Journal of Engineering and Applied Sciences 13 (Special Issue 12): 9468-9473, 2018
ISSN: 1816-949X

A Smart Fuzzing Tool for Vulnerability Analysis of Open Source Software

[1]Kwang-Jik Kim, [1]Yong-Sun Ko, [2]Jae-Pyo Park and [2]Jong-Hee Lee
[1]Department of IT Policy Management,
[2]Department of Information Security, Soongsil University, 07027 Seoul, Republic of Korea

Abstract: Recently, program developed by open source tends to be utilized in various industrial fields but inexperienced developers can produce the open source code vulnerable to security such as not complying with secure coding standard or standard of programming language, etc., rather due to the advantages of open source which anyone can view, correct and distribute source code freely. In this study, it suggests the smart fuzzing tool interworked white box test and black box test in order to analyze the effective vulnerability of open source. The smart fuzzing tool was implemented as a prototype and it was found that the accuracy of the vulnerability detection was improved through the vulnerability analysis regarding the open source. It is expected that the suggested smart fuzzing tool detects the vulnerability with automated measure, more exactly and effectively and overcomes the limit which existing fuzzing system has.

Key words: Open source, secure coding, vulnerability analysis, black and white box testing, smart fuzzing, automated measure

INTRODUCTION

Open source is the program source code or software given for free and anyone can view open source code freely. It should be kept the right of original producer of open source, so, it can be corrected and distributed (Wikitree, 2015). However, inexperienced developers can produce the open source code vulnerable to security such as not complying with secure coding standard or standard of programming language, etc., rather, due to the advantages of open source which anyone can view, correct and distribute source code freely. Also, developers may have wrong cognition that wide review and test could be progressed for open source codes, produced and distributed like this, vulnerable to security.

On the basis of wrong realization, developers distribute and use many open source code vulnerable to security or use compilers not suitable to standard without specialized analysis of source code. It can be re-produced serious vulnerabilities of security for open source such as Heartbleed of open SSL (Hacksum, 2015; Anonymous, 2015a, b) or vulnerability in execution of remote command of GNU Bash. Therefore, specialized developers should remove security vulnerabilities consciously, doing exact coding based on standard in order to increase the security not occurring security vulnerabilities of open source. Besides, CERT coding standard is being used as one of guideline for it (Kang, 2015).

In addition, although, developers try to develop the program on the basis of standard when developing, it cannot be perfect coding without any vulnerabilities, so, it should be complemented through additional analysis of vulnerabilities to minimize such security vulnerabilities (Zerkane *et al*., 2016). Thus, it suggests the smart fuzzing tool interworked white box test and black box test for secure coding of open source in this study. The proposed smart fuzzing tool was implemented as a prototype and it was found that the accuracy of the vulnerability detection was improved through the vulnerability analysis regarding the open source.

Literature review

Black box test: The black box test is the one which tester executes, just understanding which function software performs. Tester does not know how software acts internally. It tests in drawing the result value corresponding with input value, on the basis of software function and requirement only (Zhao and Liu, 2016; Michael *et al*., 2007). The black box test has advantages which not require the information and technical skill of source code but useful in big system. However, it can hardly test all input values for short periods, detect logical error and make test case without knowledge for definite functional specification.

White box test: The white box test is the one which tester accesses to code of software and inspects and tester can know the internal process of action of software. On the contrary, it can verify how software input value draws result value but it has a risk of not testing objectively, while executing test in accordance with action of code

Corresponding Author: Jong-Hee Lee, Department of Information Security, Soongsil University, 07027 Seoul, Republic of Korea

(Khan and Khan, 2012; Ron, 2006). It can be accessed to code, checked all codes clearly and distinguished the error for unveiled codes. Besides, it has advantage to be easy in inferring input value and making test scenario in white box test. To execute white box test, its disadvantage is to need practiced skill, to be expensive and not guaranteed to meet the specification of test.

Fuzzing: Fuzzing is the security test to identify the potential vulnerabilities and errors, inputting ineffective value or selected value randomly, inducing unexpected result. Fuzzing is divided into sorts depending on the selecting method of input value to test and there are dump fuzzing, inputting data randomly and smart fuzzing, understanding software properly, selecting and inputting the related data (Cha *et al*., 2015; Kim *et al*., 2016a, b). Generally, fuzzing refers to the automated verification on the program. The fuzzing test is conducted while the program uses various data including the unexpected data at random. The usefulness of fuzzing is placed in the data set creation for finding vulnerability.

Symbolic execution: The symbolic execution is basically expressed as an unknown quantity instead of the concrete value. If the program meets the branching statement, it creates the symbolic value as following routes on both sides. That is, it is a way to figure out what values each variable has at which point after watching the conditions of several branching statements. The symbolic execution engine applies this principle to the program to see the route condition the engine is accumulated and follow it. That is, we can dramatically improve the code coverage that the dynamic analysis is done through the symbolic execution (Kim *et al*., 2016a, b).

MATERIALS AND METHODS

Process of smart fuzzing: Unlike the conventional fuzzing, the smart fuzzing is an automated method to detect the vulnerable points of the software. A data model should be created regarding the targeted software that needs fuzzing and analysis is automatically performed regarding the data file and software itself.

Unlike the conventional fuzzing, proposed smart fuzzing models the vulnerability information and input file structure through the static analysis of the white box test as shown in Fig. 1. It then connects and fuzzes the modeled data structure and vulnerability code through the dynamic analysis of black box test to extract the input data and detects and reports the vulnerability.

Smart fuzzing executes code as initial input value, using generated input data value. It collects the scope for input value and generates new scope in symbolic execution. Figure 2 shows the composition of smart fuzzing and data flow of each composition.

The following shows a detailed data treatment process of the smart fuzzing tool for the vulnerability analysis.

Uploading of the analysis target source and registration of the target of fuzzing: Register the targeted open source website or open-source application as well as the basic information including URL to the smart fuzzing tool, through the vulnerability analysis DB.

Vulnerability inspection by the static analysis (white box test) and the detection of external data inflow source code: Detect vulnerability that could be attained through the static analysis by registering the open source data that are registered on the vulnerability analysis DB to the static analysis engine. In addition, the Java source code inflowing data from the external source through the static analysis and the source code that includes the logic having the suitability and possibility of flow analysis for symbolic execution is detected based on the Java source code.

Store the analysis data in the vulnerability analysis DB: The data (the source code data that will be sent to the symbolic execution engine) that are attained through the static analysis are stored in the vulnerability analysis DB.

Register fuzzing information to the fuzzer: The basic information including the URL that was initially registered in the vulnerability analysis DB is registered to the fuzzer to prepare for the black box test.

Black box test of the service web through the fuzzer: Perform a black box test through the data that are registered to fuzzer, regarding the service web (a website that is under service with the operation of open source) which is the subject of the analysis.

Store the test result in the vulnerability analysis DB: Store the resulting information through the fuzzer's black box test to the vulnerability analysis DB. The information attained through the black box test includes the detected vulnerability and the URL information that is extracted from the targeting web.

Create the target information by matching the resulting data from the static and dynamic analyses and send them to the symbolic execution engine: Match the data detected through the white box test (static analysis) that

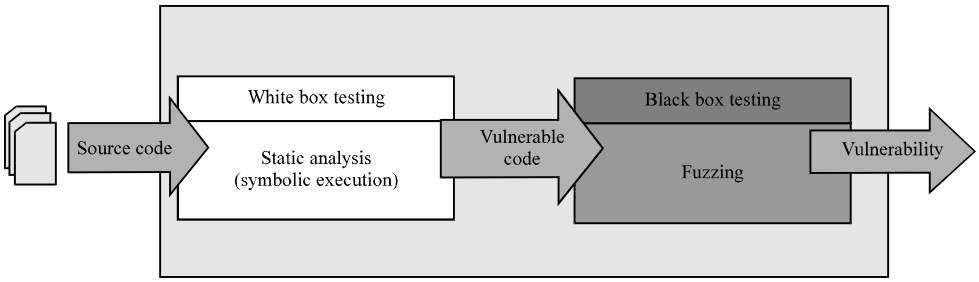

Fig. 1: Conceptual diagram for smart fuzzing

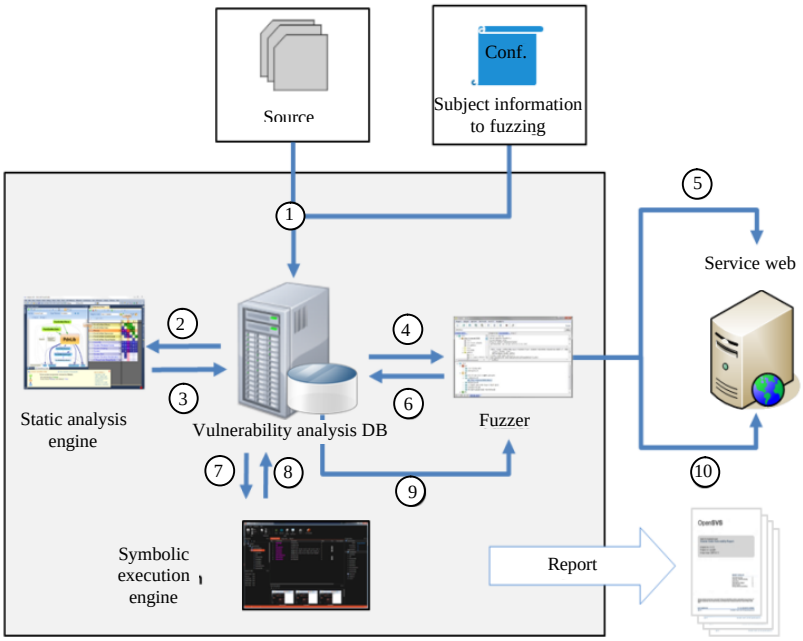

Fig. 2: Architecture and process of smart fuzzing tool

are stored in the vulnerability analysis DB with the data detected through the black box test (fuzzer). Find the source code that can create the test data through the symbolic execution engine from the matched information and send it to the symbolic execution engine to be used as the target of analysis.

Store the test data that are created through the symbolic execution engine in the vulnerability analysis DB: Store the test data set that are created through the symbolic execution engine in the vulnerability analysis DB. Since, the test data could not always be attained from all the source codes that are registered in the symbolic execution engine, select the data that could be used in the final smart fuzzing and store them in the vulnerability analysis DB.

Register the data and URL for smart fuzzing to the fuzzer: Register the data set and URL for the smart fuzzing by integrating the test information through the white box test, black box test and symbolic execution engine.

Perform the smart fuzzing through fuzzer: Perform the vulnerability inspection through the fuzzer's smart fuzzing, targeting the finally registered data.

RESULTS A AND DISCUSSION

Process of smart fuzzing: The smart fuzzing tool interworked with the designed white box test and black box test is implemented in prototype. It is the execution screen of black box test for analysis target (service web) through fuzzer in Fig. 3. It performs the black box test into service web by registered data in fuzzer.

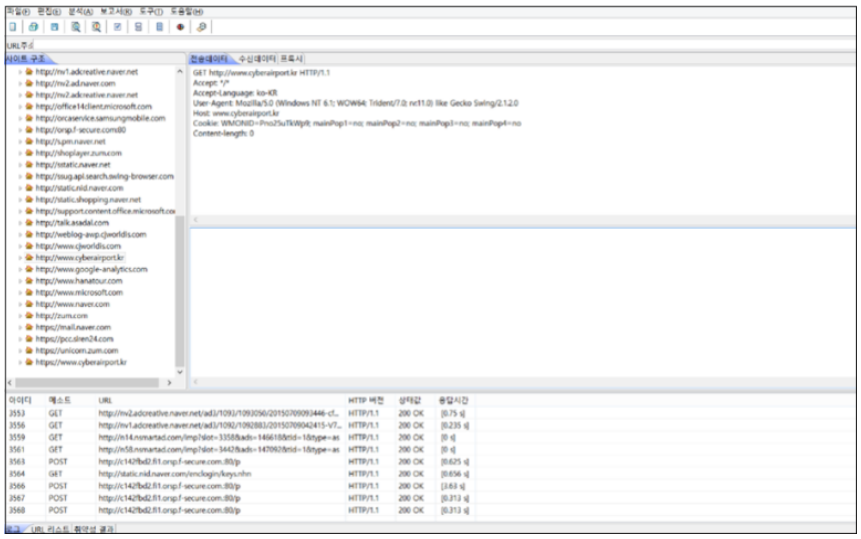

Fig. 3: Black box test by fuzzer

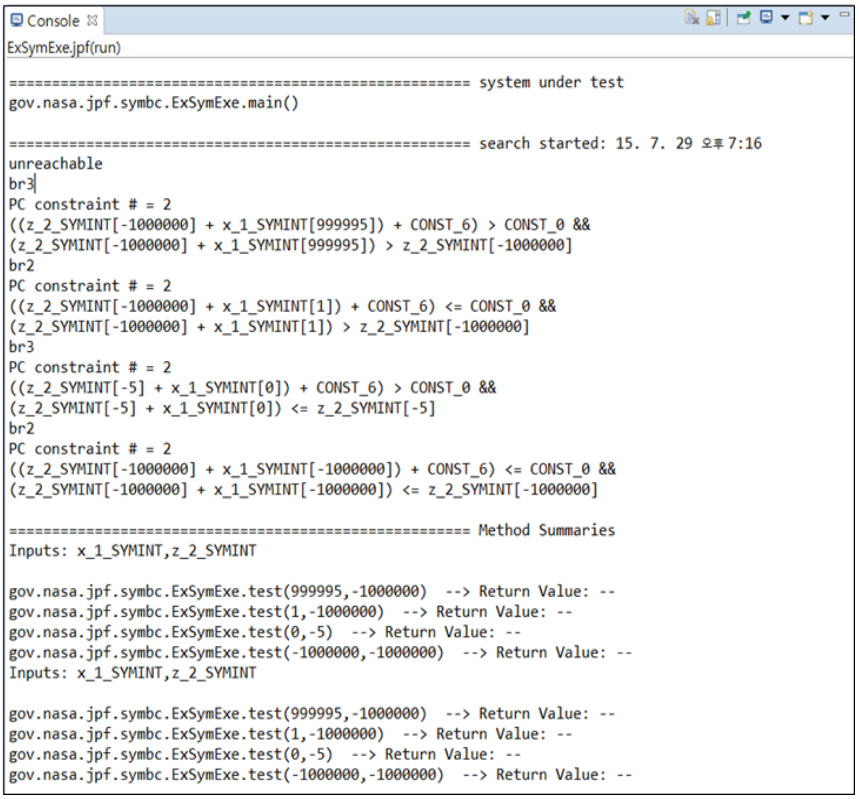

Fig. 4: Extracting test data by the execution engine

In Fig. 4, it is the screen extracting test data through symbolic execution engine, matching static and dynamic data and transmitting into symbolic execution engine of target information.

Figure 5 shows the screen when the analysis of the open source's vulnerability is being performed by the fuzzer. It is shown that a vicious script and an inappropriate method are detected.

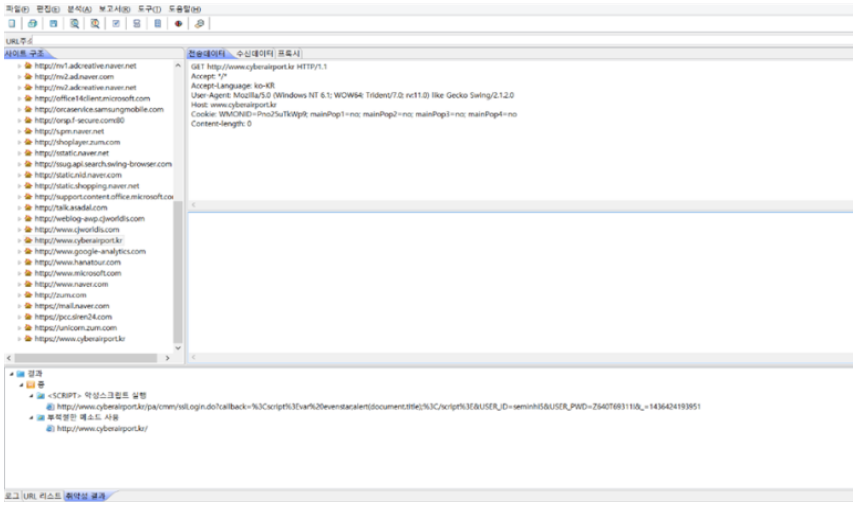

Fig. 5: Analysis of the open source's vulnerability by the fuzzer

Table 1: Test results for vulnerability detection accuracy of the smart fuzzing

	White box test				Black box test		Combined with black and white box test	
Security (vulnerabilities detection using Juliet code)	Find bugs		Symbolic execution		Fuzzing		Smart fuzzing	
	Detection time (sec)	Detection accuracy (%)	Detection time (sec)	Detection accuracy (%)	Detection time (sec)	Detection accuracy (%)	Detection time (sec)	Detection accuracy (%)
Race condition	N/A	N/A	0.71	25.2	N/A	N/A	0.82	72.5
Divide by 0	0.89	35.9	0.75	43.4	0.81	45.3	0.78	66.4
Not reachable	0.72	15.5	0.71	23.6	0.65	21.0	0.79	55.2
Null pointer exception	0.35	63.1	0.31	65.2	0.23	67.5	0.39	78.3
Assertion errors	N/A	N/A	N/A	N/A	0.21	23	0.23	58.3

*Juliet code can be downloaded from http://samate.nist.gov/SARD/testsuite.php

The vulnerability detection accuracy measurement was performed through an experiment to evaluate the performance of the realized smart fuzzing tool. Table 1 shows the results of the vulnerability detection status when the vulnerability is analyzed through the white box test, black box test and the smart fuzzing that integrates both the white box test and the black box test. The find bugs was used for the static analysis engine and for the symbolic execution engine and fuzzer, a program that, we have developed for this study was used. For the open source files, Juliet code 25,477 Java files were used as the targets of the experiment.

The results show that the vulnerability detection time was relatively longer than the vulnerability detection performed by the white box test and black box test but the accuracy was higher when the smart fuzzing was used for the vulnerability test.

CONCLUSION

In this study, we suggest the smart fuzzing tool interworked white box and black box test for secure coding of open source. In addition, it tests whether vulnerability analysis acts normally, realizing proto type of suggested smart fuzzing tool. The suggested smart fuzzing tool provides higher test coverage than existing vulnerability analysis method and it interworks with exploit. It is planned to assess and inspect the capacity for suggested tool through experiment and evaluation, after realizing the perfect tool of reporting module and service web from now on. We expect that it overcomes the limits existing fuzzing system has and detects vulnerability more exactly and efficiently in automated measure, through suggested smart fuzzing tool and interworking with white box test and black box test.

REFERENCES

Anonymous, 2015a. Heart bleed memory disclosure upgrade open SSL now!. Cisco Systems, Inc., San Jose, California, USA. http://blog.talosintelligence.com/2014/04/heartbleed-memory-disclosure-upgrade.html.

Anonymous, 2015b. To prevent a second heart bleed should increase support open source developers. CBS Interactive Inc, San Francisco, California, USA.

Cha, S.K., M. Woo and D. Brumley, 2015. Program-adaptive mutational fuzzing. Proceedings of the IEEE Symposium on Security and Privacy (SP), May 17-21, 2015, IEEE, San Jose, California, USA., ISBN:978-1-4673-6949-7, pp: 725-741.

Hacksum, 2015. OpenSSL heartbleed vulnerability (CVE-2014-0160). United States Computer Emergency Readiness Team (US-CERT), Arlington, Virginia.

Kang, M., 2015. Static analysis of hypervisor open source based on secure coding. Master Thesis, Korea University, Seoul, South Korea.

Khan, M.E. and F. Khan, 2012. A comparative study of white box, black box and grey box testing techniques. Intl. J. Adv. Comput. Sci. Appl., 3: 12-15.

Kim, J.H., M.C. Ma and J.P. Park, 2016a. An analysis on secure coding using symbolic execution engine. J. Comput. Virol. Hacking Tech., 12: 177-184.

Kim, S., W. Jo and T. Shon, 2016b. A novel vulnerability analysis approach to generate fuzzing test case in industrial control systems. Proceedings of the IEEE Conference on Information Technology, Networking, Electronic and Automation Control, May 20-22, 2016, IEEE, Chongqing, China, ISBN:978-1-4673-9195-5, pp: 566-570.

Michael, S., A. Greene and P. Amini, 2007. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, Addison-Wesley, Boston, Massachusetts, ISBN:9780321446114, Pages: 543.

Ron, P., 2006. Software Testing. 2nd Edn., Pearson Education, London, England, ISBN:978-81-7758-031-0, Pages: 377.

Wikitree, 2015. How open source is used for?. Wikitree, New York, USA.

Zerkane, S., D. Espes, P.L. Parc and F. Cuppens, 2016. Vulnerability Analysis of Software Defined Networking. In: Foundations and Practice of Security, Cuppens, F., L. Wang, N. Cuppens-Boulahia, N. Tawbi and J. Garcia-Alfaro (Eds.). Springer, Switerzland, ISBN:978-3-319-51965-4, pp: 97-116.

Zhao, M. and P. Liu, 2016. Empirical Analysis and Modeling of Black-Box Mutational Fuzzing. In: Engineering Secure Software and Systems, Caballero J., E. Bodden and E. Athanasopoulos (Eds.). Springer, Switzerland, ISBN:978-3-319-30805-0, pp: 173-189.