Automatic Software Vulnerability Detection Based on Guided Deep Fuzzing

Jun Cai
The Academy of Equipment, Beijing, China
cjgfkd@163.com

Jinquan Men
61541 troops, Beijing, China
Lss_633@126.com

Shangfei Yang
Naval Academy of Armament, Beijing, China
lunaryxp@126.com

Jun He
The Academy of Equipment, Beijing, China
hejun0123@sina.com

Abstract—**Software security has become a very import part of information security in recent years. Fuzzing has proven successful in finding software vulnerabilities which are one major cause of information security incidents. However, the efficiency of traditional fuzz testing tools is usually very poor due to the blindness of test generation. In this paper, we present Sword, an automatic fuzzing system for software vulnerability detection, which combines fuzzing with symbolic execution and taint analysis techniques to tackle the above problem. Sword first uses symbolic execution to collect program execution paths and their corresponding constrains, then uses taint analysis to check these paths, the most dangerous paths which most likely lead to vulnerabilities will be further deep fuzzed. Thus, with the guidance of symbolic execution and taint analysis, Sword generates test cases most likely to trigger potential vulnerabilities lying deep in applications.**

Keywords-software vulnerability detection; fuzzing; taint analysis; symbolic execution

I. INTRODUCTION

Software security has become a very import part of information security in recent years. Vulnerabilities are one of the root causes of security problems. Once they are exploited by attackers, they may cause serious damages. Therefore, vulnerability detection technology is gaining more and more attention in the field of information security.

Fuzzing [1-5] is a traditional vulnerability detection technique. It was first proposed and used by Barton Miller in 1989. Although it was invented more than 20 years ago, it does not obsolete and is still an important and commonly used method. The idea behind fuzzing is very simple: generating malformed inputs and feeding them to an application; if the application crashes or hangs, a potential vulnerability is detected. There are some excellent fuzzers:

Zzuf [6] is a transparent application input fuzzer. It works by intercepting file and network operations and changing random bits in the program's input. Zzuf's main purpose is to make things easier and automated.

Sulley [7] is an actively developed fuzzing engine and fuzz testing framework consisting of multiple extensible components. The goal of the frame work is to simplify not only data representation but to simplify data transmission and instrumentation.

Peach [8] is a smart fuzzer that is capable of performing both generation and mutation based fuzzing. One distinctive high level concept of peach is modeling. Peach operates by applying fuzzing to models of data and state.

Although these standard fuzzers has been proven successful in finding vulnerabilities, however, with the development of software development technology and the strengthening of software security testing, they become more and more inefficient for the vulnerability detection of modern software. This is mainly due to the blindness of test generation.

In this paper, we present an automatic fuzzing system named Sword to tackle the above problem. The key idea of our approach are (1) taking the advantage of symbolic execution to search as more execution paths of the target problem as possible, thus improving the code coverage;(2) taking the advantage of taint analysis to check each execution path, the path-dependent taint information can guide the fuzzer to generate pertinent test cases, thus improving the efficiency of fuzzing;

This paper is organized as follows: Section 2 introduces vulnerability detection techniques and the motivation of our approach. Section 3 describes our approach in detail. Section 4 explains the open problems and challenges and finally concludes the paper

II. OVERVIEW

In this section, we first analyze the advantages and disadvantages of the currently most commonly used software vulnerability detection techniques, and then describe the motivation of our approach.

A. *Software vulnearblity detction techniques*

Currently, there are three major vulnerability detection techniques: fuzzing, symbolic execution and taint analysis. Obviously, none of them is almighty, each technique has its advantages and disadvantages.

Supported by National High Technology Research and Development Program of China (2012AA012902)

The advantage of fuzzing is that it is simple and practical and can detect real bugs directly. However, traditional fuzzing technique has a well-known common drawback: it is often ineffective and has poor code coverage in breadth because most generated inputs are rejected at the early stage of program running or just go around a single execution path.

Symbolic execution [9-11] is a program analysis technique that uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. It maintains a path condition that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Test generation is performed by solving the collected constraints using a constraint solver. The advantage of symbolic execution is that it can theoretically find all the possible execution paths of a program by collecting and solving constraints for each path. By the aid of symbolic execution, we can get good code coverage in software testing. However, symbolic execution has three bottlenecks: Path-explosion, a program especially a large-scale program may have an extremely large number of paths; Path-divergence, it is challenging to compute precise path constraints for real-world programs; Complex-constraint, a constraint solver may fail to find solutions to complex but satisfiable path constraints such as constraints involving non-linear operations.

Taint analysis [12-14] is an emerging program analysis technique which has been widely used in many fields of information security in recent years. The main idea behind taint analysis is that any variable that can be modified (directly or indirectly) by the user can become a security vulnerability (the variable becomes tainted) when a tainted variable is used to execute dangerous commands. The advantage of taint analysis is that it offers the capabilities to detect most of the input validation vulnerabilities with a very low false positive rate. Taint analysis certainly have shortcomings. It is generally suffering from slow execution, and the problems are detected only for the executions path that have been executed until now (not for all executable paths) which can lead to false negatives.

B. Motivation

Vulnerability is generally a vulnerable sequences of code contain some dangerous functions. It may lie anywhere in the program, but usually hide in the depth of the program or somewhere easy to neglect. Therefore, if we can't let the program run deep enough or walk through the paths seldom executed, we will not be able to find vulnerabilities. Figure 1 illustrates the execution paths of a program visually. There are many branch points in the execution process of a program. If we describe the program execution paths as a tree, these branch points are just the tree's nodes. In Figure 1, each small circle represents a branch point, the node "1" is the tree's root or the first branch point at the entrance of the program. Vulnerabilities tend to lie in the top of the tree, for example, supposing that there is a vulnerability lying near the node "8", if we divide the program execution tree into layers, the node "8" is at the eighth layer of the tree. If we can't let the program run at least deep than layer 8, we will never find the vulnerability.

Our motivation is to fuzzing deep with guidance to directly find the vulnerabilities lying in the depth of the target program. We use symbolic execution to search the frontier nodes of various different paths, then use taint analysis to check these paths, and finally choose the most potential paths that may lead to the vulnerability (as show in Figure 1) to fuzzing deep to trigger it.

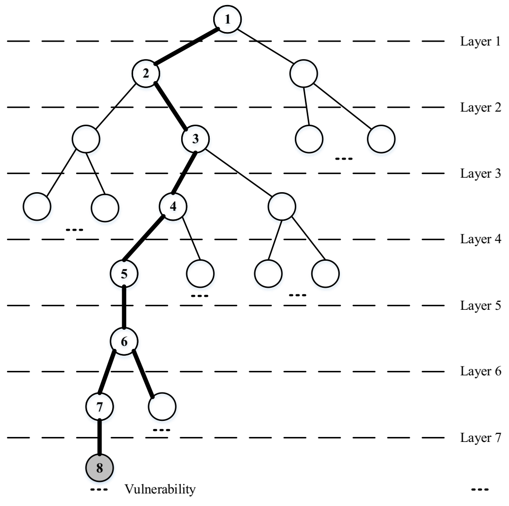

Figure 1. Program execution paths

III. DESIGN

In this section, we first give an overview of Sword and then describe its detailed design.

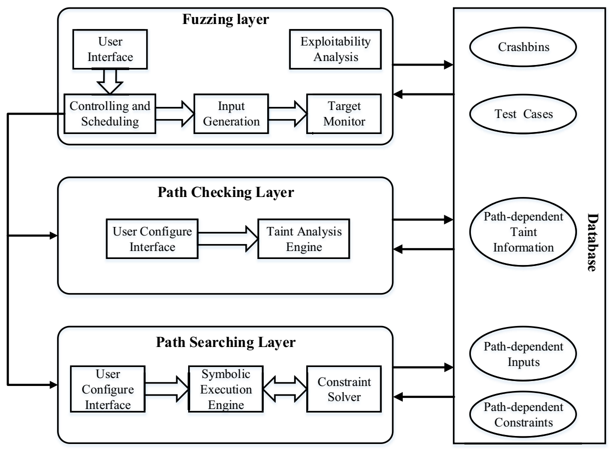

Figure 2. An overview of Sword

Figure 2 shows an overview of Sword. As show in Figure 2, Sword is divided into four major parts: path searching layer, path checking layer, fuzzing layer and database. Given a binary program P to be test, Sword works as follows: (1) using symbolic execution to search the possible execution paths of P and storing the Path-dependent inputs and constrains into the database;(2)using taint analysis to check each execution path and storing the path-dependent taint information into the database; (3)using Parma Polyhedra Library (PPL) [15, 16] to quickly generate preconditioned test cases; (4) the whole process is automatically controlled and scheduled by a module named the controlling and scheduling module.

A. Path Searching Layer

The path searching layer contains three modules: user configure interface, symbolic execution engine and constraint solver. The goal of this layer is to find all the possible execution paths of the target program within a set range.

1) User configure interface

Due to the path-explosion problem, it is difficult to symbolically execute a significantly large subset of all program paths. In reasonable time, only a small subset of all paths can be symbolically executed for symbolic execution of each program path may cost high computational overhead. Thus, we will perform symbolic execution in a certain scale which can be configured via the user configure interface.

We can set a threshold for how many layers depicted in figure 1 to take symbolic execution. For example, if we set the threshold to 5, then the symbolic execution engine will find all the paths under layer 5.

2) Smbolic execution engine

The symbolic execution engine maintains a symbolic state and a symbolic path constraint set (or formula) PC. It starts with a user given seed input. PC accumulates constraints on the inputs that trigger the execution to follow the associated path. An execution path can be represented by its PC. At every conditional statement if (e) S1 else S2, PC is enlarged with new constraint. A fresh path condition PC' is created and initialized to $PC \wedge \neg\sigma(e)$ ("else" branch) and PC is updated to $PC \wedge \sigma(e)$ ("then" branch), Note that unlike in concrete execution, both branches can be taken, resulting in two execution paths [11]. When the number of different constrains in PC reaches the threshold, the symbolic engine stopped accumulating new constrains. The path-dependent constraints will be sent to the constraint solver and stored into the database simultaneously.

3) Contraint solver

A Constraint solver is a decision procedure to solve constraints. The ability of constraint solver determines the symbolic execution engine's ability to find paths. STP [17] and Z3 [18] are two representative constraint solver. STP is aimed at solving constraints generated by program analysis tools, theorem provers, automated bug finders, intelligent fuzzers and model checkers. Z3 is a high-performance theorem prover being developed at Microsoft Research which supports Windows, OSX, Linux and FreeBSD. The input to constraint solver is a path-dependent constraint, if the constraint is satisfiable, it will generate a variable assignment which we call path-dependent input and store the assignment into the database.

B. Path Checking Layer

The path checking layer contains two modules: user configure interface and taint analysis engine. The goal of this layer is to check the execution paths found in the previous layer and generate path-dependent information to guide the fuzzer to generate pertinent test cases.

1) User configure interface

There are two taint analysis approaches: one is static taint analysis which is performed mostly at source level; the other is dynamic taint analysis which is performed during the execution of target program. We perform dynamic taint analysis in Sword.

Before performing dynamic taint analysis, it is necessary to do some configuration: (1) Taint sources, what data to tag and how to tag it (2) Propagation Policy, how tags should be propagated at runtime (3) Taint sinks, where and how tags should be checked.

2) Taint analysis engine

It is of great importance to perform taint analysis according to the features of vulnerabilities. Take integer overflow vulnerabilities for example, they usually have the following features [19] (1) Untrusted Source. (2)Incomplete or improper sanitization checks of tainted data. (3)Using an overflowed value in some sensitive operation. For example, the overflowed value is used in memory allocation functions (e.g., malloc, calloc) as a size argument, and it usually results in an insufficient memory allocation, which may eventually become an attacker's springboard to a buffer overflow.

We use the path-dependent inputs as taint source to drive the target program run along different paths. The taint analysis engine track the propagation of tainted data along the execution path and check whether tainted data is used in some dangerous ways. The most important information we want to get is to identify the sinks and its related sources (input offsets), we call this path-dependent taint information which will be stored into the database and treated as important heuristic information for fuzzing test case generation.

C. Fuzzing Layer

The fuzzing layer contains five modules: user interface, controlling and scheduling, input generation, target monitor, and exploitability analysis.

1) User interface

We would like to provide users with a graphical interface through which users can select a target program, set some important parameters such as the number of test cases generated for each path, the threshold of symbolic execution engine etc., and finally start fuzzing.

2) Controlling and scheduling

The controlling and scheduling module is responsible for controlling and scheduling the entire fuzzing process. It takes the database as an intermediary to coordinate the work between the three layers.

3) Input generation

In the path searching layer, the symbolic execution engine has found the frontier nodes of each path, then how can we use fuzzing to test further deep? The generated test cases must satisfy the path-dependent constraints first. To do this, we can invoke the constraint solver. However, this method is not efficient because every time you generate a test case you have to invoke the constraint solver once. Reference [20] has proposed a more efficient method.

In reference [20], the author uses PPL to quickly generate path-preserving test cases. PPL is a modern C++ library for the manipulation of numerical information that can be represented by points in some n-dimensional vector space. For instance, one of the key domains the PPL supports is that of rational convex polyhedral. The key idea of the method is: transforming a path's path-dependent constraints to a Z-Polytope which defining the input space of the path, then randomly choosing a point inside the Z-Polytope as a test case. With the assistance of Z-Polytope abstraction, this method can derive and produce a large number of random inputs that will respect the path constraints. The bottleneck of this method is that it is not able to handle non-linear predicates. Therefore, a more reasonable solution is to combine this method with constraint solver.

Our goal is to quickly generate as many path-preserving test cases as possible. The more test cases we generate, the deeper we test. Meanwhile, the path-dependent taint information should be taken into account in the generation of test cases to further improve their efficiency, only the security-related data will be fuzzed.

4) Target monitor

The target monitor is aimed to monitor the execution of program, automatically detect abnormities and determine which test case or test cases bring on the abnormity. There are three abnormity detection methods: the simplest method is response analysis, a more advanced method is automated debugger motioning, and the most advanced method is dynamic binary instrumentation (DBI). Currently, a good solution for DBI is valgrind [21], there are valgrind tools that can automatically detect many memory management and threading bugs.

5) Exploitability Analysis

If a bug is detected by the target monitor, Exploitability Analysis must be performed on it to determine whether it can be further exploited. This step is usually performed manually.

D. Database

The database is used to store all the media data and the results produced in the whole process of fuzzing such as path-dependent constraints, path-dependent inputs, path-dependent taint information, test cases, crashbins and so on. It is also a link connecting the various modules.

IV. Conclusion

In this paper, we propose an automatic software vulnerability detection technique which combines fuzzing with symbolic execution and taint analysis. This theoretical approach would give interesting results. Currently, we have implemented the taint analysis engine. Nevertheless several problems have to be solved. The main challenges is to quickly generate preconditioned test cases. To build reasonable vulnerability models 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] V. Ganesh, T. Leek, M. Rinard, "Taint-based Directed Whitebox Fuzzing", ICSE'09, May 16-24, 2009, Vancouver, Canada, pp. 474–484.

[3] A. Takanen. "Fuzzing: the past, the present and the future", 2009. URL http://actes.sstic.org/SSTIC09/Fuzzing-the_Past-the_Present_and_the_Future/SSTIC09-article-A-Takanen-Fuzzing-the_Past-the_Present_and_the_Future.pdf.

[4] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST '11. 2011, pp. 427–430.

[5] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A Taint Based Approach for Smart Fuzzing," in Proceedings of the 2012 Fourth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST '12. 2012, pp. 818–825.

[6] Zzuf. http://caca.zoy.org/wiki/zzuf.

[7] Sulley. https://github.com/OpenRCE/sulley

[8] Peach. http://peachfuzzer.com/

[9] S. Anand, "TECHNIQUES TO FACILITATE SYMBOLIC EXECUTION OF REAL-WORLD PROGRAMS", Georgia Institute of Technology, August 2012

[10] C. Cadar, P. Godefroid, S. Khurshid, "Symbolic Execution for Software Testing in Practice–Preliminary Assessment", ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA.

[11] C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later", Communications of the ACM, 2013, 56(2): 82-90.

[12] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In Proceedings of the Network and Distributed System Security Symposium (NDSS 2005).

[13] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework", ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

[14] E. J. Schwartz, T. Avgerinos, D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)", in the Proceedings of the 2010 IEEE Symposium on Security and Privacy.

[15] R. Bagnara, P. M. Hill, and E. Zaffanella, "The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems", Science of Computer Programming, 2008, 72(1):3-21.

[16] The Parma Polyhedra Library (PPL), http://bugseng.com/products/ppl/

[17] STP, http://people.csail.mit.edu/vganesh/STP_files/stp.html.

[18] Z3 theorem prover, http://z3.codeplex.com/.

[19] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution", In Proceedings of the 16th Network and Distributed System Security Symposium (NDSS' 09), San Diego, CA, February 2009

[20] Brian S. Pak, "Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution", School of Computer Science Carnegie Mellon University, May 2012

[21] Valgrind, http://valgrind.org.