CSEP564 : Computer Security : Reading 3

Karuna Sagar Krishna

October 16, 2024

Paper Title

AFL++: Combining Incremental Steps of Fuzzing Research

Paper Authors

Andrea Fioraldi, Dominik Maier, Heiko Eibfeldt, Marc Heuse

Problem

Prior to AFL++, fuzzing researchers forked existing fuzzing tools like AFL to implement their new proposed technique to improve fuzzing. Many of these techniques are target specific i.e. they work great for certain targets but not for others. It is hard for both industry and researchers to pick appropriate fork/technique for their targets and further it is even harder to combine multiple techniques. Effectively, researchers and engineers are slowed down and restricted by the lack of tools and infrastructure to effectively try new techniques by combining or modifying existing ones.

Approach

AFL++ addresses the above problem by providing a plugin framework to implement different fuzzing techniques at various fuzzing stages. This architecture also supports to combine multiple techniques to improve fuzzing. In short, AFL++ framework enables composable, usable and reproducible fuzzing helping both research and industry. The core idea is custom mutator API which the authors claim as approachable and future proof way to implement new techniques.

The authors explain why AFL was chosen as the baseline for their framework and briefly point out various concepts like coverage guided feedback, forkserver, mutations and scheduling employed by AFL. They provide high level insights into various techniques to improve various aspects of fuzzing that have shown up in the research community eg. AFLFast, AFLSmart, MOpt, RedQueen and LAF-Intel. The papers describes the custom mutator API, pointing out briefly how different techniques can be implemented using these APIs. AFL++ supports both source and binary code instrumentation and the paper talks about various capabilities available for each various compiler backends like LLVM, GCC, QEMU and Unicorn. Finally, the authors have used AFL++ to evaluate few techniques showcasing how to reproducible and composable aspects of AFL++. Using the results from this evaluation, the authors conclude that fuzzing is highly target dependent and hence justifying the need for AFL++ plugin framework.

Conclusions

The authors show that AFL++ provides the baseline framework with important properties like reproducible, composable and comparable by evaluating a few combinations of existing techniques implemented as plugins. The plugin system makes it easy to prototype new research ideas and for engineers to tailor the fuzzing tool to a specific target. The authors are self-attesting the use of AFL++ in their own fuzzing research. Hence, the authors hope that research community and industry standardize on AFL++ as *the* fuzzing tool.

New Ideas

This paper presents a new fuzzing tool called AFL++ which has a plugin framework aka custom mutator API. This is a great engineering idea to create tools or systems that can be extended with the intent to experiment with new fuzzing idea and/or compose different strategies together. Such tools provide a great stepping stone.

The AFL++ Optimal is yet another great idea that makes the tool more usable by lowering the barrier to entry. By having a set of default configurations and strategies, the fuzzing tool can be effectively used right out of the box for most targets.

Improvements

I had to read AFL++ documentation and "The Art, Science and Engineering of Fuzzing: A Survey" paper to get better understanding of various fuzzing concepts and terminologies. The paper could have established an brief overview of the fuzzing concepts to understand the paper better.

Additionally, the paper could have added architecture/design diagram for AFL++ showing how the fuzzing pipeline and various stages interact. This would have provided better clarity of how the custom mutator API affects various aspects of fuzzing such as scheduling and trimming.

New Directions

It seems that AFL++ is a wonderful tool which can be configured in various ways. AFL++ optimal tries to find a best default configuration for various targets. Given that the paper establishes that fuzzing is highly target dependent, I would have thought about classifying targets based on different interfaces (eg. REST API targets, parsers, GUI) and then have default/starter configurations for each of these classification bucket. This might be useful for engineers to adopt AFL++ for their target interface. Also, there might be other interesting ways to classify targets (eg. source code language).

It is not clear how long should a fuzzing campaign last before it finds a majority of the bugs. There seems to be some research in this area and it might be possible to integrate that into AFL++ itself, so that engineers don't have to randomly pick iteration or time period to stop the campaign.