2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

Finding Software Vulnerabilities by Smart Fuzzing

Sofia Bekrar
Grenoble University
Grenoble, France
sofia.bekrar@imag.fr

Chaouki Bekrar
VUPEN Security
Montpellier, France
bekrar@vupen.com

Roland Groz
Grenoble University
Grenoble, France
roland.groz@imag.fr

Laurent Mounier
Grenoble University
Grenoble, France
Laurent.Mounier@imag.fr

***Abstract*— Nowadays, one of the most effective ways to identify software vulnerabilities by testing is the use of fuzzing, whereby the robustness of software is tested against invalid inputs that play on implementation limits or data boundaries. A high number of random combinations of such inputs are sent to the system through its interfaces. Although fuzzing is a fast technique which detects real errors, its efficiency should be improved. Indeed, the main drawbacks of fuzz testing are its poor coverage which involves missing many errors, and the quality of tests. Enhancing fuzzing with advanced approaches such as: data tainting and coverage analysis would improve its efficiency and make it smarter. This paper will present an idea on how these techniques when combined give better error detection by iteratively guiding executions and generating the most pertinent test cases able to trigger potential vulnerabilities and maximize the coverage of testing.**

Keywords: fuzzing, testing, software vulnerabilities

I. INTRODUCTION

Many malicious attacks are based on the existence of vulnerabilities. When they are exploitable, these security flaws allow an attacker to break into a system. Therefore, it is crucial to identify them. Currently, several solutions can be used to spot such vulnerabilities. Although none can be guaranteed to find all of them, some of them give good results. A main technique and a widely established is Fuzzing. Fuzzing is an interface testing method aiming to detect vulnerabilities in software without access to application source code.

Although many academic and applied researches have addressed the software security issue in recent years, hundreds of new vulnerabilities are discovered, published or exploited each month. The existence of such real threats within enterprise and institution infrastructures, systems and networks, could affect their entire corporate information technology domain. The ability to find these flaws might be a witness for a need for improvement.

Therefore, it becomes essential to improve the efficiency of existing approaches.

Our goal is to propose an innovative method for vulnerability discovery to manage security risks by finding new and unpatched vulnerabilities before they can be exploited by attackers. The resulting tool will target both file processors (for DOC, JPEG, PDF format) and network protocols (TCP/IP). Since the entire source code of these targets is rarely available, we plan to use assembly level analysis, based on disassembly and reverse engineering platforms. We also assume a partial knowledge of the target because input (files, packets) structures are generally known, but not the implementation details.

The paper is organized as follows: Section 2 introduces fuzzing, the technique on which our work is based. Section 3 presents an approach for error identification. It also describes each step and its role in the process. Section 4 explains the open problems and challenges and finally concludes the paper.

II. FUZZING

Many definitions of fuzzing [1], [2] exist in the literature. All can be summarized in one definition: Fuzzing is a security testing approach based on injecting invalid or random inputs into a program in order to obtain an unexpected behavior and identify errors and potential vulnerabilities.

There is no single fuzzing method because fuzzing has no precise rules. Its efficiency depends on the creativity of its author.

Its key idea is to generate pertinent tests able to crash the target and also to choose the most appropriate tools to monitor the process. Fuzzed data generation can be performed in two ways. They can be generated randomly by modifying correct data without requiring any knowledge of the application details. This method is known as Blackbox fuzzing and was the first fuzzing concept. On the other hand, Whitebox fuzzing consists in generating tests assuming a complete knowledge of the application code and behavior. A third type is Graybox fuzzing which stands between the two methods aiming to take advantages of both. It uses only a minimal knowledge of the behavior target. It is thus the most appropriate method according to our purposes.

III. PROPOSED APPROACH

In this section we will present our method illustrated by the tool architecture depicted on Figure 1. The figure will be followed by a brief description of the main techniques used and the motivations that justify their use. As shown in Figure 1 our fuzzer can be divided into six parts.

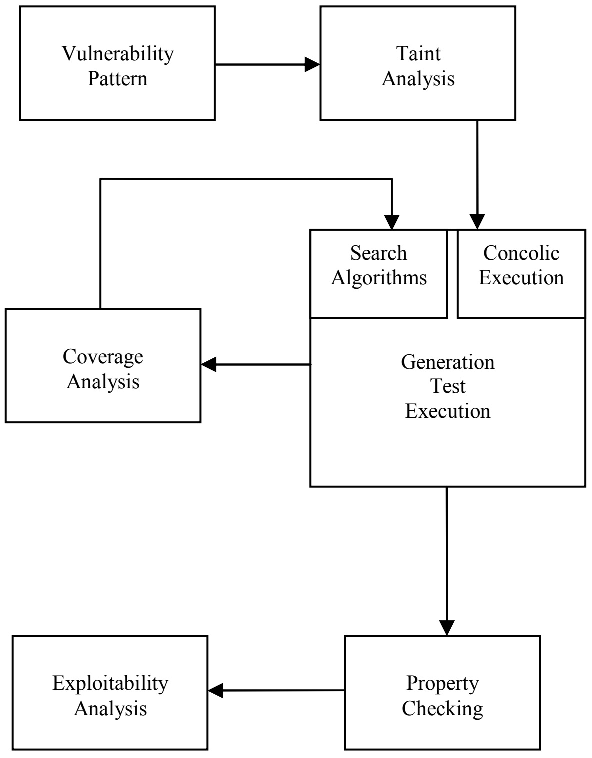

Figure 1. Proposal.

We propose to start by defining vulnerability patterns at the assembly level. These patterns would facilitate the identification of interesting functions within the binary. A taint analysis will be applied to the binary, taking into account information obtained during the previous step. The potentially dangerous data will be marked as tainted and their propagation at execution will be tracked. A test generation will be then applied using concolic execution and search algorithms in order to audit the most dangerous paths of the application under test and generate better quality tests for each new execution as described in Section C. After the generation and execution of tests, the fuzzing effectiveness will be evaluated through coverage analysis techniques. A method to detect faults will be implemented and used at execution. The potential exploitability of each detected vulnerability will be evaluated.

A. Vulnerability Pattern

The first step is to identify potentially vulnerable sequences of code in the binary in order to get an idea of the most dangerous parts in the binary that should be tested. The idea is to define vulnerability "patterns" based on already discovered vulnerabilities and VUPEN Security expertise in finding and exploiting security vulnerabilities in binaries. A vulnerability pattern represents a model at assembly level that can potentially be the cause of a fault, especially a memory corruption. An example of a vulnerable function at source code is *strcmp* where function parameters can be controlled by user. This common situation allows accessing memory in an insecure manner that can generate a buffer overflow.

B. Taint Analysis

The taint analysis approach [3] has gained momentum these last few years. One proof of that is its continuously increasing use and also the existence of many frameworks. It generally consists in marking data originating from untrusted sources, like network and user inputs, as tainted. It can be used for example to track the propagation of tainted data and check when tainted data is used in dangerous ways, for example, to overwrite return address.

Data tainting [4] would improve fuzzing by using information about the vulnerability, such as path execution, to select the most promising test sequences able to trigger potential vulnerabilities and restrict the test space.

Many taint analysis tools exist. Dytan [5] is a dynamic taint analysis framework for Windows and Linux which instrument binaries without recompilation or access to sources. MyNav [6] is a plugin for the IDA Pro disassembler [7] created to help reverse engineers in their tasks. It helps to dynamically analyze programs. It can be used in this case although it is not a taint analysis framework. Its functions include finding paths between interesting functions and inputs.

C. Test generation

Generating the fuzzed data is the most important step in the fuzzing process. This step can be conducted in different ways. Fuzzed data can be generated by applying mutations on existing data or by modeling the target (file processor/ protocol). A model can be created using the source code, learning methods, or from target specifications.

In order to trigger particular paths that might reveal faults, the choice of test values is crucial. Our goal is to generate the most pertinent test cases. For that, concolic execution and search algorithms could be used. Concolic execution [8] is a systematic path exploration technique based on executing the program using concrete values as inputs; collect constraints on these inputs negate and solve constraints to explore new path conditions. This exploration strategy has been implemented in several fuzzing tools such as SAGE [9] developed by P. Godefoird at Microsoft, or Fuzzgrind [10]. Instrumentation tools such as PIN [11], VALGRIND [12], and DYNAMORIO [13] are required to generate constraints. Once a constraint is generated it must be solved using a constraint solver (e.g., STP [14]). Concolic execution allows us to explore new execution paths by generating new inputs.

Search algorithms, such as genetic algorithms [15] could be used. The quality of tests would be improved after each execution thanks to the evolution of tests population by applying operations (mutation, crossover…) to the initial population.

We plan to apply a taint analysis at the assembly level to gather information about potentially dangerous data; the program will be then executed with concrete values to generate inputs that explore new paths and select the most promising test sequences able to trigger potential vulnerabilities taking into account information gathered in the previous step. After the first execution, we apply search algorithms such as genetic algorithms to the previous generated population of inputs in order to improve the test generation process thanks to crossover and mutation operations that, when applied to a population of tests, generate better quality tests for the next execution.

D. Coverage analysis

Once the fuzzed data generated and the target is executed with these new inputs, the effectiveness of the fuzzing process must be evaluated using code coverage techniques. Fuzzing evaluation is measuring how well the program is tested and identifying the modification necessary to expand the coverage. The idea is to implement a technique so that the binary execution is traced. It consists in tracing the basic blocks execution using PyDbg [16] to set breakpoints in each basic block and check if it is hit in order to evaluate the coverage. IDA2SQL [17] helps us in this task by exporting information about the program (basic blocks, functions …) to a MySQL database.

Estimating the efficiency of the fuzzing helps us trying to maximize it.

E. Property checking

Monitoring the program for possible faults at execution is necessary in the fuzzing process. Disassembling and debugging features facilitate such monitoring if they are attached to the program at execution.

We propose not only to monitor for exceptions but also to systematically identify violations at runtime to avoid hidden vulnerabilities. Detecting the occurrence of a fault in such way improves fault-detection capabilities.

Working at assembly level makes this task harder because of assembly programs complexity.

F. Exploitability pattern

If a fault is found in the last step, a process must be performed on it to determine its potential exploitability. Analyzing information gathered from the binary like the stack or the heap content when a vulnerability is triggered would be an interesting direction for evaluating its potential exploitability. Currently, there is a variety of tools, such as Miller's binary analysis tool BitBlaze [18], to help determine whether a crash is caused by a potentially exploitable vulnerability; however these tools do not provide reliable information.

IV. Conclusion

The theoretical approach proposed above would give interesting results; nevertheless several problems have to be solved. The main challenge is to define vulnerability patterns that can be used for detecting possible security vulnerabilities. Many researchers have worked on defining such patterns and have faced many problems. We have not only to face those open problems, but also the difficulty of defining patterns according to Assembly instruction. This makes the task more difficult than working on usual source-code. Binary analysis complexity is due to the low-level semantics, especially lack of functions. Another difficulty is dealing with registers and no with types. Diversity and complexity of Assembly instructions is also a difficulty.

References

[1] M. Sutton, A. Greene, and P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison–Wesley Professional, United States, 2007.

[2] A. Takanen, J. DeMott, and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, ARTECH HOUSE, 2008.

[3] J. Newsome and D.X. Song, "Dynamic taint analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software", In NDSS. The Internet Society, 2005.

[4] E.J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)", In IEEE Symposium on Security and Privacy, IEEE Computer Society, 2010, pp. 317-331.

[5] J.A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework", In David S. Rosenblum and Sebastian G. Elbaum, editors, ISSTA, ACM, 2007, pp. 196-206.

[6] MyNav. http://code.google.com/p/mynav/

[7] The IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro/

[8] J.C. King, "Symbolic execution and program testing". Commun. ACM, 1976, pp. 385-394.

[9] P. Godefroid, M.Y. Levin, D.A. Molnar, "Automated Whitebox Fuzz Testing", in Proc. Network and Distributed System Security Symp. (NDSS 2008), The Internet Society, Feb 2008, pp. 151-166.

[10] Fuzzgrind: an automatic fuzzing tool. http://seclabs.org/fuzzgrind/

[11] C. Luk, R.S. Cohn, R. Muth, H. Patil, A. Klauser, P.G. Lowney, S. Wallace, V.J. Reddi, and K.M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", in PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190-200.

[12] DynamoRio: Dynamic Instrumentation Tool Platform. http://code.google.com/p/dynamorio/.

[13] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", In Jeanne Ferrante and Kathryn S. McKinley, editors, PLDI, ACM, 2007, pp. 89-100.

[14] STP Constraint Solver.

http://sites.google.com/site/stpfastprover/STP-Fast-Prover.

[15] M. Last, S. Eyal, A. Kandel, “Effective Black-Box Testing with Genetic Algorithms”, In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, Haifa Verifcation Conference, volume 3875 of Lecture Notes in Computer Science, Springer, 2005,pp. 134-148. https://www.research.ibm.com/haifa/Workshops/verification2005/papers/testing/ibm05 ga2.pdf.

[16] PaiMei. http://pedram.redhive.com/PyDbg/docs/.

[17] ida2sql: exporting IDA databases to MySQL: http://blog.zynamics.com/2010/06/29/ida2sql-exporting-ida-databases-to-mysql/

[18] C. Miller, “Crash analysis with BitBlaze”, at BlackHat USA,2010.