D. Output files structure

Now you have generated all the samples that you wanted with our generator. You will find them in a folder at the same level than the folder « generator », named « generation » followed by the date and hour of the generation.

In this folder there will be a list of folder named by the name of the OWASP top 10 flaws generated. In each of these folder, there is a list of the CWE that are derived from the OWASP flaw, for example for the Injection folder, there is the CWE 78, 89, 90, 91, 95 and 98.

Now once you're in a CWE folder, there will be two folders : safe and unsafe which will contain respectively safe files and unsafe files. You can find find below a scheme that will sum up the folder structure.

In order to target easily the different files that you can search, all the files generates have a unique name that reflect the content of the program. Each file name follow this format :

CWE_XX_[(Input1)(Input2)…]_[(Sanitize1)(Sanitize2)…]_[(Construction1)(Construction2)…]

Each part bracketed is the definition of a kind of sample, for example as you can see above, the first part bracketed is for the definition of the Input.

For each sample, there is multiple level for defining exactly which kind os sample it is. Each level is declared in parenthesis, for example an input can be defined in this way :

[(array)(GET)]

This example means that the input is taken in an array where we did before a $_GET in order to get the data.

For letting you understanding completely how it is working, we will take a complete example, the name of the file is :

```
CWE_78_[(GET)]_[(func_preg_match)(letters_numbers)]_[(cat)(concatenation_simple_quote)].php
```

So with that example we can find a lot of information about how the file is constructed. First of all we can see that this sample is made for the CWE 78. After we can find information about :

- Input : The input is simply taken with a **$_GET**.
- Sanitize : In order to sanitize the data coming from the input, the function **preg_match** has been used and this function is allowing **only letters and numbers**.
- Construction : Now, with the sanitized data, we are constructing a request by **concatenating** the data with the request in order to make a **cat** OS command.

III.E. The manifest

In order to help to know easily what is produced by the generator and where it is stored, a manifest is done along with the generation of the files. You will find them in each OWASP folder with the name Manifest.xml. In order to show you how the data are presented in this file, you see below an sample of this file for the injection generation :

```
<testcase>
    <meta-data>
     <author>H. B¸hler, D. Lucas, F. Nollet, A. Reszetko</author>
     <date>09/03/15</date>
     <input>file : /tmp/tainted.txt</input>
    </meta-data>
    <file path="CWE_98/unsafe/CWE_98_[(backticks)]_[(func_preg_match)
(no_filtering)]_[(include_file_name)(concatenation_simple_quote)].php"
language="PHP">
     <flaw line="62" name ="Injection"/>
    </file>
</testcase>
```

<testcase> : description of a sample
<metadata> : contains informartion about the sample
<file> : contains information about the path of the sample and the language. If the sample is unsafe, a tag <flaw> is added and give the line of the flaw.

IV. Adding new samples or flaws to the generator

IV.A. XML files structure

As it is mentioned before, in order to generate all the samples, we are using three XML files that are storing all the parts of the samples. In order to don't loose you, you can find below a short recall about what are these XML files and how they are used.

- Input : The input is the method in order to collect the data.
- Sanitize : The sanitize is used to clean the input from malicious content that can be given by the user, it can be suitable or not for request that we want to do.
- Construction : The construction is how we are constructing the request, for example, we can do a simple concatenation or use the interpretation of the variable.

Now we will describe you how the data are stored in these files and with that, you will be able to add easily new samples to them. We will start with the easier : Input.xml, this one is used for most of the flaws implemented because most of the them need an entry from an external content in order to make the request.

```
<sample>
    <path>
        <dir>GET</dir>
    </path>
    <comment>Input get</comment>
    <code>$input=$_GET['un']</code>
    <inputType>get : un</inputType>
</sample>
```

<path> : Information permitting to create the name of the files, each <dir> is a sublevel.
<comment> : Insert comments on samples concerning this sample.
<code> : Contains the code to input.
<inputType> : Explain which kind of input it is and from where it comes.

The next XML file that we will study is Sanitize.xml. So let's start by looking at an example of this file :

```
<sample>
    <flaws>
      <flaw>CWE_89_Injection</flaw>
      <flaw>CWE_91_Injection</flaw>
    </flaws>
    <path>
      <dir>CAST</dir>
      <dir>cast_int</dir>
    </path>
    <comment>sanitize : cast into int</comment>
    <codes>
      <code>$tainted = (int) $tainted ;</code>
    </codes>
    <constraints>
      <constraint flawType = "CWE_89_Injection" type = "int" field = ""/>
      <constraint flawType = "CWE_91_Injection" type = "int" field = ""/>
    </constraints>
    <safeties>
      <safety flawType = "CWE_89_Injection" safe = "1" needQuote = "0" />
      <safety flawType = "CWE_91_Injection" safe = "1" needQuote = "0" />
    </safeties>
</sample>
```

First of all, we will give a short explanation about the meaning of each new tag, the tags which have the same name than tags for input have the meaning :

<flaws> : Show the flaws for which this sample is used.
<codes> : Contains one or multiple tags <code> which contains the code to insert.
<safety> : Show information about the safety of the satinize for a particular flaw. In this case, safe="1" means that it is safe for a flaw, 0 for unsafe. There is an additional field called needQuote which means that if the data is surrounded with quotes in the request, it can be considered as safe.
<constraint> : Permit to the generator to construct PHP programs that are valid. In order to that, the attributs of this tag are used, for example their is a type checking with what the construction sample is waiting.
A few more explanation about the needQuote attribute, and for that we will take the following PHP line :

```
exec("cat ".$test);
```

This line is done in order to get the result of the OS command « cat » for the value of $test. A normal user will simply tap the name of the file that he wants to see. But an attacker can send : « test;ls », and then the attacker will collect also the result of ls, but it is not restricted to the only use of ls, he can tap all the commands that he wants.

Now we suppose that the function addslashes() is used before on data that we collected. This function suppress all « \ », « ' » , « " » in the data given in parameter. So if we surround the variable with quotes, the user user will not be able to finish the string, so the request will be protected because all the data will be considered as one string. Here is the example illustrating this example :

```
$test=addslashes($test);
exec("cat ".$test."");
```

So if the attacker sends « test;ls » there will be only an error message saying that the file « test;ls » doesn't exist.

The last part is about construction.xml, in order to understand easily what kind of data it stores, we show you an example :

```
<sample>
    <flaws>
      <flaw>CWE_89_Injection</flaw>
    </flaws>
    <path>
      <dir>select_from_where</dir>
      <dir>concatenation</dir>
    </path>
    <comment>construction : concatenation</comment>
    <codes>
      <code>$query = "SELECT * FROM student where id=". $tainted . "";</code>
    </codes>
    <safeties>
      <safety flawType="CWE_89_Injection" quote = "0" safe = "0" />
    </safeties>
    <constraints>
      <constraint flawType="CWE_89_Injection" type = "int" field = "WHERE"/>
    </constraints>
</sample>
```

In comparison with the sanitize.xml file's structure, there isn't a lot new things, the tags with the same names have the same meaning. There is just few new things: there is a new attribute for the tag safety which is quote that only means if the data used in order to construct the request is surrounded by quotes.

The only only thing that significantly change is that for XSS flaws, the first <code> will be pointing a file that will be the beginning of the sample and the second will be pointing the file that will end the sample.

As you see there can be a different meaning for each flaw and if you have to implement new kind of flaws, you can change the meaning of few tags.

IV.B. Creating a new flaw's generator

In the folder named « Flaws_generator », you will find the existing generators. If you want to create a new one which interprets differently the data inside the XML files, you have to :

- Inspire yourself from the working of our generators and change what is different with the one you want. Dont't forget to specify the CWE in the function getType() or your generator.
- Add your generator to the file Generator_factory.py.
- Add the name to the variable « flaws » in the python code core.py.
- Add a new condition to the same file when the parameters is equal to the name of your flaw.
- Modify the usage function to show that we can generate the new flaw.

References

[1] http://www.telecomnancy.eu

[2] http://samate.nist.gov/Main_Page.html

[3] http://www.nist.gov

[4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] http://cwe.mitre.org