Fuzzing 101

DeMarcus Thomas

SWA

12/01/2022

Overview

- What is Fuzzing?
- Use Cases
- Fuzzing Types
- Common Tools
- Hands-on Examples

What is Fuzzing?

- Automatic test generation and execution with the goal of finding security vulnerabilities
  - Uses malformed/semi-malformed payloads via automation
- Within the class of dynamic analysis as it runs executables for path exploration
- Think of this as a version of stress testing for software
- Has been used to find thousands of software vulnerabilities in various software environments

Fuzzing In-Practice

- Commonly thought of being used by security professionals after software is created

- Used to find vulnerabilities in software tools and provide recommendations for mitigating problems

Fuzzing In-Practice cont.

- It is recommended as a best practice to incorporate fuzzing techniques into the actual software development life cycle
- More vulnerabilities can be discovered before they are introduced into projects
- Fuzzing should not be a one-time action, but a continuous process

Common Fuzzer Structure

- Two main components exists for fuzzers:
  - Input generator - automatically generates test cases
  - Monitor - detects errors in the program at runtime
- Complexity for both components describes the various types of fuzzers that are available

Fuzzing Types

- Black-box (Random) Fuzzing
- Mutation-based Fuzzing
- Grammar-based Fuzzing
- Coverage Guided Fuzzing
- Generation-Based Fuzzing

Black-box Fuzzing

- Randomly generates inputs for the target program
- Inputs are selected without awareness of a program's logic
- Technique can be effective for programs/applications that have not been tested before, but not very efficient

Mutation-based Fuzzing

- Uses inputs that are mutated slightly from valid originals
  - In most cases seed values are provided to start from
- Can follow simple tactics to change the valid inputs slightly to reach deeper branches of code
- The inputs that then cause deeper branches are then used as seeds to the input generator for even deeper exploration

Grammar-based Fuzzing

- Technique that can fuzz applications or programs that require a specific format
- Can be more efficient than random alternatives mentioned previously
- Often the user will provide the proper format that should be given to this type of fuzzer

Coverage Guided Fuzzing

- Send random inputs to a targeted program/application to maximize the amount of code-coverage
- Code coverage is a metric used to measure the quality of tests that are often used by developers and test engineers to decide what areas of code need more testing
- If a mutated input increases code coverage, it is added to the overall corpus and is used to generate additional inputs

Generation-Based Fuzzing

- Fuzzing strategy that can generate inputs from scratch with no sample input needed
- Can use strategies such as genetic or evolutionary algorithms for selecting new inputs
- Providing sample inputs are not required, but when provided can make the fuzzing process more efficient

What is a Fuzz Harness?

- Used to manage the interaction between a fuzzer and the program under test
- Manages the inputs from the fuzzer and passes them to the target program
- Fuzzing harnesses are commonly used to test libraries and APIs that need to be tested

Fuzz Harnesses Best Practices

- Fuzz libraries and APIs, rather than standalone programs
- Ensure the program behavior is as deterministic as possible. The same input must result in the same output
- The called library should avoid exiting (by exit() or raising signals) for valid code paths
- It should avoid mutating the global state of the program as it will confuse the fuzzer

Building a Fuzz Harness Demo

Common Fuzzing Tools

- The tools that will be the focus of this course are listed below:
  - LibFuzzer
  - AFL
  - HonggFuzz

- For each tool we will provide some basic usage and a simple example to apply it to

LibFuzzer

- Library that assists with the fuzzing of applications and other libraries
  - Characterized as an in-process, coverage –guided , evolutionary fuzzing engine
- Linked with the library under test and tracks which areas of code are reached via a target function entry point
- Mutations on the inputs are performed to maximize code coverage

LibFuzzer cont.

- Built into ***clang*** by default starting with version 6.0

- Uses a special target function to start fuzzing
  - ***LLVMFuzzerTestOneInput***

- A full example from the LibFuzzer documentation is shown below:

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    DoSomethingInterestingWithMyAPI(Data, Size);
    return 0;
}
```

LibFuzzer Demo

AFL (American Fuzzy Lop)

- Created by Michael Zelwaski
  - Released in 2013
- Fuzzer that utilizes genetic algorithms to efficiently increase code coverage of a target executable
- Very successful in finding bugs in software
  - Firefox, bash, OpenSSL, etc.

AFL (American Fuzzy Lop) cont.

- Requires user to provide a targeted application with a sample command to test
- Requires at least one test case input file for AFL to mutate
- AFL will then run to determine crashes and hangs that could be indicators of a security vulnerability

AFL Demo

HonggFuzz

- Security oriented fuzzer that supports feedback-driven, evolutionary vulnerability discovery
- Also, a multi-threaded and multi-process tool that can share a corpus among its multiple instances
- Can provide input via STDIN or use the target function mentioned previously with LibFuzzer

HonggFuzz Demo

Tool Summary

- Each tool has slightly different tactics that are used to fuzz a target application
- AFL and HonggFuzz can fuzz continuously, even if a crash or hang is found, until a user stops a fuzzing run
- Each tool can use the same harness when using the function ***LLVMFuzzerTestOneInput***