ComfortFuzz: The Smartest Dumb Fuzzer

Neil Fitzgerald

December 13, 2013

1 Introduction

ComfortFuzz is a light-weight fuzzer that intelligently narrows its fuzzing space and prioritizes tasks more likely to be successful, a project of the MRC group in CSAIL under the guidance of Martin Rinard. In this paper, I will describe the work that it builds on, my personal contribution, and work that will be done on it in the future.

2 Background

ComfortFuzz is hardly the first attempt to research a better fuzzer. ComfortFuzz builds on a broad tradition of fuzzers, including one from the MRC group itself.

2.1 Previous Work

Fuzzers are tools used to test an application for bugs, especially exploitable bugs such as integer overflow errors, using modified user inputs randomly distributed over the whole potential input space. The earliest fuzzers were so-called "black box" fuzzers, which simply randomly generate inputs and pass them through to the program without any other tricks.

Fuzzing was substantially improved by the introduction of "white box" fuzzers [2]. These fuzzers use program analysis tools to improve the accuracy of fuzzers. Some use solvers to "break" specific expressions. Others use "taint tracking" to find which variables can be influenced by the user, and at what points, to see where the program can potentially be derailed by a malicious adversary [1].

White box fuzzers are useful, but they have two downsides. Firstly, they are extremely computationally expensive, sometimes taking days to run even when the work is distributed across multiple machines. Secondly, they are usually targeted as just one type of errors: e.g., integer overflow errors. With ComfortFuzz, our goal is to maintain the low overhead and wide applicability of black box fuzzers, while using various tricks to make our fuzzer more intelligent.

2.2 Parallel Work

ComfortFuzz is based heavily on the framework of another fuzzer built by the MRC group, ClearFuzz. Although ClearFuzz itself is closer to the white box fuzzers described above, we have reused the main workflow of the fuzzer, including the initial analysis tool used on the program, the tool that allows us to manipulate fields of a seed file given the file format, and the tool that takes our fuzzing tasks and runs them. This greatly simplified some of the tactics described below, particularly finding correlated variables at critical points.

3 Contribution

Since most of the framework was already in place, my contribution mostly consisted of the code to produce fuzzing tasks from our surface analysis of the program and the output of Daikon, a tool that finds invariants over the fields of a sample set of input. I implemented the following tactics:

- **The Comfort Zone.** This is the namesake tactic of ComfortFuzz, as it is the most distinctive. The idea is to break the invariants we get from Daikon: e.g., if we find that width is always between 40 and 1920, we will replace the width with a value outside that range. This allows us to pick random values that are more likely than average to be hard for our program to deal with.
- **Shivs.** This is not an original tactic, but it's an obvious one: we choose certain deterministic values, e.g., INT_MAX, "file://c/s", that we always try.
- **Correlated fuzzing.** This is our secondary original tactic. We use our preliminary analysis of the program to find which variables (e.g., width and height) often appear together at critical, vulnerable points in the program. We then fuzz these variables simultaneously rather than separately, since some bugs may only be found at an edge case for both rather than each individually.
- **Scoring.** Finally, we rank our fuzzing tasks, which number in the hundreds in our simple test cases. This allows us to prioritize variables that have low coverage in our sample input, and shivs that are the most likely to be successful.

4 Future Work

There is still a great deal of work to be done on ComfortFuzz. Although it now works, it can and will be modified to run faster and smarter. Making it faster is a matter of optimizing the code. Making it smarter will be more difficult, and requires analysis of test cases to see how successful ComfortFuzz is at reducing the cardinality of the input space, and which shivs (and which sorts of random values) are the most successful, and adjusting scoring appropriately. This will also give us a sense of which sorts of programs and cases ComfortFuzz is best suited for, and what advantages it has over other fuzzers.

Additionally, we could add new tactics to ComfortFuzz. One of its current weaknesses is its inability to recognize and subvert unsafe sanity checks in the code. Given additional analysis capability, we could find boolean expressions in the code that are always (or never) satisfied, and pick new fuzzing values accordingly.

5 Acknowledgements

I would be greatly remiss if I did not thank the MRC group in CSAIL for their continued guidance with this project. I would like to thank Stelios Sidiroglou-Douskos for his mentorship and patience, and Nathan Rittenhouse, Eric Lahtinen, and Paolo Piselli for their assistance and advice at various points.

References

[1] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 474–484. IEEE, 2009.

[2] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.