

Fuzzing Methods

Moritz Rupp

Hochschule Albstadt-Sigmaringen, Albstadt, Germany, ruppmori@hs-albsig.de

Abstract. Modern Software has become increasingly more complex. This is connected with growing numbers of security bugs and cyberattacks. Software testing tries to counter that development. Fuzzing is a dynamic, automated way of testing systems and sees growing usage among security professionals. It has gained growing attention after several zero day vulnerabilities have been found with this approach. This paper examines different methods to use this technology.

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

1 Introduction

Development of any kind has always come with bugs, errors and unintentional 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 risks, particularly if an application is processing external data input. If not handled correctly, it can pose security vulnerabilities or data breaches.

The bigger a program gets, more often such unwanted appearances occur. A field in which this is most noticeable is web-development. Whereas 15 years ago, most websites were built on top of a few technologies, modern web-applications often use several frameworks with huge amounts of dependencies. This lead to an increasing amount of security related bugs [O'D22]. Different approaches in software testing tries to counteract that development.

Back when software was relatively simple and free of dependences, manual testing was the state of the art. That included code reviews and manual checks for potential exploitation. This was time consuming and needed experts for every specific application. Therefore, static analysis was quickly adapted. This contained new techniques such as pattern search with a control flow graph, data dependency graph and data flow analysis [McG04]. When software grew even more in complexity, security researchers needed a more scalable approach. Hence dynamic analysis was implemented. This method tries to understand software behaviour by testing and evaluating the target system while it is running.

Fuzz testing is on the cutting edge of this technique and will be explored in this paper. A particular emphasis will be laid on the different methods used in fuzzing. At first we will examine the basic concepts and functionality of the general process. On top of that different methods in fuzzing will be explored. That expresses itself in how certain functionalities of the fuzzing application are executed. This section will be the main focus of the paper. Following we will also cover the practical usage by showcasing different tools and a concrete example.

As a final point we will have a conclusion in which we provide a summary and an assessment of 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 program behaviour. Furthermore the computation results are monitored and reported, all in an automated way. Fuzzing has become increasingly more relevant among security researchers and is used by almost all big tech companies for penetration testing [li2]. Tech leaders find more than 80 percent of their bugs by using different fuzzing methods [Dec20]. Google, for example, has found over 15 thousand errors in their browser project chrome [Tun19]. Microsoft used dynamic analysis to test one of their flagship projects 'Office' and found nearly 2000 bugs using the fuzzing tool AFL [Dec20]. Also open source projects such as Linux use fuzzing methods on a great scale. Many security bugs within the Linux kernel have been found, this way [Rei20].

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 functionalities 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 will create test cases from scratch based on protocol knowledge. Both methods have pro and cons and will be covered in more detail 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 binaries et cetera. Specifically interesting are interfaces that are reachable by unprivileged 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 actual fuzz-engine. In the most basic case, it will generate pseudo random inputs based on the given seed. The seed could contain anything that makes sense in the context of the expected data [li2]. Essentially a snapshot of valid inputs of the target system. This will be mostly combinations of strings and integers.

Following the test cases can be passed to the target system. Depending on the identified interfaces, different transfer methods are used. This could involve HTTP Requests, TCP sockets or simply files or input parameters for a program. While the fuzz-engine is sending the generated inputs, the target system has to be monitored to track how the test cases effect the system behaviour. Monitoring 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 functionalities, they first have to be implemented.

More features are feedback instrumentations. Those are compile time binaries that are inserted into the target code to track and tell the fuzzer which parts of the internal programm flow is effected and has already been covered. This ensures to establish a correlation between test cases and program behaviour.

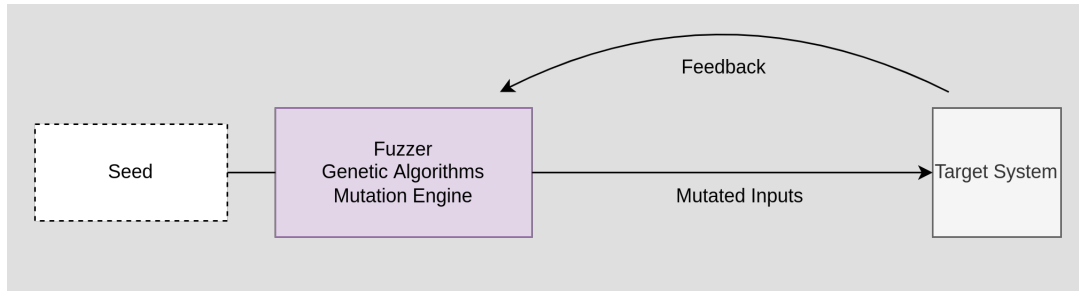


Figure 1: Basic Fuzzing application

The feedback gets sent back to the fuzzing engine and can be read out or reprocessed [Asc20].

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. The fuzzing process can now proceed as follows.

The seed streams valid inputs to the fuzzer that will modify these with the help of the mutation engine. Following the mutated inputs gets sent 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 ended, it is now possible to analyse and minimize the findings. This includes determining if a tracked exception could actually cause a vulnerability or even an exploit. Furthermore, these assessments are rated. This is done by ranking frameworks like the common vulnerability scoring system (CVSS) [NIS19]. Finally, all relevant information is summed up, reported and passed to the corresponding responsible.

4 Fuzzing Methods

Due to the huge landscape of applications, infrastructures and use cases, enormous classifications exist around fuzzing.

In this section we will further examine the different methods used to generate the fuzzing input, like random, mutation and generation based fuzzing. Afterwards we will also cover different types of target focused fuzzers such as application or file format fuzzing et cetera.

4.1 Random fuzzing

When fuzzing was invented in 1988 by Barton P. Miller, random fuzzing was the very first approach [Gar12].

When Miller held a lecture remotely via a telephone connection, a thunderstorm caused electrical disturbances in the telephone line. This resulted in garbled transmissions. Based on these random occurrences, Miller stated that bad input could be used to crash other systems. He and his students applied this observation on different Unix utilities they were using at that time. They remarked that more than 30% of the tested programs were vulnerable to bad input [Zel22].

Initially they simply manually passed pseudo random inputs to these programs. Later on, Miller made a programming assignment to his students, that stated to automate this task. The very first fuzzer was born. The basic principle still holds up today.

Random fuzzers will simply generate inputs by randomly concatenate characters that are provided in a seed. The method is mainly used for black box testing, since input

formats are mostly unknown. A clear downside are the huge amounts of invalid inputs such a fuzzer will produce.

Most of the generated test cases are getting rejected by the parser of the target program. Hence, random fuzzers are often used to test parser applications.

Another use case is to fuzz strings of program parameters. They barely follow any specific structure and are therefore an ideal target for random based fuzzers.

4.2 Mutation based fuzzing

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

```

1  seed = "https://www.hs-albsig.de/such?tx\_solr[q]"
2  Fuzzer = MutationFuzzer(seed=seed)
3  Fuzzer.mutate for i in range(5) -->
4  https://wGw.h2-albsig.re/sLh?tx\_soslr[qx=?]
5  htSpK://wWl.hs-albsisag.de/such?tx\_solr[s]a
6  htSps://o?w.h_s-lbsig.de/such?tx\_solr[q]=\p
7  h?tpIl//wsw.hs-albswgqde/such?tx\_solr[q]=q\
8  https://www.hs-albsig.de/such?tx\_solr[q]=\

```

Figure 2: Mutated fuzzing outputs

At first, in line 1 the seed is determined. Following a fuzzer object is applied and the seed is passed (see line 2). Then we call the fuzzer method to mutate the given inputs without setting up any fixed patterns. This results in several mutated URLs. The inner procedure is simple. The fuzzer engine will take a random index of the seed in the range of the string length. The index value will then be flipped. 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, several of the generated URLs will never reach our initial target, the webserver. This implementation will moreover fuzz the URL-parser than the HTTP-server.

Therefore, it is necessary to be very patient about which part of valid inputs should get passed to the seed to be fuzzed [Zel22].

In this concrete example the right approach would be to setup a static corpus and only mutate the string of the file path and parameters. These can then be combined and send out.

```

1  static = https://www.hs-albsig.de/
2  seed = suche/?tx\_solr[q]=\
3

```

Figure 3: Correct set up of the seed

Even more precision can be achieved by providing multiple seeds that represent different parts of the valid format. A simple function can then receive mutation of those seeds and concatenate them within the expected format.

Still mutation based fuzzers lack in efficiency and strongly depend on the right configuration.

4.3 Generation based fuzzing

In generation based fuzzing new data is defined based on the expected input of the target specification. Hence knowledge of the system under test is mandatory. It is necessary to have a good understanding of the input format, since test cases are generated completely from scratch. It's also important since we want to ensure to pass the parser of the target system, so our test cases are actually reaching the application. Generation based fuzzers achieve this with the help of formal languages. Those are essentially a set of strings of a predefined alphabet. With grammars it is possible to form different expressions based of that. A grammar describes how to form valid strings from the language's alphabet [Zel22]. This is achieved by assigning rules to individual parts of it. In most cases so called context free grammars are used. With them it is possible to form lots of relevant languages. They are particularly great for expressing syntactical structures. This can range from actual programming languages to formats such as JSON or standardized naming conventions like URLs. Figure 4 provides a context free grammar to generate valid URLs.

```

1  "<start>":
2      ["<url>"],
3  "<url>":
4      ["<scheme>://<authority><path><query>"],
5  "<scheme>":
6      ["http", "https", "ftp", "ftps"],
7  "<authority>":
8      ["<host>", "<host>:<port>", "<userinfo>@<host>"]
9  "<host>":
10     ["hs-albsig.de", "www.albstadt.de", "github.com"],
11  "<port>":
12     ["80", "443", "<nat>"],
13  "<nat>":
14     ["<digit>", "<digit><digit>"],
15  "<digit>":
16     ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"],
17  "<userinfo>":
18     ["user:password"],
19  "<path>":
20     ["", "/", "<id>"],
21  "<id>":
22     ["abc", "def", "x<digit><digit>"],
23  "<query>":
24     ["", "?<params>"],

```

Figure 4: URL Grammar

A grammar always begins with a start-symbol, followed by a set of expansion rules, which indicates how symbols can be expanded or built [Zel22]. Essentially, we sequentially run over expressions and apply expansion rules on the symbols. This means that the symbol on the left, can be replaced by the string on the right. In our example in Figure 4, we would replace the start symbol in line 1 with the string in line 2. This string would then be built with the further symbols and strings continuously. When fully traversed, the grammar is capable of generating various valid URLs.

```
1 http://hs-albsig.de?abc=x23
2 ftp://user:password@www.albstadt.de=?def&x23
3 https://github.com
4 ftps://user:password@hs-albsig.de=?def
5 http://www.albstadt.de/x68
6
```

Figure 5: Grammar generated URLs

We can see that all generated URLs are within an acceptable format of a web-parser, hence will actually reach our target system.

Fuzzing tools that offer generation based functionalities use such grammars in their inner working flow to create test cases. In reality it is not necessary to setup them yourself. Generation based tools instead require to split up input data into its individual parts and then assign rules to them. These creation rules determine how to handle and concatenate every single part in order to build up a valid test case.

The following example shows the first part of an HTTP request, build with such creation rules of the fuzzing tool Sully [Luk12].

```
1 s_static('GET ')
2 s_delim(" ")
3 s_string('index.html ')
4 s_string('HTTP ')
5
```

Figure 6: Creation rules in Sully

The individual parts of an HTTP request are passed to the corresponding rules. Since the protocol follows a strict syntax via HTTP verbs, the 'GET' string in line 1 is set to static. Every part is continuously assigned to rules that ensure are within the protocol format. Only parameters such as strings are set up to be fuzzable.

The clear downside of this method is the big configuration needed. It is The individual parts of an http request are passed to the corresponding rules. Since the protocol follows a strict syntax via http verbs, the 'GET' string in line 1 is set to static. Every part is continuously assigned to rules that ensure are within the protocols format. Only parameters such as strings are set up to be fuzzable.

The clear downside of this method is the big configuration needed. It is necessary to setup the fuzzer accordingly for every system from scratch.

4.4 Fuzzer types

As we have seen already, it is crucial to choose the right fuzzing method. Otherwise, we might fail to reach the target system with our test cases. That's why further classifications can be made on the different types of fuzzers. They determine the attack vectors of the system under test.

Application fuzzers will always focus on the I/O of a system. This includes the user interface, command line parameters or import/export capabilities. For a web application this also means URLs, forms or RPC requests. With the help of proxys and sniffers, it is fairly simple to record valid inputs of those vectors, hence mutation based fuzzing is often chosen as a method. Application fuzzing is also among the most used fuzzer type since it sees the most amount of user traffic.

Protocol fuzzers will mostly focus on communication patterns. This could be different data packets of network traffic like HTTP requests, TCP streams et cetera. Since such

protocols are often complex and very case sensitive, it requires good understanding of their inner working flow in order to test them. Hence, generation based fuzzers are the method of choice.

File format fuzzing will fuzz different file formats such as JPG or PNG. This is achieved by generating multiple malformed samples which are then opened in the target application. If the application crashes, the file is saved for later review.

The main goal is then to gain control over the crash. This might lead to an exploit.

Since users are more likely to interact with images than executables, it is especially dangerous if findings occur in such file types[OWA22].

5 Tooling

Since fuzzing required lots of pre configuration and setup, it is almost always executed with the help of different tools. Only on rare occasions it would make sense to manually fuzz a system. This specifically includes old software that is not compatible with modern fuzzing tools. Otherwise, all systems that offer interfaces and compute external input, are fuzzable with most modern tools.

Such fuzzing tools will mainly provide 3 things. Generate test cases as inputs for the system under test, built a connection channel to transfer those inputs to the target interface and monitor the system behaviour.

There are dozens of commercial fuzzing engines and hundreds of self made implementations. Big tech companies often built their own fuzzers to test their it infrastructures. Googles ‘OSS-Fuzz’ was able to find thousands of bugs in many different Google applications.

Among the most relevant fuzzing tool is the American fuzzing loop (AFL) [rea22]. AFL is responsible for a majority of zero day findings and will be covered in the following example.

5.1 Practical example

The American fuzzing loop is a powerful open-source fuzzer that offers all of the covered methods and techniques. On top of that, it employs genetic algorithms and feedback instrumentations to increase code coverage. In the following example, we will fuzz the C program fuzzgoat [fuz21]. It has several deliberate memory corruption bugs that are easily found by AFL. Before the fuzzing process can start, we have to do some pre configurations.

```
1  $~ export CC=afl-clang-fast
2  $~ make
3  $~ mkdir afl_in
4  $~ cp /bin/ps afl_in/
5  $~ mkdir afl_out
6  $~ afl-fuzz -i afl_in -o afl_out -- ./fuzzgoat @@
```

Figure 7: AFL configuration

At first it is necessary to compile the target binary with AFL’s built in compiler. This way the target is set up for feedback instrumentations and AFL is able to read out code paths and general code coverage.

In line 1 we tell AFL to use the compiler clang, to then compile the binary with it. Following we create a directory `afl_in`. In line 4 we copy the binary output of an Unix utility to this directory. This is essentially the seed of which AFL will create the test cases. To store results we create another directory `afl_out`. In there AFL will provide information

about crashes and bugs. On line 6 we start the actual fuzzing process by passing the seed and output locations as well as the target binary.

Afl will now take inputs of the seed, mutate them and insert the test cases in the target application's command structure. When AFL is started, it spawns an interface in the terminal that provides different information about the fuzzing process. This includes the amount of code coverage, which test cases are currently passed and the number of findings.

```

american fuzzy lop 2.52b (fuzzgoat)
process timing
  run time : 0 days, 0 hrs, 0 min, 27 sec
  last new path : 0 days, 0 hrs, 0 min, 0 sec
  last uniq crash : 0 days, 0 hrs, 0 min, 1 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 31 (22.79%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : calibration
  stage execs : 0/8 (0.00%)
  total execs : 135k
  exec speed : 5193/sec
fuzzing strategy yields
  bit flips : 2/352, 3/335, 1/301
  byte flips : 0/44, 0/27, 0/8
  arithmetics : 4/2232, 0/416, 0/20
  known ints : 1/205, 0/630, 0/308
  dictionary : 0/0, 0/0, 0/0
  havoc : 130/129k, 0/0
  trim : 99.99%/28, 0.00%
overall results
  cycles done : 0
  total paths : 136
  uniq crashes : 7
  uniq hangs : 0
map coverage
  map density : 0.09% / 0.54%
  count coverage : 2.01 bits/tuple
findings in depth
  favored paths : 49 (36.03%)
  new edges on : 72 (52.94%)
  total crashes : 67 (7 unique)
  total tmouts : 1 (1 unique)
path geometry
  levels : 3
  pending : 120
  pend fav : 34
  own finds : 134
  imported : n/a
  stability : 100.00%
[cpu:196%]
```

Figure 8: AFL fuzzing interface

Depending on the tested application, it can be necessary to run AFL for several hours to days. When finished, it is possible to examine the findings in the output folder. In there, AFL lists all the test cases that generated crashes and bugs. Following those findings can then be analysed and rated. As a final act, reports can be written and passed to the corresponding responsible.

6 Conclusion

Fuzzing is a very convenient and efficient way of testing software. It can be remarkably effective in finding bugs in applications and is capable of testing huge programs and systems in a short time, with relatively low effort. While it requires less expertise than other testing techniques it is important to have a good understanding on how to setup the fuzzer accordingly to the target system. It is crucial to choose the right method for generating test cases. Otherwise, there is a risk in targeting the wrong interface or system. Hence a generally good knowledge of It-infrastructures is required.

Since modern applications keep growing in complexity and new fields like the internet of things increase the landscape of potentially vulnerable targets, fuzzing will continuously gain relevance.

References

- [Asc20] Cornelius Aschermann. Algorithmic improvements for feedback-driven fuzzing. 2020.
- [Dec20] Sergej Dechand. Introduction and fuzzing 101, <https://www.youtube.com/watch?v=7adrxnbuew0>. 2020.
- [fuz21] fuzzstation. Welcome to fuzzgoat. 2021.
- [Gar12] Parul Garg. Fuzzing: Mutation vs. generation. 2012.
- [li2] Fuzzing: a survey author=Li, Jun and Zhao, Bodong and Zhang, Chao, journal=Cybersecurity, volume=3, number=3, pages=1–13, year=2018, publisher=SpringerOpen.
- [Luk12] Dejan Lukan. Sulley fuzzing framework intro. 2012.
- [McG04] McGraw. Static analysis for security. 2004.
- [NIS19] NIST. Vulnerability metrics. 2019.
- [O'D22] Aimee O'Driscoll. 25+ cyber security vulnerability statistics and facts of 2022. 2022.
- [OWA22] OWASP. Fuzzing. 2022.
- [rea22] readthedocs.io. fuzzing with afl fuzz. 2022.
- [Rei20] Jonathan Reimer. 5 cves found with feedback-based fuzzing, <https://www.code-intelligence.com/blog/5-cves-found-with-feedback-based-fuzzing>. 2020.
- [Tun19] Liam Tung. Google: We've open-sourced clusterfuzz tool that found 16,000 bugs in chrome. 2019.
- [Zel22] Andreas Zeller. Fuzzing: Breaking things with random inputs. 2022.