Thesis Project Proposal (30 hec)
Master of Science in Department of Computer Science
and Engineering

Date: 2016/01/26

Fuzzing: The Future or Just a Fuss?

Emil Edholm:	*TDA602 - Language-Based Security*
	EDA263 - Computer Security
	EDA491 - Network Security
	EDA092 - Operating Systems
	DAT151 - Programming Language Technology
David Göransson:	*TDA602 - Language-Based Security*
	EDA263 - Computer Security
	EDA491 - Network Security
	EDA092 - Operating Systems

Keywords: Fuzzing, Testing, Penetration Testing, Security

1 Introduction

The development of software has come a long way since its inception. It is not uncommon for projects to contain several thousands of lines of code, or more. With such a large code base, it is very easy for security related bugs to go undetected, even if there are tests. Some of these bugs may be hard to identify through the use of normal testing, such as unit tests or integration tests. The bugs may be improbable corner cases that no one thought to test for, and they may pose a serious security threat.

Fuzzing is another type of testing technique that one can employ. A fuzzing application, called a *fuzzer*, will throw massive amounts of carefully generated input to the target application with the intention of finding crashes and anomalies in state. This can be done either on a white box, i.e. an application where the tester has source code access, or a black box, where the tester does not have access to the source code. Fuzzing is normally viewed as a penetration testing tool, but can also be used for finding ordinary bugs and crashes when given extraordinary input.

Our hope is that our finished work leads to a greater understanding about fuzzing and bug-finding principles in general. This also includes raising awareness about security related issues that unexpected input can have on an application. Since fuzzers can be quite hard to configure correctly and has a steep learning curve, we believe that our work can somewhat lower this curve and make fuzzing a more approachable alternative early in the software development phase. The hope is that you should not have to be a security expert in order to use fuzzing efficiently.

2 Context

There exist two major categories of fuzzers: *generational* (sometimes called grammar-based) and *mutational*. Generational fuzzers generate random input within a known pre-configured template. This could be, for example, a protocol RFC or a file format specification. Mutational fuzzers use an existing file (or data), called a *seed*, and makes small mutations of the seed to use as input to the target application and does therefore not have to know anything about the target application.

The paper *Program-Adaptive Mutational Fuzzing* (Cha, Woo & Brumley, 2015) describes a new way of using mutational-based fuzzing on applications where the target source code is unavailable (black box testing) and compares the results of this algorithm against a number of existing fuzzing software.

The above mentioned paper used common unix applications in their testing. Godefroid, Levin & Molnar (2008) discusses whitebox testing in a Windows environment, where they test against undisclosed common Windows applications.

Although the above sources are non-exhaustive on the concept of fuzzing they cover the different techniques in broad terms and cover the basics of a few of the different techniques we wish to evaluate. There also exists many other papers that discuss different techniques for finding security related bugs in software using automated- or semi automated methods, but as far as we have been able to discern, there are no research papers that discuss and compares fuzzing as a whole.

3 Goals and Challenges

Part of our goal is to evaluate and compare the efficiency of mutational and generational fuzzers. By rigorously testing fuzzers we aim to differentiate what techniques proves most competent, what advantages they may have, and also investigate the potential for future improvements. Fuzzing is an endless process, thus one must consider the efficiency with respect to time. How do your know your have fuzzed enough? What kind of code coverage is achievable? Some techniques may find specific types of bugs in a fast manner whilst other may take longer time and vice versa. This type of questioning will be taken in consideration when evaluating the different fuzzers and algorithms. Is an advanced “smart” fuzzer, such as AFL (AFL, 2016), significantly better than a fuzzer that can be written in ten lines of code?

The first goal will be to select the fuzzers that we choose to evaluate. The criteria we have is that the fuzzers should be open source and still be in development. We define being in development as having had at least one update in the last two years.

The next goal will be to select target applications with great care, i.e. those applications that we will run the fuzzers on. Our selection of these applications will primarily be based upon applications used by cited sources that write about automated testing techniques, such as fuzzing. We will also test the fuzzers upon applications that we deem interesting but will include results from these, more as a footnote than anything concluding. We will also test an application that we have made with known bugs, in order to see if and how many of them are found when fuzzing.

One of the challenges throughout this master thesis will be to try and deceive the fuzzers that we have chosen to evaluate. By developing an anti-fuzzing algorithm, our goal is to design it in such a way that it is very hard for a fuzzer to find bugs. The reason for doing this is two-fold, first and foremost, if you are able to to "fool" the fuzzer, then you have also found an angle of improvement that could possibly be implemented, and the other reason is investigating if it is possible for a company for example, to develop an algorithm that detects that their application is being fuzzed and take countermeasures. The reason for this could be to make it harder for outside parties use automated tools to find vulnerabilities in their application.

We also aim to to touch upon the subject of how fuzzing can be more tightly integrated with software development. Currently fuzzing is a technique mainly used to find vulnerabilities and bugs post-production and require an in-depth knowledge in order to configure and run them efficiently. However, the ability to test different parts of an application with fuzzing during development in an user friendly fashion may improve code quality and the likelihood of finding bugs before they can cause any harm. Are different fuzzers similar enough to be combined into a common interface, such as an IDE plugin? Is it worth it, with respect to bugs found and time trade-off?

4 Approach

The first step will be to do an in depth research about the the different fuzzing techniques commonly employed, i.e. the different kinds of mutational fuzzing and how grammar-based fuzzers work. Based on this research we can determine which fuzzers will make up the core of the analysis to be done in the next step.

Each of the selected fuzzers will be researched to gain a deep knowledge of how they work and what the differences are between them. We will start fuzzing a selection of applications with known bugs and potential unknown bugs and see how many of them each of the different fuzzers find and what time it took to do so. The fuzzers will be applied on each target application for an appropriate and reasonable amount of CPU hours. This research will answer questions such as what percentage of crashes are (probably) exploitable as security holes?, What kind of code coverage can you achieve?, How many found faults in a certain timeframe etc. This also ties in with questions such as what types of bugs can be identified by the use of fuzzing? Can you identify behavioral anomalies?

Our hand-crafted application will be built in several iterations, to test different types of bugs and vulnerabilities that fuzzers commonly are used to find, such as privilege escalation and memory corruption. For each iteration we will try to adapt to the fuzzing techniques used and their weaknesses. We will try and introduce functionality that runs properly yet the fuzzer are unable to fuzz.

Results of the fuzzing will then be used as a baseline for future tests when our fuzzer-fooling algorithm has been applied to the target applications, including our own. The experiment will be redone with the modified applications and the results compared to the baseline experiment.

A summary of our criteria for completion is as follows:

- An evaluation of the fuzzers with tests against the selection of applications along with our own. The evaluation will address efficiency, speed and accuracy among other things.
- Testing how, or if it is possible to avoid fuzzing by implementing an anti-fuzzing algorithm and discuss possible enhancements.
- An investigation of the options for integrating and making fuzzing easier to use with regards to software development.

5 References

Cha, S. K., Woo, M., & Brumley, D. (2015). Program-Adaptive Mutational Fuzzing.

Godefroid, P., Levin, M. Y., & Molnar, D. A. (2008, February). Automated Whitebox Fuzz Testing. In *NDSS* (Vol. 8, pp. 151-166).

AFL, American Fuzzy Lop (2016, January 26). Retrieved from http://lcamtuf.coredump.cx/afl/