A Survey of Automated Root Cause Analysis of Software Vulnerability

JeeSoo Jurn, Taeeun Kim, and Hwankuk Kim(✉)

Korea Internet & Security Agency, 9, Jinheung-gil, Naju-si, Jeollanam-do 58324, Republic of Korea
{jjs0771, tekim31, rinyfeel}@kisa.or.kr

Abstract. In recent years, many researches on automatic exploit generation and automatic patch techniques have been published. Typically, in the CGC (Cyber Grand Challenge) competition hosted by DARPA, a hacking competition was held between machines to find vulnerabilities, automatically generate exploits and automatically patch them. In the CGC competition, they implemented themselves to work on their own platform, allowing only 7 system calls. However, in a real environment, there are much more system calls and the software works on complicated architecture. In order to effectively apply the vulnerability detection and patching process to the actual real environment, it is necessary to identify the point causing the vulnerability. In this paper, we introduce a method to analyze root cause of vulnerabilities divided into three parts, fault localization, code pattern similarity analysis, and taint analysis.

1 Introduction

As hacking techniques become advanced, vulnerabilities have been exponentially increasing. The number of vulnerabilities in which about 80,000 vulnerabilities are newly registered in CVE (Common Vulnerability Enumeration) from 2010 to 2015 is increasing [1]. While the number of vulnerabilities is increasing rapidly, response to vulnerabilities depends on manual analysis, which slows down the response speed. We need to develop techniques that can automatically detect vulnerabilities and patch them. To accurately patch the exploited results, a precise cause analysis process is needed. Figure 1 shows the techniques and methods for automated vulnerability detection, automated vulnerability analysis, and automatic patch. In order to analyze vulnerabilities, it is necessary to classify vulnerabilities preferentially as follows. After vulnerability classification, we track the location of the vulnerability and derive a scheme that can be patched for each vulnerability. Although there are various techniques for the cause analysis, Fault Localization technology is a technique of assigning weights based on test cases and analyzing points that caused errors. Code Pattern analysis is a technique of navigating the similarity analysis and outputting the part containing the affected code. To identify the cause of the vulnerability, it is classified into three methods: fault localization, code pattern analysis, and taint analysis. These three techniques will be described in detail in Sect. 3 below.

L. Barolli et al. (Eds.): IMIS 2018, AISC 773, pp. 756–761, 2019.
https://doi.org/10.1007/978-3-319-93554-6_74

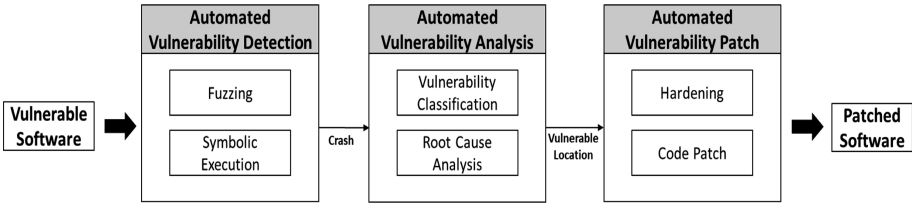

Fig. 1. Automated vulnerability remediation about vulnerable software

2 Related Work

2.1 Automated Vulnerability Detection

Automated vulnerability detection techniques can be divided into fuzzing and symbolic Execution. Fuzzing is black-box testing method that randomly change input values to find software vulnerability. Symbolic Execution is white-box testing method that find input to explore paths of the target software. Fuzzing is divided into dumb fuzzing and smart fuzzing according to input modeling. Mutation fuzzing and generation fuzzing are classified according to test case generation method. Dumb fuzzing is the simplest fuzzing technique to generate defects by randomly changing input values for the target software [2]. Because changing input values is simple, test case creation is fast, but because the scope of the code is narrow, it is difficult to find a valid crash. Smart Fuzzing is a technique for generating input values suitable for a format and generating an error through target software analysis [3–6]. Smart fuzzing has the advantage of knowing where errors can occur through software analysis. The tester can create test cases for that branch to extend the coverage of the code and generate valid conflicts. However, analyzing the target software requires expert knowledge and takes a long time to generate a template suitable for software input. Testers can create test cases for those branches to extend the coverage of the code and generate valid conflicts. However, analyzing the target software requires expert knowledge and takes a long time to generate a template suitable for software input. Mutation fuzzing is a test technique that enters the target software by modifying the data samples. Generation fuzzing is a technique for modeling the format of input values to be applied to target software and creating new test cases for that format.

Symbolic execution is largely divided into offline symbol execution and online symbol execution. Offline symbol execution is resolved by analyzing the path predicate [7] and selecting only one path to generate a new input value. The disadvantage is that the program must run from the beginning to navigate to another path, causing redo to cause overhead. Online symbolic execution is a way in which the state is replicated and the path executor is created each time the symbol executor encounters a branch statement [8, 9]. There is no overhead associated with re-execution using the online method, but significant resources are consumed because all state information must be stored and multiple states must be processed at the same time. Hybrid symbolic execution is suggested to solve this problem. Hybrid symbolic execution is performed every time a branch statement is executed, storing the status information via online

symbolic execution and proceeding until memory is exhausted [10]. If you no longer have space to store, switch to offline symbolic execution and perform a path search. Fixed a memory overflow issue in online symbolic execution by saving state information and applying later methods through hybrid symbolic execution. It also eliminates the overhead of offline symbolic execution because you do not need to run it again from the start point (Table 1).

Table 1. Automated vulnerability detection techniques comparison

Techniques	Testing method	Description
Fuzzing	Mutation fuzzing	Randomly mutate-based test case generation
	Generation fuzzing	Target modeling based test case generation
	Dumb fuzzing	Test case generation without state of target
	Smart fuzzing	Test case generation with state of target
Symbolic execution	Online symbolic execution	Fork every time executor meet a branch
	Offline symbolic execution	Only resolve branch statements for one path
	Hybrid symbolic execution	Online Symbolic Execution until memory cap Offline symbolic Execution from saved point

2.2 Automated Vulnerability Patch

Binary hardening can be divided into OS level memory hardening techniques and compiler level binary enhancement techniques. First, the technique for improving the memory at the OS level is ASLR (Address Space Layout Randomization), DEP (Data Execution Prevention/Not Executable), ASCII-Armor. ASLR is a technique to randomize image-based values when programs are mapped to virtual memory. This is a security technique that prevents attackers from attacking because it is difficult for an attacker to grasp the memory structure of the target program. DEP/NX is a technique to prevent code from being executed in data areas such as stack and heap. This will restrict the execution authority on the stack or heap area to prevent attacks. ASCII - Armor is a technique for protecting the space of shared libraries by buffer overflow attack by inserting NULL (\ x00) bytes at the beginning of that address. The inserted NULL byte cannot reach the address.

The techniques that can be applied to the compiler are PIE (Position Independent Executable), SSP (Stack Smashing Protector) RELRO (relocation read only), PIE is similar to the ASLR provided by the operating system. The difference with ASLR is that ASLR applies random address to the stack area of the heap memory area and the shared library area of the heap memory area. PIE is a way to binary random randomize the logical address and to prevent overwriting attack by overwriting the SFP (saved frame pointer) by inserting a specific value (Canary) to monitor on the stack between

SFPs, especially buffers and SFPs. Buffer overflow attack overwrites SFP Attacker overwrites canary. Buffer overflow attacks are detected by monitoring the modulated canary. Finally, RELRO create a read-only ELF binary or data area for the RELRO process so that memory is not changed. There are two types of methods, depending on the state of the GOT domain, part RELRO and complete RELRO. A partial RELRO with a writable GOT domain can quickly use the entire RELRO without consuming resources, but it is vulnerable to attacks such as GOT overwriting. On the other hand, the complete RELRO of the read - only GOT domain consumes more resources and time than the partial RELRO, but you can use the GOT domain to prevent attacks (Table 2).

Table 2. Automated vulnerability patch method and protected area

Patch level	Patch method	Protected area
OS level	ASLR	Stack, heap, shared library
	DEP	Stack, heap
	ASCII-Armor	Shared library
Compiler level	PIE	Code
	SSP	Stack
	RELRO	GOT

3 Automated Vulnerability Root Cause Analysis

In order to clearly patch the vulnerability, we must correctly analyze the point that triggered this vulnerability and execute the patch. Various studies are under way to find places that cause vulnerabilities. In this section, we introduce the techniques of automated vulnerability root cause analysis classified into the following three categories of fault localization, pattern based analysis and taint analysis.

3.1 Fault Localization

Fault Localization is a technique of finding the location of this vulnerability by assigning weights using test cases. Fault localization can be divided into four types of similarity-based, statistics-based, artificial intelligence-based, program analysis-based. Similarity-based fault localization is a method that takes advantage of the frequency of execution of statements within the case of test success and failure. The Tarantula formula is representative, and means the ratio of failed cases divided by the percentage of failed cases added the percentage of normal cases [10]. Statistics based fault localization estimates the error location by measuring the probability of searching for a different path when searching for a path using conditional probability [11]. AI-based fault localization method is a method of searching a subgraph which is frequently appeared by converting a program into a behavior graph by graph mining technique. SVM is used to classify correct and incorrect execution [12]. Program-based fault localization technique measures the suspicion of a code block by calculating the edges between suspicious code blocks in the control-flow graph [13].

3.2 Signature Based Similarity Analysis

The signature-based similarity analysis is a technique to compare bug signatures with other software if there is same bug signature in target software. In the case of source code, there are many functions, such as function-based [14], instruction-based code clone and code similarity analysis, but there are many limitations in analyzing code similarity to binary. First of all, the biggest weakness is that the binary code generated for each CPU Architecture is different. In order to solve this problem, various studies have been carried out to operate on a multi-platform, analysis technique is changed according to whether a weak code pattern is analyzed based on an intermediate language, a weak code similarity analysis is compared with an object [15, 16]. Since the executable file to be generated is different, many researches have been conducted to construct an environment for pattern similarity analysis, which is possible in a multi-platform environment.

3.3 Taint Analysis

Taint analysis is a technique for analyzing how tainted input values flow from external inputs. The derivation of taint value is called 'taint propagation'. When a tainted memory or tainted register is accessed, it is judged that the area is tainted. However, in Taint analysis, there is a problem that is not tainted even though it is affected by the actual value. It is called under taint. In order to solve this problem, DTA++ suggests overcoming by offline analysis and online taint propagation technique [17]. These results cover a slightly smaller range compared to the previous study, DYTAN, which grasped all of the branch and covered a wider range than the existing taint tool [18]. In recent years, backward taint analysis technology has been introduced, which detects the location of errors by tracing contaminated values, and is now installed as a plug-in to Microsoft's Window debugger [19].

4 Conclusion

The number of vulnerabilities is increasing rapidly due to the development of new hacking techniques. However, time-consuming software analysis depending on vulnerability analyst make it difficult to respond to attacks immediately. In order to solve this, automated analytical techniques of various vulnerabilities are announced, but in order to respond positively, it is important to clearly grasp the root cause of the error and patch it. In this research, we have studied the tendency to clearly find the root cause of error. We will study techniques to classify this vulnerability by CWE via the vulnerability analysis and to find the point which caused the error fundamentally,

Acknowledgments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00184, Self-Learning Cyber Immune Technology Development).

References

1. U.S. National Vulnerability Database. http://cve.mitre.org/cve/
2. zzuf - Caca Labs. http://caca.zoy.org/wiki/zzuf
3. Peach Fuzzer. https://www.peach.tech/
4. Sulley. https://github.com/OpenRCE/sulley
5. Aitel, D.: An introduction to SPIKE, the fuzzer creation kit. In: BlackHat USA Conference (2002)
6. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: A taint based approach for smart fuzzing. In: IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 818–825 (2012)
7. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS, vol. 8, pp. 151–166 (2008)
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
9. Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., Candea, G.: Cloud9: a software testing service. ACM SIGOPS Operating Syst. Rev., 5–10 (2010)
10. James, A., Mary, J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 273–282 (2015)
11. Liblit, B., Mayur, N., Alice X.Z., Alex, A., Micheal, I.J.: Scalable statistical bug isolation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 15–26 (2015)
12. Chao, L., Long, F., Xifeng, Y., Jiawei, H., Samuel, P.M.: Statistical debugging: a hypothesis testing-based approach. IEEE Trans. Software Eng. **32**(10), 831–848 (2006)
13. Zhao, L., Lina, W., Zouting, X., Dongming, G.: Execution-aware fault localization based on the control flow analysis. In: Information Computing and Applications, ICICA, pp. 158–165 (2010)
14. Seulbae, K., Seunghoon, W., Heejo, L., Hakjoo, O.: VUDDY: a scalable approach for vulnerable code clone discovery. In: IEEE Symposium on Security and Privacy, pp. 595–614 (2017)
15. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: IEEE Symposium on Security and Privacy, pp. 709–724 (2015)
16. Feng, Q., Wang, M., Zhang, M., Zhou, R., Henderson, A., Yin, H.: Extracting conditional formulas for cross-platform bug search. In: ASIA CCS 2017 (2017)
17. MinGyung, K., Stephen, M.C., Pongsin, P., Dawn, S.: DTA++: dynamic taint analysis with targeted control-flow propagation. In: NDSS (2011)
18. James, C., Wanchun, L., Alessandro, O.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 196–206 (2007)
19. Weidong, C., Marcus, P., SangKil, C., Yanick, F., Vasileios, P.K.: RETracer: triaging crashes by reverse execution from partial memory dumps. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 820–831 (2016)