Towards Building a Generic Vulnerability Detection Platform by Combining Scalable Attacking Surface Analysis and Directed Fuzzing

Xiaoning Du(✉)

Nanyang Technological University, Singapore, Singapore
duxi0002@ntu.edu.sg

1 Introduction

Vulnerabilities are one of the major threats to software security. Usually, they are hunted by security experts via manual code audits, or with some automated tools like fuzzers (e.g., [1,5,12]) and symbolic execution (e.g., [4,7,10,13]), which can provide concrete inputs to trigger and validate the vulnerabilities. As fuzzy static scanners usually flag a list of potential vulnerable codes or functions with high rate of false positive, we deem them in the spectrum of attack surface identification approaches. The scalability of symbolic execution is extremely restricted by the path exploration problem and solver capability, which makes it not a preferable choice for large scale vulnerability detection. Coverage-based undirected fuzzing is hardly scalable and effective in general due to the large size of the program and the lack of good seeds to trigger various behaviors or executions. Faced with the fact that all existing static and dynamic detection tools are concerned with the trade-off problem between scalability and precision, a generic and scalable vulnerability detection platform is desirable.

As only a few vulnerabilities are scattered across a large amount of code, vulnerability hunting is a challenging task that requires intensive knowledge and skills and is comparable to finding "a needle in a haystack" [17]. Identifying potentially vulnerable locations in a code base is critical as a pre-step for effective vulnerability assessment. Metric-based techniques, inspired by bug prediction [11], leverage machine learning to predict vulnerable code at the granularity level of a source file. It cannot work well due to the severe imbalance between non-vulnerable and vulnerable code as well as the lack of features to reflect characteristics of vulnerabilities. Pattern-based use patterns of known vulnerabilities to identify potentially vulnerable code through static analysis. The patterns are formulated by security experts using their domain knowledge, e.g., missing security checks on security-critical objects [16], security properties [14], and vulnerability specifications [15]. Due to the requirement on prior knowledge of known vulnerabilities, it can only identify similar but not new types of vulnerabilities.

Among the automated assessment tools, directed fuzzing [5,8] stands out for its ability to reach a target program location efficiently and fuzz it effectively.

J. Sun and M. Sun (Eds.): ICFEM 2018, LNCS 11232, pp. 464–468, 2018.
https://doi.org/10.1007/978-3-030-02450-5_36

Experimentally fed with a limited portion of heuristically selected attack surface, AFLGo [5] is reported to outperform directed symbolic-execution-based whitebox fuzzing and undirected fuzzing. We believe its vulnerability-hunting power can be further boosted with wisely identified attack surface. Currently, the guiding in AFLGo is achieved just via power scheduling, which can be obtuse and insensitive. Much improvement can be done to make the guiding strategy more swift and intelligent.

In this study, we aim at combining attack surface identification and directed fuzzing for building a generic and scalable vulnerability detection platform.

2 Our Approach

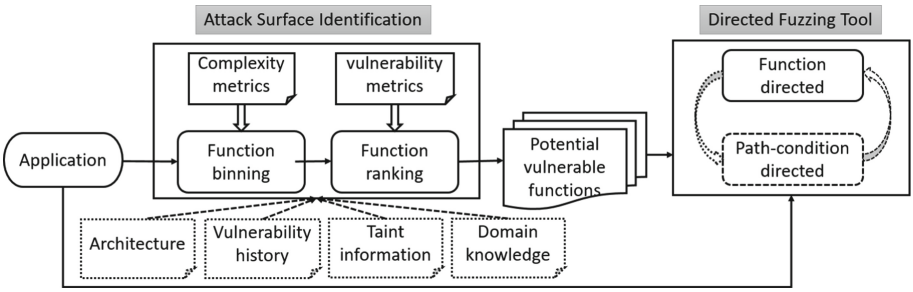

Fig. 1. An overview of the proposed framework

An overview of our proposed framework is shown in Fig. 1. Given an application's source code, the attack surface identification component is used to generate a list of potential vulnerable functions based on the complexity and vulnerability metrics of the application. These functions can be directly fed to the directed fuzzing tool as targets to confirm the vulnerability with concrete triggering input. Within the fuzzing tool, we use the function-directed fuzzing to reach the target function, and combine with path-directed method to penetrate the target function to trigger the vulnerability. Note that finished components of the framework are drawn with solid lines and explained below, and unfinished ones are drawn with dashed lines as explained in next section.

We have proposed and implemented a *generic*, *lightweight* and *extensible* framework, named LEOPARD, to identify attack surfaces at the function level through program metrics. LEOPARD does not require any prior knowledge about known vulnerabilities. It works in two steps by combining two sets of systematically derived metrics. Complexity metrics capture the complexity of a function in two dimensions: the control structures in the function, and the loop structures in the function. Vulnerability metrics reflect the characteristics of vulnerabilities in three dimensions: the constants, pointers, and coupling level of predicates in a function. Details about the metrics and some supplementary experimental

results are available at our website [2]. First, it uses complexity metrics to group the functions in a target application into a set of bins. Then, it leverages vulnerability metrics to rank the functions in each bin and identifies the top ones as potentially vulnerable. Experimental results on nine real-life projects have demonstrated that LEOPARD can cover 74% of vulnerable functions by identifying 25% of functions as vulnerable; and LEOPARD can outperform machine learning-based techniques. Based on the identified vulnerable functions in the current stable release of `PHP`, a security expert discovered six zero-day vulnerabilities.

For the directed fuzzing, we have integrated the attack surface identification framework with some off-the-shell directed fuzzing tools. We choose FOT [3], which is a versatile, configurable and extensible fuzzing framework. It provides a basic function-level directed fuzzing interface, requiring only a list of target functions. The initial evaluation of the combined approach of using attack surface identified by LEOPARD and feeding it to FOT demonstrates very encouraging results with tens of crashes and zero-day vulnerabilities identified in popular libraries like MJS, GNU bc, GNU diffutils, gpac, radare2, FLIF, libsass, libpff, liblnk and jsmn. For the path-condition directed fuzzing, we have developed the first penetration fuzzer by guiding the fuzzing to focus on the useful program executions related to the vulnerable code, which by only considering the statements with the positive effectiveness to the vulnerable code. The initial experiments has shown positive results on CGC benchmark with complicated program logics, where most existing fuzzers have failed. More investigation is needed to evaluate the efficiency of different combination strategies of the two directed fuzzing techniques, which have been further discussed in next section.

3 Future Work and Conclusion

3.1 Metrics Extension

The set of complexity and vulnerability metrics can be refined and extended, by adjusting scores of existing metrics or incorporating new metrics, to highlight interesting functions via capturing different perspectives. To this end, we have identified the following information to be vital to further improve our findings.

Taint Information. Leveraging taint information will help an analyst to identify the functions that process the external (i.e., taint) input.

Vulnerability History. In general, recently patched functions are straightforward attack surface due to the verified reachability, with considerable risks of incomplete patch or introducing new issues, but functions that are patched long before the release of the current version tend to involve no vulnerabilities.

Domain Knowledge. Domain knowledge can play a vital role in prioritizing the interesting functions for further assessment. Information such as the modules that are currently fuzzed by others can be used to refine the ranking. It is also

interesting to explore what information can be mined from mailing list, twitters and security blogs.

Architecture. Strong correlations between bug/vulnerability-prone files and architecture design flaws [6,9] can also be considered in to light up attack surface identification with some high level information.

3.2 Directed Fuzzing

To enhance the directed fuzzing for the effective usage of the potential vulnerable functions, we are looking at two directions. Firstly, we want to investigate how to combine the two directed fuzzing techniques into one holistic approach. Function-level directed fuzzing is good at reaching target vulnerable functions, however to trigger the vulnerability in the vulnerable function requires further penetration. Therefore we can conduct the directed fuzzing in two steps by invoking the two directed techniques sequentially. However, function-level directed fuzzing may stuck in some code to reach the target function due to the lack of low-level penetration in local path conditions. To address this, we need to invoke the path-condition directed fuzzing together with the function-level directed fuzzing. These phenomenons require a better interplay between the two techniques and dynamic scheduling of them based on the progress. We are planning to propose a runtime scheduler to orchestrate the two techniques dynamically.

Secondly, the metrics used to generated during the attack surface identification step is statically calculated, which may not be precise. Hence the ranking of vulnerable functions is less ideal. To address this, we can combine the fuzzer deeper with the metrics calculation and vulnerable function ranking so that we use the runtime information generated by fuzzer and adjust ranking of the vulnerable function dynamically. For example, this approach can directly remove the easily reachable functions with high vulnerability metrics hence improving the effectiveness of the approach.

3.3 Conclusion

This paper presented a generic framework for effectively finding vulnerabilities in source code level. The key idea is to combine the scalable static analysis and directed fuzzing to balance the trade off between scalability and accuracy.

References

1. American fuzzy lop. http://lcamtuf.coredump.cx/afl/ (2017)
2. Leopard. https://sites.google.com/site/leopardsite2017/ (2017)
3. FOT. https://sites.google.com/view/fot-the-fuzzer (2018)
4. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: ISSTA, pp. 12–22 (2011)
5. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: CCS, pp. 2329–2344 (2017)

6. Cai, Y., Xiao, L., Kazman, R., Mo, R., Feng, Q.: Design rule spaces: a new model for representing and analyzing software architecture. TSE (2018)
7. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: SP, pp. 725–741 (2015)
8. Chen, H., et al.: Hawkeye: towards a desired directed grey-box fuzzer. In: CCS (2018)
9. Feng, Q., Kazman, R., Cai, Y., Mo, R., Xiao, L.: Towards an architecture-centric approach to security analysis. In: WICSA, pp. 221–230 (2016)
10. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
11. Malhotra, R.: A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. **27**(C), 504–518 (2015)
12. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. In: NDSS (2017)
13. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
14. Vanegue, J., Lahiri, S.K.: Towards practical reactive security audit using extended static checkers. In: SP, pp. 33–47 (2013)
15. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: SP, pp. 590–604 (2014)
16. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: exposing missing checks in source code for vulnerability discovery. In: CCS, pp. 499–510 (2013)
17. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: predicting security vulnerabilities for windows vista. In: ICST, pp. 421–428 (2010)