Go Fuzzing

- Overview
- Writing fuzz tests
  - Requirements
  - Suggestions {#suggestions}
- Running fuzz tests
  - Command line output
  - Failing input
  - Custom settings {#custom-settings}
- Corpus file format
- Resources
- Glossary {#glossary}
- Feedback

Go supports fuzzing in its standard toolchain beginning in Go 1.18. Native Go fuzz tests are supported by OSS-Fuzz.

Try out the tutorial for fuzzing with Go.

Overview

Fuzzing is a type of automated testing which continuously manipulates inputs to a program to find bugs. Go fuzzing uses coverage guidance to intelligently walk through the code being fuzzed to find and report failures to the user. Since it can reach edge cases which humans often miss, fuzz testing can be particularly valuable for finding security exploits and vulnerabilities.

Below is an example of a fuzz test, highlighting its main components.

Example code showing the overall fuzz test, with a fuzz target within it. Before the fuzz target is a c
Example code showing the overall fuzz test, with a fuzz target within it. Before the fuzz target is a c

Writing fuzz tests

Requirements

Below are rules that fuzz tests must follow.

- A fuzz test must be a function named like `FuzzXxx`, which accepts only a `*testing.F`, and has no return value.
- Fuzz tests must be in *_test.go files to run.
- A fuzz target must be a method call to `(*testing.F).Fuzz` which accepts a `*testing.T` as the first parameter, followed by the fuzzing arguments. There is no return value.
- There must be exactly one fuzz target per fuzz test.
- All seed corpus entries must have types which are identical to the fuzzing arguments, in the same order. This is true for calls to `(*testing.F).Add` and any corpus files in the testdata/fuzz directory of the fuzz test.
- The fuzzing arguments can only be the following types:
  - `string`, `[]byte`
  - `int`, `int8`, `int16`, `int32`/`rune`, `int64`
  - `uint`, `uint8`/`byte`, `uint16`, `uint32`, `uint64`
  - `float32`, `float64`
  - `bool`

Suggestions {#suggestions}

Below are suggestions that will help you get the most out of fuzzing.

- Fuzz targets should be fast and deterministic so the fuzzing engine can work efficiently, and new failures and code coverage can be easily reproduced.
- Since the fuzz target is invoked in parallel across multiple workers and in nondeterministic order, the state of a fuzz target should not persist past the end of each call, and the behavior of a fuzz target should not depend on global state.

Running fuzz tests

There are two modes of running your fuzz test: as a unit test (default `go test`), or with fuzzing (`go test -fuzz=FuzzTestName`).

Fuzz tests are run much like a unit test by default. Each seed corpus entry will be tested against the fuzz target, reporting any failures before exiting.

To enable fuzzing, run `go test` with the `-fuzz` flag, providing a regex matching a single fuzz test. By default, all other tests in that package will run before fuzzing begins. This is to ensure that fuzzing won't report any issues that would already be caught by an existing test.

Note that it is up to you to decide how long to run fuzzing. It is very possible that an execution of fuzzing could run indefinitely if it doesn't find any errors. There will be support to run these fuzz tests continuously using tools like OSS-Fuzz in the future, see Issue #50192.

Note: Fuzzing should be run on a platform that supports coverage instrumentation (currently AMD64 and ARM64) so that the corpus can meaningfully grow as it runs, and more code can be covered while fuzzing.

Command line output

While fuzzing is in progress, the fuzzing engine generates new inputs and runs them against the provided fuzz target. By default, it continues to run until a failing input is found, or the user cancels the process (e.g. with Ctrl^C).

The output will look something like this:

```
~ go test -fuzz FuzzFoo
fuzz: elapsed: 0s, gathering baseline coverage: 0/192 completed
fuzz: elapsed: 0s, gathering baseline coverage: 192/192 completed, now fuzzing with 8 workers
fuzz: elapsed: 3s, execs: 325017 (108336/sec), new interesting: 11 (total: 202)
fuzz: elapsed: 6s, execs: 680218 (118402/sec), new interesting: 12 (total: 203)
fuzz: elapsed: 9s, execs: 1039901 (119895/sec), new interesting: 19 (total: 210)
fuzz: elapsed: 12s, execs: 1386684 (115594/sec), new interesting: 21 (total: 212)
PASS
ok      foo 12.692s
```

The first lines indicate that the "baseline coverage" is gathered before fuzzing begins.

To gather baseline coverage, the fuzzing engine executes both the seed corpus and the generated corpus, to ensure that no errors occurred and to understand the code coverage the existing corpus already provides.

The lines following provide insight into the active fuzzing execution:

- elapsed: the amount of time that has elapsed since the process began

- execs: the total number of inputs that have been run against the fuzz target (with an average execs/sec since the last log line)
- new interesting: the total number of "interesting" inputs that have been added to the generated corpus during this fuzzing execution (with the total size of the entire corpus)

For an input to be "interesting", it must expand the code coverage beyond what the existing generated corpus can reach. It's typical for the number of new interesting inputs to grow quickly at the start and eventually slow down, with occasional bursts as new branches are discovered.

You should expect to see the "new interesting" number taper off over time as the inputs in the corpus begin to cover more lines of the code, with occasional bursts if the fuzzing engine finds a new code path.

Failing input

A failure may occur while fuzzing for several reasons:

- A panic occurred in the code or the test.
- The fuzz target called `t.Fail`, either directly or through methods such as `t.Error` or `t.Fatal`.
- A non-recoverable error occurred, such as an `os.Exit` or stack overflow.
- The fuzz target took too long to complete. Currently, the timeout for an execution of a fuzz target is 1 second. This may fail due to a deadlock or infinite loop, or from intended behavior in the code. This is one reason why it is suggested that your fuzz target be fast.

If an error occurs, the fuzzing engine will attempt to minimize the input to the smallest possible and most human readable value which will still produce an error. To configure this, see the custom settings section.

Once minimization is complete, the error message will be logged, and the output will end with something like this:

```
    Failing input written to
testdata/fuzz/FuzzFoo/a878c3134fe0404d44eb1e662e5d8d4a24beb05c3d68354903670ff65513ff49
    To re-run:
    go test -run=FuzzFoo/a878c3134fe0404d44eb1e662e5d8d4a24beb05c3d68354903670ff65513ff49
FAIL
exit status 1
FAIL    foo 0.839s
```

The fuzzing engine wrote this failing input to the seed corpus for that fuzz test, and it will now be run by default with `go test`, serving as a regression test once the bug has been fixed.

The next step for you will be to diagnose the problem, fix the bug, verify the fix by re-running `go test`, and submit the patch with the new testdata file acting as your regression test.

Custom settings {#custom-settings}

The default go command settings should work for most use cases of fuzzing. So typically, an execution of fuzzing on the command line should look like this:

```
$ go test -fuzz={FuzzTestName}
```

However, the `go` command does provide a few settings when running fuzzing. These are documented in the `cmd/go` package docs.

To highlight a few:

- `-fuzztime`: the total time or number of iterations that the fuzz target will be executed before exiting, default indefinitely.
- `-fuzzminimizetime`: the time or number of iterations that the fuzz target will be executed during each minimization attempt, default 60sec. You can completely disable minimization by setting `-fuzzminimizetime 0` when fuzzing.
- `-parallel`: the number of fuzzing processes running at once, default `$GOMAXPROCS`. Currently, setting -cpu during fuzzing has no effect.

Corpus file format

Corpus files are encoded in a special format. This is the same format for both the seed corpus, and the generated corpus.

Below is an example of a corpus file:

```
go test fuzz v1
[]byte("hello\\xbd\\xb2=\\xbc ⌘")
int64(572293)
```

The first line is used to inform the fuzzing engine of the file's encoding version. Although no future versions of the encoding format are currently planned, the design must support this possibility.

Each of the lines following are the values that make up the corpus entry, and can be copied directly into Go code if desired.

In the example above, we have a `[]byte` followed by an `int64`. These types must match the fuzzing arguments exactly, in that order. A fuzz target for these types would look like this:

```
f.Fuzz(func(*testing.T, []byte, int64) {})
```

The easiest way to specify your own seed corpus values is to use the `(*testing.F).Add` method. In the example above, that would look like this:

```
f.Add([]byte("hello\\xbd\\xb2=\\xbc ⌘"), int64(572293))
```

However, you may have large binary files that you'd prefer not to copy as code into your test, and instead remain as individual seed corpus entries in the testdata/fuzz/{FuzzTestName} directory. The `file2fuzz` tool at golang.org/x/tools/cmd/file2fuzz can be used to convert these binary files to corpus files encoded for `[]byte`.

To use this tool:

```
$ go install golang.org/x/tools/cmd/file2fuzz@latest
$ file2fuzz -h
```

Resources

- **Tutorial**:
  - Try out the tutorial for fuzzing with Go for a deep dive into the new concepts.
  - For a shorter, introductory tutorial of fuzzing with Go, please see the blog post.
- **Documentation**:
  - The `testing` package docs describes the `testing.F` type which is used when writing fuzz tests.
  - The `cmd/go` package docs describe the flags associated with fuzzing.
- **Technical details**:
  - Design draft
  - Proposal

Glossary {#glossary}

corpus entry: An input in the corpus which can be used while fuzzing. This can be a specially-formatted file, or a call to `(*testing.F).Add`.

coverage guidance: A method of fuzzing which uses expansions in code coverage to determine which corpus entries are worth keeping for future use.

failing input: A failing input is a corpus entry that will cause an error or panic when run against the fuzz target.

fuzz target: The function of the fuzz test which is executed for corpus entries and generated values while fuzzing. It is provided to the fuzz test by passing the function to `(*testing.F).Fuzz`.

fuzz test: A function in a test file of the form `func FuzzXxx(*testing.F)` which can be used for fuzzing.

fuzzing: A type of automated testing which continuously manipulates inputs to a program to find issues such as bugs or vulnerabilities to which the code may be susceptible.

fuzzing arguments: The types which will be passed to the fuzz target, and mutated by the mutator.

fuzzing engine: A tool that manages fuzzing, including maintaining the corpus, invoking the mutator, identifying new coverage, and reporting failures.

generated corpus: A corpus which is maintained by the fuzzing engine over time while fuzzing to keep track of progress. It is stored in `$GOCACHE`/fuzz. These entries are only used while fuzzing.

mutator: A tool used while fuzzing which randomly manipulates corpus entries before passing them to a fuzz target.

package: A collection of source files in the same directory that are compiled together. See the Packages section in the Go Language Specification.

seed corpus: A user-provided corpus for a fuzz test which can be used to guide the fuzzing engine. It is composed of the corpus entries provided by f.Add calls within the fuzz test, and the files in the testdata/fuzz/{FuzzTestName} directory within the package. These entries are run by default with `go test`, whether fuzzing or not.

test file: A file of the format xxx_test.go that may contain tests, benchmarks, examples and fuzz tests.

vulnerability: A security-sensitive weakness in code which can be exploited by an attacker.

Feedback

If you experience any problems or have an idea for a feature, please file an issue.

For discussion and general feedback about the feature, you can also participate in the #fuzzing channel in Gophers Slack.

□□□ #2
□ □□□ □□□ 2026-10-26 20:26:29 UTC
□ □□□ □□□ 2026-10-26 20:26:29 UTC