Fuzzing Methods

Anonymous Submission

refs.bib

Abstract. Modern Software[**DUMMY:1**] has become increasingly more complex. This is connected with growing nummbers of security bugs and cyberattacks. Software testing tries to oppose that development. Fuzzing is a dynamic automated way of testing and sees growing usage among security profesionals. This paper examines different approaches and use cases of this technology.

Keywords: Offensive Security · Fuzzing · Brute-Forcing · It-Security · Testing

1 Introduction

Development of any kind has always come with bugs, errors and unintentionally behaviour. Software is no exception. Increasingly complex programs and growing technology stacks contribute to this problem. On top of that, finished components are often continuously integrated. As a result, it becomes ever more difficult to predict final program behaviour. This yields in risks, particular if an application is processing external data input. If not handled correctly it can pose security vulnerabilitys or data breaches.

The bigger a programm gets, more often such unwanted appearances occur. A field in which this is most noticable is web-development. Whereas 15 years ago, most websites were built on top of a view technologys, modern web-applications often use several framework with huge amounts of dependencys. This lead to an increasing amount of security bugs. Different approaches in software testing tries to oppose that development.

Back when software was relativly simple and free of dependencys, manuel testing was the state of play. That included code reviews and manuel checks for potentiel exploitation. This was time consuming and needed experts for every specific application. Therefore static analyis was quickly adapted. This contained new techniques such as pattern search with a control flow graph, data dependency graph and data flow analysis. When software grew even more in complexity, security researchers needed a more scalable approach. Hence dynamic analyis was implemented. This method tries to understand software behaviour by inputing different types of data. Fuzz-testing is the cutting edge of this technique and will be explored in this paper. At first we will examine the basic concept and functionality. Following different methods in Fuzzing will be explored. Finally we will have a conclusion that covers the pros and cons of Fuzz-testing.

2 Fuzzing background

Fuzz testing or fuzzing is a method of testing software to detect security holes in applications, operating systems, and networks. It involves flooding their input interfaces with random data, called fuzz, to make them crash or trigger unexpected programm behaviour. Furthermore the computation results are monitered and reported, all in an automated way. Fuzzing has become increasingly more relevant among security researchers and is used by almost all big tech companys for penetration testing. Tech leaders find more than 80 of their bugs by using different fuzzing Methods. Google for example has found over 19

thousend errors in their browser projekt chrome.[1] Microsoft used dynamic analysis to test one of their flagship projects 'office' and found nearly 2000 bugs using the fuzzing tool afl. Also open source project such linux use Fuzzing methods on a great scale. Many security bugs within the Linux Kernel have been found this way.
Generally a distinction can be made on how fuzzing is executed on a specific interface. Depending on the application or infrastructure of the target system, different types of fuzzers can be used. Application Fuzzing will approach a system by testing functionalitys such as buttons, input fields or command line parameters. File format fuzzing tries to generate corrupted files and feed them to the corresponding target for processing. Other types of Fuzzing targets include Protocol Fuzzing and Network fuzzing.
In addition more distinction can be made on how the fuzzing inputs or test cases are generated. There are two approaches. Mutation based Fuzzers alter existing data samples to create new test cases. Generation-Based Fuzzers define new data based on the expected input of the target specification. Both methods have pro and cons and will be covered in more details later on.

3 Functionality

This section will provide a perspective on the basic concept of Fuzzing. Regardless of the specific implementation, the core process of all fuzzers remains the same. First the target interface has to be identified. Relevant are all interfaces that allow external data input. This ranges from network sockets, user-input on a website to linux binarys et cetera. Specifally interesting are interfaces that are reachable by unpriviliged users. The act of finding those target interfaces highly depends on the tested application and could include things like code reviews or portscanning. Based on the findings, the fuzzer input can now be generated. This is one main task of the actuall fuzz-engine. In the most basic case, it will generate pseudo random inputs based of the given seed. The seed could contain anything that makes sense in the context of the expected data, essentially a snapshoot of valid inputs for the target system. Mostly combinations of strings and intergers. Following the test cases can be passed to the target system. Depending on the identfied interfaces, different transfer methods are used. This could involve HTTP Requests, tcp sockets or simply files as input parameters for a programm. While the Fuzz-eingine is sending the generated inputs, the target system has to be monitered to track how the testcases effect the system behaviour. Monitering and tracking can be achieved with different approaches. A convinient but not always available case are logging functions of the target system. They can be simply read out during or after the fuzzing process. If the target is lacking such functionalitys it is neccecarry to implement feedback instrumentations. Those are compile time binarys that are inserted into the target code to track and tell the fuzzer which parts of the internal programmflow is effected and has already been covered. This ensures to establish a correlation between test cases and programm behaviour. The Feedback gets send back to the fuzzing-engine and can be read out or reprocessed.

[1]Source 3

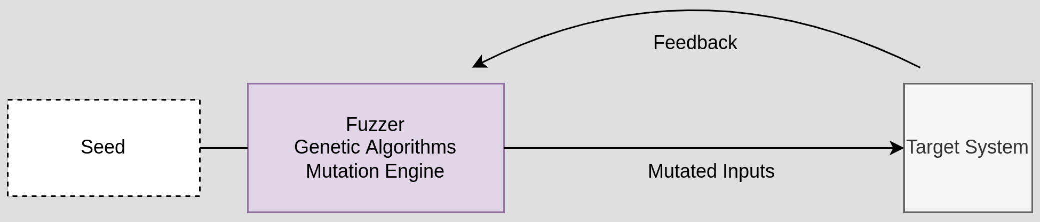

Figure 1: Basic Fuzzing application[2]

Figure 1 provides an overview of a basic mutation based fuzzing implementation. The Fuzzer as seen between the Seed and the target system includes a mutation engine and genetic algorithms.[3] The Fuzzing process can now lapse as follows.
The Seed streams valid inputs to the fuzzer that will modifiy these with the help of the mutation engine. Following the mutated inputs gets send to the target system in which feedback instrumentations listen for exceptions, log them and provide feedback to the fuzzer. Based on that genetic algorithms produce further inputs. Once the fuzzing process has endet, it is now possible to analyse and minimize the findings. This includes to determine if a tracked exeption could actually cause a vulnability or even an exploit. Furthermore these assessments are rated. This is done by ranking frameworks like the common vulnability scoring system(CVSS). Finally all relevant informations are summed up, reported and passed to the corresponding responsible.

4 Fuzzing Methods

Due to the huge landscape of applications, infrastructures and use cases, enourmes classifications exist around fuzzing. In this section we will, further examine the different methods used to generate the fuzzing input, like mutation and generation based fuzzing. Afterward we will also cover different types of target focused fuzzers such as application or file format fuzzing et cetera.

4.1 Mutation based fuzzing

As we already know, this method will simply alter already existing data. This can be based of available informations in case the target inputs are known. Otherwise it is first neccecarry to record valid inputs. This can be achieved via proxys that will record communication or interaction patterns. These valid inputs can then be mutated, either in a completly random manner or after fixed patterns. Figure 2 provides a mutation based implementation in pseudo code.

```
seed = "https://www.hs-albsig.de/such?tx\_solr[q]=\\"
```

[3]source 1

```
Fuzzer = MutationFuzzer(seed=seed)
Fuzzer.mutate for i in alphabet -->
"https://wGw.h2-albsig.re/sLh?tx\_soslr[qx=\\"
"htSpK://wwl.hs-albsisag.de/such?tx\_solr[s]=\\"
"htSps://oww.hs-albsig.de/such?tx\_solr[q]=\\"
"https://www.hs-albsig.de/such?tx\_solr[q]=\\"
"https://www.hs-albsig.de/such?tx\_solr[q]=\\"
"https://www.hs-albsig.de/such?tx\_solr[q]=\\"
```

Figure 2: Mutated Fuzzing outputs

First the seed is determined. Following a Fuzzer object is applied and the seed passed. Then we call the fuzzer method to mutate the given input without setting up any fixed patterns. Here we instantly see why mutation based fuzzers are often called 'dumb fuzzers'. Most of the generated outputs are invalid urls, and will be rejected by the parser of the target system. In fact in this concrette approach we moreover will fuzz the parser than the webserver.

Therefore it is neccecarry to setup grammars that will follow the expected format.

4.2 Generation based fuzzing

Smart Fuzzers, since they generate data from the go…

4.3 Application Fuzzing

4.4 Protocoll Fuzzing

4.5 File format Fuzzing

4.6 Network Fuzzing

5 Tooling

Hundrets of fuzzing tools are available. Like wfzuu, afl and zfuzz etc..

5.1 Practical example

We will fuzz a webserver with afl…

6 Conclusion

abbrev3,crypto,biblio