fuzz

- Prerequisites
- Create a folder for your code {#create_folder}
- Add code to test {#code_to_test}
  - Write the code
  - Run the code
- Add a unit test {#unit_test}
  - Write the code
  - Run the code
- Add a fuzz test {#fuzz_test}
  - Write the code
  - Run the code
- Fix the invalid string error {#fix_invalid_string_error}
  - Diagnose the error
    - Write the code
    - Run the code
  - Fix the error
    - Write the code
    - Run the code
- Fix the double reverse error {#fix_double_reverse_error}
  - Diagnose the error
    - Write the code
    - Run the code
  - Fix the error
    - Write the code
    - Run the code
- Conclusion {#conclusion}
- Completed code

This tutorial introduces the basics of fuzzing in Go. With fuzzing, random data is run against your test in an attempt to find vulnerabilities or crash-causing inputs. Some examples of vulnerabilities that can be found by fuzzing are SQL injection, buffer overflow, denial of service and cross-site scripting attacks.

In this tutorial, you'll write a fuzz test for a simple function, run the go command, and debug and fix issues in the code.

For help with terminology throughout this tutorial, see the Go Fuzzing glossary.

You'll progress through the following sections:

1. Create a folder for your code.
2. Add code to test.
3. Add a unit test.
4. Add a fuzz test.
5. Fix two bugs.
6. Explore additional resources.

Note: For other tutorials, see Tutorials.

Note: Go fuzzing currently supports a subset of built-in types, listed in the Go Fuzzing docs, with support for more built-in types to be added in the future.

Prerequisites

- **Go.** We recommend using the latest version of Go to follow this tutorial. For installation instructions, see Installing Go.
- **A tool to edit your code.** Any text editor you have will work fine.
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.
- **An environment that supports fuzzing.** Go fuzzing with coverage instrumentation is only available on AMD64 and ARM64 architectures currently.

Create a folder for your code {#create_folder}

To begin, create a folder for the code you'll write.

1. Open a command prompt and change to your home directory.
   On Linux or Mac:

   ```
   $ cd
   ```

   On Windows:

   ```
   C:\> cd %HOMEPATH%
   ```

   The rest of the tutorial will show a $ as the prompt. The commands you use will work on Windows too.
2. From the command prompt, create a directory for your code called fuzz.

   ```
   $ mkdir fuzz
   $ cd fuzz
   ```

3. Create a module to hold your code.
   Run the `go mod init` command, giving it your new code's module path.

   ```
   $ go mod init example/fuzz
   go: creating new go.mod: module example/fuzz
   ```

   Note: For production code, you'd specify a module path that's more specific to your own needs. For more, be sure to see Managing dependencies.

Next, you'll add some simple code to reverse a string, which we'll fuzz later.

Add code to test {#code_to_test}

In this step, you'll add a function to reverse a string.

Write the code

1. Using your text editor, create a file called main.go in the fuzz directory.
2. Into main.go, at the top of the file, paste the following package declaration.

   ```
   package main
   ```

   A standalone program (as opposed to a library) is always in package `main`.
3. Beneath the package declaration, paste the following function declaration.

```
func Reverse(s string) string {
    b := []byte(s)
    for i, j := 0, len(b)-1; i {{raw "<"}} len(b)/2; i, j = i+1, j-1 {
        b[i], b[j] = b[j], b[i]
    }
    return string(b)
}
```

This function will accept a `string`, loop over it a `byte` at a time, and return the reversed string at the end.
Note: This code is based on the `stringutil.Reverse` function within golang.org/x/example.

4. At the top of main.go, beneath the package declaration, paste the following `main` function to initialize a string, reverse it, print the output, and repeat.

```
func main() {
    input := "The quick brown fox jumped over the lazy dog"
    rev := Reverse(input)
    doubleRev := Reverse(rev)
    fmt.Printf("original: %q\n", input)
    fmt.Printf("reversed: %q\n", rev)
    fmt.Printf("reversed again: %q\n", doubleRev)
}
```

This function will run a few `Reverse` operations, then print the output to the command line. This can be helpful for seeing the code in action, and potentially for debugging.

5. The `main` function uses the fmt package, so you will need to import it.
The first lines of code should look like this:

```
package main


import "fmt"
```

Run the code

From the command line in the directory containing main.go, run the code.

```
$ go run .
original: "The quick brown fox jumped over the lazy dog"
reversed: "god yzal eht revo depmuj xof nworb kciuq ehT"
reversed again: "The quick brown fox jumped over the lazy dog"
```

You can see the original string, the result of reversing it, then the result of reversing it again, which is equivalent to the original.

Now that the code is running, it’s time to test it.

Add a unit test {#unit_test}

In this step, you will write a basic unit test for the `Reverse` function.

Write the code

1. Using your text editor, create a file called reverse_test.go in the fuzz directory.
2. Paste the following code into reverse_test.go.

```
package main

import (
    "testing"
)

func TestReverse(t *testing.T) {
    testcases := []struct {
        in, want string
    }{
        {"Hello, world", "dlrow ,olleH"},
        {" ", " "},
        {"!12345", "54321!"},
    }
    for _, tc := range testcases {
        rev := Reverse(tc.in)
        if rev != tc.want {
                t.Errorf("Reverse: %q, want %q", rev, tc.want)
        }
    }
}
```

   This simple test will assert that the listed input strings will be correctly reversed.

Run the code

Run the unit test using `go test`

```
$ go test
PASS
ok      example/fuzz  0.013s
```

Next, you will change the unit test into a fuzz test.

Add a fuzz test {#fuzz_test}

The unit test has limitations, namely that each input must be added to the test by the developer. One benefit of fuzzing is that it comes up with inputs for your code, and may identify edge cases that the test cases you came up with didn't reach.

In this section you will convert the unit test to a fuzz test so that you can generate more inputs with less work!

Note that you can keep unit tests, benchmarks, and fuzz tests in the same *_test.go file, but for this example you will convert the unit test to a fuzz test.

Write the code

In your text editor, replace the unit test in reverse_test.go with the following fuzz test.

```
func FuzzReverse(f *testing.F) {
    testcases := []string{"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc)  // Use f.Add to provide a seed corpus
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev := Reverse(orig)
        doubleRev := Reverse(rev)
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
```

```
}
```

Fuzzing has a few limitations as well. In your unit test, you could predict the expected output of the `Reverse` function, and verify that the actual output met those expectations.

For example, in the test case `Reverse("Hello, world")` the unit test specifies the return as `"dlrow ,olleH"`.

When fuzzing, you can't predict the expected output, since you don't have control over the inputs.

However, there are a few properties of the `Reverse` function that you can verify in a fuzz test. The two properties being checked in this fuzz test are:

1. Reversing a string twice preserves the original value
2. The reversed string preserves its state as valid UTF-8.

Note the syntax differences between the unit test and the fuzz test:

- The function begins with FuzzXxx instead of TestXxx, and takes `*testing.F` instead of `*testing.T`
- Where you would expect to see a `t.Run` execution, you instead see `f.Fuzz` which takes a fuzz target function whose parameters are `*testing.T` and the types to be fuzzed. The inputs from your unit test are provided as seed corpus inputs using `f.Add`.

Ensure the new package, `unicode/utf8` has been imported.

```
package main


import (
    "testing"
    "unicode/utf8"
)
```

With the unit test converted to a fuzz test, it’s time to run the test again.

Run the code

1. Run the fuzz test without fuzzing it to make sure the seed inputs pass.

   ```
   $ go test
   PASS
   ok      example/fuzz  0.013s
   ```

You can also run `go test -run=FuzzReverse` if you have other tests in that file, and you only wish to run the fuzz test.

2. Run `FuzzReverse` with fuzzing, to see if any randomly generated string inputs will cause a failure. This is executed using `go test` with a new flag, `-fuzz`, set to the parameter `Fuzz`. Copy the command below.

```
$ go test -fuzz=Fuzz
```

Another useful flag is `-fuzztime`, which restricts the time fuzzing takes. For example, specifying `-fuzztime 10s` in the test below would mean that, as long as no failures occurred earlier, the test would exit by default after 10 seconds had elapsed. See this section of the cmd/go documentation to see other testing flags.
Now, run the command you just copied.

```
$ go test -fuzz=Fuzz
fuzz: elapsed: 0s, gathering baseline coverage: 0/3 completed
fuzz: elapsed: 0s, gathering baseline coverage: 3/3 completed, now fuzzing with 8
workers
fuzz: minimizing 38-byte failing input file...
--- FAIL: FuzzReverse (0.01s)
    --- FAIL: FuzzReverse (0.00s)
        reverse_test.go:20: Reverse produced invalid UTF-8 string "\x9c\xdd"

    Failing input written to
testdata/fuzz/FuzzReverse/af69258a12129d6cbba438df5d5f25ba0ec050461c116f777e77ea7c9a0
d217a
    To re-run:
    go test -
run=FuzzReverse/af69258a12129d6cbba438df5d5f25ba0ec050461c116f777e77ea7c9a0d217a
FAIL
exit status 1
FAIL    example/fuzz  0.030s
```

A failure occurred while fuzzing, and the input that caused the problem is written to a seed corpus file that will be run the next time `go test` is called, even without the `-fuzz` flag. To view the input that caused the failure, open the corpus file written to the testdata/fuzz/FuzzReverse directory in a text editor. Your seed corpus file may contain a different string, but the format will be the same.

```
go test fuzz v1
string("")
```

The first line of the corpus file indicates the encoding version. Each following line represents the value of each type making up the corpus entry. Since the fuzz target only takes 1 input, there is only 1 value after the version.

3. Run `go test` again without the `-fuzz` flag; the new failing seed corpus entry will be used:

```
$ go test
--- FAIL: FuzzReverse (0.00s)
    --- FAIL:
FuzzReverse/af69258a12129d6cbba438df5d5f25ba0ec050461c116f777e77ea7c9a0d217a (0.00s)
        reverse_test.go:20: Reverse produced invalid string
FAIL
exit status 1
FAIL    example/fuzz  0.016s
```

Since our test has failed, it's time to debug.

Fix the invalid string error {#fix_invalid_string_error}

In this section, you will debug the failure, and fix the bug.

Feel free to spend some time thinking about this and trying to fix the issue yourself before moving on.

Diagnose the error

There are a few different ways you could debug this error. If you are using VS Code as your text editor, you can set up your debugger to investigate.

In this tutorial, we will log useful debugging info to your terminal.

First, consider the docs for `utf8.ValidString`.

```
ValidString reports whether s consists entirely of valid UTF-8-encoded runes.
```

The current `Reverse` function reverses the string byte-by-byte, and therein lies our problem. In order to preserve the UTF-8-encoded runes of the original string, we must instead reverse the string rune-by-rune.

To examine why the input (in this case, the Chinese character 日) is causing `Reverse` to produce an invalid string when reversed, you can inspect the number of runes in the reversed string.

Write the code

In your text editor, replace the fuzz target within `FuzzReverse` with the following.

```
f.Fuzz(func(t *testing.T, orig string) {
    rev := Reverse(orig)
    doubleRev := Reverse(rev)
    t.Logf("Number of runes: orig=%d, rev=%d, doubleRev=%d", utf8.RuneCountInString(orig),
utf8.RuneCountInString(rev), utf8.RuneCountInString(doubleRev))
    if orig != doubleRev {
        t.Errorf("Before: %q, after: %q", orig, doubleRev)
    }
    if utf8.ValidString(orig) && !utf8.ValidString(rev) {
        t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
    }
})
```

This `t.Logf` line will print to the command line if an error occurs, or if executing the test with `-v`, which can help you debug this particular issue.

Run the code

Run the test using go test

```
$ go test
--- FAIL: FuzzReverse (0.00s)
    --- FAIL: FuzzReverse/28f36ef487f23e6c7a81ebdaa9feffe2f2b02b4cddaa6252e87f69863046a5e0
(0.00s)
        reverse_test.go:16: Number of runes: orig=1, rev=3, doubleRev=1
        reverse_test.go:21: Reverse produced invalid UTF-8 string "\x83\xb3\xe6"
FAIL
exit status 1
FAIL    example/fuzz    0.598s
```

The entire seed corpus used strings in which every character was a single byte. However, characters such as 日 can require several bytes. Thus, reversing the string byte-by-byte will invalidate multi-byte characters.

Note: If you're curious about how Go deals with strings, read the blog post Strings, bytes, runes and characters in Go for a deeper understanding.

With a better understanding of the bug, correct the error in the `Reverse` function.

Fix the error

To correct the `Reverse` function, let's traverse the string by runes, instead of by bytes.

Write the code

In your text editor, replace the existing Reverse() function with the following.

```
func Reverse(s string) string {
    r := []rune(s)
    for i, j := 0, len(r)-1; i {{raw "<"}} len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

The key difference is that `Reverse` is now iterating over each `rune` in the string, rather than each `byte`. Note that this is just an example, and does not handle combining characters correctly.

Run the code

1. Run the test using `go test`

   ```
   $ go test
   PASS
   ok      example/fuzz  0.016s
   ```

   The test now passes!
2. Fuzz it again with `go test -fuzz`, to see if there are any new bugs.

   ```
   $ go test -fuzz=Fuzz
   fuzz: elapsed: 0s, gathering baseline coverage: 0/37 completed
   fuzz: minimizing 506-byte failing input file...
   fuzz: elapsed: 0s, gathering baseline coverage: 5/37 completed
   --- FAIL: FuzzReverse (0.02s)
       --- FAIL: FuzzReverse (0.00s)
   ```

```
        reverse_test.go:33: Before: "\x91", after: "�"


    Failing input written to
testdata/fuzz/FuzzReverse/1ffc28f7538e29d79fce69fef20ce5ea72648529a9ca10bea392bcff28c
d015c
    To re-run:
    go test -
run=FuzzReverse/1ffc28f7538e29d79fce69fef20ce5ea72648529a9ca10bea392bcff28cd015c
FAIL
exit status 1
FAIL    example/fuzz  0.032s
```

We can see that the string is different from the original after being reversed twice. This time the input itself is invalid unicode. How is this possible if we're fuzzing with strings? Let's debug again.

Fix the double reverse error {#fix_double_reverse_error}

In this section, you will debug the double reverse failure and fix the bug.

Feel free to spend some time thinking about this and trying to fix the issue yourself before moving on.

Diagnose the error

Like before, there are several ways you could debug this failure. In this case, using a debugger would be a great approach.

In this tutorial, we will log useful debugging info in the `Reverse` function.

Look closely at the reversed string to spot the error. In Go, a string is a read only slice of bytes, and can contain bytes that aren't valid UTF-8. The original string is a byte slice with one byte, `'\x91'`. When the input string is set to `[]rune`, Go encodes the byte slice to UTF-8, and replaces the byte with the UTF-8 character `�`. When we compare the replacement UTF-8 character to the input byte slice, they are clearly not equal.

Write the code

1. In your text editor, replace the `Reverse` function with the following.

```
func Reverse(s string) string {
    fmt.Printf("input: %q\n", s)
    r := []rune(s)
    fmt.Printf("runes: %q\n", r)
    for i, j := 0, len(r)-1; i {{raw "<"}} len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

   This will help us understand what is going wrong when converting the string to a slice of runes.

Run the code

This time, we only want to run the failing test in order to inspect the logs. To do this, we will use `go test -run`.

To run a specific corpus entry within FuzzXxx/testdata, you can provide {FuzzTestName}/{filename} to `-run`. This can be helpful when debugging. In this case, set the `-run` flag equal to the exact hash of the failing test. Copy and paste the unique hash from your terminal; it will be different than the one below.

```
$ go test -run=FuzzReverse/28f36ef487f23e6c7a81ebdaa9feffe2f2b02b4cddaa6252e87f69863046a5e0
input: "\x91"
runes: ['']
input: ""
runes: ['']
--- FAIL: FuzzReverse (0.00s)
    --- FAIL: FuzzReverse/28f36ef487f23e6c7a81ebdaa9feffe2f2b02b4cddaa6252e87f69863046a5e0
(0.00s)
        reverse_test.go:16: Number of runes: orig=1, rev=1, doubleRev=1
        reverse_test.go:18: Before: "\x91", after: ""
FAIL
exit status 1
FAIL    example/fuzz    0.145s
```

Knowing that the input is invalid unicode, let's fix the error in our `Reverse` function.

Fix the error

To fix this issue, let's return an error if the input to `Reverse` isn't valid UTF-8.

Write the code

1. In your text editor, replace the existing `Reverse` function with the following.

```
func Reverse(s string) (string, error) {
    if !utf8.ValidString(s) {
        return s, errors.New("input is not valid UTF-8")
    }
    r := []rune(s)
    for i, j := 0, len(r)-1; i {{raw "<"}} len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r), nil
}
```

   This change will return an error if the input string contains characters which are not valid UTF-8.

2. Since the Reverse function now returns an error, modify the `main` function to discard the extra error value. Replace the existing `main` function with the following.

```
func main() {
    input := "The quick brown fox jumped over the lazy dog"
    rev, revErr := Reverse(input)
    doubleRev, doubleRevErr := Reverse(rev)
    fmt.Printf("original: %q\n", input)
    fmt.Printf("reversed: %q, err: %v\n", rev, revErr)
    fmt.Printf("reversed again: %q, err: %v\n", doubleRev, doubleRevErr)
}
```

   These calls to `Reverse` should return a nil error, since the input string is valid UTF-8.

3. You will need to import the errors and the unicode/utf8 packages. The import statement in main.go should look like the following.

```
import (
    "errors"
    "fmt"
    "unicode/utf8"
)
```

4. Modify the reverse_test.go file to check for errors and skip the test if errors are generated by returning.

```
func FuzzReverse(f *testing.F) {
    testcases := []string {"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc)  // Use f.Add to provide a seed corpus
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev, err1 := Reverse(orig)
        if err1 != nil {
            return
        }
        doubleRev, err2 := Reverse(rev)
        if err2 != nil {
             return
        }
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
}
```

Rather than returning, you can also call `t.Skip()` to stop the execution of that fuzz input.

Run the code

1. Run the test using go test

```
$ go test
PASS
ok      example/fuzz  0.019s
```

2. Fuzz it with `go test -fuzz=Fuzz`, then after a few seconds has passed, stop fuzzing with `ctrl-C`. The fuzz test will run until it encounters a failing input unless you pass the `-fuzztime` flag. The default is to run forever if no failures occur, and the process can be interrupted with `ctrl-C`.

```
$ go test -fuzz=Fuzz
fuzz: elapsed: 0s, gathering baseline coverage: 0/38 completed
fuzz: elapsed: 0s, gathering baseline coverage: 38/38 completed, now fuzzing with 4 workers
fuzz: elapsed: 3s, execs: 86342 (28778/sec), new interesting: 2 (total: 35)
```

```
fuzz: elapsed: 6s, execs: 193490 (35714/sec), new interesting: 4 (total: 37)
fuzz: elapsed: 9s, execs: 304390 (36961/sec), new interesting: 4 (total: 37)
...
fuzz: elapsed: 3m45s, execs: 7246222 (32357/sec), new interesting: 8 (total: 41)
^Cfuzz: elapsed: 3m48s, execs: 7335316 (31648/sec), new interesting: 8 (total: 41)
PASS
ok      example/fuzz  228.000s
```

3. Fuzz it with `go test -fuzz=Fuzz -fuzztime 30s` which will fuzz for 30 seconds before exiting if no failure was found.

```
$ go test -fuzz=Fuzz -fuzztime 30s
fuzz: elapsed: 0s, gathering baseline coverage: 0/5 completed
fuzz: elapsed: 0s, gathering baseline coverage: 5/5 completed, now fuzzing with 4
workers
fuzz: elapsed: 3s, execs: 80290 (26763/sec), new interesting: 12 (total: 12)
fuzz: elapsed: 6s, execs: 210803 (43501/sec), new interesting: 14 (total: 14)
fuzz: elapsed: 9s, execs: 292882 (27360/sec), new interesting: 14 (total: 14)
fuzz: elapsed: 12s, execs: 371872 (26329/sec), new interesting: 14 (total: 14)
fuzz: elapsed: 15s, execs: 517169 (48433/sec), new interesting: 15 (total: 15)
fuzz: elapsed: 18s, execs: 663276 (48699/sec), new interesting: 15 (total: 15)
fuzz: elapsed: 21s, execs: 771698 (36143/sec), new interesting: 15 (total: 15)
fuzz: elapsed: 24s, execs: 924768 (50990/sec), new interesting: 16 (total: 16)
fuzz: elapsed: 27s, execs: 1082025 (52427/sec), new interesting: 17 (total: 17)
fuzz: elapsed: 30s, execs: 1172817 (30281/sec), new interesting: 17 (total: 17)
fuzz: elapsed: 31s, execs: 1172817 (0/sec), new interesting: 17 (total: 17)
PASS
ok      example/fuzz  31.025s
```

   Fuzzing passed!
   In addition to the `-fuzz` flag, several new flags have been added to `go test` and can be viewed in the documentation.

   See Go Fuzzing for more information on terms used in fuzzing output. For example, "new interesting" refers to inputs that expand the code coverage of the existing fuzz test corpus. The number of "new interesting" inputs can be expected to increase sharply as fuzzing begins, spike several times as new code paths are discovered, then taper off over time.

Conclusion {#conclusion}

Nicely done! You've just introduced yourself to fuzzing in Go.

The next step is to choose a function in your code that you'd like to fuzz, and try it out! If fuzzing finds a bug in your code, consider adding it to the trophy case.

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

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

Check out the documentation at go.dev/security/fuzz for further reading.

Completed code

--- main.go ---

```
package main

import (
    "errors"
    "fmt"
    "unicode/utf8"
)

func main() {
    input := "The quick brown fox jumped over the lazy dog"
    rev, revErr := Reverse(input)
    doubleRev, doubleRevErr := Reverse(rev)
    fmt.Printf("original: %q\n", input)
    fmt.Printf("reversed: %q, err: %v\n", rev, revErr)
    fmt.Printf("reversed again: %q, err: %v\n", doubleRev, doubleRevErr)
}

func Reverse(s string) (string, error) {
    if !utf8.ValidString(s) {
        return s, errors.New("input is not valid UTF-8")
    }
    r := []rune(s)
    for i, j := 0, len(r)-1; i {{raw "<"}} len(r)/2; i, j = i+1, j-1 {
```

```
        r[i], r[j] = r[j], r[i]
    }
    return string(r), nil
}
```

--- reverse_test.go ---

```
package main

import (
    "testing"
    "unicode/utf8"
)

func FuzzReverse(f *testing.F) {
    testcases := []string{"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc) // Use f.Add to provide a seed corpus
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev, err1 := Reverse(orig)
        if err1 != nil {
            return
        }
        doubleRev, err2 := Reverse(rev)
        if err2 != nil {
            return
        }
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
}
```

Back to top

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