Mid-Term
Binary Function Fuzzer (BFF)
for CS5371 Soft Test for Mobile & Emb Sys

Jeremy Solmonson
School of Security Engineering
University of Colorado at Colorado Springs
Colorado Springs, CO 80922
Email: jsolmons@uccs.edu

***Abstract*—Vulnerability mitigation is essential for application development organizations. Without the ability to timely find and mitigate software vulnerabilities, computer attacks will use the underlying weakness to take advantage of a computer system. Finding vulnerabilities as early as possible within the Software Development Life Cycle (SDLC), is the most cost efficient approach for application security. Unfortunately, most vulnerability analysis tools assess the software against the entire system. As a result, entire program must be completely written before the first vulnerability assessment is conducted. To integrate vulnerability identification earlier within the SDLC, we developed the Binary Function Fuzzer (BFF) tool. This tool allows software developers to integrate vulnerability identification within their unit tests and subsequently fixing the underlying issue. Additionally by bringing vulnerability identification to the individual developer enables faster mitigation strategies and stronger code ownership.**

I. Introduction

The process of reducing the vulnerabilities within software is extremely time intensive because programs are becoming larger and more complex. These complex program require extra scrutiny as the underlying logic may not be well known or well tested. Without through testing, vulnerabilities may be overlooked and released into the final program. Additionally, commercial software developers are rewarded for releasing their products as quickly as possible, which can limit the rigorous testing needed to reach hard-to-find vulnerabilities.

Exploitable vulnerabilities are extremely concerning for software development organizations. An exploitable vulnerability allows a computer attacker to use the underlying software in unintended methods. An attacker that can insert their own instructions as a substitute for the original programs is said to achieve Remote Code Execution (RCE). This can be considered the worst type of vulnerability as it allows the attack to execute their instructions up to the privileges of the application. For system services or processes that run with elevated permission, a RCE vulnerability allows the attacker complete control of the system. Testing for vulnerabilities on a continual basis reduces the potential for severely exploitable vulnerabilities.

To meet the organizational goals of balancing time with software quality, testing process is integrated into the earliest stages of software development known as unit tests. Unit tests allow the developer to ensure their features work as intended. However, unit tests do not usually include vulnerability testing. The problem with testing for vulnerabilities at the unit level is the time required to perform repetitive tests and observing unintended behavior. For large programs, execution from beginning to end can be severely time intensive.

To save time during testing, developers usually write units tests to ensure the successful functionality of their code. Writing tests that perform multiple failures is an after thought or non-existent practice. However, by writing tests that ensure a fault results in graceful program shutdown should be an integral part of software development. By augmenting the testing procedures within a unit level fuzzer allows the developers to identify vulnerabilities in a timely manner. In this paper, we discuss a new program called Binary Function Fuzzer (BFF), that can be used at the unit test level to search for vulnerabilities in individual units of code. This allows security testing to be integrated at the lowest level of software development. By assisting software developers as they write the code, security vulnerabilities can be identified and fixed earlier within the SDLC. The final software product will have reduced vulnerabilities.

The problem with fuzzing unit inputs is their unreliability due to manipulations by previous blocks of code. For instance, if a program first checks the user input for non-ascii characters and ensures the length is less than a maximum amount, then these characteristics are followed into the subsequent execution blocks. A unit level vulnerability fuzzer should observe these constraints and inject inputs that are consistent with the input the program would receive.

Continually fuzzing inputs into individual functions poses environmental challenges. If the execution environment depends on specific variables, then the program needs to maintain the state of those variables in further testing. Take a file for instance, if the function enters with the current file pointer at a specific location within the file, the variable should returned to the same spot after the test completes. If the pointer is not returned, the future tests may be unreliable. This same logic is true for dynamically allocated memory, the current stack, and

registers. Any block level test needs to be aware of saving the non-input variables for further execution.

The BFF tool solve the above problems by providing a fuzzing capability that is integrated into unit level testing. This reduces the overall time required to test the system as the individual blocks of code are tested during development. Additionally, only the code up to the current function is analyzed. Alternative execution paths are not assessed. This allows the developer to better understand their integration into the current program. Further, by using a control flow graph with taint analysis, the BFF tool is able to identify input constraints and pass those to the subsequent fuzzer.

This paper provides the following contributions:

- Introduces the BFF program for fuzzing individual functions within the resulting binary
- A method that enables software developers to incorporate vulnerability identification within unit tests

II. What you need to know for the problem? (Background)

Below is a quick list of topics that is needed for background knowledge.

- Taint Analysis
- Function Call Interception
- Concolic testing / Symbolic Execution
- Control Flow Graph
- Dynamic Analysis
- Static Analysis

The main idea for this paper is to fuzz a select portion of a target program. By fuzzing a subset of the entire program, vulnerabilities can be identified in a deeper portion of the code. Usually fuzzers that locate hard to find, or deep, bugs can only run a few individual programs [cite]. The scalability of these tools can be overcome by analyzing the resulting file.

To abstract a portion of the program into a subset of the program requires Function Call Interception (FCI) [cite]. This allows a second program to use instrumentation by substituting, or modifying, the underlying main program. By modifying the main program, a developer can gain deep insight into a portion target program behavior. FCI allows the developer to substitute instructions or data variables to alter the main programs execution. While used for mostly debugging purposes, this can be extended to fuzzing.

Fuzzing is injecting semi-valid inputs into a target program with the intent to find bugs. This technique has been an area of recent research [cite] to fund unknown vulnerabilities. One of the problems with fuzzing is executing deep into the program to find hard to reach bugs. This issue has been partly solved with symbolic execution and constraint solvers[cite]. However, the problem still remains due to resource exhaustion. As programs become larger and more complex, more resources are needed to fuzz deeper into the program. One method to overcome this challenge is to allow the developer to identify areas of interest to fuzz. By fuzzing a portion of the program, the time needed to test can be significantly reduced. Further, this technique can be integrated into the software development process. Fuzzing a software feature as it becomes available will lead to higher quality coding practices.

To assist the developer further, taint analysis is able to identify user inputs and mark, or taint, them as untrusted. Following these variables within a program can identify execution paths that leads to user triggered bugs. The execution paths can be traced using static or dynamic analysis. Static analysis allows the tester to view the underlying program logic, commonly through a disassembler. The program logic can be further analyzed through a Control Flow Graph (CFG). The combination of static analysis and a CFG allows the user to view program execution through the basic blocks of code. As an analogy, the static analysis CFG view provide a road-map of the underlying program, while dynamic analysis drives down the road map. Dynamic analysis assesses a program during execution and provides relevant information back to the user. During dynamic analysis a program can be halted, modified by changing memory contents or registers, and then continue executing. This allows a user to assess different combinations to trigger function execution. By combining a control flow graph and tainting the input variables, a user is able to visually identify areas of potential concern within an application. Then using dynamic analysis, these input paths can be executed and observed in real time to provide further information.

The one issue with dynamic analysis is finding the right combination of inputs to execute a specific path. To overcome this challenge, symbolic execution with constraint solvers is used to identify inputs for path execution. By using a constraint solver to find the conditions for branch execution, a single execution path can be identified. This enables a repeatable process to narrow down a source of program feature. One tool in recent research that performs this activity is Angr [cite]. Angr is a binary analysis tool that is modifyable through the python programming interface. By adding to this exisitng tools, a program can be created to limit the scope of binary execution to a single path.

Combining the above techniques allows a software developer to fuzz their individual software feature.

III. How does this relate to others work? (Related Work)

Angr is a binary analysis tool that allows a program to use the output of Angr into a python program. [cite]

Driller is a fuzzer that uses symbolic execution to find hard to reach bugs. [cite]

Fuzzing State of the Art outlines problems within current fuzzing tools [cite]

TaintTrace flow tracing with dynamic binary rewrite [cite]

Concolic Execute Fuzzing Based on Control-Flow Analysis [cite]

Static program analysis assisted dynamic taint tracking for software vulnerability discovery [cite]

A binary analysis approach to retrofit security in input parsing routines [cite]

IV. Experiment Design

N/A for this assignment

The following steps are the design of this program

1) Create a Control Flow Graph (CFG) to trace the execution of the program (static analysis)
2) Use taint analysis to trace the program inputs throughout the CFG from step 1
3) Identify a function (or selection of binary) to fuzz within the program
4) Execute the program upto the entry of the selected function (or selection of binary) from step 3
5) Save the environment state (registers, memory addresses, files, etc.) for use within the next step
6) Identify input variables that are worthwhile to fuzz (preferably from step 2)
7) Use a fuzzer to loop through the saved state and fuzz the desired variables while executing the selected function (dynamic analysis)
8) Record identified vulnerabilities

Once the BFF tool is created the experiment will be conducted as follows.

This experiment was conducted with the first 100 programs in the Juliet test suite.

The BFF tool requires that a specific unit of code is selected for test.

Further, the BFF tool was tested against code that was likely to contain no vulnerabilities. The purpose of this test was to assess the false positive rate.

Each of the tests where the BFF tool identified a vulnerable application, a manual test was performed to verify the correctness of the found vulnerability.

For each application, we recorded the following data points.

- Start time
- Time to first vulnerability identification
- Number of identified vulnerabilities
- Number of correctly identified vulnerabilities
- Number of incorrect identified vulnerabilities
- Number of non-identified vulnerabilities
- Sequence of input to trigger each vulnerability

V. What do you discover? (Results)

N/A for this assignment

References

[1] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 300–311. [Online]. Available: http://ieeexplore.ieee.org/document/7985671/

[2] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," vol. 67, no. 3, pp. 1199–1218.

[3] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," vol. 37, no. 6, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2382756.2382798

[4] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, pp. 94–105. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2931037.2931054

[5] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, pp. 559–570. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884853

[6] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," pp. 233–244. [Online]. Available: http://arxiv.org/abs/1707.09038

[7] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 33–44.

[8] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 643–653. [Online]. Available: http://ieeexplore.ieee.org/document/7985701/

[9] P. McAfee, M. Wiem Mkaouer, and D. E. Krutz, "CATE: Concolic android testing using java PathFinder for android applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, pp. 213–214. [Online]. Available: http://ieeexplore.ieee.org/document/7972811/

[10] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 1–11.

[11] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *11th IEEE Symposium on Computers and Communications (ISCC'06)*. IEEE, pp. 749–754. [Online]. Available: http://ieeexplore.ieee.org/document/1691114/