Large Scale Generation of Complex and Faulty PHP Test Cases

Bertrand Stivalet, Elizabeth Fong

Software and Systems Divison,
National Institute of Standards and Technology
Gaithersburg, MD, 20899, USA
{bertrand.stivalet, efong}@nist.gov

***Abstract*—Developing good test cases is an intellectually demanding and critical task, and it has a strong impact on the effectiveness and efficiency of the whole testing process. This paper presents an automated generator of test cases, which are designed to evaluate source code security analyzers. The generator produces PHP: Hypertext Preprocessor (PHP) programs with most common vulnerabilities embedded in various code complexities. It also produces programs without vulnerabilities to test for false positives. The generator is modular and extensible. We describe its internal design and how it works. The generated PHP test cases were added to the Software Assurance Reference Dataset (SARD) and will be used to assess the effectiveness of static analyzers. We conclude with the current state of the tool, its benefits and future work.**

Keywords—automated testing; Common Weakness Enumeration; security testing; test cases generation; software vulnerability

DISCLAMER

Certain instruments, software, or materials are identified in this paper to specify the exposition adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the instruments, software or materials are necessarily the best available for the purpose. NIST assumes no responsibility whatsoever for the use of this test suite by other parties, and makes no guaranties, expressed or implied, about its quality, reliability, or any other characteristics.

I. INTRODUCTION

With the rise of Web 2.0, the use of PHP: Hypertext Preprocessor (PHP) has exploded. Today it is the most-used programming language in web applications. People without technical background can use automated tools or existing templates to build dynamic websites. PHP evolved rapidly; however its security features did not grow as fast. In 2014, 13.7 % of all vulnerabilities listed by the National Vulnerability Database were related to PHP [1]. Nowadays, security testing has become a fundamental part of a software development life cycle. It is (or should be) included at every stage of the application development process. By definition, software testing is the investigation conducted to detect and trace abnormal software behavior caused by bugs and can be performed by several means. Dynamic Application Security Testing (DAST) [2] is one of the most documented and popular techniques to detect weaknesses, but it can only be performed when the application is at a later stage of development. To prevent vulnerabilities from being implemented at the earliest stages, Static Application Security Testing (SAST) [3] can analyze the source code as soon as it is written. It is a mature practice when developing software in languages like C/C++ and Java, but there is a lack of testing material for PHP. This paper documents the development of the first publicly available PHP Test Suite generator. The PHP synthetic test cases generator produces thousands of unique test cases expressing common flaws with different complexities to test the effectiveness and capabilities of PHP static analyzers.

A. Background

Software Assurance Metrics and Tool Evaluation (SAMATE) is a project at the National Institute of Standards and Technology (NIST) with the goal of improving software assurance by developing materials, specifications, and methods to test tools and techniques and measure their effectiveness. In order to test the effectiveness of tools, we developed a repository of example code with known weaknesses, the Software Assurance Reference Dataset (SARD) [4]. Everyone can access the content of this database and download test cases to evaluate static analyzers.

This repository hosts nearly 155 000 test cases in different programming languages, but not all the languages are well represented. For instance, only 15 vulnerable PHP test cases were available before our PHP test suite was added. We needed more testing materials to properly test PHP analyzers. To achieve this objective efficiently, we developed a generator capable of producing thousands of vulnerable and non-vulnerable synthetic test cases.

B. Test Case Generation

Software systems have become more and more complex, with components developed by different vendors, using different techniques in different programming languages and even running on different platforms. Static analyzers need to be very capable to handle these. To test static analyzers running on such diverse source code requires thousands of test cases with many variants for each known weakness. They can be produced manually, but it is very time consuming.

A great amount of research effort in past decades has been spent on automatic test case generation [5]. Generally speaking, test cases are generated based upon information including program structure and/or source code, the software flaws using both valid and faulty data, information about the input/output data space, and information dynamically obtained from program execution. Thus, building a test case generator that can produce large-scale test cases required creativity.

C. Similar Work

Kratkiewicz [6] listed a buffer overflow taxonomy designed for detecting buffer overflow in C code. This project provides a test suite generator and other tools that can be used to evaluate source code analysis tools with respect to buffer overflow detection.

Two test suites were generated: Juliet C/C++ and Java, and Securely Taking on Software of Uncertain Provenance (STONESOUP). See Table I for their characteristics.

The Juliet test suite generator was developed by the National Security Agency (NSA) Center for Assured Software [7]. The test suite is a collection of C/C++ and Java programs with known flaws comprising 57 099 test cases in C/C++ and 23 957 test cases in Java. Juliet covers 181 different kinds of flaws, mapped to the Common Weakness Enumeration (CWE) [8]. The test cases are synthetic; that is, they were created as examples with well-characterized weaknesses. Each case targets only one flaw.

Another similar work is STONESOUP by the Intelligence Advanced Research Projects Activity (IARPA) [9]. This test case generation process starts with an open source program, injects a weakness into that program, manually inspects that the weakness was implemented correctly, and creates variants (snippets) off this base program by changing where the weakness was injected. Input/output pairs are also created as part of the test case generation process.

TABLE I. CHARACTERISTICS OF EXISTING TEST SUITES

	Juliet v1.2		**STONESOUP**	
	C/C++	Java	C/C++	Java
CWE	118	112	56	50
Test cases	57 099	23 957	4582	3188
	Total: **81 056**		Total: **7770**	

D. Users

The targeted audience of the PHP Vulnerability Test suite can be distinguished into three categories. The first is the static analysis tools makers. They can use the PHP test suite against their tools in order to test and improve their products. Tool users can run the trial version of static analyzers on the test suite to test and compare products before buying one. Also, security researchers conducting studies on new static analysis approaches could test the behavior of their tools on the test suite and perform comparisons with existing tools.

II. OVERVIEW OF THE PHP VULNERABILITY TEST SUITE GENERATOR

The diversity of C/C++ and Java synthetic test cases that are available shows the maturity of the static analysis discipline for those two programming languages. But the situation for other languages like PHP is different. For PHP web application testing, people would concentrate their tests on the dynamic aspect. [10] shows that static and dynamic analysis are not overlapping, but are complementary. Static analysis allows testers to find weaknesses at the earlier stages of the development process. In order to promote the use of PHP static analyzers, we need to make testing materials available to let people experiment with and test those static analyzers. Until now, no PHP test suite contained a wide range of weakness variants embedded in various code complexities. All of these reasons convinced us to create the PHP Vulnerability Test Suite generator.

Bertrand Stivalet and Aurelien Delaitre, from the NIST SAMATE Team, managed and designed the architecture of the project, which has been implemented by students from TELECOM Nancy [11] composed of Herve Buhler, David Lucas, Xavier Marchal, Fabien Nollet, Guillaume Pighi, Axel Reszetko and Jonathan Retterer.

A. Objectives and Challenges

Being able to test static analysis products and compare them should increase the competitiveness of these products and improve both the state-of-the-art and the state-of-the-practice of SAST in PHP. To complete this task as precisely as possible, we need to provide those tools with the ideal data set. The main objective was to create a generic algorithm which can produce test suites for different programming languages without editing the core of the program. To address the characteristics of each programming language, we needed the generation process to be reusable and modular, so we can easily add custom rules, classes of weaknesses and complexities. From a user's perspective, the tool should be customizable and easy to run without prior knowledge of the project.

To measure the value of static analyzers, we need to define which attributes and characteristics should be considered, then define the corresponding metrics. Some prevalent measures for tool effectiveness are described in [12], e.g. precision, recall, discrimination, overlap and coverage. Table II defines some of the metrics used to assess software assurance tools.

TABLE II. METRICS TO TEST SOFTWARE ASSURANCE TOOLS

Metrics	**Definition**
Precision	Proportion of correct warnings produced by a tool
Recall	Proportion of actual defects found in the code
Coverage	Classes of vulnerabilities found by tools
Discrimination	Ability to report a vulnerability when it is present in code and keep quiet when there is no vulnerability

B. *Design of Test Cases*

Our first concern is to cover the most common vulnerabilities. For that, we decided to follow the 2013 Open Web Application Security Project (OWASP) Top 10 [13]. We want our test cases to be representative of real software and statistically significant, meaning a large enough test suite containing many occurrences of the same defect types with wide weakness-type diversity. We also need test cases where all weaknesses are known in order to facilitate the automation of the testing process. To find out whether a static analyzer is reporting false positives (or noise), we need test cases with correct code. To find out whether an analyzer misses defects or flaws, we need test cases with flaws. The program should generate test cases in pairs with valid code and vulnerable code. Most importantly, test cases should be like real code.

A test case generator needs also to produce various complexities, which are time-consuming for a human to implement systematically. Based upon the characteristics of the test cases, i.e., statistical significance and ground truth, users can successfully measure different aspects of static analyzers and compare them methodically.

C. *Tool Usage*

The PHP vulnerability test suite generator is written in Python 3.3. The tool has a command line interface. Users can generate, with a single command, the entire test suite by executing the script without options. In addition, the tool can accept many options based on several factors, including options to generating test cases for selected OWASP categories or CWE weaknesses. Possible invocation commands are shown in Figure 2.1.

This command shows the available options and the help message.

```
$python core.py -h
```

This command runs the entire script.

```
$python core.py
```

The following commands are equivalent. They generate flaws related to the XSS and Injection in OWASP's categories options.

```
$python core.py -flaw=XSS,Injection
$python core.py -f XSS,Injection
```

The following commands are equivalent. They generate XSS and SQL Injection test cases in CWE options.

```
$python core.py --cwe=79,89
$python core.py -c 79,89
```

Figure 2.1 Command Line arguments

Table III summarizes the available options.

D. *Internal Structure*

The internal structure of the PHP vulnerability test suite generator consists of three modules: Input, Filtering, and Sink, corresponding to the most important components of a vulnerability.

- *Input* is where untrusted data are injected, e.g., command line, variable or form methods.
- *Filtering* is used to filter the inputs with sanitization functions, casting, deprecated functions, or simply no filtering.
- *Sink* is where a sensitive operation (such as a database query) is executed with potentially untrusted input and where the vulnerability is triggered.

TABLE III. AVAILABLE OPTIONS

CWE Option	
-c / --cwe=	**Weaknesses Description**
78	Command OS Injection
79	Cross Site Scripting - 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
OWASP Category Option	
-f / --flaw=	**Weaknesses Description**
IDOR	Insecure Direct Object Reference
Injection	Injection (SQL, LDAP, XPATH, OS Command, Code)
SDE	Sensitive Data Exposure
SM	Security Misconfiguration
URF	URL Redirects and Forwards
XSS	Cross-site Scripting

Each sample is constructed based on the specification of one *Input*, one *Filtering* and one *Sink*. Then the *Filtering* module is embedded inside the *Complexity* templates. However, multiple *Input* methods, *Filtering* methods, and *Sink* methods can be specified with permutations and combinations of these which results in thousands of test cases with or without flaws with various complexities. See Figure 2.2.

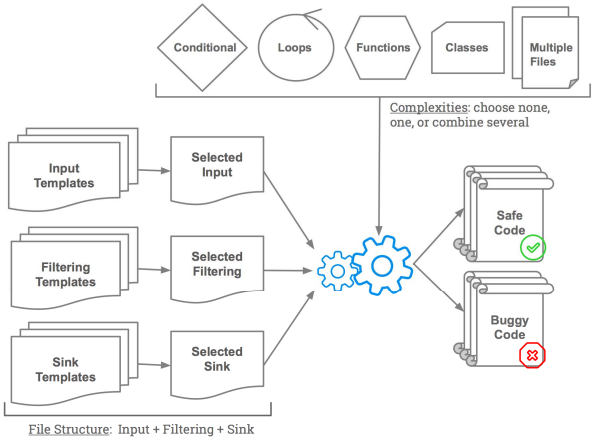

Figure 2.2: Design of the generator

We use three XML files to store the metadata that detail the characteristics of the modules and their information. See respectively Figure 2.3, Figure 2.4 and Figure 2.5 which describe the internal structure of the *Input* XML file, *Filtering* XML file and *Sink* XML file. Each of these structures has been used to generate the given example in Figure 2.7.

Input.xml stores the functions used to get external data that can be malicious. Almost all the test cases generated need external data to trigger the weakness. The XML tags in Figure 2.3 are described in Table IV:

TABLE IV. *INPUT* XML TAGS DESCRIPTION

<Tags>	Description
path	Information used to create the filename. Each <dir> tag contains key words of the *Input* method used, and is a sub-section of the filename
comment	Description of the *Input* method
code	Contains the source code
inputType	Describes the type of input and its source

```
<sample>
  <path>
    <dir>POST</dir>
  </path>
  <comment>INPUT : get the field UserData from the
           variable $_POST</comment>
  <code>$input = $_POST['UserData'];</code>
  <inputType>variable : $_POST['UserData']</inputType>
</sample>
```

Figure 2.3 *Input* XML structure

Filtering.xml lists the filtering methods that can be used by several classes of vulnerability. It contains safe functions, deprecated functions and misused functions that can be safe for a specific class of vulnerability but unsafe for others. The XML tags in Figure 2.4 are described in Table V.

TABLE V. *FILTERING* XML TAGS DESCRIPTION

<Tags>	Description
flaws	Describes the vulnerability categories where the sample can be used
path	Information used to create the filename. Each <dir> tag contains key words of the *Filtering* method used, and is a sub-string of the file name
comment	Description of the *Filtering* method
codes	Contains the source code
constraints	Used by the generator to construct a valid PHP program by specifying the type of input the function needs
safeties	Information about the soundness of the filtering function for the different vulnerability classes. *safe="1"*means that it is safe, *safe="0"* it is not

Sink.xml contains the source code where the flaw can be exploited, and other pieces of source code that are related to the potential exploit. The XML tags in Figure 2.5 are described in Table VI.

TABLE VI. *SINK* XML TAGS DESCRIPTION

<Tags>	Description
flaws	Describes the vulnerability categories where the sample can be used
path	Information used to create the filename. Each <dir> tag contains key words of the *Sink* method used, and is a sub-string of the file name
comment	Description of the *Sink* function
codes	Contains several pieces of source code depending on the complexity chosen
constraints	Used by the generator to construct a valid PHP program by specifying the type of input the function needs
safeties	Information about the soundness of the sink function for the different vulnerability classes. *safe="1"*means it is safe, *safe="0"* it is not

```
<sample>
  <flaws>
    <flaw>CWE_89_Injection</flaw>
    <flaw>CWE_78_Injection</flaw>
  </flaws>
  <path>
    <dir>func_mysql_real_escape_string</dir>
  </path>
  <comment>FILTERING : use of mysql_real_escape
           string</comment>
  <codes>
    <code>$tainted =
          mysql_real_escape_string($input);</code>
  </codes>
  <constraints>
    <constraint flawType = "CWE_89_Injection" type =
                "string" />
    <constraint flawType = "CWE_78_Injection" type =
                "string" />
  </constraints>
  <safeties>
    <safety flawType = "CWE_89_Injection" safe = "0"/>
    <safety flawType = "CWE_78_Injection" safe = "0"/>
  </safeties>
</sample>
```

Figure 2.4 *Filtering* XML structure

```
<sample>
  <flaws>
    <flaw>CWE_89_Injection</flaw>
  </flaws>
  <path>
    <dir>select_from_where</dir>
    <dir>concatenation</dir>
  </path>
  <comment>SINK : SQL query with concatenation
  </comment>
  <codes>
    <code>$query = "SELECT * FROM student where id=".
          $tainted . "";</code>
    <code> [source code removed for clarity] </code>
  </codes>
  <constraints>
    <constraint flawType="CWE_89_Injection" type =
                "string" />
  </constraints>
  <safeties>
    <safety flawType="CWE_89_Injection" quote = "0"
            safe = "0" />
  </safeties>
 </sample>
```

Figure 2.5 Sink XML structure

The complexity of the test case is wrapped around one or more modules. It can be expressed by adding control flow and/or data flow elements, and can be constructed recursively:

- if-true : <complexity type="if-true">
- if-false : <complexity type="if-false">
- for : <complexity type="loop" kind="for">
- while : <complexity type="loop" kind="while">
- function : <complexity type="function">
- class : <complexity type="class">
- file : <complexity type="file">

The complexity elements in the XML document described in Figure 2.6 produces a test case that calls a *Filtering* method from a class located in a separated file. The *sink* code is written inside a conditional block, which is always true.

```
<program>
  <input/>
  <complexity type="file">
    <complexity type="class">
      <filtering/>
    </complexity>
  </complexity>
  <complexity type="if-true">
  <sink/>
  </complexity>
</program>
```

Figure 2.6 *Complexity* XML structure

E. *Output Results*

In a very short time, the tool generates a lot of different test cases, both with and without flaws, with various complexities. It takes about 40 seconds to run the script and generate the entire test suite (42 212 test cases) in a Virtual Machine configured with 8 cores and 4 GB of RAM. A summary of how many flawless and vulnerable test cases have been produced is displayed after each vulnerability category is generated.

Figure 2.7 provides an example of a produced SQL Injection test case (without any code complexity) with the three internal structures *Input-Filtering-Sink*.

```
<?php

    $input = $_POST['UserData'];                                   Input

    $tainted = mysql_real_escape_string($input);                   Filtering

    $query = "SELECT * FROM student where id=". $tainted . "";

    $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
    mysql_select_db('dbname') ;
    echo "query : ". $query ."<br /><br />" ;

    $res = mysql_query($query); //flaw                             Sink

    while($data = mysql_fetch_array($res)){
        print_r($data) ;
        echo "<br />" ;
    }

    mysql_close($conn);
?>
```

Figure 2.7: Sample generated vulnerable PHP test case

Table VII details the number of generated files, grouped by OWASP TOP 10 categories and their associated CWEs. 29 258 samples are safe, and 12 954 samples are faulty.

TABLE VII. SUMMARY OF THE GENERATED TEST CASES

Vulnerability	CWE	Safe	Unsafe	Total
Insecure Direct Object Reference	862 - Missing Authorization	400	80	**480**
Injection	78 - OS Command Injection	1872	624	**2496**
	89 - SQL Injection	8640	912	**9552**
	90 - LDAP Injection	1728	2112	**3840**
	91 - XML Injection	4784	1264	**6048**
	95 - File Injection	1296	336	**1632**
	98 - PHP Remote File Inclusion	2592	672	**3264**
Sensitive Data Exposure	311 - Missing Encryption of Sensitive Data	2	2	**4**
	327 - Use of a Risky Cryptographic Algorithm	3	5	**8**
Security Misconfiguration	209 - Information Exposure Through an Error Message	5	3	**8**
URL Redirects and Forwards	601 - URL Redirection to Untrusted Site	2208	2592	**4800**
Cross Site Scripting	79 - Cross-Site Scripting (XSS)	5728	4352	**10 080**
	Total	**29 258**	**12 954**	**42 212**

The generated test cases are stored in categorized directories in order to make the samples easier to retrieve. First, each generated test suite is available in a new folder with the date and time as an attribute. Then the test cases are split by *Vulnerability categories* and by their associated *CWE ids*. Finally, inside these directories, the PHP files are split in safe and vulnerable directories. See Figure 2.8.

```
PHPTestSuite_MM_DD_YYYY_HHhMMmSS/
└── Vulnerability categories/
        └── manifest.xml
        └── CWE numbers/
            ├── Safe samples/
            │       └── PHP files
            └── Unsafe samples/
                    └── PHP files
```

Figure 2.8: Directory tree generated

The generated file from Figure 2.7 would be located under the directory:

`PHPTestSuite_05_04_2015_13h37m42/Injection/89/unsafe/`

To identify a test case based on its name, the generator uses a naming convention. Each generated PHP file name is unique. It follows a specific pattern including the CWE number, the *input* method, the *filtering* function, and the *sink*. An example of a generic filename is given in Figure 2.9.

CWE_[*CWE_number*]_[*Input*]_[*Filtering*]_[*Sink*].php

Figure 2.9: File name pattern

The filename in the example illustrated in Figure 2.7 would be:

CWE_89__POST__func_mysql_real_escape_string__select_from_where-concatenation.php

A manifest file, called *manifest.xml*, is generated in every *Vulnerability categories* directory (see Figure 2.10). Its role is to keep track of every test case that has been produced and make the analysis easier. It reports them with their associated attributes as described in Table VIII.

TABLE VIII. *MANIFEST* XML TAGS DESCRIPTION

<Tags>	Description
testcase	Description of a sample
metadata	Contains information about the sample
file	Describes the generated test case with its path and its language

If the sample is unsafe, a tag <flaw> is added to indicate the type of vulnerability and the flawed line. Note that a unique manifest file for the whole generation would have been too large, and may crash the generator depending on the memory allocated on the host.

```
<testcase>
      <meta-data>
           <author>Bertrand STIVALET, Aurelien
                  DELAITRE</author>
           <date>17/12/15</date>
           <input>Variable: $_POST[‘UserData‘] </input>
      </meta-data>
      <file path=
"CWE_89/unsafe/CWE_89__POST__func_mysql_real_escape_strin
g__select_from_where-concatenation.php " language="PHP">
          <flaw line="15" name ="Injection"/>
      </file>
</testcase>
```

Figure 2.10: Manifest file example

III. STATE OF THE TOOL

The PHP Vulnerability Test Suite generator is open source and has been made publicly available on Github [14]. The generated test suite is hosted in the NIST's SARD [15] among other large test suites written in other programming languages. All the test cases are individually accessible using the search engine without the need of running the generator. Researchers have been using it to conduct studies. In the short time (a few months) since its release, three research papers have used the test suite.

- [16] proposes a novel feature extraction algorithm to extract basic and context features from the source code of web applications. It is based on various machine-learning models for predicting context-sensitive Cross-Site Scripting (XSS) security vulnerabilities. They have performed their tests on the PHP Vulnerability Test Suite, as it provides mostly all the cases required for XSS.
- [17] uses the PHP generated test suite to propose a context-sensitive approach based on static taint analysis and pattern matching techniques to detect and mitigate the XSS vulnerabilities in the source code of web applications.
- Students from TELECOM Nancy, University of Lorraine, used the PHP Vulnerability Test Suite to analyze the behavior of three PHP static source code analyzers: HP Fortify, Syhunt and RIPS.

The generator provides a valuable PHP test suite. It will be used for the 6th edition of the Static Analysis Tool Exposition (SATE) [18] run by researchers at NIST. Until now, they conducted their researches on C/C++ and Java static analyzers only due to the lack of PHP materials.

A. *Benefits*

It is not within the scope of this paper to describe the results of the static analyzers evaluation, but we will describe our experiences in using such an approach for software testing in general. Automated test cases generation reduces tester's time and increases productivity by reducing the effort required to perform repetitive tasks. Test automation, unlike manual testing, provides much greater coverage when it comes to processing large data volumes. It also eliminates human mistakes.

Because the test cases indicate where flaws occur, users can evaluate the reports' appropriateness semi-automatically. Another benefit is that users can select a particularly important set of flaws to examine; hence, studying a wide range of tools' capabilities is possible.

B. *Future Work*

The test cases are synthetic, that is, they were created as examples with well-characterized weaknesses. Each test case targets only one flaw. As a result, the cases have a much simpler structure than most weaknesses in production code. Therefore, the test cases are not representative of real software.

Not all the vulnerability categories described in the OWASP TOP 10 could be addressed in source code testing. For instance, the 2013 A7 - Missing Function Level Access Control category is unlikely to be found by static analyzers.

In short, to get the ideal test suite generator, it should produce more complex test cases and cover more vulnerabilities.

C. Conclusion

We designed a PHP test case generator that produces thousands vulnerable as well as safe code embedded in various complexities expressing different vulnerabilities. These test cases can be used to analyze the reporting of static analyzers and compute the metrics to test their effectiveness. This approach allows us to measure all the metrics listed in Table II, especially *coverage*, since by design the test cases contains an extensive panel of vulnerability types. Running the tools on both easily established good from bad cases makes reading the tool report easier and allows computing the *discrimination*. Lastly, *overlap* can be calculated on all test sets by having several tools analyze the same data and comparing their findings.

This global testing approach should lead to better PHP static analysis tools, and therefore, more secure web applications.

Now that the generator is available and in use, we plan to generate more test cases in other languages. Based on the engine built for this generator, we are conducting a similar project which builds about 60 000 C# vulnerable and non-vulnerable test cases.

ACKNOWLEGMENT

We wish to thank Aurelien Delaitre, Herve Buhler, David Lucas, Fabien Nollet, Axel Reszetko, Xavier Marchal, Guillaume Pighi, Jonathan Retterer for their help in developing the PHP Vulnerability Test Suite generator. We also thank Vadim Okun for numerous contributions to this paper.

REFERENCES

[1] National Vulnerability Database (NVD), NIST https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&query=PHP

[2] Dynamic Application Security Testing (DAST) https://www.techopedia.com/definition/30958/dynamic-application-security-testing-dast

[3] Application Security Testing (SAST) https://www.techopedia.com/definition/30521/static-application-security-testing-sast

[4] SAMATE Software Assurance Reference Dataset. NIST http://samate.nist.gov/SARD

[5] A.Saswat, et. Al, “An Orchestrated Survey on Automated Software Test Case Generation," Journal of Systems and Software, February 2013.

[6] K. Kratkiewicz (2005) "Evaluating Static Analysis Tools for Detecting Buffer Overflow in C Code", Master Thesis, Harvard University, Cambridge, MA, 285 pages

[7] T. Boland and P. E. Black, "The Juliet 1.1 C/C++ and Java Test Suite," Published by the IEEE Computer Society, Computer, October 2012, pp. 82-84.

[8] Common Weakness Enumeration (CWE), MITRE https://cwe.mitre.org

[9] C. Oliveira, "Real World software Assurance test Suite: STONESOUP," in Proceedings of Software Technology Conference (STC), October 13-15, 2015, Long Beach, CA. Accessed http://conference.usu.edu/stc/Schedule/Grid_Details.cfm?eid=14612&pg=none&aid=1955&ty=grid&des=reg

[10] D. Balzarotti , et al, “Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications”, published for IEEE Symposium on Security and Privacy, 2008 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4531166

[11] TELECOM Nancy http://telecomnancy.univ-lorraine.fr/en

[12] A. Delaitre, B. Stivalet, V.Okun, and E. Fong, "Evaluating Bug Finders" First International Workshop on Complex faults and Failures in Large Software System, (COUFLESS 2015)

[13] OWASP “Usage Top 10, 2013,” List https://www.owasp.org/index.php/Top_10_2013-Top_10

[14] B. Stivalet and .A. Delaitre, "PHP Vulnerability test suites generator”, Access: 2015, https://github.com/stivalet/PHP-Vuln-test-suite-generator

[15] Test Suites hosted into the Software Assurance Reference Dataset, https://samate.nist.gov/SARD/testsuite.php

[16] M. K. Gupta, et al, “Security Vulnerabilities in Web Applications," IEEE 2015, published for the 12th International Joint Conference on Computer Science and Software Engineering (JCSSE), July 2015 in Thailand. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7219789.

[17] M. K. Gupta, et al, "XSSDM: Towards Detection and Mitigation of Cross-Site Scripting Vulnerabilities in Web Applications ," IEEE 2015, published for the 4th International Conference on Advances in Computing, Communications and Informatics (ICACCI), August 2015 in India. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7275912.

[18] SATE Static Analysis Tool Exposition. NIST https://samate.nist.gov/SATE.html