Approximating Symbolic Execution Using Dynamic Taint Analysis

Subtitle Text, if any

Clark Wood
Florida State University
clark.w.wood@gmail.com

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Introduction

Finding exploitable bugs in binaries is a difficult and lucrative endeavour. Fuzzing, or the feeding of random input into processes to try and detect vulnerable conditions [CITE], is one technique used by researchers and analysts. Applications tested for vulnerabilities are often black boxes because source code is unavailable. This piles difficulties onto an already complex problem, of which much of current research attempts to solve or reduce [CITES]. Among these issues are i) efficient and thorough dynamic taint analysis [CITE], ii) the path explosion problem [CITE], and iii) the code coverage problem [CITE?]. Dynamic taint analysis is difficult because taint can propagate directly via assignment, or indirectly via affecting control flows which affect assignment [D], which make proper implementation tricky. The path explosion problem arises from the nature of assembly languages. Jump instructions, which may make up as much as X% of machine instructions in programs [CITE], each multiply the total number of existing paths in a binary by two. Considering the size of binaries, this quickly leads to an un-tenably large set of possible paths to explore via conventional techniques like symbolic execution. And even if it is possible to explore all paths in a binary, analysts still need sample input which can trigger the vulnerable condition, perhaps proving that the binary is exploitable along the way. Such automated exploit generation is the subject of much research [H, I, CITES].

This work aims to reduce the severity of the latter two problems while being wary of the first during implementation. Instead of trying to attain more reasonable performance by using symbolic execution less or more wisely [J], I focus instead on approximating symbolic execution by using dynamic taint analysis to drive mutation. This is possible by tracking taint at the instruction level, then backsolving from every jump to the source of taint to come up with smarter ways to mutate user input and create new test cases. For instance, in the below example:

```
mv eax, userInput1
mv ebx, userInput2
cmp eax, ebx
jge 0xDEADBEEF
```

If we already had a test case where userInput1 equalled 10 and userInput2 equalled 5, this test case would jump to 0xDEADBEEF. To generate a test case where we would not jump, we take the complement of the jge instruction's comparison function, which would be "less than". We can find the last instruction that modified the Z flag, which is how jge decides to jump or not. We see that this compare instruction was tained by the eax and ebx registers, which were in turn tainted by userInputs 1 and 2. Since we can impact both values in the comparison, it is easy to modify user input smartly to alter the value of the Z flag and jump during the next execution. Perhaps we could switch the values, or auto-increment/decrement one until we satisfy the needed comparison, or mutate values randomly.

There are a finite number of ways to compare data in x86, so I can create cases for all possible ways jumps are decided. I only want to prove this is possible however, so I will only implement a subset and instead comment on how to implement the other comparisons.

2. Assumptions and Scoping

I assume ASLR is disabled for the binary to be fuzzed. This allows the program to easily track and compare execution paths by creating a list of jump locations. Since the world is moving more and more toward 64-bit architectures, I plan to implement the technique for x86_64.

I also assume the presence of an algorithm which will detect an exploitable condition, provided it is given the proper binary and input to the binary. The program will then attempt to concretely execute each path within the binary by backsolving, hopefully at a greater speed than symbolic execution.

I am hopeful about the results because current selective symbolic execution engines still appear to introduce significant overhead (Between 6 and 78X overhead more than QEMU for S2E. Just QEMU is between 4 and 10 times slower on some benchmarks, more like 15 on others) [J, K, L]. In contrast, basic block counting using Intel Pin, the framework I am leveraging, introduced between 2 and 4X overhead [M]. However, the pintool needed for my proposed technique will introduce greater levels of overhead, since I will be working at the instruction level instead of the basic block level.

I am leveraging Intel PIN to perform dynamic binary instrumentation. I plan to test first on dummy programs with many paths that are easy to backsolve, similar to the example above. A C program which uses sequential switch statements to check user input against a secret, when compiled with tcc, which performs very little optimization, should create an appropriate binary. I then plan to test against S2E and Peach Fuzzer.

3. Research

Dynamic taint analysis (DTA) is the process of locating sources of taint and following their propagation through a binary to data sinks. Since, for exploitation, bugs must be triggered by user input, DTA usually identifies all user input (files, command line arguments, UI interactions, ...) as tainted, and marks it accordingly.

The granularity of taint marking is variable. Assigning everything from a generic binary value of tainted/not tainted to bit-level marking of which part of user input affected which variable is conceivable, although finer-grained taint analysis is more computationally expensive and difficult to determine [CITE].

Taint can propagate via explicit or implicit flows, sometimes also called data and control dependencies respectively [G]. Explicit flows involve a tainted variable, x, which is used in an assignment expression to compute a new variable, y. In this situation, x taints y, and if y is involved in any further assignment to a variable z, then x taints z by transitivity. In contrast, implicit data flows involve a tainted variable used to affect control flow within a program which subsequently sets the value of another variable, for instance at a branch [D].

Because of subtle implicit flow cases referenced by Clause, J. et all [D], which can be difficult to spot (Page 2, Figure 2b). Implicit data flows are not always considered by dynamic tainting techniques [E].

For my project, implicit data flows are very important, so I need to implement DTA which catches both implicit and explicit flows.

DTA has been used to implement smart mutation fuzzers [A], but I think my work is unique from and complementary to Bekrar's work. In [A], DTA is proposed as a way to intelligently decide which parts of user input should be mutated. I plan to add on to this idea by allowing smarter mutations to be selected by backsolving to decide which values are most likely to result in new paths being explored.

4. Implementation

I plan to implement dynamic taint analysis for files as input sources. I am basing my DTA heavily on Jonathan Salwańs work [N]. I've been in contact with him and received permission to use his sample code. I can watch for all reads after a file is opened, then tag user input and follow it as it moves throughout the binary. I will need to implement a system which keeps track of how taint propagates at the instruction level. A directed graph structure, with initial nodes representing the bytes of user input which are

5. References

A Bekrar, Sofia, et al. "A taint based approach for smart fuzzing." Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012.

B DeMott, J., Enbody, R., and Punch, W. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing", BlackHat and Defcon 2007.

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

D Clause, J. Li, W., Orso, A. "Dytan: a generic dynamic taint analysis framework". 2007 Int'l symposium on Software testing and analysis. ACM, 2007.

E ... "Beyond Instruction Level Taint Propagation".

F Yamaguchi, Fabian, Felix Lindner, and Konrad Rieck. "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning." Proceedings of the 5th USENIX conference on Offensive technologies. USENIX Association, 2011.

G Bao, Tao, et al. "Strict control dependence and its effect on dynamic information flow analyses." Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010.

H Cha, Sang Kil, et al. "Unleashing mayhem on binary code." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.

I Avgerinos, Thanassis, et al. "AEG: Automatic Exploit Generation." NDSS. Vol. 11. 2011.

J Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems." ACM SIGARCH Computer Architecture News 39.1 (2011): 265-278.

K Bellard, Fabrice. ”QEMU, a Fast and Portable Dynamic Translator.” USENIX Annual Technical Conference, FREENIX Track. 2005.

L Guillon, Christophe. ”Program Instrumentation with QEMU.” 1st International QEMU Users Forum. 2011.

M Luk, Chi-Keung, et al. ”Pin: building customized program analysis tools with dynamic instrumentation.” Acm Sigplan Notices. Vol. 40. No. 6. ACM, 2005.

N Salwan, Jonathan. Shell-storm.org