Fuzz Testing

1 Introduction

Fuzz Testing, or Fuzzing, is a software testing family of techniques to find security vulnerabilities repeatedly providing the program under test with generated inputs, usually unexpected or invalid.

The program is monitored for faults, such as crashes or assertion errors, and inputs that trigger faults are reported to the user.

Almost all the software that expects some kind of input can be tested using Fuzzing, but the setup can be difficult for some programs with complex input mechanisms, such as stateful servers that expect a request-response pattern.

For applications that take inputs in standard ways, such as reading from a file, a fuzzer is easy to use and almost automatic.

In the latest 20 years, fuzzers becomes a key asset for Security Researchers that launch them and wait for shallow bugs while doing other tasks in the case of naive fuzzing, or that build custom optimized fuzzers for each application to uncover deep bugs.

2 Fuzzers Classification

Fuzzers are commonly classified looking at 3 properties:

1. How many information they need from the actual program under test;
2. How they generate new testcases;
3. If the exploration of the program behaviors is guided using some information;

2.1 Program Structure Awareness

Fuzzing can be a Random Testing technique or a Structural one depending on the amount of information from the program that the fuzzer uses.

Typically, there are 3 categories:

2.1.1 White-box Fuzzers

White-box fuzzers require a full picture of the program. Concolic executors are fuzzer of this kind because they collected a model of the program in terms of logic constraints during the executions.

2.1.2 Black-box Fuzzers

The black-box ones are blind regards the program, for instance, they have just the knowledge about the input format and randomly generate input regardless of the actual implementation.

2.1.3 Grey-box Fuzzers

Grey-box fuzzers use an approach that is in the middle. They require some information, but not a large amount, making this kind of fuzzers effective and still fast. Which part of the code is covered by a testcase is an example of minimal information required.

2.2 Testcase Generation

Fuzzer can generate testcases from scratch, using a model of the input format for instance, or by mutation. Mutational fuzzers derive testcases from other testcases.

2.3 Guided Fuzzing

Fuzzers can be guided by some information in order to explore better some program behavior.

A very effective fuzzing technique is Coverage Guided Fuzzing, a popular twist that introduces code coverage as feedback for testcase generation. There is strong empirical evidence that this technique uncovers more bugs in software than blind fuzzing.

The exploration in this kind of fuzzers is a novelty search, when a new testcase that triggers a new code portion that was previously unexplored is generated, it is saved in a corpus for further processing. Typically, this approach is combined with the generation by mutation, making fuzzing an evolutionary algorithm.

Other feedbacks can be used to guide fuzzers, for instance, the size of memory allocations can be maximized to catch out-of-memory bugs.

3 Challenges in Fuzzing

The exploration of the code can suffer from some roadblocks, such as checksum checks, that makes difficult the exploration of the code behind these roadblocks.

Another important challenge in Fuzzing is Path Explosion, when a fuzzer considers a testcase as interesting too often and the size of the corpus becomes too large. This is a typical problem of white-box fuzzers that tries to enumerate all the paths in the Control Flow Graph.

An important direction of research is generating testcases that are almost valid without the need for an input model, which is manual work.

At the time of writing, public available fuzzers have difficulties in scaling thorough CPU cores and multiple machines, and the synchronized scaling of these programs is a challenge.