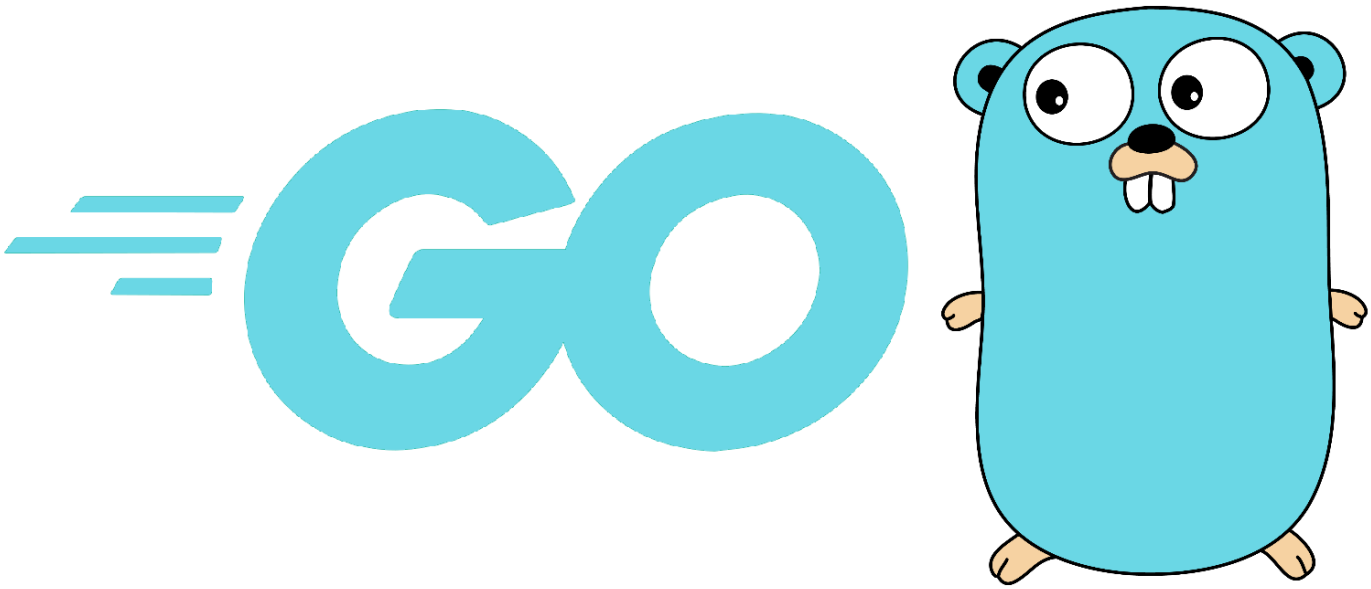
Hacking with Go



This is my attempt at filling the gap in Go security tooling. When starting to learn Go, I learned from a lot of tutorials but I could find nothing that is geared towards security professionals.

These documents are based on the `Gray/Black Hat Python/C#` series of books. I like their style. Join me as I learn more about Go and attempt to introduce Go to security denizens without fluff and through practical applications.

Table of Contents

- 01 - Setting up a Go development environment
- 02 - Basics
  - 02.1 - Packages, functions, variables, basic types, casting and constants
  - 02.2 - for, if, else, switch and defer
  - 02.3 - Pointers, structs, arrays, slices and range
  - 02.4 - Methods and interfaces
  - 02.5 - Printf, Scanf, bufio readers and maps
  - 02.6 - Goroutines and channels
  - 02.7 - Error handling
- 03 - Useful Go packages - WIP
  - 03.1 - flag package
  - 03.2 - log package

    - 04.1 - Basic TCP and UDP clients
    - 04.2 - TCP servers
    - 04.3 - TCP proxy
    - 04.4 - SSH clients
    - 04.5 - SSH Harvester
- 05 - Parsing Files
    - 05.1 - Extracting PNG Chunks
- 06 - Go-Fuzz
    - 06.1 - Go-Fuzz Quickstart
    - 06.2 - Fuzzing iprange with Go-Fuzz
    - 06.2 - Fuzzing goexif2 with Go-Fuzz

FAQ

Why not use Python?
Python reigns supreme in security and for good reason. It's a powerful programming language. There are a lot of supporting libraries out there both in security and for general use. However, I think Go has its merits and can occupy a niche.

Why not use other tutorials?
There are a lot of tutorials for Go out there. None are geared towards security professionals. Our needs are different, we want to write quick and dirty scripts that work (hence Python is so successful). Similar guides are available in Python and other programming languages.

Why not just use Black Hat Go?
There's a book named Black Hat Go by No Starch in production. Looking at the author list, I cannot compete with them in terms of experience and knowledge. That is a proper book with editors and a publisher while I am just some rando learning as I go. It does not take a lot of CPU power to decide the book will be better.

But the book is not out yet. Today is December 6th 2017 and the book is marked for release in August 2018. The book page does not have any released chapters or material. We can assume it's going to be similar to the other `gray|black hat` books. This repository and that book are inevitably going to have a lot of overlap. Think of this as warm up while we wait.

Update February 2020: Black Hat Go has been released. Please see the code samples at https://github.com/blackhat-go/bhg.

Rewrite in Rust/Haskell
Honestly I will be very much interested in a similar guide for Rust/Haskell geared for security people. Please let me know if you create one.

Feedback

I am always interested in feedback. There will be errors and there are always better ways to code. Please create an issue here. If this has helped you please let me know, it helps with the grind.

Other resources

There are tons of Go resources online. I am going to try not to re-hash what has been already created. `Hacking with Go` is not meant to be self-contained. When in doubt, use one of these resources or just search.

The following links helped me get started:

- GoDoc: https://godoc.org/
- A Tour of Go: https://tour.golang.org/
- Go by Example: https://gobyexample.com/
- Go playground: https://play.golang.org/
- Effective Go: https://golang.org/doc/effective_go.html

Similar resources to `Hacking with Go` :

- Security with Go published by Packt: https://github.com/PacktPublishing/Security-with-Go
- `goHackTools` : https://github.com/dreddsa5dies/goHackTools
- Go programming language secure coding practices guide

License

- Code in this repository is licensed under GPLv3.
- Non-code content is licensed under Creative Commons Attribution-NonCommercial 4.0 (CC BY-NC 4.0).

01 - Setting up a Go development environment

I am going to use a Windows 10 x64 Virtual Machine (VM) but Go is available for most popular platforms. I can already hear the infosec pros grunt. The Getting Started section on Go website has how-tos for most popular platforms. You can find binaries and building instructions.

You can get free Windows VMs from modern.ie. Make a snapshot after you everything is set up. They expire in 90 days and you can only re-arm them multiple times.

- Installation on Windows 10 VM
- GOPATH
- Test application

- Editor
  - Go playground
  - Offline coding
- gofmt
- Starting curly brace

Installation on Windows 10 VM

1. Go to https://golang.org/doc/install and download the MSI binary.
2. Install the MSI, choose the default location.
3. Choose a development directory. I have created a shared directory in my VM. This way I can code in host and run the guest. In my case it 's `Z:\Go` where Z is the shared drive/directory.
4. Set the following environmental variables (installer might have already set some up):
   - `GOROOT : C:\Go`
   - `GOPATH : Z:\Go` or the directory from step 3.
5. Add `C:\Go\Bin` to PATH.
6. Open a new cmd and run `go env` . You should see what you have setup.

Output of `go env` in my Windows 10 VM is:

```
$ go env
set GOARCH=amd64
set GOBIN=
set GOEXE=.exe
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GOOS=windows
set GOPATH=Z:\Go\
set GORACE=
set GOROOT=C:\Go
set GOTOOLDIR=C:\Go\pkg\tool\windows_amd64
set GCCGO=gccgo
set CC=gcc
set GOGCCFLAGS=-m64 -mthreads -fmessage-length=0
 -fdebug-prefix-map=C:\Users\IEUser\AppData\Local\Temp\go-build352203231=/tmp/go-build
 -gno-record-gcc-switches
set CXX=g++
set CGO_ENABLED=1
set CGO_CFLAGS=-g -O2
set CGO_CPPFLAGS=
set CGO_CXXFLAGS=-g -O2
set CGO_FFLAGS=-g -O2
set CGO_LDFLAGS=-g -O2
set PKG_CONFIG=pkg-config
```

GOPATH

You can write Go code anywhere but only code in a `GOPATH` directory can be executed with `go run` [2].

Go to the development path in step 3 of last section and create three directories inside it:

- `src` : Source code.
- `bin` : Compiled files.
- `pkg` : Executables.

You can clone this repository in `src` and then run everything in `code` . The directory structure looks like in the Windows 10 VM:

```
Z:\Go>tree /F

Z:.
├───bin
├───pkg
└───src
    └───Hacking-with-Go
        └───code
            └───01
                    01-01-HelloWorld.go
```

Test application

Let's write a quick "Hello World" application and run it.

```
package main

import "fmt"

func main() {

    fmt.Println("Hello World!")
}
```

And we can run it with `go run 01-01-HelloWorld.go` .

```
Z:\Go\src\hacking-with-go\code\01>go run 01-01-HelloWorld.go
Hello World!
```

Editor

Choose whatever you like. There are many editors with Go support (you will see below) to choose from. Some in no particular order are:

- SublimeText using GoSublime package.
- Atom via go-plus package.
- Visual Studio Code with Go extension.
- Vim-go.
- Emacs go-mode.

I personally use Sublime Text 3 and GoSublime.

Go playground

The online go playground at https://play.golang.org/ is good for prototyping/testing and sharing quick scripts. It's pretty useful when Go is not installed on the machine. For more information read Inside the Go Playground.

Offline coding

It's possible to run both the playground and documentation server offline.

- `godoc -http :1234` will run the the documentation server at `localhost:1234`.
- `go tool tour` will start an offline version of Tour of Go at `localhost:3999`. This allows coding offline in browser in Go playground.

gofmt

`gofmt` is Go's official formatting tool. It automatically modifies source code. The main reason behind choosing an editor with Go support is running `gofmt` automatically on your code.

I personally do not agree with `gofmt`. For example it uses tabs (I like spaces). Tab-width is fixed at four (I like two). But it's better if our code adheres to language standards.

For more information read go fmt your code. For usage see Command gofmt.

Starting curly brace

The starting curly brace needs to be on the same line as the the keyword starting the block (e.g. `for` or `if`). This is a Go standard enforced by the compiler. It's explained in the Go FAQ.

This is wrong:

```
func main()
{
```

```
    fmt.Println("Hello World!")
}
```

This is correct:

```
func main() {
    fmt.Println("Hello World!")
}
```

Continue reading ⇒ 02 - Basics

02 - Basics

This is a quick introduction to Go. This section assumes you know other programming languages (most likely Python) and are familiar with basic programming structures.

These notes were originally created during the tutorials at Tour of Go and some other sources. Then more were added to make it a reference/cheat sheet.

Table of Contents

- 02.1 - Packages, functions, variables, basic types, casting and constants
- 02.2 - for, if, else, switch and defer
- 02.3 - Pointers, structs, arrays, slices and range
- 02.4 - Methods and interfaces
- 02.5 - Printf, Scanf, bufio readers and maps
- 02.6 - Goroutines and channels
- 02.7 - Error handling

02.1 - Packages, functions, variables, basic types, casting and constants

- Packages
  - Exported names
- Functions
  - Functions can return multiple values
  - Named return values
  - init function
- Variables

    - Initialization
    - Initialization Values
    - Short variable declarations
- Basic types
- Casting
- Constants
- Raw strings

Packages

Go is divided into packages. Packages are the equivalent of modules in Python. Only the `main` package can be executed with `go run`.

We can import packages with `import`. The Hello World application imported the `fmt` package. Multiple imports are similar:

```
import (
    "fmt"
    "math/rand"
    "otherimport"
)
```

Exported names

In Go, a name is exported if it begins with a capital letter.

When importing a package, you can refer only to its exported names. Unexported names are not accessible from outside the package.

Functions

Unlike C, type comes after variable name except for pointers.

```
// 02.1-01-multiply.go
package main

import "fmt"

func multiply(x int, y int) int {
    return x * y
}

func main() {
```

```
    fmt.Println(multiply(10,20))
}
```

https://play.golang.org/p/jZrNpGAEWds

Functions can return multiple values

A function can return any number of values. Gone are the days when we had to use pointers in function parameters as extra return values.

```
// 02.1-02-addTwo.go
package main

import "fmt"

func addTwo(x int, y int) (int, int) {
    return x+2, y+2
}

func main() {
    fmt.Println(addTwo(10,20))
}
```

https://play.golang.org/p/sH0LeYIBpOM

If multiple variables have the same type we can declare them like this:

```
func addTwo(x, y int) (int, int) {
    return x+2, y+2
}
```

https://play.golang.org/p/Dwl94tWctK8

Named return values

Return values can be named. If so, they are treated as variables defined in the function.

A return statement without arguments returns the named return values. This is known as a "naked" return. *Using named return values and naked return is frowned upon unless it helps readability.*

```
// 02.1-03-addTwo2.go
package main

import "fmt"

func addTwo2(x int, y int) (xPlusTwo int, yPlusTwo int) {
    xPlusTwo = x + 2
```

```
    yPlusTwo = y + 2

    return xPlusTwo, yPlusTwo
}

func main() {
    fmt.Println(addTwo2(20,30))
}
```

https://play.golang.org/p/wiC9HJ0uxDN

init function

`init` function is used to set up the state. A common practice is to declare flags in it.

1. Imported packages are initialized.
2. Variable declarations evaluate their initializers.
3. `init` function executes.

```
// 02.1-09-init.go
package main

import "fmt"

func init() {
    fmt.Println("Executing init function!")
}

func main() {
    fmt.Println("Executing main!")
}
```

https://play.golang.org/p/HfL8YjGMsmw

Resulting in:

```
$ go run 02.1-09-init.go
Executing init function!
Executing main!
```

Like any other function, variables declared in `init` are only valid there.

Variables

Use `var`.

- `var x int`

Can be combined for multiple variables:

- `var x,y int == var x int, y int`

Initialization

Variables can be initialized.

- `var a, b int = 10, 20`

or

- `var a int = 10`
- `var b int = 20`

If initialized value is present during declaration, type can be omitted:

- `var sampleInt, sampleBoolean, sampleString = 30, true, "Hello"`

or

- `var sampleInt = 30`
- `var sampleBoolean = true`
- `var sampleString = "Hello"`

```
// 02.1-04-variables.go
package main

import "fmt"

func main() {
    var a, b int = 10, 20
    var sampleInt, sampleBoolean, sampleString = 30, true, "Hello"

    fmt.Println(a, b , sampleInt, sampleBoolean, sampleString)
}
```

https://play.golang.org/p/TnRrIC43-NR

Initialization Values

If no initial value is assigned to a declared variable, it will get a `zero` value:

- `0` for numeric types (int, float, etc.).
- `false` for the boolean type.
- `""` (the empty string) for strings.

Short variable declarations

Inside a function (including `main`), the `:=` short assignment statement can be used in place of a `var` declaration with implicit type.

Outside a function, every statement begins with a keyword (`var`, `func`) so the `:=` construct is not available.

```
// 02.1-05-short-declaration.go
package main

import "fmt"

func main() {
    sampleInt, sampleBoolean, sampleString := 30, true, "Hello"

    fmt.Println(sampleInt, sampleBoolean, sampleString)
}
```

https://play.golang.org/p/RMC-9h4eBLD

`var` statements can be put in different lines (increases readability):

```
var (
    sampleInt           = 30
    sampleBoolean       = true
    sampleString        = "Hello"
)
```

Several other Go constructs use the same format. For example `import` and `const`.

Basic types

```
bool

string

int  int8  int16  int32  int64  // use int unless you want a specific size
uint uint8 uint16 uint32 uint64 uintptr // ditto, use uint

byte // alias for uint8

rune // alias for int32
     // represents a Unicode char

float32 float64
```

```
complex64 complex128
```

Casting

Casting needs to be explicit, unlike C where some castings worked out of the box.

```
// 02.1-06-casting.go
package main

import (
    "fmt"
)

func main() {
    var a, b int = 20, 30
    // Need to convert a and b to float32 before the division
    var div float32 = float32(a) / float32(b)
    // Cast float32 to int
    var divInt = int(div)
    fmt.Println(div, divInt)
}
```

https://play.golang.org/p/wKtudyE9f8q

Constants

Declared with `const` keyword. Can be character, string, boolean or numeric. Cannot use `:=`. Coding standard requires constants to start with a capital letter.

```
// 02.1-07-const.go
package main

import "fmt"

const Whatever = "whatever"

func main() {
    fmt.Println(Whatever)

    const One = 1
    fmt.Println(One)
}
```

https://play.golang.org/p/RaNzEnRlFZ4

Multiple constants can be declared together:

```
const (
    Const1 = "Constant String"
    Int1 = 12345
    True = true
)
```

Raw strings

Go has two types of strings:

- Interpreted strings: The typical `string` type created with `"`. Can contain anything except new line and unescaped `"`.
- Raw strings: Encoded between "`" (backticks) can contain new lines and other artifacts.

```
// 02.1-08-rawstring.go
package main

import "fmt"

func main() {

    rawstr :=

        `First line

some new lines

more new lines

"double quotes"
    `

    fmt.Print(rawstr)
}
```

https://play.golang.org/p/D8TwnBhwM0o

Continue reading ⇒ 02.2 - for, if, else, switch and defer

02.2 - for, if, else, switch and defer

- For
- ++ and --

- if
- Short statements
- else
- switch
- defer

For

Similar to C with two differences:

- No parenthesis around the three components. Having parenthesis will give result in an error.
- Curly braces `{ }` are always required and the first one needs to be in the same line as for, if, etc.

It has three components:

- `for init; condition; post { }`

```
// 02.2-01-for1.go
package main

import "fmt"

func main() {
    // var sum int
    sum := 0
    for i := 0; i < 20; i++ {
        sum += i
    }

    fmt.Println(sum)
}
```

Init and post (first and last) components are optional and turn `for` into `while`:

```
// 02.2-02-for2.go
package main

import "fmt"

func main() {
    // var sum int
    sum, i := 0
    for i < 20 { // while (i<20)
        sum += i
        i++
    }
```

```
    fmt.Println(sum)
}
```

Without the condition it turns into `for(;;)` or `while(1)`

```
for {   // while(1)
    ...
}
```

++ and --

Don't be fooled by their use in `for` examples. According to the FAQ, they are "statements" and not "expressions." In other words we can use them to increase or decrease a variable by one but cannot assign the result to a different one.

This will not work:

```
// 02.2-03-incdec.go
package main

import "fmt"

func main() {
    // var sum int
    sum, i := 0
    // This will not work
    sum = i++
    fmt.Println(sum)
}
```

```
Z:\Go\src\Hacking-with-Go\code\02>go run 02.2-03-incdec.go
# command-line-arguments
.\02.2-03-incdec.go:9:9: syntax error: unexpected ++ at end of statement
```

if

Does not need parenthesis but needs curly braces.

```
// 02.2-04-if1.go
package main

import "fmt"
```

```
func main() {

    a := 10
    b := 20

    if b > a {
        fmt.Println(b, ">", a)
    }
}
```

Short statements

Short statements are interesting. They are statements that are executed before the condition. It's not a unique idea to Go because we have already seen them in `for` constructs in almost every language.

They can be used in `if` s.

```
// 02.2-05-if2.go
package main

import "fmt"

func main() {

    if var1 := 20; var1 > 10 {
        fmt.Println("Inside if:", var1)
    }
    // Cannot use the variable var1 here
}
```

In this code `var1 := 20` is executed before the `if` condition. Any variables declared in the short statement are only in scope in the `if` block and are destroyed after.

Short statements are usually used for executing a function and checking the return value with an `if` .

else

`else` is similar to C else.

If the corresponding `if` has a short statement then any variables declared in the short statement are also in scope in the `else` block.

```
// 02.2-06-else.go
package main
```

```
import "fmt"

func main() {

    if var1 := 20; var1 > 100 {
        fmt.Println("Inside if:", var1)
    } else {
        // Can use var1 here
        fmt.Println("Inside else:", var1)
    }
    // Cannot use var1 here
}
```

switch

Similar to C switch with some differences:

- Doesn't automatically go to the next `switch` statement unless you have `fallthrough` in the end. The `fallthrough` only works if it's the last statement in the case.
- Can have a short statement like `if` .

```
// 02.2-07-switch1.go
package main

import (
    "fmt"
    "math/rand" // This is not cryptographically secure!
    "time"
)

func main() {
    // Seeding rand
    rand.Seed(time.Now().UnixNano())
    fmt.Println("Choosing a random number:")

    switch num := rand.Intn(3); num {
    case 1:
        fmt.Println("1")
    case 2:
        fmt.Println("2")
    default:
        fmt.Println("3")
    }
}
```

Cases can have `if` conditions if we use a switch with an empty value:

```
// 02.2-08-switch2.go
package main

import (
    "fmt"
    "math/rand" // This is not cryptographically secure!
    "time"
)

func main() {
    // Seeding rand
    rand.Seed(time.Now().UnixNano())
    fmt.Println("Choosing a random number:")

    switch num := rand.Intn(100); {
    case num < 50:
        fmt.Println("Less than 50")
    default:
        fmt.Println("More than 50")
    }
}
```

The short statement does not have to be part of the switch:

```
// 02.2-09-switch3.go
package main

import (
    "fmt"
    "math/rand" // This is not cryptographically secure!
    "time"
)

func main() {
    // Seeding rand
    rand.Seed(time.Now().UnixNano())
    fmt.Println("Choosing a random number:")

    num := rand.Intn(100)
    switch {
    case num < 50:
        fmt.Println("Less than 50")
    default:
        fmt.Println("More than 50")
    }
}
```

defer

`defer` is another interesting feature in Go. It defers the execution of a function until the calling function returns.

It works like a stack, every time program reaches a `defer`, it will push that function with its argument values. When surrounding function returns, deferred functions are popped from the stack and executed.

```
// 02.2-10-defer1.go
package main

import "fmt"

func main() {
    defer fmt.Println("This runs after main")

    fmt.Println("Main ended")
}
```

Results in:

```
Z:\Go\src\Hacking-with-Go\code\02>go run 02.2-10-defer1.go
Main ended
This runs after main
```

Argument values are saved when the `defer` statement is reached but it is executed later.

```
// 02.2-11-defer2.go
package main

import "fmt"

func main() {
    num := 1
    defer fmt.Println("After main returns", num)

    num++
    fmt.Println("Inside main", num)
}
```

```
$ go run 02.2-11-defer2.go
Inside main 2
After main returns 1
```

The value of `num` was `1` when the print was deferred.

Continue reading ⇒ 02.3 - Pointers, structs, arrays, slices and range

02.3 - Pointers, structs, arrays, slices and range

- Pointers
  - Function arguments: variables vs. pointers
- Structs
- Arrays
- Slices
  - Slice length and capacity
  - make
  - append
- range

Pointers

Similar to C:

- Point with `*`: `var p *int == int *p;`
- Generate pointer (get address of) with `&`: `i := 1` and `p = &i`

No pointer arithmetic.

Function arguments: variables vs. pointers

Functions/methods accept both variables and pointers. The golden rule is:

- **Pass pointers when function/method needs to modify the parameter.**

When a variable is passed, the function/method gets a copy and the original copy is not modified. With pointers the underlying value is modified.

Structs

Go does not have classes. It has structs like C.

Exported field names need to be uppercase to be visible outside the defining package.

```
// 02.3-01-structs.go
package main

import "fmt"

type Student struct {
```

```
    FirstName string
    LastName  string
}

func main() {
    // Make an instance
    studentOne := Student{"Ender", "Wiggin"}

    // Now we can access fields
    fmt.Println(studentOne.FirstName)

    // We can just assign fields using names, anything not assigned will be
    // initialized with "zero" as we have seen before
    studentTwo := Student{FirstName: "Petra"}

    // We will print "{Petra }" notice the space after Petra which is supposed
    // to be the delimiter between the fields, LastName is nil because it is not
    // given a value
    fmt.Println(studentTwo)

    // Can also make a pointer to a struct
    p := &studentOne

    // Now instead of *p.LastName (doesn't work) we can just use p.LastName
    // fmt.Println((*p).LastName) will not work with error message: invalid indirect o
    fmt.Println(p.LastName)

    // Which is the same as
    fmt.Println(studentOne.LastName)

    // We can just create a pointer out of the blue
    p2 := &Student{"Hercule", "Poirot"}
    fmt.Println(p2)
}
```

Tour of Go says, we have to create a pointer to a struct to access fields while we can just do it directly as we saw in the code.

Arrays

`var a [10]int == int a[10]; .`

Arrays cannot be resized.

```
// 02.3-02-array.go
package main

import "fmt"

func main() {
```

```
    var a [5]int
    a[0] = 10
    a[4] = 20

    fmt.Println(a) // [10 0 0 0 20]

    // Array can be initialized during creation
    // characters[2] is empty
    characters := [3]string{"Ender", "Pentra"}

    fmt.Println(characters) // [Ender Pentra ]
}
```

Slices

Slice is a dynamic view of an array. Slices *don't store anything* by themselves, they reference an array. If we change something via the slice, the array is modified.

Think of slices as dynamic arrays. When a slice is created out of the blue, an underlying array is also initialized and can be modified by the slice.

```
// 02.3-03-slice1.go
package main

import "fmt"

func main() {

    // Create an array of strings with 3 members
    characters := [3]string{"Ender", "Petra", "Mazer"}

    // Last index is exclusive
    // allMembers []string := characters[0:3]
    var allMembers []string = characters[0:3]
    fmt.Println("All members", allMembers)

    var lastTwo []string = characters[1:3]
    fmt.Println("Last two members", lastTwo)

    // Replace Mazer with Bean
    fmt.Println("Replacing Mazer with Bean")
    allMembers[2] = "Bean"

    fmt.Println("All members after Bean swap", characters)

    fmt.Println("Last two members after Bean swap", lastTwo)
}
```

We can create array and slice literals. Meaning we can just initialize them by their members instead of assigning a length and then add more members. If a slice literal is created, the underlying array is also created.

```
// 02.3-04-slice2.go
package main

import "fmt"

func main() {

    // Slice literal of type struct, the underlying array is created automatically
    sliceStruct := []struct {
        a, b int
    }{
        {1, 2},
        {3, 4},
        {5, 6}, // need this comma in the end otherwise it will not work
    }

    fmt.Println(sliceStruct)
}
```

If a length is not specified during array creation, the result is a slice literal as seen above.

If we do not want to specify a length we can use `[...]`.

```
// 02.3-05-slice3.go
package main

import "fmt"

func main() {

    characters := [...]string{"Ender", "Petra", "Mazer"}

    fmt.Println(characters)
}
```

Slice length and capacity

Slices have length and capacity.

- **Length** is the current number of items in the slice. Returned by `len(slice)`.
- **Capacity** is the maximum number of items in the slice. Returned by `cap(slice)`. Capacity is determined by the number of items in the original array from the start of the slice and *not the*

size of array. For example if the slice starts from the second item (index 1) of an array, slice capacity is `len(array)-1`. This ensures that the slice cannot go past the array.

In most cases, we do not care about capacity. Create slices and append to them.

```
// 02.3-06-slice4.go
package main

import "fmt"

func main() {

    ints := [...]int{0, 1, 2, 3, 4, 5}
    fmt.Println(ints)

    slice1 := ints[2:6]

    // len=4 and cap=4 (from 3rd item of the array until the end)
    printSlice(slice1)

    slice1 = ints[2:4]

    // len=2 but cap will remain 4
    printSlice(slice1)
}

// Copied from the tour
func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

make

To create dynamically-sized arrays use `make`. `make` creates a zero-ed array and returns a slice pointing to it.

- `slice1 := make([]int, 10)` creates an int array of length 10.
- `slice2 := make([]int, 5, 10)` creates an int array of length 5 and capacity of 10.

We can **append** stuff to slices and it grows as needed:

- `slice1 = append(slice1, 1)`

We can append multiple elements:

- `slice1 = append(slice1, 1, 2, 3)`

append

In order to append one slice to another (obviously they should be of the same type), we have to use `...` as follows:

- `slice1 = append(slice1, slice2...)`

`append` is a variadic function, meaning it can an arbitrary number of arguments. By passing `slice2...`, we are essentially passing each member of `slice2` one by one to `append`.

This is pretty useful later on when we want to append two byte slices together.

```
// 02.3-07-slice-append.go
package main

import "fmt"

func main() {

    // Create a slice pointing to an int array
    s1 := make([]int, 5)

    fmt.Println(s1) // [0 0 0 0 0]

    for i := 0; i < len(s1); i++ {
        s1[i] = i
    }

    fmt.Println(s1) // [0 1 2 3 4]

    s2 := make([]int, 3)

    for i := 0; i < len(s2); i++ {
        s2[i] = i
    }

    fmt.Println(s2) // [0 1 2]

    s3 := append(s1, s2...)

    fmt.Println(s3) // [0 1 2 3 4 0 1 2]
}
```

range

`range` iterates over slices. It returns an index and *a copy of the item* stored at that index.

- `for index, value := range slice`

`value` is optional but `index` is not. Ignore either with `_`.

```go
// 02.3-08-range.go
package main

import "fmt"

func main() {
    characters := [3]string{"Ender", "Petra", "Mazer"}
    for i, v := range characters {
        fmt.Println(i, v)
    }

    // 0 Ender
    // 1 Petra
    // 2 Mazer

    fmt.Println("-----------")

    // Only using index
    for i := range characters {
        fmt.Println(i, characters[i])
    }

    fmt.Println("-----------")

    // Ignoring index
    for _, v := range characters {
        // No non-elaborate way to get index here
        fmt.Println(v)
    }

    // Ender
    // Petra
    // Mazer
}
```

Continue reading ⇒ 02.4 - Methods and interfaces# 02.4 - Methods and interfaces

- Methods
  - Create methods for slices
  - Value vs. pointer receivers
  - Pointer Receivers
  - When to use methods vs. functions
- Interfaces
- Type switch
- Stringers
  - Solution to the Stringers exercise

Methods

Methods can be defined for types (e.g. structs). A method is a function with a special *receiver*, receiver is the type that a method is defined for.

Create methods for slices

Let's say want to create a method for a string array that prints the members. First problem is that we cannot create a method for type `[]string` because it's an `unnamed type` and they cannot be method receivers. The trick is to declare a new type for `[]string` and then define the method for that type.

```
// 02.4-01-method1.go
package main

import "fmt"

// Create a new type for []string
type StringSlice []string

// Define the method for StringSlice
func (x StringSlice) PrintSlice() {
    for _, v := range x {
        fmt.Println(v)
    }
}

func main() {

    // Create an array of strings with 3 members
    characters := [3]string{"Ender", "Petra", "Mazer"}

    // Create a StringSlice
    var allMembers StringSlice = characters[0:3]

    // Now we can call the method on it
    allMembers.PrintSlice()

    // Ender
    // Petra
    // Mazer

    // allMembers.PrintSlice()
    // allMembers.PrintSlice undefined (type []string has no field or method PrintSlic
}
```

Note that we cannot call `PrintSlice()` on `[]string` although they are essentially the same type.

Value vs. pointer receivers

In the previous example we created a value receiver. In methods with value receivers, the method gets a *copy* of the object and the initial object is not modified.

We can also designate a pointer as receiver. In this case, any changes on the pointer inside the method are reflected on the referenced object.

Pointer receivers are usually used when a method changes the object or when it's called on a large struct. Because value receivers copy the whole object, a large struct will consume a lot of memory but pointer receivers do not have this overhead.

Pointer Receivers

Pointer receivers get a pointer instead of a value but can modify the referenced object.

In the following code, Tuple's fields will be modified by `ModifyTuplePointer()` but not by `ModifyTupleValue()`.

However, **this is not the case for slices** (e.g. `IntSlice` in the code). Both value and pointer receivers modify the slice.

Pointer receivers are more efficient because they do not copy the original object.

All methods for one type should either have value receivers or pointer receivers, do not mix and match like the code below :).

```
// 02.4-02-method2.go
package main

import "fmt"

// Tuple type
type Tuple struct {
    A, B int
}

// Should not change the value of the object as it works on a copy of it
func (x Tuple) ModifyTupleValue() {
    x.A = 2
    x.B = 2
}

// Should change the value of the object
func (x *Tuple) ModifyTuplePointer() {
    x.A = 3
    x.B = 3
}

type IntSlice []int
```

```
func (x IntSlice) PrintSlice() {
    fmt.Println(x)
}

// Modifies the IntSlice although it's by value
func (x IntSlice) ModifySliceValue() {
    x[0] = 1
}

// Modifies the IntSlice
func (x *IntSlice) ModifySlicePointer() {
    (*x)[0] = 2
}

func main() {

    tup := Tuple{1, 1}

    tup.ModifyTupleValue()
    fmt.Println(tup) // {1 1} - Does not change

    tup.ModifyTuplePointer()
    fmt.Println(tup) // {3 3} - Modified by pointer receiver

    var slice1 IntSlice = make([]int, 5)
    slice1.PrintSlice() // [0 0 0 0 0]

    slice1.ModifySliceValue()
    slice1.PrintSlice() // [1 0 0 0 0]

    slice1.ModifySlicePointer()
    slice1.PrintSlice() // [2 0 0 0 0]
}
```

When to use methods vs. functions

Methods are special functions. In general use methods when:

- The output is based on the state of the receiver. Functions do not care about states.
- The receiver must to be modified.
- The method and receiver are logically connected.

Interfaces

An *interface* is not Generics! An interface can be one type of a set of types that implement a set of specific methods.

For example we will define an interface which has the method `MyPrint()`. If we define and implement `MyPrint()` for type B, a variable of type B can be assigned to an interface of that type.

```
// 02.4-03-interface1.go
package main

import "fmt"

// Define new interface
type MyPrinter interface {
    MyPrint()
}

// Define a type for int
type MyInt int

// Define MyPrint() for MyInt
func (i MyInt) MyPrint() {
    fmt.Println(i)
}

// Define a type for float64
type MyFloat float64

// Define MyPrint() for MyFloat
func (f MyFloat) MyPrint() {
    fmt.Println(f)
}

func main() {

    // Define interface
    var interface1 MyPrinter

    f1 := MyFloat(1.2345)
    // Assign a float to interface
    interface1 = f1
    // Call MyPrint() on interface
    interface1.MyPrint() // 1.2345

    i1 := MyInt(10)
    // Assign an int to interface
    interface1 = i1
    // Call MyPrint() on interface
    interface1.MyPrint() // 10
}
```

Empty Interface is `interface {}` and can hold any type. We are going to use empty interfaces a lot in functions that handle unknown types.

```
// 02.4-04-interface2.go
package main

import "fmt"

var emptyInterface interface{}

type Tuple struct {
    A, B int
}

func main() {

    // Use int
    int1 := 10
    emptyInterface = int1
    fmt.Println(emptyInterface) // 10

    // Use float
    float1 := 1.2345
    emptyInterface = float1
    fmt.Println(emptyInterface) // 1.2345

    // Use custom struct
    tuple1 := Tuple{5, 5}
    emptyInterface = tuple1
    fmt.Println(emptyInterface) // {5 5}
}
```

We can access the value inside the interface after casting. But if the interface does not contain a float, it will trigger a panic:

- `myFloat := myInterface(float64)`

In order to prevent panic we can check the error returned by casting and handle the error.

- `myFloat, ok := myInterface(float64)` .

If the interface has a float, `ok` will be `true` and otherwise `false` .

```
// 02.4-05-interface3.go
package main

import "fmt"

func main() {
    var interface1 interface{} = 1234.5

    // Only print f1 if cast was successful
    if f1, ok := interface1.(float64); ok {
```

```
        fmt.Println("Float")
        fmt.Println(f1) // 1234.5
    }

    f2 := interface1.(float64)
    fmt.Println(f2) // 1234.5 No panic but not recommended

    // This will trigger a panic
    // i1 = interface1.(int)

    i2, ok := interface1.(int) // No panic
    fmt.Println(i2, ok)        // 0 false
}
```

Type switch

Type switches are usually used inside functions that accept empty interfaces. They are used to determine the type of data that inside the interface and act accordingly.

A type switch is a switch on `interface.(type)` and some cases.

```
// 02.4-06-typeswitch.go
package main

import "fmt"

func printType(i interface{}) {
    // Do a type switch on interface
    switch val := i.(type) {
    // If an int is passed
    case int:
        fmt.Println("int")
    case string:
        fmt.Println("string")
    case float64:
        fmt.Println("float64")
    default:
        fmt.Println("Other:", val)
    }
}

func main() {
    printType(10)      // int
    printType("Hello") // string
    printType(156.32)  // float64
    printType(nil)     // Other: <nil>
    printType(false)   // Other: false
}
```

Stringers

Stringers overload print methods. A Stringer is a method named `String()` that returns a `string` and is defined with a specific type as receiver (usually a struct).

```
type Stringer interface {
    String() string
}
```

After the definition, if any `Print` function is called on the struct, the Stringer will be invoked instead. For example if a struct is printed with `%v` format string verb (we will see later that this verb prints the value of an object), Stringer is invoked.

```
// 02.4-07-stringer1.go
package main

import "fmt"

// Define a struct
type Tuple struct {
    A, B int
}

// Create a Stringer for Tuple
func (t Tuple) String() string {
    // Sprintf is similar to the equivalent in C
    return fmt.Sprintf("A: %d, B: %d", t.A, t.B)
}

func main() {

    tuple1 := Tuple{10, 10}
    tuple2 := Tuple{20, 20}
    fmt.Println(tuple1) // A: 10, B: 10
    fmt.Println(tuple2) // A: 20, B: 20
}
```

Solution to the Stringers exercise

Make the `IPAddr` type implement `fmt.Stringer` to print the address as a dotted quad. For instance, `IPAddr{1, 2, 3, 4}` should print as `1.2.3.4`.

```
// 02.4-08-stringer2.go
package main

import "fmt"
```

```
type IPAddr [4]byte

// TODO: Add a "String() string" method to IPAddr.
func (ip IPAddr) String() string {
    return fmt.Sprintf("%v.%v.%v.%v", ip[0], ip[1], ip[2], ip[3])
}

func main() {
    hosts := map[string]IPAddr{
        "loopback":  {127, 0, 0, 1},
        "googleDNS": {8, 8, 8, 8},
    }
    for name, ip := range hosts {
        fmt.Printf("%v: %v\n", name, ip)
    }
}
```

Continue reading ⇒ 02.5 - Printf, Scanf, bufio readers and maps

02.5 - Printf, Scanf, bufio readers and maps

- Print
  - Print verbs
    - Decimals
    - Floats
    - Value
    - Strings
    - Others
    - Print verbs in action
- Scan
  - Scan verbs
    - Reading user input with Scanln
    - What's wrong with Scanln?
- bufio.Reader
- Maps

Print

The `fmt` package contains `printf`. It's very similar to the C equivalent.

These three need a format string:

- `fmt.Sprintf` returns a string.

- `fmt.Fprintf` takes any objects that implements `io.Writer` for example `os.Stdout` and `os.Stderr`.
- `fmt.Printf` prints to stdout.

The following are similar but do not need a format string:

- `fmt.Print` and `fmt.Println`
- `fmt.Fprint`
- `fmt.Sprint`

Print verbs

To format strings we can use verbs (also known as switches). For more information on switches, see the fmt package source.

Decimals

- `%d` : digits = numbers.
- `%nd` : n = width of number. Right justified and padded with spaces. To left justify use `-` like `%-nd`. If `n` is less than the number of digits nothing happens.
- `%b` : number in binary.
- `%c` : `chr(int)`, prints the character corresponding to the number.
- `%x` : hex.

Floats

- `%f` : float.
- `%n.mf` : n = decimal width, m = float width. Right justified. To left justify use `-` like `%-n.mf`. If n is less than the number of digits nothing happens.
- `%e` and `%E` : scientific notation (output is a bit different from each other).

Value

- `%v` or value: catch all format. Will print based on value.
- `%+v` : will print struct's field names if we are printing a struct. Has no effect on anything else.
- `%#v` : prints code that will generate that output. For example for a struct instance it will give code that creates such a struct instance and initializes it with the current values of the struct instance.

Strings

- `%q` : double-quotes the strings before printing and also prints any invisible characters.
- `%s` : string.

- `%ns` : control width of string. Right justified, padded with spaces. To left justify use `-` like `%-ns` . If n is less than the length of the string, nothing happens.

Others

- `%t` : boolean.
- `%T` : prints the type of a value. For example `int` or `main.myType` .

Print verbs in action

```
// 02.5-01-print-verbs.gos
package main

import "fmt"

type myType struct {
    field1 int
    field2 string
    field3 float64
}

func main() {

    // int
    fmt.Println("int:")
    int1 := 123
    fmt.Printf("%v\n", int1)     // 123
    fmt.Printf("%d\n", int1)     // 123
    fmt.Printf("|%6d|\n", int1)  // |   123|
    fmt.Printf("|%-6d|\n", int1) // |123   |
    fmt.Printf("%T\n", int1)     // int
    fmt.Printf("%x\n", int1)     // 7b
    fmt.Printf("%b\n", int1)     // 1111011
    fmt.Printf("%e\n", int1)     // %!e(int=123)
    fmt.Printf("%c\n", int1)     // { - 0x7B = 123
    fmt.Println()

    // float
    fmt.Println("float:")
    float1 := 1234.56
    fmt.Printf("%f\n", float1)       // 1234.560000
    fmt.Printf("|%3.2f|\n", float1)  // |1234.56|
    fmt.Printf("|%-3.2f|\n", float1) // |1234.56|
    fmt.Printf("%e\n", float1)       // 1.234560e+03
    fmt.Printf("%E\n", float1)       // 1.234560E+03
    fmt.Println()

    // string
    fmt.Println("string:")
    string1 := "Petra"
    fmt.Printf("%s\n", string1)      // Petra
```

```
    fmt.Printf("|%10s|\n", string1)  // |     Petra|
    fmt.Printf("|%-10s|\n", string1) // |Petra     |
    fmt.Printf("%T\n", string1)      // string
    fmt.Println()

    // boolean
    fmt.Println("boolean:")
    boolean1 := true
    fmt.Printf("%t\n", boolean1) // true
    fmt.Printf("%T\n", boolean1) // bool
    fmt.Println()

    // struct type
    fmt.Println("struct:")
    struct1 := myType{10, "Ender", -10.2}
    fmt.Printf("%v\n", struct1)  // {10 Ender -10.2}
    fmt.Printf("%+v\n", struct1) // {field1:10 field2:Ender field3:-10.2}
    fmt.Printf("%#v\n", struct1) // main.myType{field1:10, field2:"Ender", field3:-10.
    fmt.Printf("%T\n", struct1)  // main.myType
}
```

Scan

As expected Go has `Scan` functions for reading input. Like `Printf` the package description is comprehensive.

The functions read from standard input (`os.Stdin`):

- `Scan` : treats new lines as spaces.
- `Scanf` : parses arguments according to a format string.
- `Scanln` : reads one line.

These read from `io.Reader` s:

- `Fscan`
- `Fscanf`
- `Fscanln`

These read from an argument string:

- `Sscan`
- `Sscanf`
- `Sscanln`

As you can guess, the following stop at the first new line or EOF:

- `Scanln`

- `Fscanln`
- `Sscanln`

While these treat new lines as spaces:

- `Scan`
- `Fscan`
- `Sscan`

Similar to `Printf` we can use format strings with these functions:

- `Scanf`
- `Fscanf`
- `Sscanf`

Scan verbs

Scan verbs are the same as `Print`. `%p`, `%T` and `# +` flags are not implemented.

Apart from `%c` every other verb discards leading whitespace (except new lines).

Reading user input with Scanln

Let's start by something simple like reading a line from input:

```
// 02.5-02-scan1.go
package main

import "fmt"

func main() {

    var s string
    n, err := fmt.Scanln(&s)
    if err != nil {
        panic(err)
    }

    fmt.Printf("Entered %d word(s): %s", n, s)
}
```

All is well when input does not have any whitespace (e.g. space):

```
$ go run 02.5-02-scan1.go
HelloHello
Entered 1 word(s): HelloHello
```

But When input has whitespace:

```
$ go run 02.5-02-scan1.go
Hello Hello
panic: expected newline

goroutine 1 [running]:
main.main()
        Z:/Go/src/Hacking-with-Go/code/02/02.5/02.5-02-scan1.go:10 +0x1ae
exit status 2
```

What's wrong with Scanln?

1. `Scan` does not return the number of characters as we expect from the C equivalent. It returns the number of words entered.
2. `Scan` and friends separate words by whitespace. Meaning when we entered `Hello Hello`, they are counted as two words. `Scanln` stored the first `Hello` in `s` and was expecting a new line or EOF to finish that. Instead it got a new word and panicked.

If we wanted to just read a number or anything without whitespace, it would have worked.

If we replace `Scanln` with `Scan` in the code, the program will not panic but will ignore anything after the first whitespace.

Lesson learned **Don't use Scan for reading user input with whitespace**.

bufio.Reader

An easier way to read user input is through `bufio` readers. We are looking for the quickest ways to get things done after all.

```
// 02.5-03-bufioreadstring.go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {

    reader := bufio.NewReader(os.Stdin)
    // ReadString will read until first new line
    input, err := reader.ReadString('\n') // Need to pass '\n' as char (byte)
    if err != nil {
        panic(err)
```

```
    }

    fmt.Printf("Entered %s", input)
}
```

ReadString reads until the first occurrence of its argument (delimiter). The delimiter should be a `byte` hence we need to pass a `char` using single quotes (`\n`). `"\n"` is a string and will not work.

`bufio.Reader` has more methods for reading different types. For example we can directly read user input and convert it to bytes with [ReadBytes][bufio-readbytes].

```
// 02.5-04-bufioreadbytes.go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {

    reader := bufio.NewReader(os.Stdin)
    // Read bytes until the new line
    input, err := reader.ReadBytes('\n') // Need to pass '\n' as char (byte)
    if err != nil {
        panic(err)
    }

    // Print type of "input" and its value
    fmt.Printf("Entered type %T, %v\n", input, input)
    // Print bytes as string
    fmt.Printf("Print bytes as string with %%s %s", input)
}
```

We are printing the type of `input` and its value as-is first. Then we print the bytes as string with `%s` .

```
$ go run 02.5-04-bufioreadbytes.go
Hello 0123456789
Entered type []uint8, [72 101 108 108 111 32 48 49 50 51 52 53 54 55 56 57 13 10]
Print bytes as string with %s Hello 0123456789
```

As you can see `bytes` are just `uint8` (unsigned ints) and printing them yields decimal values and not hex. Don't worry about bytes and strings now. We will have a byte manipulation chapter.

Maps

Fast lookup/add/delete. Each key is associated with a value (similar to Python dict).

Declare an initialized map:

- `mapName := make(map[KeyType]ValueType)` .

`KeyType` needs to be a `comparable` type. `ValueType` can be anything.

If a key does not exist, the result is a zero value. For example `0` for `int` .

To check if a key exists or not (0 might be a valid value in the map) use:

- `value, ok := mapName[key]`

If `ok` is true then the key exists (and false if the key is not there).

To test for a key without getting the value drop the value like this `_, ok := mapName[key]` and then just check `ok` .

`range` iterates over the contents of a map like arrays/slices. But we get keys instead of indexes. Typically we use the range with a `for` :

- `for key, value := range mapName`

We can create a map using data:

- `m := map[string]int{"key0": 0, "key1": 1}`

Delete a key/value pair with `delete` :

- `delete(m, "key0")`

```
// 02.5-05-maps.go
package main

import "fmt"

type intMap map[int]int

// Create a Stringer for this map type
func (i intMap) String() string {

    var s string
    s += fmt.Sprintf("Map type %T\n", i)
    s += fmt.Sprintf("Length: %d\n", len(i))

    // Iterate through all key/value pairs
    for k, v := range i {
        s += fmt.Sprintf("[%v] = %v\n", k, v)
    }
    return s
```

```
}

func main() {

    // Create a map
    map1 := make(intMap)

    // Add key/value pairs
    map1[0] = 10
    map1[5] = 20

    // Print map - Stringer will be called
    fmt.Println(map1)
    // Map type main.intMap
    // Length: 2
    // [0] = 10
    // [5] = 20

    // Delete a key/value pair
    delete(map1, 0)

    fmt.Println(map1)
    // Map type main.intMap
    // Length: 1
    // [5] = 20

    // Create a map on the spot using members
    map2 := map[string]string{"key1": "value1", "key2": "value2"}

    fmt.Println(map2)
    // map[key1:value1 key2:value2]
}
```

Continue reading ⇒ 02.6 - Goroutines and channels

[bufio-readbytes]: https://golang.org/pkg/bufio/#Reader.ReadBytes# 02.6 - Goroutines and channels

- Goroutines
  - Spawning anonymous goroutines
- Channels
  - Buffered channels
  - Closing channels
  - Checking channel status
  - Reading from channels with range
  - select
  - Directed channels
- Synching goroutines

- sync.WaitGroup

Goroutines

> Concurrency is not parallelism
>
> - Rob "Commander" Pike

With that said, let's look at one of Go's main selling points, the `goroutine`. `go function(a, b)` runs the function in parallel and continues with the rest of the program.

```
// 02.6-01-goroutine1.go
package main

import "fmt"

func PrintMe(t int, count int) {
    for i := 0; i < count; i++ {
        fmt.Printf("Printing from %d\n", t)
    }
}

func main() {

    go PrintMe(0, 100)

    fmt.Println("Main finished!")
}
```

But we never see anything printed. `main` returns before goroutine is spun up and start printing:

```
$ go run 02.6-01-goroutine1.go
Main finished!
```

Lesson learned: **Always wait for goroutines to finish! (if applicable).**

Continuing the C tradition, we can wait for a key-press before ending `main`.

```
// 02.6-02-goroutine2.go
package main

import "fmt"

func PrintMe(t int, count int) {
    for i := 0; i < count; i++ {
        fmt.Printf("Printing from %d\n", t)
    }
```

```
}

func main() {

    go PrintMe(0, 10)

    // Wait for a keypress
    fmt.Scanln()
    fmt.Println("Main finished!")
}
```

This time we can see the goroutine's output:

```
$ go run 02.6-02-goroutine2.go
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
Printing from 0
e
Main finished!
```

Spawning anonymous goroutines

We can also spawn new goroutines on the spot:

```
// 02.6-03-goroutine3.go
package main

import "fmt"

func main() {

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Printf("Printing from %d\n", 0)
        }
    }()

    // Wait for a keypress
    fmt.Scanln()
    fmt.Println("Main finished!")
}
```

Channels

Channels go hand-in-hand with gorotuines. They are typed. For example if we create a channel of type `int`, we can only use it to transfer `int`s. Values are transfered using `<-`. Channels must be created before use.

Let's make a channel in honor of famous hacker 4chan and use it to transfer some numbers around:

```
// 02.6-04-channel1.go
// This will not run
package main

import "fmt"

func main() {

    fourChan := make(chan int)

    i1 := 10

    // Send i1 to channel
    fourChan <- i1
    fmt.Printf("Sent %d to channel\n", i1)

    // Receive int from channel
    i2 := <-fourChan
    fmt.Printf("Received %d from channel\n", i2)
}
```

But it doesn't work:

```
$ go run 02.6-04-channel1.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        Z:/Go/src/Hacking-with-Go/code/02/02.6/02.6-04-channel1.go:12 +0x75
exit status 2
```

[Unbuffered] Channels will not start until the other side is ready.

Our channel's "other side" is also in `main` and the channel is unbuffered (we will talk about it in a bit). Meaning there's nothing on the other side listening.

We can either send or receive the data in a goroutine (or both):

```
// 02.6-05-channel2.go
package main

import "fmt"

func main() {

    fourChan := make(chan int)

    go func() {
        // Send i1 to channel
        i1 := 10
        fourChan <- i1 // fourChan <- 10
        fmt.Printf("Sent %d to channel\n", i1)
    }()

    go func() {
        // Receive int from channel
        i2 := <-fourChan
        fmt.Println(i2)
        fmt.Printf("Received %d from channel\n", i2)
    }()

    // Wait for goroutines to finish
    fmt.Scanln()
    fmt.Println("Main Finished!")
}
```

This time we have another goroutine listening on the other side:

```
$ go run 02.6-05-channel2.go
10
Received 10 from channel
Sent 10 to channel
e
Main Finished!
```

Buffered channels

Buffered channels have capacity and only block when the buffer is full. Buffer size (as far as I know) is specified during declaration:

- `bc := make(chan int, 10)` makes an `int` channel with size `10`.

Using buffered channels we can send and receive in main:

```
// 02.6-06-channel3.go
package main
```

```
import "fmt"

func main() {

    fourChan := make(chan int, 2)

    // Send 10 to channel
    fourChan <- 10
    fmt.Printf("Sent %d to channel\n", 10)

    // Receive int from channel
    // We can also receive directly
    fmt.Printf("Received %d from channel\n", <-fourChan)
}
```

If the channel goes over capacity, we get the same fatal runtime error as before.

Closing channels

Channels can be closed. To close a channel we can use `close(fourChan)`.

Sending items to a closed channel will cause a panic.

Checking channel status

When reading from channels, we can also get a second return value:

- `i1, ok := <- fourChan`

If channel is open `ok` will be `true`. `false` means channel is closed.

Reading from a closed channel will return a zero value (e.g. `0` for most number types). See this example. `i2` is 10 before reading something from a closed channel. After it's `0`.

```
// 02.6-07-channel4.go
package main

import "fmt"

func main() {

    fourChan := make(chan int, 2)

    close(fourChan)

    i2 := 10
    fmt.Println("i2 before reading from closed channel", i2) // 10
    i2, ok := <-fourChan
```

```
    fmt.Printf("i2: %d - ok: %t", i2, ok) // i2: 10 - ok: false
}
```

Reading from channels with range

Use a `range` in a `for` to receive values from the channel in a loop until it closes like `for i:= range fourChan .`

```
// 02-08-channel5.go
package main

import "fmt"

func main() {

    fourChan := make(chan int, 10)

    go func() {
        // Send 0-9 to channel
        for i := 0; i < 10; i++ {
            fourChan <- i
        }
    }()

    go func() {
        // Receive from channel
        for v := range fourChan {
            fmt.Println(v)
        }
    }()

    // Wait for goroutines to finish
    fmt.Scanln()
    fmt.Println("Main Finished!")
}
```

If we attempt to read something from an open channel and there's nothing there, the program will block and wait until it gets something. We can use channels to sync goroutines instead of waiting for `Scanln` . Here's our example from `02.6-03-goroutine3.go` :

```
// 02.6-09-channel6.go
package main

import "fmt"

func main() {

    c := make(chan bool)
```

```
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Printf("Printing from %d\n", 0)
        }

        // Send true to channel when we are done
        c <- true
    }()

    // Main will wait until it receives something from c
    <-c
}
```

select

Another way to wait for channels to be ready is using `select`. `select` has some `case`s. It will block until one of the cases is ready and runs it. If multiple are ready, it will choose one at random.

```
// 02.6-10-channel7.go
package main

import "fmt"

func main() {

    c := make(chan int, 2)

    for i := 0; i < 10; i++ {

        select {
        case c <- i:
            // If we can write to channel, send something to it
            fmt.Println("Sent to channel", i)
        case i2 := <-c:
            // If we can read from channel, read from it and print
            fmt.Println("Received from channel", i2)
        default:
            // This is run when nothing else can be done
        }
    }
}
```

Break is never reached because there's always something to do. Increase the size of the channel and re-run the program a few times to see `select`'s randomness when multiple choices are valid.

```
$ go run 02.6-10-channel7.go
Sent to channel 0
Received from channel 0
Sent to channel 2
```

```
Sent to channel 3
Received from channel 2
Received from channel 3
Sent to channel 6
Received from channel 6
Sent to channel 8
Sent to channel 9
```

If channel is unbuffered, `default` is always triggered because there's nothing listening on the other side.

Directed channels

Channels can be directed. Meaning you can only read or write to them.

- `c1 := make(chan<- int)` : write-only int channel.
- `c2 := make(<-chan int)` : read-only int channel.

However, declaring directed channels is not useful. Because if we can never write to a read-only channel, it will never have data. Instead they are used when passing channels to functions/goroutines.

Rewriting `02.6-05-channel2.go` using directed channels:

```go
// 02.6-11-channel8.go
package main

import "fmt"

// Directed write-only channel
func Sender(c chan<- int) {
    for i := 0; i < 10; i++ {
        fmt.Println("Sent", i)
        c <- i
    }
}

func Receiver(c <-chan int) {
    for i := range c {
        fmt.Println("Received", i)
    }
}

func main() {

    fourChan := make(chan int)

    go Sender(fourChan)
    go Receiver(fourChan)
```

```
    // Wait for goroutines to finish
    fmt.Scanln()
    fmt.Println("Main Finished!")
}
```

```
$ go run 02.6-11-channel8.go
Sent 0
Sent 1
Received 0
Received 1
Sent 2
Sent 3
Received 2
Received 3
Sent 4
Sent 5
Received 4
Received 5
Sent 6
Sent 7
Received 6
Received 7
Sent 8
Sent 9
Received 8
Received 9
d
Main Finished!
```

Synching goroutines

In our previous example, we used both `Scanln` and a blocking channel to force `main` wait for goroutines to finish. There's a better way of doing this using sync.WaitGroup.

Let's assume we are generating a list of strings that need to processed. To take advantage of Go's concurrency model, we spawn a goroutine to generate the list and send each to a channel. Then we read from the channel and spawn a new goroutine for each string and process it.

This way we can start processing the generated strings as they are being generated and we do not have to create a large string slice to hold the results.

```
// 02.6-12-waitgroup1.go
package main

import "fmt"

// generateStrings generated n strings and sends them to channel.
// Channel is closed when string generation is done.
```

```
func generateStrings(n int, c chan<- string) {

    // Close channel when done
    defer close(c)
    // Generate strings
    for i := 0; i < n; i++ {
        c <- fmt.Sprintf("String #%d", i)
    }
}

// consumeString reads strings from channel and prints them.
func consumeString(s string) {
    fmt.Printf("Consumed %s\n", s)
}

func main() {
    // Create channel
    c := make(chan string)
    // Generate strings
    go generateStrings(10, c)

    for {
        select {
        // Read from channel
        case s, ok := <-c:
            // If channel is closed stop processing and return
            if !ok {
                fmt.Println("Processing finished")
                return
            }
            // Consume the string read from channel
            go consumeString(s)
        }
    }
}
```

This looks correct but it's not. Not all strings are consumed. Because the channel is closed and we return from main when generateStrings is done. However, not all consumerString goroutines are done when by then. We need to find a way to signal main to wait until all goroutines have returned.

sync.WaitGroup

We accomplish this with `sync.WaitGroup`. Before spawning each consumerString goroutine we `wg.Add(1)` to it. Every time a consumerString goroutine is finished, we subtract the counter by one with `wg.Done()` and then we wait before returning with `wg.Wait()` which blocks execution until the counter is zero.

```
package main

import (
```

```
    "fmt"
    "sync"
)

var wg sync.WaitGroup

// generateStrings generated n strings and sends them to channel.
// Channel is closed when string generation is done.
func generateStrings(n int, c chan<- string) {

    // Close channel when done
    defer close(c)
    // Generate strings
    for i := 0; i < n; i++ {
        c <- fmt.Sprintf("String #%d", i)
    }
}

// consumeString reads strings from channel and prints them.
func consumeString(s string) {
    // Decrease waitgroup's counter by one
    defer wg.Done()
    fmt.Printf("Consumed %s\n", s)
}

func main() {
    // Create channel
    c := make(chan string)
    // Generate strings
    go generateStrings(10, c)

    for {
        select {
        // Read from channel
        case s, ok := <-c:
            // If channel is closed stop processing and return
            if !ok {
                // Wait for all goroutines to finish
                wg.Wait()
                // Return
                fmt.Println("Processing finished")
                return
            }
            // Increase wg counter by one for each goroutine
            // Note this is happening inside main before spawning the goroutine
            wg.Add(1)
            // Consume the string
            go consumeString(s)
        }
    }
}
```

Continue reading ⇒ 02.7 - Error handling

sync-waitgroup: https://golang.org/pkg/sync/#WaitGroup# Error handling

- Error handling
- Errors
  - Avoiding if err != nil fatigue
  - Solution to the Errors exercise

Error handling

Go does not have try/catch or try/except. Instead almost every function returns (or should return) an error value. It's good practice to return an error value in every function and also check it after reading values from functions/channels/etc. As a result it's very common to see `if err != nil` code blocks.

Go's error handling is very controversial. Some call it genius and others not so much. For more information read the Error handling and Go blog.

Errors

`error` type is similar to `Stringer()` .

```
type error interface {
    Error() string
}
```

Create a method for the struct type named `Error()` to return error codes/messages.

```
func (e MyType) Error() string {
    return fmt.Sprintf("error message")
}
```

According to Go docs, errors strings should not be capitalized. Most built-in and package methods return an error value if an error occurs, otherwise they will return `nil` for error which means no error.

Avoiding if err != nil fatigue

Checking for errors after every function call will result in a lot of `if err != nil` blocks. One good way is to create a function to check the error and perform actions based on it. For example:

```
func checkError(err) {
    if err != nil {
```

```
            // Do something
        }
    }
```

Solution to the Errors exercise

The errors exercise is part of tour of go.

```
// 02.7-01-errors1.go
package main

import (
    "fmt"
    "math"
)

type ErrNegativeSqrt float64

func (e ErrNegativeSqrt) Error() string {
    return fmt.Sprintf("cannot Sqrt negative number: %v", float64(e))
}

func Sqrt(x float64) (float64, error) {

    if x < 0 {
        return 0, ErrNegativeSqrt(x)
    }

    // Don't need else here - why?
    return math.Sqrt(x), nil
}

func main() {
    fmt.Println(Sqrt(2))
    fmt.Println(Sqrt(-2))
}
```

Instead of creating an `Error()` method, we could create a new error type and return that using `fmt.Errorf`:

```
// 02.7-02-errors2.go
package main

import (
    "fmt"
    "math"
)

func Sqrt(x float64) (float64, error) {
```

```
    if x < 0 {
        return 0, fmt.Errorf("cannot Sqrt negative number: %v", float64(x))
    }

    return math.Sqrt(x), nil
}

func main() {
    fmt.Println(Sqrt(2))
    fmt.Println(Sqrt(-2))
}
```

Continue reading ⇒ 03 - Useful Go packages

03 - Useful Go packages

This is where this repo is going to divert a bit from BH books. This section is going to be simple guides to Go packages that will be used later. For example the `flag` package is used to create and process command line parameters and is a building block of almost every security script that hopes to be re-used. Some packages like `net` are more complex and better learned in action while building/using tools.

As I move forward and learn more, I will return and add more tutorials here.

- 03.1 - flag package: Parsing command line parameters.
- 03.2 - log package: Logging.

flag package

flag package is the Go equivalent of Python argparse. While not as powerful, it does what we expect it to do. It simplifies adding and parsing command line parameters, leaving us to concentrate on the tools. Most of our tools will need them to be actually useful (hardcoding URLs and IPs get old too fast).

- Alternative community packages
- Basic flags
  - Flag use
- Declaring flags in the init function
- Custom flag types and multiple values
- Required flags
- Alternate and shorthand flags
- Non-flag arguments

- Subcommands

Alternative community packages

Some community packages offer what `flag` does and more. In this guide I am trying to stick to the standard library. Some of these packages are:

- Cobra: A Commander for modern Go CLI interactions
- cli: A simple, fast, and fun package for building command line apps in Go

Basic flags

Declaring basic flags is easy. We can create basic types such as: `string`, `bool` and `int`.

A new flag is easy to add:

- `ipPtr := flag.String("ip", "127.0.0.1", "target IP")`
  - `String` : Flag type.
  - `ipPtr` : Pointer to flag's value.
  - `ip` : Flag name, meaning flag can be called with `-ip`.
  - `127.0.0.1` : Flag's default value if not provided.
  - `target IP` : Flag description, displayed with `-h` switch.

It's also possible to pass a pointer directly:

- `var port int`
- `flag.IntVar(&port, "port", 8080, "Port")`

```
// 03.1-01-flag1.go
package main

import (
    "flag"
    "fmt"
)

func main() {

    // Declare flags
    // Remember, flag methods return pointers
    ipPtr := flag.String("ip", "127.0.0.1", "target IP")

    var port int
    flag.IntVar(&port, "port", 8080, "Port")

    verbosePtr := flag.Bool("verbose", true, "verbosity")
```

```
    // Parse flags
    flag.Parse()

    // Hack IP:port
    fmt.Printf("Hacking %s:%d!\n", *ipPtr, port)

    // Display progress if verbose flag is set
    if *verbosePtr {
        fmt.Printf("Pew pew!\n")
    }
}
```

This program contains a mistake! Can you spot it? If not, don't worry.

`-h/-help` print usage:

```
$ go run 03.1-01-flag1.go -h
Usage of ... \_obj\exe\03.1-01-flag1.exe:
  -ip string
        target IP (default "127.0.0.1")
  -port int
        Port (default 8080)
  -verbose
        verbosity (default true)
exit status 2
```

Without any flags, default values are used:

```
$ go run 03.1-01-flag1.go
Hacking 127.0.0.1:8080!
Pew pew!
```

Flag use

Flag use is standard.

```
$ go run 03.1-01-flag1.go -ip 10.20.30.40 -port 12345
Hacking 10.20.30.40:12345!
Pew pew!
```

The problem is the default value of our boolean flag. A boolean flag is `true` if it occurs and `false` if it. We set the default value of `verbose` to `true` meaning with our current knowledge we cannot set verbose to `false` (we will see how below but it's not idiomatic).

Fix that line and run the program again:

```
$ go run 03.1-02-flag2.go -ip 10.20.30.40 -port 12345
Hacking 10.20.30.40:12345!

$ go run 03.1-02-flag2.go -ip 10.20.30.40 -port 12345 -verbose
Hacking 10.20.30.40:12345!
Pew pew!
```

`=` is allowed. Boolean flags can also be set this way (only way to set verbose to `false` in our previous program):

```
$ go run 03.1-02-flag2.go -ip=20.30.40.50 -port=54321 -verbose=true
Hacking 20.30.40.50:54321!
Pew pew!

$ go run 03.1-02-flag2.go -ip=20.30.40.50 -port=54321 -verbose=false
Hacking 20.30.40.50:54321!
```

`--flag` is also possible:

```
$ go run 03.1-02-flag2.go --ip 20.30.40.50 --port=12345 --verbose
Hacking 20.30.40.50:12345!
Pew pew!
```

Declaring flags in the init function

`init` function is a good location to declare flags. `init` function is executed after variable initialization values and before `main`. There's one little catch, variables declared in `init` are out of focus outside (and in `main`) hence we need to declare variables outside and use `*Var` methods:

```
package main

import (
    "flag"
    "fmt"
)

// Declare flag variables
var (
    ip      string
    port    int
    verbose bool
)

func init() {
    // Declare flags
    // Remember, flag methods return pointers
```

```
    flag.StringVar(&ip, "ip", "127.0.0.1", "target IP")

    flag.IntVar(&port, "port", 8080, "Port")

    flag.BoolVar(&verbose, "verbose", false, "verbosity")
}

func main() {

    // Parse flags
    flag.Parse()

    // Hack IP:port
    fmt.Printf("Hacking %s:%d!\n", ip, port)

    // Display progress if verbose flag is set
    if verbose {
        fmt.Printf("Pew pew!\n")
    }
}
```

Custom flag types and multiple values

Custom flag types are a bit more complicated. Each custom type needs to implement the flag.Value interface. This interface has two methods:

```
type Value interface {
        String() string
        Set(string) error
}
```

In simple words:

1. Create a new type `mytype`.
2. Create two methods with `*mytype` receivers named `String()` and `Set()`.
    - `String()` casts the custom type to a `string` and returns it.
    - `Set(string)` has a `string` argument and populates the type and returns an error if applicable.
3. Create a new flag without an initial value:
    - Call `flag.NewFlagSet(&var,` instead of `flag.String(`.
    - Call `flag.Var(` instead of `flag.StringVar(` or `flag.IntVar(`.

Now we can modify our previous example to accept multiple comma-separated IPs. Note, we are using the same structure of generateStrings and consumeString from section 02.6 - sync.WaitGroup. In short, we are going to generate all permutations of IP:ports and then "hack" each of them in one goroutine.

The permutation happens in its own goroutine and is results are sent to channel one by one. When all permutations are generated, channel is closed.

In main, we read from channel and spawn a new goroutine to hack each IP:port. When channel is closed, we wait for all goroutines to finish and then return.

```
package main

import (
    "errors"
    "flag"
    "fmt"
    "strings"
    "sync"
)

// 1. Create a custom type from a string slice
type strList []string

// 2.1 implement String()
func (str *strList) String() string {
    return fmt.Sprintf("%v", *str)
}

// 2.2 implement Set(*strList)
func (str *strList) Set(s string) error {
    // If input was empty, return an error
    if s == "" {
        return errors.New("nil input")
    }
    // Split input by ","
    *str = strings.Split(s, ",")
    // Do not return an error
    return nil
}

// Declare flag variables
var (
    ip      strList
    port    strList
    verbose bool
)

var wg sync.WaitGroup

func init() {
    // Declare flags
    // Remember, flag methods return pointers
    flag.Var(&ip, "ip", "target IP")

    flag.Var(&port, "port", "Port")
```

```
    flag.BoolVar(&verbose, "verbose", false, "verbosity")
}

// permutations creates all permutations of ip:port and sends them to a channel.
// This is preferable to returing a []string because we can spawn it in a
// goroutine and process items in the channel while it's running. Also save
// memory by not creating a large []string that contains all permutations.
func permutations(ips strList, ports strList, c chan<- string) {

    // Close channel when done
    defer close(c)
    for _, i := range ips {
        for _, p := range ports {
            c <- fmt.Sprintf("%s:%s", i, p)
        }
    }
}

// hack spawns a goroutine that "hacks" each target.
// Each goroutine prints a status and display progres if verbose is true
func hack(target string, verbose bool) {

    // Reduce waitgroups counter by one when hack finishes
    defer wg.Done()
    // Hack the planet!
    fmt.Printf("Hacking %s!\n", target)

    // Display progress if verbose flag is set
    if verbose {
        fmt.Printf("Pew pew!\n")
    }
}

func main() {

    // Parse flags
    flag.Parse()

    // Create channel for writing and reading IP:ports
    c := make(chan string)

    // Perform the permutation in a goroutine and send the results to a channel
    // This way we can start "hacking" during permutation generation and
    // not create a huge list of strings in memory
    go permutations(ip, port, c)

    for {
        select {
        // Read a string from channel
        case t, ok := <-c:
            // If channel is closed
            if !ok {
```

```
                // Wait until all goroutines are done
                wg.Wait()
                // Print hacking is finished and return
                fmt.Println("Hacking finished!")
                return
            }
            // Otherwise increase wg's counter by one
            wg.Add(1)
            // Spawn a goroutine to hack IP:port read from channel
            go hack(t, verbose)
        }
    }
}
```

Result:

```
$ go run 03.1-04-flag4.go -ip 10.20.30.40,50.60.70.80 -port 1234
Hacking 50.60.70.80:1234!
Hacking 10.20.30.40:1234!
Hacking finished!

$ go run 03.1-04-flag4.go -ip 10.20.30.40,50.60.70.80 -port 1234,4321
Hacking 10.20.30.40:4321!
Hacking 10.20.30.40:1234!
Hacking 50.60.70.80:4321!
Hacking 50.60.70.80:1234!
Hacking finished!

$ go run 03.1-04-flag4.go -ip 10.20.30.40,50.60.70.80 -port 1234,4321 -verbose
Hacking 10.20.30.40:4321!
Pew pew!
Hacking 50.60.70.80:4321!
Pew pew!
Hacking 10.20.30.40:1234!
Pew pew!
Hacking 50.60.70.80:1234!
Pew pew!
Hacking finished!
```

Required flags

`flag` does not support this. In Python we can use `parser.add_mutually_exclusive_group()`. Instead we have to manually check if a flag is set. This can be done by comparing a flag with it's default value or the initial zero value of type in case it does not have a default value.

This can get complicated when the flag can contain the zero value. For example an `int` flag could be set with value `0` which is the same as the default value for `int`s. Something that can help is

the number of flags after parsing available from `flag.NFlag()`. If number of flags is less than expected, we know something is wrong.

Alternate and shorthand flags

`flag` does not have support for shorthand or alternate flags. They need to be declared in a separate statement.

```
flag.BoolVar(&verbose, "verbose", false, "verbosity")
flag.BoolVar(&verbose, "v", false, "verbosity")
```

Non-flag arguments

After `flag.Parse()` it's possible to read other arguments passed to the application with `flag.Args()`. The number of them is available from `flag.NArg()` and they individually can be accessed by index using `flag.Arg(i)`.

```
// 03.1-05-args.go
package main

import (
    "flag"
    "fmt"
)

func main() {
    // Set flag
    _ = flag.Int("flag1", 0, "flag1 description")
    // Parse all flags
    flag.Parse()
    // Enumererate flag.Args()
    for _, v := range flag.Args() {
        fmt.Println(v)
    }
    // Enumerate using flag.Arg(i)
    for i := 0; i < flag.NArg(); i++ {
        fmt.Println(flag.Arg(i))
    }
}
```

Running the program with non-flag arguments results in:

```
$ go run 03.1-05-flag5.go -flag1 12 one two 3
one
two
3
```

```
one
two
3
```

Subcommands

Subcommands are possible using flag.NewFlagSet.

- `func NewFlagSet(name string, errorHandling ErrorHandling) *FlagSet`

We can decide what happens if parsing that subcommand fails with the second parameter:

```
const (
    ContinueOnError ErrorHandling = iota // Return a descriptive error.
    ExitOnError                          // Call os.Exit(2).
    PanicOnError                         // Call panic with a descriptive error.
)
```

After that we need to parse the subcommand. This is usually done by reading `os.Args[1]` (second argument after program name should be subcommand) and parsing the detected subcommand.

```
// 03.1-06-subcommand.go
package main

import (
    "flag"
    "fmt"
    "os"
)

var (
    sub1 *flag.FlagSet
    sub2 *flag.FlagSet

    sub1flag  *int
    sub2flag1 *string
    sub2flag2 int

    usage string
)

func init() {
    // Declare subcommand sub1
    sub1 = flag.NewFlagSet("sub1", flag.ExitOnError)
    // int flag for sub1
    sub1flag = sub1.Int("sub1flag", 0, "subcommand1 flag")

    // Declare subcommand sub2
```

```
    sub2 = flag.NewFlagSet("sub2", flag.ContinueOnError)
    // string flag for sub2
    sub2flag1 = sub2.String("sub2flag1", "", "subcommand2 flag1")
    // int flag for sub2
    sub2.IntVar(&sub2flag2, "sub2flag2", 0, "subcommand2 flag2")
    // Create usage
    usage = "sub1 -sub1flag (int)\nsub2 -sub2flag1 (string) -sub2flag2 (int)"
}

func main() {
    // If subcommand is not provided, print error, usage and return
    if len(os.Args) < 2 {
        fmt.Println("Not enough parameters")
        fmt.Println(usage)
        return
    }

    // Check the sub command
    switch os.Args[1] {

    // Parse sub1
    case "sub1":
        sub1.Parse(os.Args[2:])

    // Parse sub2
    case "sub2":
        sub2.Parse(os.Args[2:])

    // If subcommand is -h or --help
    case "-h":
        fallthrough
    case "--help":
        fmt.Printf(usage)
        return
    default:
        fmt.Printf("Invalid subcommand %v", os.Args[1])
        return
    }

    // If sub1 was provided and parse, print the flags
    if sub1.Parsed() {
        fmt.Printf("subcommand1 with flag %v\n", *sub1flag)
        return
    }

    // If sub2 was provided and parse, print the flags
    if sub2.Parsed() {
        fmt.Printf("subcommand2 with flags %v, %v\n", *sub2flag1, sub2flag2)
        return
    }
}
```

As you can see there's a lot of manual work in sub commands and they are not as elegant as normal flags.

Continue reading ⇒ 03.2 - log package

log package

log package is used for logging. The examples (unlike some other packages) are not very helpful. It's very bare bones and has only two logging levels.

For anything complicated use Google's glog package.

- Basic logging
- Custom logger
  - Log to file
    - Logging to multiple files/streams
  - Flag
  - Prefix
- Logging levels

Basic logging

Basic logging is similar to other languages.

```
// 03.2-01-basic-logging.go
package main

import (
    "log"
)

func main() {

    a, b := 10, 20

    log.Print("Use Print to log.")
    log.Println("Ditto for Println.")
    log.Printf("Use Printf and format strings. %d + %d = %d", a, b, a+b)
}
```

Each log is on a new line:

```
$ go run 03.2-01-basic-logging.go
2017/12/25 22:18:38 Use Print to log.
```

```
2017/12/25 22:18:38 Ditto for Println.
2017/12/25 22:18:38 Use Printf and format strings. 10 + 20 = 30
```

We can also forward the output to a file (or any number of `io.Writer` s) with [log.SetOutput] [setoutput1-log-pkg].

```
logFile, err := os.Create("log1.txt")
if err != nil {
    panic("Could not open file")
}

log.SetOutput(logFile)
```

Custom logger

We can setup a custom logger with logger.New.

```
func New(out io.Writer, prefix string, flag int) *Logger
```

- `out` : Log destination. Any `io.Writer` like files.
- `prefix` : Appears before each log entry. Think `Warning/Info/Error` .
- `flag` : Defines logging properties (e.g. the date time format).

Log to file

Using `out` we can log to files.

```
// 03.2-02-log-file.go
package main

import (
    "log"
    "os"
)

func main() {

    // Create a file
    logFile, err := os.Create("log1.txt")
    if err != nil {
        panic("Could not open file")
    }

    // Close the file after main returns
    defer logFile.Close()
```

```
    a, b := 10, 20

    // We will not use the other options
    myLog := log.New(logFile, "", 0)

    myLog.Print("Use Print to log.")
    myLog.Println("Ditto for Println.")
    myLog.Printf("Use Printf and format strings. %d + %d = %d", a, b, a+b)
}
```

`log1.txt` will contain:

```
Use Print to log.
Ditto for Println.
Use Printf and format strings. 10 + 20 = 30
```

After `New`, `mylog.SetOutput(w io.Writer)` can redirect the logger.

Logging to multiple files/streams

It's also possible to log to multiple files (or `io.Writer`s) with io.MultiWriter. This is useful when we want to both output to stdout and to files.

```
// 03.2-03-log-multiple-files.go
package main

import (
    "bytes"
    "fmt"
    "io"
    "log"
    "os"
)

func main() {

    // Create a file
    logFile, err := os.Create("log1.txt")
    if err != nil {
        panic("Could not open file")
    }

    // Close the file after main returns
    defer logFile.Close()

    // Create a second file
    logFile2, err := os.Create("log2.txt")
    if err != nil {
```

```
        panic("Could not open file2")
    }

    defer logFile2.Close()

    // Create a buffer
    var buflog bytes.Buffer

    multiW := io.MultiWriter(logFile, logFile2, &buflog, os.Stdout)

    a, b := 10, 20

    // Log to multiW
    myLog := log.New(multiW, "", 0)

    myLog.Print("Use Print to log.")
    myLog.Println("Ditto for Println.")
    myLog.Printf("Use Printf and format strings. %d + %d = %d", a, b, a+b)

    // Print buffer
    fmt.Println("Buffer:")
    fmt.Println(buflog.String())
}
```

We can see what we logged in both stdout and buffer:

```
$ go run 03.2-03-log-multiple-files.go
Use Print to log.
Ditto for Println.
Use Printf and format strings. 10 + 20 = 30
Buffer:
Use Print to log.
Ditto for Println.
Use Printf and format strings. 10 + 20 = 30
```

Flag

Prefix should be next but by discussing flag we can see if it appears before flag format or not. `flag` is an integer and is a collection of bits (like `FLAGS` CPU register). The flags are defined as constants:

```
// https://godoc.org/log#pkg-constants

const (
    // Bits or'ed together to control what's printed.
    // There is no control over the order they appear (the order listed
    // here) or the format they present (as described in the comments).
    // The prefix is followed by a colon only when Llongfile or Lshortfile
    // is specified.
```

```
    // For example, flags Ldate | Ltime (or LstdFlags) produce,
    //  2009/01/23 01:23:23 message
    // while flags Ldate | Ltime | Lmicroseconds | Llongfile produce,
    //  2009/01/23 01:23:23.123123 /a/b/c/d.go:23: message
    Ldate         = 1 << iota     // the date in the local time zone: 2009/01/23
    Ltime                         // the time in the local time zone: 01:23:23
    Lmicroseconds                 // microsecond resolution: 01:23:23.123123.  assumes
    Llongfile                     // full file name and line number: /a/b/c/d.go:23
    Lshortfile                    // final file name element and line number: d.go:23.
    LUTC                          // if Ldate or Ltime is set, use UTC rather than the
    LstdFlags     = Ldate | Ltime // initial values for the standard logger
)
```

There's only room for a few bits of customization (see what I did there?).

```
// 03.2-04-log-flags.go
package main

import (
    "log"
    "os"
)

func main() {

    a, b := 10, 20

    // New logger will output to stdout with flags
    // Only log date and file
    myLog := log.New(os.Stdout, "", log.Ldate|log.Lshortfile)

    myLog.Print("Use Print to log.")
    myLog.Println("Ditto for Println.")
    myLog.Printf("Use Printf and format strings. %d + %d = %d", a, b, a+b)
}
```

We log date and filename:

```
$ go run 03.2-04-log-flags.go
2017/12/26 03.2-04-log-flags.go:25: Use Print to log.
2017/12/26 03.2-04-log-flags.go:26: Ditto for Println.
2017/12/26 03.2-04-log-flags.go:27: Use Printf and format strings. 10 + 20 = 30
```

Prefix

`prefix` adds a string to the beginning of each log line.

```
// 03.2-05-log-prefix.go
package main

import (
    "log"
    "os"
)

func main() {

    a, b := 10, 20

    // New logger will output to stdout with prefix "Log1: " and flags
    // Note the space in prefix
    myLog := log.New(os.Stdout, "Log1: ", log.Ldate|log.Lshortfile)

    myLog.Print("Use Print to log.")
    myLog.Println("Ditto for Println.")
    myLog.Printf("Use Printf and format strings. %d + %d = %d", a, b, a+b)
}
```

Prefix is printed before flags:

```
$ go run 03.2-05-log-prefix.go
Log1: 2017/12/26 03.2-05-log-prefix.go:16: Use Print to log.
Log1: 2017/12/26 03.2-05-log-prefix.go:17: Ditto for Println.
Log1: 2017/12/26 03.2-05-log-prefix.go:18: Use Printf and format strings. 10 + 20 = 30
```

Logging levels

`log` only supports two logging levels:

- Fatal: `log.Print` and calls `os.Exit(1)`.
- [Panic][panic-log-pkg]: `log.Print` and calls `panic()`.

Both of these support `ln` and `f` variants (e.g. `Fatalf`, `Panicln`).

Continue reading ⇒ 04 - Go networking

[panic-log-pkg]: https://godoc.org/log#Panic# 04 - Go networking Now that we are done with the basics and packages, we can start learning about networking. Following Black Hat Python, let's start with networking basics and then move on.

Go's networking capabilities are in the net package and its sub-packages like net/http. The Python equivalent to `net` is `socket` and `net/http` can be compared to the 3rd party `Requests` module.

Table of Contents

- 04.1 - Basic TCP and UDP clients
- 04.2 - TCP servers
- 04.3 - TCP proxy
- 04.4 - SSH clients
- 04.5 - SSH Harvester

04.1 - Basic TCP and UDP clients

- TCP client
  - net.Dial - TCP
  - net.DialTCP
- UDP client
  - net.Dial - UDP
  - net.DialUDP
- Lessons learned

TCP client

The building blocks for the basic TCP client is explained in the net package overview.

net.Dial - TCP

net.Dial is the general-purpose connect command.

- First parameter is a string specifying the network. In this case we are using `tcp`.
- Second parameter is a string with the address of the endpoint in format of `host:port`.

```
// 04.1-01-basic-tcp1.go
package main

import (
    "bufio"
    "flag"
    "fmt"
    "net"
)

var (
    host, port string
)
```

```
func init() {
    flag.StringVar(&port, "port", "80", "target port")
    flag.StringVar(&host, "host", "example.com", "target host")
}

func main() {

    flag.Parse()

    // Converting host and port to host:port
    t := net.JoinHostPort(host, port)

    // Create a connection to server
    conn, err := net.Dial("tcp", t)
    if err != nil {
        panic(err)
    }

    // Write the GET request to connection
    // Note we are closing the HTTP connection with the Connection: close header
    // Fprintf writes to an io.writer
    req := "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
    fmt.Fprintf(conn, req)

    // Another way to do it to directly write bytes to conn with conn.Write
    // However we must first convert the string to bytes with []byte("string")
    // reqBytes := []byte(req)
    // conn.Write(reqBytes)

    // Reading the response

    // Create a new reader from connection
    connReader := bufio.NewReader(conn)

    // Create a scanner
    scanner := bufio.NewScanner(connReader)

    // Combined into one line
    // scanner := bufio.NewScanner(bufio.NewReader(conn))

    // Read from the scanner and print
    // Scanner reads until an I/O error
    for scanner.Scan() {
        fmt.Printf("%s\n", scanner.Text())
    }

    // Check if scanner has quit with an error
    if err := scanner.Err(); err != nil {
        fmt.Println("Scanner error", err)
    }
}
```

The only drawback with `scanner` is having to close the HTTP connection with the `Connection: close` header. Otherwise we have to manually kill the application.

```
$ go run 04.1-01-basic-tcp1.go -host example.com -port 80
HTTP/1.1 200 OK
Cache-Control: max-age=604800
Content-Type: text/html
Date: Sat, 16 Dec 2017 05:21:33 GMT
Etag: "359670651+gzip+ident"
Expires: Sat, 23 Dec 2017 05:21:33 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (dca/53DB)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1270
Connection: close

<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
...
```

Instead of using a `scanner` we can use `ReadString(0x00)` and stop when we reach an error (in this case `EOF`):

```
// 04.1-02-basic-tcp2.go

...
// Read until a null byte (not safe in general)
// Response will not be completely read if it has a null byte
if status, err := connReader.ReadString(byte(0x00)); err != nil {
    fmt.Println(err)
    fmt.Println(status)
}
...
```

Using `0x00` as delimiter is not ideal. If the response payload contains NULL bytes, we are not reading everything. But it works in this case.

net.DialTCP

net.DialTCP is the TCP specific version of `Dial`. It's a bit more complicated to call:

- `func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)`
- `network` is the same as `net.Dial` but can only be `tcp`, `tcp4` and `tcp6`.

- `laddr` is local address and can be chosen. If `nil`, a local address is automatically chosen.
- `raddr` is remote address and is the endpoint.

The type for both local and remote address is `*TCPAddr`:

```
type TCPAddr struct {
        IP   IP
        Port int
        Zone string // IPv6 scoped addressing zone
}
```

We can pass the `network` (e.g. "tcp") along with `host:port` or `ip:port` string to net.ResolveTCPAddr to get a `*TCPAddr`.

`DialTCP` returns a *TCPConn. It's a normal connection but with extra methods like `SetLinger`, `SetKeepAlive` or `SetKeepAlivePeriod`.

Let's re-write the TCP client with TCP-specific methods:

```
// 04.1-03-basic-tcp-dialtcp.go
// Basic TCP client using TCPDial and TCP specific methods
package main

import (
    "bufio"
    "flag"
    "fmt"
    "net"
)

var (
    host, port string
)

func init() {
    flag.StringVar(&port, "port", "80", "target port")
    flag.StringVar(&host, "host", "example.com", "target host")
}

// CreateTCPAddr converts host and port to *TCPAddr
func CreateTCPAddr(target, port string) (*net.TCPAddr, error) {
    return net.ResolveTCPAddr("tcp", net.JoinHostPort(host, port))
}

func main() {

    // Converting host and port
    a, err := CreateTCPAddr(host, port)
    if err != nil {
```

```
        panic(err)
    }

    // Passing nil for local address
    tcpConn, err := net.DialTCP("tcp", nil, a)
    if err != nil {
        panic(err)
    }

    // Write the GET request to connection
    // Note we are closing the HTTP connection with the Connection: close header
    // Fprintf writes to an io.writer
    req := "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
    fmt.Fprintf(tcpConn, req)

    // Reading the response

    // Create a scanner
    scanner := bufio.NewScanner(bufio.NewReader(tcpConn))

    // Read from the scanner and print
    // Scanner reads until an I/O error
    for scanner.Scan() {
        fmt.Printf("%s\n", scanner.Text())
    }

    // Check if scanner has quit with an error
    if err := scanner.Err(); err != nil {
        fmt.Println("Scanner error", err)
    }
}
```

This is a bit better.

UDP client

Similar to TCP, we can make a UDP client with both `net.Dial` and `net.DialUDP`.

net.Dial - UDP

Creating a UDP client is very similar. We will just call `net.Dial("udp", t)`. Being UDP, we will use `net.DialTimeout` to pass a timeout value.

```
// 04.1-04-basic-udp.go

// Create a connection to server with 5 second timeout
conn, err := net.DialTimeout("udp", t, 5*time.Second)
if err != nil {
```

```
        panic(err)
    }
```

Each second is one `time.Second` (remember to import the `time` package).

net.DialUDP

net.DialUDP is similar to the TCP equivalent:

- `func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)`
- `*UDPAddr` is acquired through net.ResolveUDPAddr.
- `network` should be `udp` .

```
// 04.1-05-udp-dialudp.go
package main

import (
    "bufio"
    "flag"
    "fmt"
    "net"
)

var (
    host, port string
)

func init() {
    flag.StringVar(&port, "port", "80", "target port")
    flag.StringVar(&host, "host", "example.com", "target host")
}

// CreateUDPAddr converts host and port to *UDPAddr
func CreateUDPAddr(target, port string) (*net.UDPAddr, error) {
    return net.ResolveUDPAddr("udp", net.JoinHostPort(host, port))
}

func main() {

    // Converting host and port to host:port
    a, err := CreateUDPAddr(host, port)
    if err != nil {
        panic(err)
    }

    // Create a connection with DialUDP
    connUDP, err := net.DialUDP("udp", nil, a)
    if err != nil {
        panic(err)
    }
```

```
    // Write the GET request to connection
    // Note we are closing the HTTP connection with the Connection: close header
    // Fprintf writes to an io.writer
    req := "UDP PAYLOAD"
    fmt.Fprintf(connUDP, req)

    // Reading the response

    // Create a scanner
    scanner := bufio.NewScanner(bufio.NewReader(connUDP))

    // Read from the scanner and print
    // Scanner reads until an I/O error
    for scanner.Scan() {
        fmt.Printf("%s\n", scanner.Text())
    }

    // Check if scanner has quit with an error
    if err := scanner.Err(); err != nil {
        fmt.Println("Scanner error", err)
    }
}
```

Lessons learned

1. Convert `int` to `string` using strconv.Itoa. strconv.Atoi does the opposite (note `Atoi` also returns an `err` so check for errors after using it.
2. `String(int)` converts the integer to corresponding Unicode character.
3. Create TCP connections with net.Dial.
4. We can read/write bytes directly to connections returned by `net.Dial` or create a `Scanner`.
5. Convert a string to bytes with `[]byte("12345")`.
6. Get seconds of type `Duration` with `time.Second`.
7. `net` package has TCP and UDP specific methods.

Continue reading ⇒ 04.2 - TCP servers

04.2 - TCP servers

Now we will create TCP and UDP servers.

- net.Listen
  - No logging with io.Copy()
  - Logging with extra goroutines
- net.TCPListen

- Lessons learned

net package overview also shows us how to create a generic TCP server. When creating a server we can take advantage of goroutines and spawn one for each connection.

net.Listen

The generic `net.Listen` method is capable of doing both TCP and UDP.

No logging with io.Copy()

Building on the example from `net` package we can build a simple TCP server:

```
// 04.2-01-tcpserver1.go
package main

import (
    "flag"
    "fmt"
    "io"
    "net"
)

var (
    host, port string
)

func init() {
    flag.StringVar(&port, "port", "12345", "target port")
    flag.StringVar(&host, "host", "example.com", "target host")
}

// handleConnectionNoLog echoes everything back without logging (easiest)
func handleConnectionNoLog(conn net.Conn) {

    rAddr := conn.RemoteAddr().String()
    defer fmt.Printf("Closed connection from %v\n", rAddr)

    // This will accomplish the echo if we do not want to log
    io.Copy(conn, conn)
}

func main() {

    flag.Parse()

    // Converting host and port to host:port
    t := net.JoinHostPort(host, port)

    // Listen for connections on BindIP:BindPort
```

```
    ln, err := net.Listen("tcp", t)
    if err != nil {
        // If we cannot bind, print the error and quit
        panic(err)
    }

    // Wait for connections
    for {
        // Accept a connection
        conn, err := ln.Accept()
        if err != nil {
            // If there was an error, print it and go back to listening
            fmt.Println(err)
            continue
        }

        fmt.Printf("Received connection from %v\n", conn.RemoteAddr().String())

        // Spawn a new goroutine to handle the connection
        go handleConnectionNoLog(conn)
    }
}
```

Most of the code in `main` is similar to Python. We listen on a `host:port` and then accept each connection. With each new connection, a new goroutine is spawned to handle it.

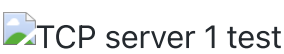
```
// handleConnectionNoLog echoes everything back without logging (easiest)
func handleConnectionNoLog(conn net.Conn) {

    rAddr := conn.RemoteAddr().String()
    defer fmt.Printf("Closed connection from %v\n", rAddr)

    // This will accomplish the echo if we do not want to log
    io.Copy(conn, conn)
}
```

This is where the magic happens:

- `io.Copy(conn, conn)`

You copy one connection to the other. It's super easy! And it works.

We can telnet to the server and see.

 TCP server 1 test

Logging with extra goroutines

Things become complicated when we want to log info that we have received. The main structure of the program is the same but we spawn two extra goroutines inside the `handleConnection` goroutine.

```
// 04.2-02-tcpserver2.go
// handleConnectionLog echoes everything back and logs messages received
func handleConnectionLog(conn net.Conn) {

    // Create buffered channel to pass data around
    c := make(chan []byte, 2048)

    // Spawn up two goroutines, one for reading and another for writing

    go readSocket(conn, c)
    go writeSocket(conn, c)

}
```

A buffered channel is created and passed to each goroutine. As you can imagine `readSocket` reads from the connection and writes to channel. Note the argument is a directed channel (this prevents from accidentally reading from it instead of writing):

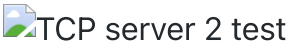
```
// readSocket reads data from socket if available and passes it to channel
// Note the directed write-only channel designation
func readSocket(conn net.Conn, c chan<- []byte) {

    // Create a buffer to hold data
    buf := make([]byte, 2048)
    // Store remote IP:port for logging
    rAddr := conn.RemoteAddr().String()

    for {
        // Read from connection
        n, err := conn.Read(buf)
        // If connection is closed from the other side
        if err == io.EOF {
            // Close the connction and return
            fmt.Println("Connection closed from", rAddr)
            return
        }
        // For other errors, print the error and return
        if err != nil {
            fmt.Println("Error reading from socket", err)
            return
        }
        // Print data read from socket
        // Note we are only printing and sending the first n bytes.
        // n is the number of bytes read from the connection
        fmt.Printf("Received from %v: %s\n", rAddr, buf[:n])
        // Send data to channel
```

```
            c <- buf[:n]
        }
    }
```

This is pretty straightforward. The only important part is `n`. `n` is the number of bytes read from the socket after `conn.Read`. When sending the data to the channel we are only interested in the first `n` bytes (if we send the whole buffer, the other side will get 2048 bytes every time).

```
// writeSocket reads data from channel and writes it to socket
func writeSocket(conn net.Conn, c <-chan []byte) {

    // Create a buffer to hold data
    buf := make([]byte, 2048)
    // Store remote IP:port for logging
    rAddr := conn.RemoteAddr().String()

    for {
        // Read from channel and copy to buffer
        buf = <-c
        // Write buffer
        n, err := conn.Write(buf)
        // If connection is closed from the other side
        if err == io.EOF {
            // Close the connction and return
            fmt.Println("Connection closed from", rAddr)
            return
        }
        // For other errors, print the error and return
        if err != nil {
            fmt.Println("Error writing to socket", err)
            return
        }
        // Log data sent
        fmt.Printf("Sent to %v: %s\n", rAddr, buf[:n])
    }
}
```

`writeSocket` is easier. We use a directed channel to read data into a buffer and send it off. This server is also not echo-ing back the first character.



net.TCPListen

As we have seen, there are TCP specific methods in the `net` package. The code is pretty much the same. We just use `TCPListen` and pass a `*TCPAddr` to it. The result is a `TCPConn` which is `net.Conn` under the hoods. Everything else remains the same.

```
// 04.2-03-tcpserver3.go

// CreateTCPAddr converts host and port to *TCPAddr
func CreateTCPAddr(host, port string) (*net.TCPAddr, error) {
    return net.ResolveTCPAddr("tcp", net.JoinHostPort(host, port))
}

func main() {

    // Converting host and port to TCP address
    t, err := CreateTCPAddr(bindHost, bindPort)
    ...

    // Listen for connections on bindHost:bindPort
    ln, err := net.ListenTCP("tcp", t)
    ...

    for {
        conn, err := ln.AcceptTCP()
        ...

        go handleConnectionLog(conn)
    }
...
```

Lessons learned

1. `io.Copy(conn, conn)` is magic.
2. Goroutines are pretty easy to spawn for socket read/writes.

Continue reading ⇒ 04.3 - TCP proxy# 04.3 - TCP Proxy

Building a non-TLS terminating TCP proxy is pretty easy. It's very similar to the TCP server we have already created.

We listen for TCP connections. After one is established, we create a new connection to the forwarding IP:port and send all data. Without logging this can be done with a simple `io.Copy(connDest, connSrc)`. With logging we have to use multiple goroutines (as we have seen before).

Only forwardConnection is different. Instead of calling `handleConnection` we call `forwardConnection` in a new goroutine.

Inside, we create a TCP connection to server and two channels. Then we handle each side of the connection like the echo TCP server.

```go
// 04.3-01-tcp-proxy.go
package main

import (
    "flag"
    "fmt"
    "io"
    "net"
)

var (
    bindIP, bindPort, destIP, destPort string
)

func init() {
    flag.StringVar(&bindPort, "bindPort", "12345", "bind port")
    flag.StringVar(&bindIP, "bindIP", "127.0.0.1", "bind IP")
    flag.StringVar(&destPort, "destPort", "12345", "bind port")
    flag.StringVar(&destIP, "destIP", "127.0.0.1", "bind IP")
}

// readSocket reads data from socket if available and passes it to channel
func readSocket(conn net.Conn, c chan<- []byte) {

    // Create a buffer to hold data
    buf := make([]byte, 2048)
    // Store remote IP:port for logging
    rAddr := conn.RemoteAddr().String()

    for {
        // Read from connection
        n, err := conn.Read(buf)
        // If connection is closed from the other side
        if err == io.EOF {
            // Close the connction and return
            fmt.Println("Connection closed from", rAddr)
            return
        }
        // For other errors, print the error and return
        if err != nil {
            fmt.Println("Error reading from socket", err)
            return
        }
        // Print data read from socket
        // Note we are only printing and sending the first n bytes.
        // n is the number of bytes read from the connection
        fmt.Printf("Received from %v: %s\n", rAddr, buf[:n])
        // Send data to channel
        c <- buf[:n]
    }
}
```

```
// writeSocket reads data from channel and writes it to socket
func writeSocket(conn net.Conn, c <-chan []byte) {

    // Create a buffer to hold data
    buf := make([]byte, 2048)
    // Store remote IP:port for logging
    rAddr := conn.RemoteAddr().String()

    for {
        // Read from channel and copy to buffer
        buf = <-c
        // Write buffer
        n, err := conn.Write(buf)
        // If connection is closed from the other side
        if err == io.EOF {
            // Close the connction and return
            fmt.Println("Connection closed from", rAddr)
            return
        }
        // For other errors, print the error and return
        if err != nil {
            fmt.Println("Error writing to socket", err)
            return
        }
        // Log data sent
        fmt.Printf("Sent to %v: %s\n", rAddr, buf[:n])
    }
}

// forwardConnection creates a connection to the server and then passes packets
func forwardConnection(clientConn net.Conn) {

    // Converting host and port to destIP:destPort
    t := net.JoinHostPort(destIP, destPort)

    // Create a connection to server
    serverConn, err := net.Dial("tcp", t)
    if err != nil {
        fmt.Println(err)
        clientConn.Close()
        return
    }

    // Client to server channel
    c2s := make(chan []byte, 2048)
    // Server to client channel
    s2c := make(chan []byte, 2048)

    go readSocket(clientConn, c2s)
    go writeSocket(serverConn, c2s)
    go readSocket(serverConn, s2c)
    go writeSocket(clientConn, s2c)
```

```
}
func main() {

    flag.Parse()

    // Converting host and port to bindIP:bindPort
    t := net.JoinHostPort(bindIP, bindPort)

    // Listen for connections on BindIP:BindPort
    ln, err := net.Listen("tcp", t)
    if err != nil {
        // If we cannot bind, print the error and quit
        panic(err)
    }

    fmt.Printf("Started listening on %v\n", t)

    // Wait for connections
    for {
        // Accept a connection
        conn, err := ln.Accept()
        if err != nil {
            // If there was an error print it and go back to listening
            fmt.Println(err)

            continue
        }
        fmt.Printf("Received connection from %v\n", conn.RemoteAddr().String())

        go forwardConnection(conn)
    }
}
```

Continue reading ⇒ 04.4 - SSH clients# 04.4 - SSH clients

Next in line is creating SSH clients. The /x/crypto/ssh provides SSH support. It's not one of the standard libraries so you need to `go get golang.org/x/crypto/ssh` before use.

We can authenticate using either user/pass or certificate.

- Basic interactive session with user/pass
- Verifying host
  - ssh.FixedHostKey
  - Custom host verifier
- Login with SSH key
- Login and run a command
  - Run a command with CombinedOutput
  - Run a command with Run

Basic interactive session with user/pass

First program is a typical interactive session based on the example in the docs. We login with a user/pass combo.

```
// 04.4-01-sshclient-login-password.go
// Interactive SSH login with user/pass.

package main

import (
    "flag"
    "fmt"
    "io"
    "net"
    "os"

    // Importing crypto/ssh
    "golang.org/x/crypto/ssh"
)

var (
    username, password, serverIP, serverPort string
)

// Read flags
func init() {
    flag.StringVar(&serverPort, "port", "22", "SSH server port")
    flag.StringVar(&serverIP, "ip", "127.0.0.1", "SSH server IP")
    flag.StringVar(&username, "user", "", "username")
    flag.StringVar(&password, "pass", "", "password")
}

func main() {
    // Parse flags
    flag.Parse()

    // Check if username has been submitted - password can be empty
    if username == "" {
        fmt.Println("Must supply username")
        os.Exit(2)
    }

    // Create SSH config
    config := &ssh.ClientConfig{
        // Username
        User: username,
        // Each config must have one AuthMethod. In this case we use password
        Auth: []ssh.AuthMethod{
            ssh.Password(password),
        },
```

```
        // This callback function validates the server.
        // Danger! We are ignoring host info
        HostKeyCallback: ssh.InsecureIgnoreHostKey(),
    }

    // Server address
    t := net.JoinHostPort(serverIP, serverPort)

    // Connect to the SSH server
    sshConn, err := ssh.Dial("tcp", t, config)
    if err != nil {
        fmt.Printf("Failed to connect to %v\n", t)
        fmt.Println(err)
        os.Exit(2)
    }

    // Create new SSH session
    session, err := sshConn.NewSession()
    if err != nil {
        fmt.Printf("Cannot create SSH session to %v\n", t)
        fmt.Println(err)
        os.Exit(2)
    }

    // Close the session when main returns
    defer session.Close()

    // For an interactive session we must redirect IO
    session.Stdout = os.Stdout
    session.Stderr = os.Stderr
    input, err := session.StdinPipe()
    if err != nil {
        fmt.Println("Error redirecting session input", err)
        os.Exit(2)
    }

    // Setup terminal mode when requesting pty. You can see all terminal modes at
    // https://github.com/golang/crypto/blob/master/ssh/session.go#L56 or read
    // the RFC for explanation https://tools.ietf.org/html/rfc4254#section-8
    termModes := ssh.TerminalModes{
        ssh.ECHO: 0, // Disable echo
    }

    // Request pty
    // https://tools.ietf.org/html/rfc4254#section-6.2
    // First variable is term environment variable value which specifies terminal.
    // term doesn't really matter here, we will use "vt220".
    // Next are height and width: (40,80) characters and finall termModes.
    err = session.RequestPty("vt220", 40, 80, termModes)
    if err != nil {
        fmt.Println("RequestPty failed", err)
        os.Exit(2)
    }
```

```
    // Also
    // if err = session.RequestPty("vt220", 40, 80, termModes); err != nil {
    //  fmt.Println("RequestPty failed", err)
    //  os.Exit(2)
    // }

    // Now we can start a remote shell
    err = session.Shell()
    if err != nil {
        fmt.Println("shell failed", err)
        os.Exit(2)
    }

    // Same as above, a different way to check for errors
    // if err = session.Shell(); err != nil {
    //  fmt.Println("shell failed", err)
    //  os.Exit(2)
    // }

    // Endless loop to capture commands
    // Note: After exit, we need to ctrl+c to end the application.
    for {
        io.Copy(input, os.Stdin)
    }
}
```

First we create a config (note it's a pointer):

```
// Create SSH config
config := &ssh.ClientConfig{
    // Username
    User: username,
    // Each config must have one AuthMethod. In this case we use password
    Auth: []ssh.AuthMethod{
        ssh.Password(password),
    },
    // This callback function validates the server.
    // Danger! We are ignoring host info
    HostKeyCallback: ssh.InsecureIgnoreHostKey(),
}
```

Each config should have an `AuthMethod`. We are using a password in this program.

Next on the config is `HostKeyCallback` and is used to verify the server.

The familiar `Dial` method connects to the server. Then we create a session (each connection can have multiple sessions).

We set stdin, stdout and stderr for session and then terminal modes. Finally we request a pseudo-terminal with `RequestPty` and a shell. We capture commands on stdin by basically copying `os.Stdin` to the connection's input.

Note: Depending on your SSH server and the terminal mode, you might see color codes. For example you will see ANSI color codes if you run it from Windows cmd, but not in PowerShell. With Windows OpenSSH, it does not matter what TERM is sent, the color codes will not go away in cmd.

Verifying host

Usually when creating small programs in security, we do not care about the host. But it's always good to check.

`HostKeyCallback` in config can be used in three ways:

- `ssh.InsecureIgnoreHostKey()` : Ignore everything!
- `ssh.FixedHostKey(key PublicKey)` : Returns a function to check the hostkey.
- Custom host verifier: Return `nil` if host is ok, otherwise return an error.

ssh.FixedHostKey

This is an easy check. We pass a host key and the method checks if it matches the one returned by the connection.

```
// https://github.com/golang/crypto/blob/master/ssh/client.go#L265
// FixedHostKey returns a function for use in
// ClientConfig.HostKeyCallback to accept only a specific host key.
func FixedHostKey(key PublicKey) HostKeyCallback {
    hk := &fixedHostKey{key}
    return hk.check
}
```

Looking at the source, it just unmarshals two publickeys and checks if they match.

```
// https://github.com/golang/crypto/blob/master/ssh/client.go#L253
func (f *fixedHostKey) check(hostname string, remote net.Addr, key PublicKey) error {
    if f.key == nil {
        return fmt.Errorf("ssh: required host key was nil")
    }
    if !bytes.Equal(key.Marshal(), f.key.Marshal()) {
        return fmt.Errorf("ssh: host key mismatch")
    }
    return nil
}
```

It's straightforward to use. The new program is only a little different from the old one:

- Create a variable of type `ssh.PublicKey` to hold the key.
- Pass `HostKeyCallback: ssh.FixedHostKey(var_from_above)` in config.

```
// Define host's public key
var hostPubKey ssh.PublicKey

// Populate hostPubKey

// Create SSH config
config := &ssh.ClientConfig{
    // Username
    User: username,
    // Each config must have one AuthMethod. In this case we use password
    Auth: []ssh.AuthMethod{
        ssh.Password(password),
    },
    // Danger! We are ignoring host info
    HostKeyCallback: ssh.FixedHostKey(hostPubKey),
}
```

Custom host verifier

This has more flexibility. We can also use this callback function to grab and store a server's public key. It can have any number of arguments (usually we use these arguments to pass info to the host checker). It should **return a function of type** ssh.HostKeyCallback:

```
type HostKeyCallback func(hostname string, remote net.Addr, key PublicKey) error
```

In other words, it's a function of this type:

```
func hostChecker(arg1 type1, arg2 type2, ...) ssh.HostKeyCallback {
    // ...
}
```

Returned function can be a separate function or an anonymous function created inside `hostChecker`. Here's an example of an anonymous function used by `InsecureIgnoreHostKey` from `ssh` package's source:

```
// https://github.com/golang/crypto/blob/master/ssh/client.go#L240

// InsecureIgnoreHostKey returns a function that can be used for
// ClientConfig.HostKeyCallback to accept any host key. It should
// not be used for production code.
func InsecureIgnoreHostKey() HostKeyCallback {
    return func(hostname string, remote net.Addr, key PublicKey) error {
        return nil
```

```
    }
}
```

Now we know enough to create our own custom host checker and pass it to `HostKeyCallback`:

```
// 04.4-02-sshclient-check-host.go

// hostChecker returns a function to be used as callback for HostKeyCallback.
func hostChecker() ssh.HostKeyCallback {
    return printServerKey
}

// printServerKey prints server's info instead of checking it.
// It's of type HostKeyCallback
func printServerKey(hostname string, remote net.Addr, key ssh.PublicKey) error {
    // Just print everything
    fmt.Printf("Hostname: %v\nRemote address: %v\nServer key: %+v\n",
        hostname, remote, key)
    // Return nil so connection can continue without checking the server
    return nil
}
```

We can see server info in the callback function:

```
$ go run 04.4-02-sshclient2.go -user user -pass 12345
Hostname: 127.0.0.1:22
Remote address: 127.0.0.1:22
Server key: &{Curve:{CurveParams:0xc04204e100}
X:+954465638301905397235496463871348043734210257636293704534954817288090285709 67
Y:+716900309222867669321485639591608190512087182623530768120363479250069216548 63}
...
```

Login with SSH key

It's also possible to pass another `AuthMethod` and login with a key. Luckily, the package has another example. We read the PEM encoded private key and use it in `ClientConfig`.

```
// 04.4-03-sshclient-login-key.go

// Now we must read the private key
pKey, err := ioutil.ReadFile(pKeyFile)
if err != nil {
    fmt.Println("Failed to read private key from file", err)
    os.Exit(2)
}

// Create a signer with the private key
```

```
signer, err := ssh.ParsePrivateKey(pKey)
if err != nil {
    fmt.Println("Failed to parse private key", err)
    os.Exit(2)
}

// Create SSH config
config := &ssh.ClientConfig{
    // Username
    User: username,
    // Each config must have one AuthMethod. Now we use key
    Auth: []ssh.AuthMethod{
        ssh.PublicKeys(signer),
    },
    // This callback function validates the server.
    // Danger! We are ignoring host info
    HostKeyCallback: ssh.InsecureIgnoreHostKey(),
}
```

Login and run a command

Interactive login is useful but there are SSH clients for that. Automated tools usually want to login, run commands, capture the output and move on to the next host.

Each session can only run one command. A new session must be created for each new command (one SSH connection can support multiple sessions). We can run commands using one of these methods:

- Start: Runs a single command on the server.
- Run: Same as above. In fact, run calls start internally.
- Output: Runs the command but returns standard output.
- CombinedOutput: Runs the command and returns both stdout and stderr.

1. Not all of these methods return the output directly (in `[]byte`). For those that do not, we need to read `session.Stdout/Stderr`.
2. All of them return errors. After execution, check the errors.
3. For obvious reasons, it seems like `CombinedOutput` will work best.

Run a command with CombinedOutput

We re-use the code from first example but stop after the session is created. Then we run the command, check for errors and print the output.

```
// 04.4-04-sshclient-run-combinedoutput.go

// Close the session when main returns
```

```
defer session.Close()

// Run a command with CombinedOutput
o, err := session.CombinedOutput(command)
if err != nil {
    fmt.Println("Error running command", err)
}

fmt.Printf("Output:\n%s", o)
```

Results from my VM (don't get excited, it's the default user/pass for https://modern.ie VMs):

```
$ go run .\04.4-04-sshclient-run-combinedoutput.go -user IEUser -pass Passw0rd! -cmd d
Output:
 Volume in drive C is Windows 10
 Volume Serial Number is C436-9552

 Directory of C:\Users\IEUser

12/19/2017  08:28 PM    <DIR>          .
12/19/2017  08:28 PM    <DIR>          ..
10/02/2017  12:50 AM    <DIR>          .gradle
12/24/2017  07:02 PM    <DIR>          .ssh
03/23/2017  12:29 PM                 6 .vbox_version
03/23/2017  11:18 AM    <DIR>          Contacts
12/24/2017  01:50 AM    <DIR>          Desktop
...
```

Of course, we can always cheat by running multiple commands. On Windows use `&` and `&&`.

```
$ go run .\04.4-04-sshclient-run-combinedoutput.go -user IEUser -pass Passw0rd! -cmd "
Output:
 Volume in drive C is Windows 10
 Volume Serial Number is C436-9552

 Directory of C:\Users

12/24/2017  01:53 AM    <DIR>          .
12/24/2017  01:53 AM    <DIR>          ..
12/19/2017  08:28 PM    <DIR>          IEUser
03/23/2017  11:18 AM    <DIR>          Public
12/24/2017  01:53 AM    <DIR>          SSHD
               0 File(s)              0 bytes
               5 Dir(s)  21,863,829,504 bytes free
```

Run a command with Run

Using `Run` is similar, we buffer `session.Stdout/Stderr` before we execute the command and print them after. This is based on the package example:

```
// 04.4-05-sshclient-run-run.go

// Close the session when main returns
defer session.Close()

// Create buffers for stdout and stderr
var o, e bytes.Buffer

session.Stdout = &o
session.Stderr = &e

// Run a command with Run and read stdout and stderr
if err := session.Run(command); err != nil {
    fmt.Println("Error running command", err)
}

// Convert buffer to string
fmt.Printf("stdout:\n%s\nstderr:\n%s", o.String(), e.String())
```

Continue reading ⇒ 04.5 - SSH Harvester

04.5 - SSH harvester

This is a copy of my blog Simple SSH Harvester in Go. Sometime in the future, I will return and continue working on the tool. For now I want to move on to new things.

I realized I cannot find any examples of SSH certificate verification. There are a few examples for host keys here and there. Even the `certs_test.go` file just checks the host name. There was a typo in an error message[^1] in the `crypto/ssh` package but I think because this is not very much used, had gone unreported.

Here's my step by step guide to writing this tool by piggybacking on SSH host verification callbacks. Hopefully this will make it easier for the next person.

TL;DR: verifying SSH servers

1. Create an instance of ssh.CertChecker.
2. Set callback functions for `IsHostAuthority`, `IsRevoked` and optionally `HostKeyFallback`.
    - `IsHostAuthority` 's callback should return `true` for valid certificates.
    - `IsRevoked` 's callback should return `false` for valid certificates.
    - `HostKeyFallback` 's callback should return `nil` for valid certificates.
3. Create an instance of ssh.ClientConfig.

4. Set `HostKeyCallback` in `ClientConfig` to `&ssh.CertChecker.CheckHostKey`.
5. CheckHostKey will verify the certificate based on other callback functions.
6. The certificate can be accessed in `IsRevoked` callback function.

Go to `Parsing SSH certificates` to skip the fodder.

Table of Contents

- Code analysis
  - Constants and usage
  - Init function
  - Custom flag type
  - SSHServer struct
  - SSHServers struct
  - Struct to JSON
  - Utilities
- Parsing SSH certificates <-- This is the important part
  - Step 1: Create ssh.CertChecker
  - Step 2: Set Callback functions
  - IsHostAuthority
    - IsHostAuthority callback
  - IsRevoked
    - IsRevoked callback
      - Question!!!!
  - HostKeyFallback
  - Step 3: Create ssh.ClientConfig
    - Banner callback
  - Step 4: ClientConfig.HostKeyCallback
    - Other ways of verifying servers
  - Step 5: Connecting to SSH servers
    - discover method
    - Goroutines and sync.WaitGroups
- SSH Harvester in action
- Conclusion

Code analysis

Let's look at the code.

Constants and usage

We can either pass a file with `-in` . The file should have one address on each line:

```
127.0.0.1:22
[2001:db8::68]:1234
```

Or we can pass addresses with `-t` separated by commas:

- `SSHHarvester.exe -t 127.0.0.1:22,[2001:db8::68]:1234`

Output file is specified with `-out` .

```
const (
    mUsage = "SSH Harvester gathers and publishes info about SSH servers.\n" +
        "Addresses should be in format of 'host:port'.\n" +
        "Input file should have one address on each line " +
        "and addresses provided to -targets should be separated by commas.\n" +
        "-in and -targets are mutually exclusive, use one.\n" +
        "Examples:\n" +
        "go run SSHHarvester1.go -t 127.0.0.1:12334,192.168.0.10:22\n" +
        "go run SSHHarvester1.go -i inputfile.txt\n" +
        "go run SSHHarvester1.go -i inputfile.txt -out output.txt\n"
    outUsage = "output report file"
    inUsage  = "input file"
    tUsage   = "addresses separated by comma"
    vUsage   = "print extra info"

    // Delimiter for host:port
    AddressDelim = ":"
    // // Delimiter for IPv6 addresses
    // IPv6Delim = "[]"

    // Log prefix - note the trailing space
    LogPrefix = "[*] "

    // Test SSH username/password - not really important
    TestUser     = "user"
    TestPassword = "password"

    // Timeout in seconds
    Timeout = 5 * time.Second
)

// Usage string
func usage() {
    usg := mUsage
    usg += fmt.Sprintf("\n  -i, -in\tstring\t%s", inUsage)
    usg += fmt.Sprintf("\n  -o, -out\tstring\t%s", outUsage)
    usg += fmt.Sprintf("\n  -t, -targets\tstring\t%s", tUsage)
    usg += fmt.Sprintf("\n  -v, -verbose\tstring\t%s", vUsage)
    usg += fmt.Sprintf("\n")
```

```
    fmt.Println(usg)
}
```

This is pretty standard. You might want to change the default username/password. Ultimately we do not care about logging in, we just want to connect and get host info.

Init function

We setup flags, logging and check flags. `flag` package does not have `mutually_exclusive_group` from Python's `Argparse` package. It needs to be done manually. I will most likely move to a community cli package after this.

```
func init() {
    // Setup flags
    flag.StringVar(&out, "out", "", outUsage)
    flag.StringVar(&out, "o", "", outUsage)
    flag.StringVar(&in, "in", "", inUsage)
    flag.StringVar(&in, "i", "", inUsage)
    flag.Var(&targets, "targets", tUsage)
    flag.Var(&targets, "t", tUsage)
    flag.BoolVar(&verbose, "verbose", false, vUsage)
    flag.BoolVar(&verbose, "v", false, vUsage)

    // Set flag usage
    flag.Usage = usage

    // Parse flags
    flag.Parse()

    // Setting up logging
    logSSH = log.New(os.Stdout, LogPrefix, log.Ltime)

    // Check if we have enough arguments
    if len(os.Args) < 2 {
        flag.Usage()
        errorExit("not enough arguments", nil)
    }

    // Check if both in and targets are supported
    if (in != "") && (targets != nil) {
        errorExit("-in and -targets are mutually exclusive, use one", nil)
    }
}
```

`errorExit` just calls `logger.Fatalf` with a message. Logging the message and returning from main with status code 1.

```
// errorExit logs an error and then exits with status code 1.
func errorExit(m string, err error) {
    // If err is provided print it, otherwise don't
    if err != nil {
        logSSH.Fatalf("%v - stopping\n%v\n", m, err)
    }
    logSSH.Fatalf("%v - stopping\n", m)
}
```

Custom flag type

We are using a custom flag type for `-t`. This allows us to pass multiple addresses separated by `,` and get a slice of addresses directly. This is done through implementing the flag.value which contains two methods `String()` and `Set()`. In simple words:

1. Create a new type `mytype`.
2. Create two methods with `*mytype` receivers named `String()` and `Set()`.
    - `String()` casts the custom type to a `string` and returns it.
    - `Set(string)` has a `string` argument and populates the type, returns an error if applicable.
3. Create a new flag without an initial value:
    - Call `flag.NewFlagSet(&var,` instead of `flag.String(`.
    - Call `flag.Var(` instead of `flag.StringVar(` or `flag.IntVar(`.

I have written more about the `flag` package in Hacking with Go - 03.1.

```
// Custom flag type for -t (code re-used from flag section)
// Create a custom type from a string slice
type strList []string

// Implement String()
func (str *strList) String() string {
    return fmt.Sprintf("%v", *str)
}

// Implement Set(*strList)
func (str *strList) Set(s string) error {
    // If input was empty, return an error
    if s == "" {
        return errors.New("nil input")
    }
    // Split input by ","
    *str = strings.Split(s, ",")
    // Do not return an error
    return nil
}
```

SSHServer struct

We use a struct and some methods to hold server info. The `SSHServer` struct has these fields:

```
// Struct to hold server data
type SSHServer struct {
    Address   string          // host:port
    Host      string          // IP address
    Port      int             // port
    IsSSH     bool            // true if server is running SSH on address:port
    Banner    string          // banner text, if any
    Cert      ssh.Certificate // server's certificate
    Hostname  string          // hostname
    PublicKey ssh.PublicKey   // server's public key
}
```

Not all fields will be populated. For example `Hostname` and `PublicKey` are only populated if the server responds with a public key. If it has a cert, then `Cert` will be populated instead.

New `*SSHServer` s are created by `NewSSHServer` .

```
// NewSSHServer returns a new SSHServer with address, host and port populated.
// If address cannot be processed, an error will be returned.
func NewSSHServer(address string) (*SSHServer, error) {
    // Process address, return error if it's not in the correct format
    host, port, err := net.SplitHostPort(address)
    if err != nil {
        return nil, err
    }

    var s SSHServer

    s.Address = address
    s.Host = host
    s.Port, err = strconv.Atoi(port)
    if err != nil {
        return nil, err
    }
    // If port is not in (0,65535]
    if 0 > s.Port || s.Port > 65535 {
        return nil, errors.New(port + " invalid port")
    }
    return &s, nil
}
```

`net.SplitHostPort` splits `host:port` into two strings but it does not check the validity of either part. Meaning you can pass `500.500.500.500:70000` and it will be accepted because the format is correct.

To check if the IP is valid, we can use `net.ParseIP` and check the result (it's `nil` if it was not parsed correctly). However, we do not know if we are dealing with hostnames like `example.com:1234`. But we can check if ports are in the correct range.

SSHServers struct

`SSHServers` is a slice of `SSHServer` pointers. It has a Stringer method (a `String` method that returns a string representation of receiver).

```
type SSHServers []*SSHServer

// String converts []*SSHServer to JSON. If it cannot convert to JSON, it
// will convert each member to string using fmt.Sprintf("%+v").
func (servers *SSHServers) String() string {
    var report string
    // Try converting to JSON
    report, err := ToJSON(servers, true)
    // If cannot convert to JSON
    if err != nil {
        // Save all servers as string (this is not as good as JSON)
        for _, v := range *servers {
            report += fmt.Sprintf("%+v\n%s\n", v, strings.Repeat("-", 30))
        }
        return report
    }
    return report
}
```

Struct to JSON

`ToJSON` converts a struct to a JSON string. If the second argument is `true`, it pretty prints it by indenting.

```
// ToJSON converts input to JSON. If prettyPrint is set to True it will call
// MarshallIndent with 4 spaces.
// If your struct does not work here, make sure struct fields start with a
// capital letter. Otherwise they are not visible to the json package methods.
// We could also rewrite this as a method for ([]*SSHServer).
func ToJSON(s interface{}, prettyPrint bool) (string, error) {
    var js []byte
    var err error

    // Pretty print if specified
    if prettyPrint {
        js, err = json.MarshalIndent(s, "", "    ") // 4 spaces
    } else {
        js, err = json.Marshal(s)
    }
```

```
    // Check for marshalling errors
    if err != nil {
        return "", nil
    }

    return string(js), nil
}
```

This is one of the useful things I learned while working on this code. It's a pretty cool way of converting structs into strings. When printing with `"%+v"` format string, field pointers are not dereferenced and it will print the memory address. However, marshalling to JSON dereferences every field.

Note: When JSON-ing structs, make sure to mark fields as exportable by starting their names with capital letters. The JSON package cannot see them otherwise.

Utilities

There are a couple of misc functions.

`readTargetFile` reads addresses from a file (one address on each line) and returns a `[]string`.

`writeReport` gets a slice of `SSHServer`s (`SSHServers` to be exact), converts it to string (the Stringer we saw earlier will try to convert it to JSON first) and writes it to a file. The final file will be a JSON object that can be parsed.

Parsing SSH certificates <-- This is the important part

Inside ssh.ClientConfig there's a callback `HostKeyCallback`. This function should return `nil` if host is verified. Read Phil Pennock's blogpost Golang SSH Security for the history behind it.

Let's expand the tl;dr steps:

Step 1: Create ssh.CertChecker

We are interested in the following three ssh.CertChecker fields. All of them are callback functions:

```
certCheck := &ssh.CertChecker{
    IsHostAuthority: hostAuthCallback(),
    IsRevoked:       certCallback(s),
    HostKeyFallback: hostCallback(s),
}
```

Don't worry about the functions for now. But remember these callback functions are only required to have a specific **return value but can have any number of arguments**. This is very useful we can pass our `SSHServer` objects and populate them inside these functions.

Step 2: Set Callback functions

Set callback functions for these three fields.

IsHostAuthority

`IsHostAuthority` must be defined. If not, we get a run-time error:

```
golang.org/x/crypto/ssh.(*CertChecker).CheckHostKey(0xc04206a140, 0xc0420080c0,
    0xc, 0x68d700, 0xc042058450, 0x68df80, 0xc0420a2000, 0x1, 0x8)
        Z:/Go/src/golang.org/x/crypto/ssh/certs.go:301 +0xae
golang.org/x/crypto/ssh.(*CertChecker).CheckHostKey-fm(0xc0420080c0, 0xc,
    0x68d700, 0xc042058450, 0x68df80, 0xc0420a2000, 0x0, 0x0)
        Z:/Go/src/hackingwithgo/04.5-01-ssh-harvester.go:205 +0x70
...
```

To discover the error cause, one must look at the source code for CheckHostKey. We'll see that `CheckHostKey` calls `IsHostAuthority`.

```
// CheckHostKey checks a host key certificate. This method can be
// plugged into ClientConfig.HostKeyCallback.
func (c *CertChecker) CheckHostKey(addr string, remote net.Addr, key PublicKey) error
    cert, ok := key.(*Certificate)
    if !ok {
        if c.HostKeyFallback != nil {
            return c.HostKeyFallback(addr, remote, key)
        }
        return errors.New("ssh: non-certificate host key")
    }
    if cert.CertType != HostCert {
        return fmt.Errorf("ssh: certificate presented as a host key has type %d", cert
    }
    // If IsHostAuthority is not defined, run-time error occurs here
    if !c.IsHostAuthority(cert.SignatureKey, addr) {
        return fmt.Errorf("ssh: no authorities for hostname: %v", addr)
    }

    hostname, _, err := net.SplitHostPort(addr)
    if err != nil {
        return err
    }

    // Pass hostname only as principal for host certificates (consistent with OpenSSH)
    return c.CheckCert(hostname, cert)
}
```

So what does this function do?

First it tries to get a certificate from `key PublicKey` (by casting). If the cast is not successful, it uses `HostKeyFallBack` to verity server's public key instead.

Then the function checks if the certificate type is `HostCert`. SSH differentiates between host and client certificates. For example OpenSSH's `keygen` uses the `-h` switch to sign and create a host key.

Another of our callbacks, `IsHostAuthority` is called next. If it returns `false`, the certificate is not valid. The docs say:

```
// IsHostAuthority should report whether the key is recognized as
// an authority for this host. This allows for certificates to be
// signed by other keys, and for those other keys to only be valid
// signers for particular hostnames. This must be set if this
// CertChecker will be checking host certificates.
```

This is just fancy talk for verifying the CA and performing certificate pinning. In other words we can check:

1. Is the certificate signed by *a valid CA*? Note, unlike TLS certs, most SSH certs are signed by internal CAs. Often we are relying on a hardcoded CA for verification.
2. Is the certificate signed by *the valid CA*? We don't want certs signed by other CAs.

`net.SplitHostPort` (we already used it above) splits `host:port` into `host` and `port` and passes `hostname` to `CheckCert`.

`CheckCert` does a couple of more checks. Most notably it calls another one of our functions `IsRevoked`.

```
// CheckCert checks CriticalOptions, ValidPrincipals, revocation, timestamp and
// the signature of the certificate.
func (c *CertChecker) CheckCert(principal string, cert *Certificate) error {
    if c.IsRevoked != nil && c.IsRevoked(cert) {
        return fmt.Errorf("ssh: certicate serial %d revoked", cert.Serial)
    }
    ...
```

IsHostAuthority callback

Not every function can be a callback function. Each function needs to return certain type. `IsHostAuthority` requires the callback function to have this return type:

- `func(ssh.PublicKey, string) bool`

In other words, our callback function needs to **return a function of that type**.

First we create a custom type (it's not defined in the package) and then create a function that returns that type:

```
// Define custom type for IsHostAuthority
type HostAuthorityCallBack func(ssh.PublicKey, string) bool

// hostAuthCallback is the callbackfunction for IsHostAuthority. Without
// it, ssh.CertChecker will not work.
func hostAuthCallback() HostAuthorityCallBack {
    // Return true because we just want to make this work
    return func(p ssh.PublicKey, addr string) bool {
        return true
    }
}
```

If we want the connection to continue, the internal function needs to return `true`.

IsRevoked

`IsRevoked` is not mandatory. If it's not set, it's ignored. Meaning there's no automatic certificate revocation checks happening without it. ~~Note the typo in the error message:~~ ~~`certicate`~~. The typo has now been corrected. Honestly, I think this just means programs do not use this function (or I am terribly wrong and am using something which should not be used). If certificate is valid, this function must return `nil` or `false`.

IsRevoked callback

For the goal of grabbing the certificate and processing it, `IsRevoked` is the most useful. It gets the certificate as a parameter and we can do parse or verify it inside the function. `IsRevoked` must return:

- `func(cert *Certificate) bool`

Again we define that function type and declare our own function:

```
// Create IsRevoked function callback type
type IsRevokedCallback func(cert *ssh.Certificate) bool

// certCallback processes the SSH certificate. It is piggybacked on the
// IsRevoked callback function. It must return false (or nil) to keep the
// connection alive.
func certCallback(s *SSHServer) IsRevokedCallback {

    return func(cert *ssh.Certificate) bool {
        // Grab the certificate
        s.Cert = *cert
        s.IsSSH = true
```

```
            // Always return false
            return false
        }
    }
```

Inside `IsRevoked` we have access to the SSH certificate. Here we just assign it to the `Cert` field.

If you want to verify the certificate, this is the place.

Question!!!!

Help me if you can. I don't like returning unnamed functions like this. But unless I create global variables, I need to be able to access `s *SSHServer` inside `certCallback` to populate it. The function type is strict so I cannot add arguments.

I think defining the inside function as a method will work. Am I write? Wrong? Please let me know if you know the answer.

HostKeyFallback

Not all servers have SSH certificates. In fact, most servers probably do not. If server does not send a certificate, this function will be called (and the connection will terminate if this function is not defined).

If server is valid this function should return nil.

```
// hostCallback is the callback function for HostKeyCallback in SSH config.
// It can access hostname, remote address and server's public key.
func hostCallback(s *SSHServer) ssh.HostKeyCallback {
    return func(hostname string, remote net.Addr, key ssh.PublicKey) error {
        s.Hostname = hostname
        s.PublicKey = key
        // Return nil because we want the connection to move forward
        return nil
    }
}
```

Here we grab server's public key and hostname.

With these three callbacks set, we can move to the next step.

Step 3: Create ssh.ClientConfig

ssh.ClientConfig is needed for every SSH connection in Go. You can read about creating SSH connections in Hacking with Go - 04.4.

```
// Create SSH config
config := &ssh.ClientConfig{
    // Test username and password
    User: TestUser,
    Auth: []ssh.AuthMethod{
        ssh.Password(TestPassword),
    },
    HostKeyCallback: certCheck.CheckHostKey,
    BannerCallback:  bannerCallback(s),
    Timeout:         Timeout, // timeout
}
```

`Timeout` is also important. we do not want goroutines to wait forever connecting to inaccessible addresses. It's set to 5 seconds by default. Can be changed in the constants.

Banner callback

Banner callback is another important function for information gathering. By now, you know the drill.

```
// bannerCallback is the callback function for BannerCallback in SSH config.
// Grabs server banner and stores it in the SSHServer object.
func bannerCallback(s *SSHServer) ssh.BannerCallback {
    return func(message string) error {
        // Store the banner
        s.Banner = message
        // Return nil because we want the connection to move forward
        return nil
    }
}
```

We store the banner message and return `nil`. Any other return value will terminate the connection.

Step 4: ClientConfig.HostKeyCallback

This callback starts the server verification chain. It needs a function with ssh.HostKeyCallback type:

- `type HostKeyCallback func(hostname string, remote net.Addr, key PublicKey) error`

The package actually suggests (*CertChecker) CheckHostKey (we looked at its source code earlier). Looking inside `ClientConfig`, you can see I am passing it like this:

- `HostKeyCallback: certCheck.CheckHostKey,`

This is where everything clicks. We created a `certCheck` and set its callback functions. Now we are passing it to be called when we connect to a server.

Other ways of verifying servers

If you do not want to verify server's certificate, you can plug in three different types of functions here.

- `ssh.FixedHostKey(key PublicKey)` : Returns a function to check the hostkey.
- `ssh.InsecureIgnoreHostKey()` : Ignore everything! **Danger! Will Robinson!**
- `Custom host verifier` : Return nil if host is ok, otherwise return an error.

Read more about them in the verifying host.

A note about ssh.InsecureIgnoreHostKey()
After the breaking change as a consequence of the Golang SSH security blog post linked earlier, everyone seems to be using this. I am not in the position to tell you how to write your code. But make sure you know what you are doing when using this function. *cough* hashicorp packer *cough*.

Step 5: Connecting to SSH servers

Here comes the concurrent part. We have a list of addresses and our callbacks are set correctly. Time to connect to servers with `discover` .

discover method

```
// discover connects to ip:port and attempts to make an SSH connection.
// If successful, some SSH properties will be populated (most importantly isSSH
// and isAlive).
func (s *SSHServer) discover() {
    // Release waitgroup after returning
    defer discoveryWG.Done()

    defer logSSH.Println("finished connecting to", s.Address)

    certCheck := &ssh.CertChecker{
        IsHostAuthority: hostAuthCallback(),
        IsRevoked:       certCallback(s),
        HostKeyFallback: hostCallback(s),
    }

    // Create SSH config
    config := &ssh.ClientConfig{
        // Test username and password
        User: TestUser,
        Auth: []ssh.AuthMethod{
            ssh.Password(TestPassword),
        },
        HostKeyCallback: certCheck.CheckHostKey,
        BannerCallback:  bannerCallback(s),
        Timeout:         Timeout, // timeout
    }

    logSSH.Println("starting SSH connection to ", s.Address)
```

```
        sshConn, err := ssh.Dial("tcp", s.Address, config)
        if err != nil {
            // If error contains "unable to authenticate", there's something there
            logSSH.Println("error ", err)
            return
        }

        // Close connection if we succeed (almost never happens)
        sshConn.Close()
    }
```

First we defer releasing the waitgroup and the log message. This waitgroup will be explained later. In short, it's here to ensure that all `discover` goroutines are finished before starting the next stage.

Next are `CertCheck` and `ClientConfig`. We have already seen them. And finally we are connecting with `ssh.Dial`.

Goroutines and sync.WaitGroups

Each connection is done in its own goroutine. This means, we have to wait for these to complete before processing the results. We use `sync.WaitGroups`. For a longer version please read Hacking with Go - 02.6 - Syncing goroutines. But a tl;dr description is:

1. Every time a goroutine is started, we add one to the waitgroup (note we need to do this in the calling function, not inside the goroutine).
2. When the goroutine returns we subtract one (the `defer discoveryWG.Done()` in `discover`).
3. Wait in main for all goroutines to finish with `discoveryWG.Wait()`. This will block the program until they all return.

```
for _, v := range servers {
    // Before each goroutine add 1 to waitgroup
    discoveryWG.Add(1)
    go v.discover()
}

// Wait for all discovery goroutines to finish
discoveryWG.Wait()
```

SSH Harvester in action

And finally we can see the tool in action.

If the server returns a certificate:

If it returns a public key, `HostKeyFallBack` is triggered and we can it:

Note, server's have different keys for different ciphersuits. For example `dsa`, `ecdsa`, `rsa` and `ed25519` (the DJB curve). Depending what ciphersuite client supports, you may see one of these. That's another TODO.

Conclusion

It took me a couple of days to figure everything out because I could not find any examples or tutorials. But now we know how to verify SSH certificates. Hope this is useful, if you have any feedback please let me know.
[^1]: I should have actually sent a patch. But signing up for Gerrit was a pain. Would have been the easiest way to become a "Golang contributor" and put it in my Twitter bio/resume (kidding).

Continue reading ⇛ 05 - Parsing Files

05 - Parsing Files

Usually when we need to open and parse file formats, the normal parsers are not useful. Either files are badly formatted or something is hidden. Making our own file parser is the way to go.

Table of Contents

- 05.1 - Extracting PNG Chunks

05.1 - Extracting PNG Chunks

This is a copy of my [blog post][png-chunk].

I wrote some quick code that parses a PNG file, extracts some information, identifies chunks and finally extracts chunk data. The code has minimal error handling (if chunks are not formatted properly). We also do not care about parsing `PLTE` and `tRNS` chunks although we will extract them.

Code is in the `05/05.1` directory.

Golang's https://golang.org/src/image/png/reader.go does a decent job of explaining the rendering. But we are not interested in rendering.

Instead we look at `libpng` documentation at http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html. I am going to use a simple example (just a black rectangle which was supposed to

be a square lol) to demonstrate:

```
00000000  89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 00 6f 00 00 00 73 08 02 00 00 00 19 b3 cb  |...o...s......³Ë|
00000020  d7 00 00 00 01 73 52 47 42 00 ae ce 1c e9 00 00  |×....sRGB.®Î.é..|
00000030  00 04 67 41 4d 41 00 00 b1 8f 0b fc 61 05 00 00  |..gAMA..±..üa...|
00000040  00 09 70 48 59 73 00 00 0e c3 00 00 0e c3 01 c7  |..pHYs...Ã...Ã.Ç|
00000050  6f a8 64 00 00 00 3c 49 44 41 54 78 5e ed c1 01  |o¨d...<IDATx^íÁ.|
00000060  0d 00 00 00 c2 a0 f7 4f 6d 0f 07 04 00 00 00 00  |....Â ÷Om.......|
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000090  70 ae 06 96 0a 00 01 1e c4 f7 41 00 00 00 00 49  |p®......Ä÷A....I|
000000a0  45 4e 44 ae 42 60 82                              |END®B`.|
```

PNG Header

PNG starts with an 8-byte magic header:

- `89 50 4E 47 0D 0A 1A 0A`
- `const pngHeader = "\x89PNG\r\n\x1a\n"` from https://golang.org/src/image/png/reader.go.

When you open a PNG file, you can see `PNG` in the signature.

After the signature, there are a number of chunks.

PNG Chunks

Each chunk has four fields:

- `unint32` length in big-endian. This is the length of the data field.
- Four-byte chunk type. Chunk type can be anything [Footnote 1].
- Chunk data is a bunch of bytes with a fixed length read before.
- Four-byte CRC-32 of Chunk 2nd and 3rd field (chunk type and chunk data).

```
// Each chunk starts with a uint32 length (big endian), then 4 byte name,
// then data and finally the CRC32 of the chunk data.
type Chunk struct {
    Length int    // chunk data length
    CType  string // chunk type
    Data   []byte // chunk data
```

```
    Crc32  []byte // CRC32 of chunk data
}
```

First chunk or `IHDR` looks like this:

Converting big-endian `uint32` s to int is straightforward:

```
// uInt32ToInt converts a 4 byte big-endian buffer to int.
func uInt32ToInt(buf []byte) (int, error) {
    if len(buf) == 0 || len(buf) > 4 {
        return 0, errors.New("invalid buffer")
    }
    return int(binary.BigEndian.Uint32(buf)), nil
}
```

Trick #1: When reading chunks, I did something I had not done before. I passed in an `io.Reader` . This let me pass anything that implements that interface to the method. As each chunk is populated, reader pointer moves forward and gets to the start of next chunk. Note this assumes chunks are formatted correctly and does not check the CRC32 hash.

```
// Populate will read bytes from the reader and populate a chunk.
func (c *Chunk) Populate(r io.Reader) error {

    // Four byte buffer.
    buf := make([]byte, 4)

    // Read first four bytes == chunk length.
    if _, err := io.ReadFull(r, buf); err != nil {
        return err
    }
    // Convert bytes to int.
    // c.length = int(binary.BigEndian.Uint32(buf))
    var err error
    c.Length, err = uInt32ToInt(buf)
    if err != nil {
        return errors.New("cannot convert length to int")
    }

    // Read second four bytes == chunk type.
    if _, err := io.ReadFull(r, buf); err != nil {
        return err
    }
    c.CType = string(buf)

    // Read chunk data.
    tmp := make([]byte, c.Length)
    if _, err := io.ReadFull(r, tmp); err != nil {
```

```
            return err
        }
        c.Data = tmp

        // Read CRC32 hash
        if _, err := io.ReadFull(r, buf); err != nil {
            return err
        }
        // We don't really care about checking the hash.
        c.Crc32 = buf

        return nil
    }
```

IHDR Chunk

IHDR is a special chunk that contains file information. It's always 13 bytes and has:

```
    // Width:              4 bytes
    // Height:             4 bytes
    // Bit depth:          1 byte
    // Color type:         1 byte
    // Compression method: 1 byte
    // Filter method:      1 byte
    // Interlace method:   1 byte
```

These will go directly into the `PNG` struct:

```
    type PNG struct {
        Width             int
        Height            int
        BitDepth          int
        ColorType         int
        CompressionMethod int
        FilterMethod      int
        InterlaceMethod   int
        chunks            []*Chunk // Not exported == won't appear in JSON string.
        NumberOfChunks    int
    }
```

Trick #2: `chunks` does not start with a capital letter. It's not exported, so it is not parsed when we convert the struct to JSON.

Parsing the header pretty easy:

```
    // Parse IHDR chunk.
    // https://golang.org/src/image/png/reader.go?#L142 is your friend.
```

```
func (png *PNG) parseIHDR(iHDR *Chunk) error {
    if iHDR.Length != iHDRlength {
        errString := fmt.Sprintf("invalid IHDR length: got %d - expected %d",
            iHDR.Length, iHDRlength)
        return errors.New(errString)
    }

    tmp := iHDR.Data
    var err error

    png.Width, err = uInt32ToInt(tmp[0:4])
    if err != nil || png.Width <= 0 {
        errString := fmt.Sprintf("invalid width in iHDR - got %x", tmp[0:4])
        return errors.New(errString)
    }

    png.Height, err = uInt32ToInt(tmp[4:8])
    if err != nil || png.Height <= 0 {
        errString := fmt.Sprintf("invalid height in iHDR - got %x", tmp[4:8])
        return errors.New(errString)
    }

    png.BitDepth = int(tmp[8])
    png.ColorType = int(tmp[9])

    // Only compression method 0 is supported
    if int(tmp[10]) != 0 {
        errString := fmt.Sprintf("invalid compression method - expected 0 - got %x",
            tmp[10])
        return errors.New(errString)
    }
    png.CompressionMethod = int(tmp[10])

    // Only filter method 0 is supported
    if int(tmp[11]) != 0 {
        errString := fmt.Sprintf("invalid filter method - expected 0 - got %x",
            tmp[11])
        return errors.New(errString)
    }
    png.FilterMethod = int(tmp[11])

    // Only interlace methods 0 and 1 are supported
    if int(tmp[12]) != 0 {
        errString := fmt.Sprintf("invalid interlace method - expected 0 or 1 - got %x"
            tmp[12])
        return errors.New(errString)
    }
    png.InterlaceMethod = int(tmp[12])

    return nil
}
```

Our example's `IHDR` is:

```
{
    "Width": 111,
    "Height": 115,
    "BitDepth": 8,
    "ColorType": 2,
    "CompressionMethod": 0,
    "FilterMethod": 0,
    "InterlaceMethod": 0,
    "NumberOfChunks": 6
}
```

IDAT Chunks

IDAT chunks contain the image data. They are compressed using deflate. If you look at the first chunk, you will see the `zlib` magic header. This stackoverflow answer lists them:

- `78 01` - No Compression/low
- `78 9C` - Default Compression
- `78 DA` - Best Compression

Another answer has more info:

```
Level | ZLIB  | GZIP
  1   | 78 01 | 1F 8B
  2   | 78 5E | 1F 8B
  3   | 78 5E | 1F 8B
  4   | 78 5E | 1F 8B
  5   | 78 5E | 1F 8B
  6   | 78 9C | 1F 8B
  7   | 78 DA | 1F 8B
  8   | 78 DA | 1F 8B
  9   | 78 DA | 1F 8B
```

I have seen a lot of random looking blobs starting with `78 9C` when reversing custom protocols at work. I have never seen the other two headers.

In Go we can `inflate` the blob (decompress them) with zlib.NewReader:

```
package main

import (
    "compress/zlib"
    "io"
    "os"
```

```
)

func main() {
    zlibFile, err := os.Open("test.zlib")
    if err != nil {
        panic(err)
    }
    defer zlibFile.Close()

    r, err := zlib.NewReader(zlibFile)
    if err != nil {
        panic(err)
    }
    defer r.Close()

    outFile, err := os.Create("out-zlib")
    if err != nil {
        panic(err)
    }
    defer outFile.Close()

    io.Copy(outFile, r)
}
```

Note that each chunk is not compressed individually. All IDAT chunks need to be extracted, concatenated and decompressed together.

In our case, `IDAT` chunk has the `78 5E` header:

```
00000000  78 5e ed c1 01 0d 00 00 00 c2 a0 f7 4f 6d 0f 07  |x^íÁ.....Â ÷Om..|
00000010  04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 70 ae 06 96 0a 00 01              |.....p®.....|
```

Everything else is straightforward after this.

Tool Operation

Operation is pretty simple. PNG is passed by `-file` . Tool will display the PNG info like height and width. `-c` flag will display the chunks and their first 20 bytes. Chunks can be saved to file individually. Modifying the program to collect, decompress and store the IDAT chunks is also simple.

[Footnote 1]: We can hide data in random chunks. The hidden chunk must be added before/after

IDAT chunks. The standard expects the chain of IDAT chunks to be uninterrupted.
[png-chunk]: https://parsiya.net/blog/2018-02-25-extracting-png-chunks-with-go/# 06 - Go-Fuzz
This section talks about [Go-Fuzz][go-fuzz]. Go-Fuzz is a coverage-guided fuzzer for Go code.
This section might be renamed to `Fuzzing` when new content arrives.

Start by reading the Quickstart guide and a few of the examples. Then move to sections 2 and 3 for hands-on practice.

Table of Contents

- 06.1 - Go-Fuzz Quickstart
- 06.2 - Fuzzing iprange with Go-Fuzz
- 06.2 - Fuzzing goexif2 with Go-Fuzz

[go-fuzz]: https://github.com/dvyukov/go-fuzz# Go-Fuzz Quickstart

1. Get Go-fuzz with `go get github.com/dvyukov/go-fuzz`.
2. Build and install `go-fuzz` and `go-fuzz-build`.
    - `cd src\github.com\dvyukov\go-fuzz\go-fuzz`
    - `go install`
    - `cd ..\go-fuzz-build`
    - `go install`
3. Get the target package and store it in `GOPATH`. I usually keep it under `src\github.com\author\project`.
4. Create a new file in the target package named `Fuzz.go`.
5. Create a function named `Fuzz` inside `Fuzz.go` with this signature `func Fuzz(data []byte) int`.
6. `Fuzz` should return `1` if input is good and `0` otherwise.
7. Create fuzzing directory, e.g. `go-fuzz-project-name`.
8. `go-fuzz-build github.com/author/project` (note forward slashes even on Windows). Copy the resulting file (`project-fuzz.zip`) to the fuzzing directory.
9. Make a directory called `corpus` and store samples there.
10. `go-fuzz -bin=project-fuzz.zip -workdir=.` to begin fuzzing.

Examples

- "Fuzzing the new unit testing" by Dmitry Vyukov (Go-Fuzz creator): https://go-talks.appspot.com/github.com/dvyukov/go-fuzz/slides/fuzzing.slide#1
- "go-fuzz github.com/arolek/ase" by Damian Gryski: https://medium.com/@dgryski/go-fuzz-github-com-arolek-ase-3c74d5a3150c
- "Going down the rabbit hole with go-fuzz" by Nemanja Mijailovic: https://mijailovic.net/2017/07/29/go-fuzz/

- DNS parser, meet Go fuzzer by Filippo Valsorda: https://blog.cloudflare.com/dns-parser-meet-go-fuzzer/
  - "Automated Testing with Go-Fuzz" GothamGo 2015: https://www.youtube.com/watch?v=kOZbFSM7PuI
- "Fuzzing Markdown parser written in Go" by Krzysztof Kowalczyk: https://blog.kowalczyk.info/article/n/fuzzing-markdown-parser-written-in-go.html

Continue reading ⇒ 06.2 - Fuzzing iprange with Go-Fuzz# 06.2 - Fuzzing iprange with Go-Fuzz

This article will show how to use Go-Fuzz to fuzz a Go library named `iprange` at:

- [https://github.com/malfunkt/iprange][iprange-github]

Code and fuzzing artifacts are at code/06/06.2/.

Setup

The article assumes you have a working Go installation and have `go-fuzz` and `go-fuzz-build` executables in `PATH`. If not, use the quickstart or any other tutorial to do so and return here when you are done.

The Fuzz Function

The `Fuzz` function is the fuzzer's entry point. It's a function with the following signature:

- `func Fuzz(data []byte) int`

It takes a byte slice from the fuzzer and returns an integer. This gives us great flexibility in deciding what we want to fuzz. `Fuzz` is part of the target package so we can also fuzz package internals.

The output of `Fuzz` is our feedback to the fuzzer. If the input was valid (usually in the correct format), it should return `1` and `0` otherwise.

Having roughly correctly formatted input is important. Usually, we are dealing with formatted data. Just randomly sending byte blobs to the target is not going to do much. We want data that can bypass preliminary format checks. We pass the blob to either the target package or another function (e.g. some format converter) and check if it passes the parser check without any errors. If so, `Fuzz` must return `1` to tell `go-fuzz` that our format was good.

For a good example, look at the `PNG` fuzz function from the readme file:

```
func Fuzz(data []byte) int {
    img, err := png.Decode(bytes.NewReader(data))
    if err != nil {
        if img != nil {
```

```
                panic("img != nil on error")
            }
            return 0
        }
        var w bytes.Buffer
        err = png.Encode(&w, img)
        if err != nil {
            panic(err)
        }
        return 1
    }
```

Fuzzing iprange

We can use the usage section in the [iprange][iprange-github] readme to become familiar with the package.

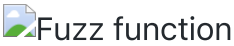
Then we need to get the package with `go get github.com/malfunkt/iprange`. This will copy package files to `$GOPATH\src\github.com\malfunkt\iprange`.

Note: I am using commit `3a31f5ed42d2d8a1fc46f1be91fd693bdef2dd52`, if the bug gets fixed, use this specific commit to reproduce the crashes.

Fuzz Function

Now we create a new file inside the package named `Fuzz.go` and write our fuzz function:

```
package iprange

func Fuzz(data []byte) int {
    _, err := ParseList(string(data))
    if err != nil {
        return 0
    }
    return 1
}
```


Fuzz function

We are converting the input from `go-fuzz` to a string and passing it to `ParseList`. If the parser returns an error, then it's not good input and we will return `0`. If it passes the check, we return `1`. Good input will be added to the original corpus.

If `go-fuzz` achieves more coverage with a specific input, it will be added to corpus even if we return `0`. But we do not need to care about that.

go-fuzz-build

Next step is using `go-fuzz-build` to make the magic blob. Create a directory (I always use my `src` directory`) and run this command inside it:

- `go-fuzz-build github.com/malfunkt/iprange`

Note you need to use forward slashes on Windows too. If `Fuzz` was written correctly we will get a zip file named `iprange-fuzz.zip`.

Note: This step usually takes a while. If the command line is not responsive after a few minutes, press enter a couple of times to check if it has finished. Sometimes the file is created but the command line windows is not updated.


Building go-fuzz-build

Corpus

To have meaningful fuzzing, we need to provide good samples. Create a directory named `corpus` inside the work directory and add one sample per file (file name does not matter).

Copy the items from [supported formats][iprange-supported] section of `iprange` readme. I created three files `test1/2/3`:

```
test1: 10.0.0.1, 10.0.0.5-10, 192.168.1.*, 192.168.10.0/24

test2: 10.0.0.1-10,10.0.0.0/24,
10.0.0.0/24

test3: 10.0.0.*, 192.168.0.*, 192.168.1-256
```

Fuzzing

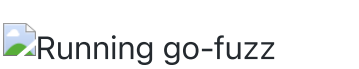
Now we can run `go-fuzz`.

- `go-fuzz -bin=iprange-fuzz.zip -workdir=.`

Note `workdir` should point to the parent of `corpus` directory.

Fuzzing Results

We will quickly get a crash and some new files will be added to the `corpus`.


Running go-fuzz

Analyzing the Crash

While we are fuzzing, we can analyze the current crash. `go-fuzz` has created two other directories besides `corpus`.

- `suppressions` contains crash logs. This allows `go-fuzz` to skip reporting the same exact crash.
- `crashers` has our loot. Each crash has three files and the file name is `SHA-1` hash of input. In this crash we have:
  - `17ee301be06245aa20945bc3ff3c4838abe13b52` contains the input that caused the crash `0.0.0.0/40`.
  - `17ee301be06245aa20945bc3ff3c4838abe13b52.quoted` is the input but quoted as a string.
  - `17ee301be06245aa20945bc3ff3c4838abe13b52.output` contains the crash dump.

Crash dump is:

```
panic: runtime error: index out of range

goroutine 1 [running]:
encoding/binary.binary.bigEndian.Uint32(...)
    /Temp/go-fuzz-build049016974/goroot/src/encoding/binary/binary.go:111
github.com/malfunkt/iprange.(*ipParserImpl).Parse(0xc04209d800, 0x526cc0, 0xc042083040
    /Temp/go-fuzz-build049016974/gopath/src/github.com/malfunkt/iprange/y.go:510 +0x2b
github.com/malfunkt/iprange.ipParse(0x526cc0, 0xc042083040, 0xa)
    /Temp/go-fuzz-build049016974/gopath/src/github.com/malfunkt/iprange/y.go:308 +0x8f
github.com/malfunkt/iprange.ParseList(0xc042075ed0, 0xa, 0xa, 0x200000, 0xc042075ed0,
    /Temp/go-fuzz-build049016974/gopath/src/github.com/malfunkt/iprange/y.go:63 +0xd6
github.com/malfunkt/iprange.Fuzz(0x3750000, 0xa, 0x200000, 0x3)
    /Temp/go-fuzz-build049016974/gopath/src/github.com/malfunkt/iprange/fuzz.go:4 +0x8
go-fuzz-dep.Main(0x5196e0)
    /Temp/go-fuzz-build049016974/goroot/src/go-fuzz-dep/main.go:49 +0xb4
main.main()
    /Temp/go-fuzz-build049016974/gopath/src/github.com/malfunkt/iprange/go.fuzz.main/m
exit status 2
```

bigEndian.Uint32

First stop is the Go standard library for `encoding/binary.binary.bigEndian.Uint32`. The source code for this method is at:

- [https://github.com/golang/go/blob/master/src/encoding/binary/binary.go#L110][bigendian-uint32]

```
func (bigEndian) Uint32(b []byte) uint32 {
    _ = b[3] // bounds check hint to compiler; see golang.org/issue/14808
```

```
    return uint32(b[3]) | uint32(b[2])<<8 | uint32(b[1])<<16 | uint32(b[0])<<24
}
```

Going to the issue in the comment, we land at [https://github.com/golang/go/issues/14808][issue-14808]. We can see what the bounds check is for. It's checking if the input has enough bytes and if not, it will panic before bytes are accessed. So this part of the chain is "working as intended."

This small piece of code results in a panic:

```
// Small program to test panic when calling Uint32(nil).
package main

import (
    "encoding/binary"
)

func main() {
    _ = binary.BigEndian.Uint32(nil)
    // _ = binary.BigEndian.Uint32([]byte(nil))
}
```

And the crash is similar to what we have seen:

```
$ go run test1.go
panic: runtime error: index out of range

goroutine 1 [running]:
encoding/binary.binary.bigEndian.Uint32(...)
        C:/Go/src/encoding/binary/binary.go:111
main.main()
        C:/Users/test-user/Go/src/gofuzz-stuff/malfunkt-iprange/test1.go:9 +0x11
exit status 2
```

Parse

Next item in the chain is at [https://github.com/malfunkt/iprange/blob/master/y.go#L309][iprange-parse]. It's a huge method but we know the method that was called so we can just search for `Uint32` . The culprit is inside [case 5][iprange-case5].

```
case 5:
    ipDollar = ipS[ippt-3 : ippt+1]
    //line ip.y:54
    {
        mask := net.CIDRMask(int(ipDollar[3].num), 32)
        min := ipDollar[1].addrRange.Min.Mask(mask)
        maxInt := binary.BigEndian.Uint32([]byte(min)) + // <----
            0xffffffff -
```

```
            binary.BigEndian.Uint32([]byte(mask)) // <----
        maxBytes := make([]byte, 4)
        binary.BigEndian.PutUint32(maxBytes, maxInt)
        maxBytes = maxBytes[len(maxBytes)-4:]
        max := net.IP(maxBytes)
        ipVAL.addrRange = AddressRange{
            Min: min.To4(),
            Max: max.To4(),
        }
    }
```

We can see two calls. The first is for `min` and the second is for `mask`. `mask` comes from the output of [net.CIDRMask][godoc-net-cidrmask]. Looking at the source code, we can see that it returns `nil` if mask is not valid:

```
// CIDRMask returns an IPMask consisting of `ones' 1 bits
// followed by 0s up to a total length of `bits' bits.
// For a mask of this form, CIDRMask is the inverse of IPMask.Size.
func CIDRMask(ones, bits int) IPMask {
    if bits != 8*IPv4len && bits != 8*IPv6len {
        return nil
    }
    if ones < 0 || ones > bits {
        return nil
    }
    // removed
}
```

We can investigate this by modifying the local `iprange` package code and printing `ipDollar[3].num` and `mask`.

```
case 5:
    ipDollar = ipS[ippt-3 : ippt+1]
    //line ip.y:54
    {
        fmt.Printf("ipdollar[3]: %v\n", ipDollar[3].num) // print ipdollar[3]
        mask := net.CIDRMask(int(ipDollar[3].num), 32)
        fmt.Printf("mask: %v\n", mask)                   // print mask
        min := ipDollar[1].addrRange.Min.Mask(mask)
        fmt.Printf("min: %v\n", min)                     // print min
        maxInt := binary.BigEndian.Uint32([]byte(min)) +
            0xffffffff -
            binary.BigEndian.Uint32([]byte(mask))
        maxBytes := make([]byte, 4)
        binary.BigEndian.PutUint32(maxBytes, maxInt)
        maxBytes = maxBytes[len(maxBytes)-4:]
        max := net.IP(maxBytes)
        ipVAL.addrRange = AddressRange{
            Min: min.To4(),
```

```
            Max: max.To4(),
        }
    }
```

Reproducing the Crash

Reproducing the crash is easy, we already have input and can just plug it into a small program using our `Fuzz` function:

```
// Small program to investigate a panic in iprange for invalid masks.
package main

import "github.com/malfunkt/iprange"

func main() {
    _ = Fuzz([]byte("0.0.0.0/40"))
}

func Fuzz(data []byte) int {
    _, err := iprange.ParseList(string(data))
    if err != nil {
        return 0
    }
    return 1
}
```

Note: We could write an easier test but I wanted to keep the `Fuzz` function intact.

```
$ go run test2.go
ipdollar[3]: 40
mask: <nil>
min: <nil>
panic: runtime error: index out of range

goroutine 1 [running]:
encoding/binary.binary.bigEndian.Uint32(...)
        C:/Go/src/encoding/binary/binary.go:111
github.com/malfunkt/iprange.(*ipParserImpl).Parse(0xc04209e000, 0x500920, 0xc04209c050
        yaccpar:354 +0x202f
github.com/malfunkt/iprange.ipParse(0x500920, 0xc04209c050, 0xa)
        yaccpar:153 +0x5f
github.com/malfunkt/iprange.ParseList(0xc042085ef8, 0xa, 0xa, 0x20, 0xc042085ef8, 0xa,
        ip.y:93 +0xbe
main.Fuzz(0xc042085f58, 0xa, 0x20, 0xc042085f58)
        C:/Users/test-user/Go/src/gofuzz-stuff/malfunkt-iprange/test1.go:10 +0x6c
main.main()
        C:/Users/test-user/Go/src/gofuzz-stuff/malfunkt-iprange/test1.go:6 +0x69
exit status 2
```

We can see `40` is passed to `net.CIDRMask` function and the result is `nil`. That causes the crash. We can see `min` is also `nil`.

Both min and mask are nil and result in a panic.

More Crashes?

I let the fuzzer run for another 20 minutes but it did not find any other crashes. Corpus was up to `60` items like:

- `2.8.0.0/4,0.0.0.5/0,2.8.0.0/4,0.0.0.5/0,2.8.0.0/4,0.0.0.5/0`
- `0.0.0.0/4,0.0.0.5-0,2.8.1.*,2.8.0.0/2`

Solution

Just pointing out bugs is not useful. Being a security engineer is not just finding vulnerabilities.

The quick solution is checking the values of `min` and `mask` before calling `Uint32`.

A better solution is to check the input for validity and good format before processing. For example, for IPv4 masks we can check if they are in the `16-30` range.

Continue reading ⇒ 06.3 - Fuzzing goexif2 Go-Fuzz
[go-fuzz]: https://github.com/dvyukov/go-fuzz [iprange-github]:
https://github.com/malfunkt/iprange [iprange-supported]:
https://github.com/malfunkt/iprange#supported-formats [bigendian-uint32]:
https://github.com/golang/go/blob/master/src/encoding/binary/binary.go#L110 [issue-14808]:
https://github.com/golang/go/issues/14808 [iprange-parse]:
https://github.com/malfunkt/iprange/blob/master/y.go#L309 [iprange-case5]:
https://github.com/malfunkt/iprange/blob/master/y.go#L498 [godoc-net-cidrmask]:
https://golang.org/pkg/net/#CIDRMask [net-cidrmask-github]:
https://github.com/golang/go/blob/master/src/net/ip.go#L68 # 06.3 - Fuzzing goexif2 with Go-Fuzz
This time we will be looking After `goexif` at [https://github.com/rwcarlsen/goexif][goexif-github].
Being a file parser, it's a prime target for `Go-Fuzz`. Unfortunately it has not been updated for a while. Instead, we will be looking at a fork at [https://github.com/xor-gate/goexif2][goexif2-github].
Code and fuzzing artifacts are at code/06/06.3/.

TL;DR

Steps are similar to the previous part.

1. `go get github.com/xor-gate/goexif2/exif`
2. `go get github.com/xor-gate/goexif2/tiff`

3. Create `Fuzz.go`.
4. Build with `go-fuzz-build`.
    - `go-fuzz-build github.com/xor-gate/goexif2/exif`
5. Fuzz
6. ???
7. Crashes!

If panics have been fixed, you can clone the commit `e5a111b2b4bd00d5214b1030deb301780110358d`.

Fuzz

The `Fuzz` function is easy straight forward:

```
// +build gofuzz

package exif

import "bytes"

func Fuzz(data []byte) int {
    _, err := Decode(bytes.NewReader(data))
    if err != nil {
        return 0
    }
    return 1
}
```

Samples

For samples, we need some pictures that contain exif data. The package comes with some samples inside the `samples` directory but I used samples at the following repository minus `corrupted.jpg`:

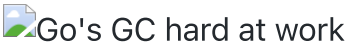
- https://github.com/ianare/exif-samples/tree/master/jpg

Running Out of Memory

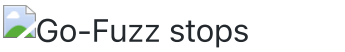
During fuzzing I got a lot of crashes that were caused by lack of memory. This usually happens when random bytes are read as field sizes and the size is not evaluated, thus the package will allocate very large chunks of memory.

We are instrumenting the application around 10000 times a second, this adds up and the garbage collector cannot keep up. Soon we need to `download more RAM`. You can see memory usage in the

following picture:



Looking at the fuzzer, we can see our `restarts` ratio is crap. This is the ratio of restarts to executions. We want it to be around `1/10000` but we have fallen to `1/1500`. This means we are crashing a lot. After a while, `Go-Fuzz` might even stop working (see stagnating total number of execs in the picture below).



Looking inside crash dumps, we see most of them are about running out of memory:

```
runtime: out of memory: cannot allocate 25769803776-byte block (25832882176 in use)
fatal error: out of memory

runtime stack:
runtime.throw(0x547da6, 0xd)
    /go-fuzz-build214414686/goroot/src/runtime/panic.go:616 +0x88
runtime.largeAlloc(0x600000000, 0x440001, 0x5f8330)
    /go-fuzz-build214414686/goroot/src/runtime/malloc.go:828 +0x117
runtime.mallocgc.func1()
    /go-fuzz-build214414686/goroot/src/runtime/malloc.go:721 +0x4d
runtime.systemstack(0x0)
    /go-fuzz-build214414686/goroot/src/runtime/asm_amd64.s:409 +0x7e
runtime.mstart()
    /go-fuzz-build214414686/goroot/src/runtime/proc.go:1175
```

This means we are running out of memory and it's not a legitimate crash. Before continuing we need to go and investigate the root cause.

Lesson #0: Fix `Go-Fuzz` running out of memory:

- Fix bugs that result in the allocation of large chunks of memory.
- Run fewer workers with `-procs`. By default `Go-Fuzz` uses all of your CPU cores (including virtual).

Analyzing Crashes

Let's look at our crashes.

```
05/03/2018  12:16 AM               365 171e8e5ca3e3d609322376915dcfa3dd56938845
05/03/2018  12:16 AM             3,651 171e8e5ca3e3d609322376915dcfa3dd56938845.output
05/03/2018  12:16 AM               912 171e8e5ca3e3d609322376915dcfa3dd56938845.quoted
05/01/2018  11:53 PM               186 3f5b7d448a0791f5739fa0a2371bb2492b64f835
05/01/2018  11:53 PM             1,928 3f5b7d448a0791f5739fa0a2371bb2492b64f835.output
05/01/2018  11:53 PM               312 3f5b7d448a0791f5739fa0a2371bb2492b64f835.quoted
```

```
05/01/2018  11:25 PM               114 49dfc363adbbe5aac9c2f8afbb0591c3ef1de2c3
05/01/2018  11:25 PM             1,383 49dfc363adbbe5aac9c2f8afbb0591c3ef1de2c3.output
05/01/2018  11:25 PM               186 49dfc363adbbe5aac9c2f8afbb0591c3ef1de2c3.quoted
05/01/2018  11:26 PM                22 a59a2ad5701156b88c6a132e1340fe006f67280c
05/01/2018  11:26 PM             1,677 a59a2ad5701156b88c6a132e1340fe006f67280c.output
05/01/2018  11:26 PM                63 a59a2ad5701156b88c6a132e1340fe006f67280c.quoted
```

Reproducing Crashes

As we know `Go-Fuzz` conveniently stores the inputs in files. We can use the following code snippet to reproduce crashes:

```
// Sample app to test crash a5 for xor-gate/goexif2.
package main

import (
    "fmt"
    "os"

    "github.com/xor-gate/goexif2/exif"
)

func main() {
    f, err := os.Open("crashers\\a59a2ad5701156b88c6a132e1340fe006f67280c")
    if err != nil {
        panic(err)
    }
    defer f.Close()

    _, err = exif.Decode(f)
    if err != nil {
        fmt.Println("err:", err)
        return
    }
    fmt.Println("no err")
}
```

A5 and 3F Crashes

These two panics are similar:

```
panic: runtime error: makeslice: len out of range

goroutine 1 [running]:
github.com/xor-gate/goexif2/tiff.(*Tag).convertVals(0xc04205a280, 0xc042080480, 0xc042
    /go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/tiff/tag.go:258 +0x
github.com/xor-gate/goexif2/tiff.DecodeTag(0x30a0000, 0xc042080480, 0x5605c0, 0x613170
```

```
    /go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/tiff/tag.go:182 +0x
github.com/xor-gate/goexif2/tiff.DecodeDir(0x30a0000, 0xc042080480, 0x5605c0, 0x613170

// removed
```

`A5` crash payload is:

```
00000000  49 49 2a 00 08 00 00 00 30 30 30 30 05 00 00 00  |II*.....0000....|
00000010  00 a0 30 30 30 30                                 |. 0000|
```

The panic is happening at https://github.com/xor-gate/goexif2/blob/develop/tiff/tag.go#L258:

```
case DTRational:
    t.ratVals = make([][]int64, int(t.Count))
    for i := range t.ratVals {
```

We can add some print statements to the local copy the package and investigate it:

```
case DTRational:
    fmt.Println("t.count: ", t.Count)
    t.ratVals = make([][]int64, int(t.Count))
    for i := range t.ratVals {
```

Running `test-crash-a5.go` we get the value:

```
$ go run test-crash-a5.go
t.count:  2684354560
panic: runtime error: makeslice: len out of range

goroutine 1 [running]:
github.com/xor-gate/goexif2/tiff.(*Tag).convertVals(0xc04205a1e0, 0xc042082018, 0xc042
```

Bonus: int Overflow and Go Playground's Operating System

As you have noticed, the constant `2684354560` is more than the maximum of signed int32 (`2147483647`). However, when trying to cast this value locally in Windows 10 64-bit VM or on the Go playground we get different results.

Consider this mini-example:

```
// Testing overflow on int.
package main

import "fmt"
```

```
func main() {
    i := int(2684354560)
    fmt.Println(i)
}
```

Running this in the Windows 10 64-bit VM, does not return an error. While running the same program in Go playground returns this error `prog.go:8:11: constant 2684354560 overflows int32` .

This means the playground is using 32 bit `int` s and locally we are using 64 bit ones. Local is obvious because we are in a 64 bit OS. To get the OS of the Go playground we can use this other small program:

```
// Get OS and architecture.
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println(runtime.GOOS)
    fmt.Println(runtime.GOARCH)
}
```

And we get:

```
nacl
amd64p32
```

`amd64p32` means it's a 64-bit OS using 32-bit pointers and `int` s. We can use `unsafe.Sizeof` to see this.

```
// Get int and pointer size.
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var i int
    var p *int
    var p2 *float32
```

```
    fmt.Printf("Size of int      : %d\n", unsafe.Sizeof(i))
    fmt.Printf("Size of *int     : %d\n", unsafe.Sizeof(p))
    fmt.Printf("Size of *float32 : %d\n", unsafe.Sizeof(p2))
}
```

On Go playground we get:

```
Size of int      : 4
Size of *int     : 4
Size of *float32 : 4
```

But locally we get:

```
$ go run int-pointer-size.go
Size of int : 8
Size of int*: 8
Size of *float32 : 8
```

Note: Pointers are just memory addresses. It does not matter what they are pointing to. As you can see `*float32` has the same size as a `*int32` or `*int64`.

Lesson #1: `int` is OS dependent. It's better to use data types with explicit lengths like `int32` and `int64`. Also if you do not need negative numbers, use unsigned versions (but be careful of underflows).

makeslice: len out of range

Now let's get back to the crash. We are trying to create a large slice and the result is an error. We can trace back this error to slice.go in Go source:

```
func makeslice(et *_type, len, cap int) slice {
    // NOTE: The len > maxElements check here is not strictly necessary,
    // but it produces a 'len out of range' error instead of a 'cap out of range' erro
    // when someone does make([]T, bignumber). 'cap out of range' is true too,
    // but since the cap is only being supplied implicitly, saying len is clearer.
    // See issue 4085.
    maxElements := maxSliceCap(et.size)
    if len < 0 || uintptr(len) > maxElements {
        panic(errorString("makeslice: len out of range"))
    }

    if cap < len || uintptr(cap) > maxElements {
        panic(errorString("makeslice: cap out of range"))
    }

    p := mallocgc(et.size*uintptr(cap), et, true)
```

```
        return slice{p, len, cap}
    }

    // maxSliceCap from the same file.
    // maxSliceCap returns the maximum capacity for a slice.
    func maxSliceCap(elemsize uintptr) uintptr {
        if elemsize < uintptr(len(maxElems)) {
            return maxElems[elemsize]
        }
        return _MaxMem / elemsize
    }
```

`_MaxMem` is calculated in malloc.go and it dictates how much memory can be allocated. On Windows 64-bit it seems to be 32GB or 35 bits.

Root cause analysis: We are allocating too much memory.

Lesson #2: Amount of memory available for malloc is OS dependent and somewhat arbitrary.

Lesson #3: Manually check size before allocating memory for slices.

But `t.Count` has to come from somewhere.

t.Count's Origin

`t.Count` is calculated a bit further up at line 133.

```
    err = binary.Read(r, order, &t.Count)
    if err != nil {
        return nil, newTiffError("tag component count read failed", err)
    }

    // There seems to be a relatively common corrupt tag which has a Count of
    // MaxUint32. This is probably not a valid value, so return early.
    if t.Count == 1<<32-1 {
        return t, newTiffError("invalid Count offset in tag", nil)
    }
```

We are reading 4 bytes (`Count` is `uint32`) and populating `t.Count` . According to RFC2306 - Tag Image File Format (TIFF) - F Profile for Facsimile:

> TIFF fields (also called entries) contain a tag, its type (e.g. short, long, rational, etc.), a count (which indicates the number of values/offsets) and a value/offset.

So we get `2684354560` when we read `A0 00 00 00` from our payload in little-endian:

```
00000000  49 49 2a 00 08 00 00 00 30 30 30 30 05 00 00 00  |II*.....0000....|
00000010  00 a0 30 30 30 30                                 |. 0000|
```

Lesson #4: After reading data, check them for validity. This is more important for field lengths.

Fix A5 and 3F Crashes

I could not find anything about the maximum number of types in a tag in the RFC. But it's a `dword` (4 bytes) so it can contain values that cause the panic in `makeslice`. We can choose a large enough value that does not cause the panic. I think `2147483647` or `1<<31-1` is a good compromise.

We can add our new check to the current check:

```
// There seems to be a relatively common corrupt tag which has a Count of
// MaxUint32. This is probably not a valid value, so return early.
// Also check for invalid count values.
if t.Count == 1<<32-1 || t.Count >= 1<<31-1 {
    return t, newTiffError("invalid Count offset in tag", nil)
}
```

Now both crashes are avoided:

```
$ go run test-crash-a5.go
err: exif: decode failed (tiff: invalid Count offset in tag)

$ go run test-crash-3f.go
err: loading EXIF sub-IFD: exif: sub-IFD ExifIFDPointer decode failed: tiff: invalid C
```

49 Crash

This crash payload is:

```
00000000  4d 4d 00 2a 00 00 00 08 00 07 30 30 30 30 30 30  |MM.*......000000|
00000010  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30  |0000000000000000|
00000020  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30  |0000000000000000|
00000030  30 30 30 30 30 30 30 30 30 30 87 69 00 04 00 00  |0000000000.i....|
00000040  00 00 30 30 30 30 30 30 30 30 30 30 30 30 30 30  |..00000000000000|
00000050  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30  |0000000000000000|
00000060  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30  |0000000000000000|
00000070  30 30                                            |00|
```

And results in:

```
panic: runtime error: index out of range

goroutine 1 [running]:
github.com/xor-gate/goexif2/tiff.(*Tag).Int64(...)
```

```
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/tiff/tag.go:363
github.com/xor-gate/goexif2/exif.loadSubDir(0xc042080510, 0x547f15, 0xe, 0xc042080390,
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/exif/exif.go:211 +0x
github.com/xor-gate/goexif2/exif.(*parser).Parse(0x613170, 0xc042080510, 0xc0420804b0,
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/exif/exif.go:190 +0x
github.com/xor-gate/goexif2/exif.Decode(0x560240, 0xc042080480, 0x5ae92f8f, 0x212abedc
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/exif/exif.go:331 +0x
github.com/xor-gate/goexif2/exif.Fuzz(0x38f0000, 0x72, 0x200000, 0xc042047f48)
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/exif/Fuzz.go:8 +0xba
go-fuzz-dep.Main(0x550580)
    go-fuzz-build214414686/goroot/src/go-fuzz-dep/main.go:49 +0xb4
main.main()
    go-fuzz-build214414686/gopath/src/github.com/xor-gate/goexif2/exif/go.fuzz.main/ma
exit status 2
```

This can be reproduced by running `test-crash-49.go`. At this point we know the drill. Looking at tag.go:363:

```
// Int64 returns the tag's i'th value as an integer. It returns an error if the
// tag's Format is not IntVal. It panics if i is out of range.
func (t *Tag) Int64(i int) (int64, error) {
    if t.format != IntVal {
        return 0, t.typeErr(IntVal)
    }
    return t.intVals[i], nil
}
```

It's known that this method can panic. We need to modify it (and the other similar ones) to return an error instead.

Fix 49 Crash

The fix is straightforward. Before accessing `t.intVals[i]` we need to check if the index is valid. This can be accomplished by checking it against `len(t.intVals[i])`.

```
// Int64 returns the tag's i'th value as an integer. It returns an error if the
// tag's Format is not IntVal. It panics if i is out of range.
func (t *Tag) Int64(i int) (int64, error) {
    if t.format != IntVal {
        return 0, t.typeErr(IntVal)
    }
    if i >= len(t.intVals) {
        return 0, newTiffError("index out of range in intVals", nil)
    }
    return t.intVals[i], nil
}
```

Lesson #5: Check index against array length before access.

Now we do not panic but there's no error because it's suppressed at exif.go:211:

```
func loadSubDir(x *Exif, ptr FieldName, fieldMap map[uint16]FieldName) error {
    tag, err := x.Get(ptr)
    if err != nil {
        return nil
    }
    offset, err := tag.Int64(0)
    if err != nil {     // error is suppressed here
        return nil
    }
    // removed
```

The new error check needs to be added to these methods:

- `Rat2`
- `Int64`
- `Int`
- `Float`

A bit further down inside the `MarshalJSON` method we can see errors being ignored:

```
// removed
for i := 0; i < int(t.Count); i++ {
    switch t.format {
    case RatVal:
        n, d, _ := t.Rat2(i)
        rv = append(rv, fmt.Sprintf(`"%v/%v"`, n, d))
    case FloatVal:
        v, _ := t.Float(i)
        rv = append(rv, fmt.Sprintf("%v", v))
    case IntVal:
        v, _ := t.Int(i)
        rv = append(rv, fmt.Sprintf("%v", v))
    }
}
// removed
```

Looking at the function we can see by ignoring the errors, we will have garbage data in the JSON. However, I don't think we need to return errors here but I could be wrong.

Adding Crashes to Tests

After things are fixed, we need to add the crashes to tests. This will discover if these bug regress in the future. Unfortunately, the package uses `go generate` to generate tests and I have no clue how

to use it. But I know how to write normal Go test using the testing package. Our payloads are pretty small so we will embed them in the test file instead of adding extra files to the package.

```
package exif

import (
    "bytes"
    "fmt"
    "os"
    "testing"
)

var goFuzzPayloads = make(map[string]string)

// Populate payloads.
func populatePayloads() {

    goFuzzPayloads["3F"] = "II*\x00\b\x00\x00\x00\t\x000000000000" +
        "0000000000000000000" +
        "0000000000000000000" +
        "0000000000000000000" +
        "0000000000000000000" +
        "000000i\x87\x04\x00\x01\x00\x00\x00\xac\x00\x00\x0000" +
        "0000000000000000000" +
        "0000000000000000000" +
        "0000000000000000\x05\x00\x00\x00" +
        "\x00\xe00000"

    goFuzzPayloads["49"] = "MM\x00*\x00\x00\x00\b\x00\a0000000000" +
        "0000000000000000000" +
        "0000000000000000\x87i" +
        "\x00\x04\x00\x00\x00\x0000000000000000" +
        "0000000000000000000" +
        "00000000000000"

    goFuzzPayloads["A5"] = "II*\x00\b\x00\x00\x000000\x05\x00\x00\x00\x00\xa000" +
        "00"

}

// Test for Go-fuzz crashes.
func TestGoFuzzCrashes(t *testing.T) {
    for k, v := range goFuzzPayloads {
        t.Log("Testing gofuzz payload", k)
        v, err := Decode(bytes.NewReader([]byte(v)))
        t.Log("Results:", v, err)
    }
}

func TestMain(m *testing.M) {
    populatePayloads()
    ret := m.Run()
```

```
    os.Exit(ret)
}
```

Lesson #6: Add `Go-Fuzz` crashes to unit tests. This is useful for regression testing.

Lessons Learned

- `Go-Fuzz` can crash when running out of memory and return false positives. We can throttle it or fix memory allocation bugs before resuming.
- Use data types with explicit lengths such as `int32` and `int64` instead of OS dependent ones like `int`.
- Amount of memory available for malloc is OS dependent and somewhat arbitrary.
- Manually check the size before allocating memory for slices.
- Check data (esp. field lengths) for validity after reading them.
- Check index against array length before access.
- Add `Go-Fuzz` crashes to unit tests.