Building (Awesome) Forensics Tools in Go

Joe Sylve, M.S. @jtsylve

Vico Marziale, Ph.D. @vicomarziale

DFRWS 2015

BlackBag™ TECHNOLOGIES

504ENSICS LABS

Grab an Updated Copy of These Slides

https://github.com/jtsylve/slides

$whoarewe

- Senior Research Developers at BlackBag
- Owners 504ENSICS Labs
- UNO graduates
- BSides NOLA organizers
- Margarita enthusiasts
- Developers of open source stuff
  - Scalpel (c)
  - LiME (c)
  - Registry Decoder (python)
  - Spotlight Inspector (python)
  - DAMM (python)

Why Are We Here

- Not to start a flame war …
- Programming languages all have advantages and disadvantages
  - C: memory management, unsafe typing, close to the metal, but speed from all of the above
  - Python: higher level constructs, interpreted, just plain easier to read and write (for most tasks), slower because of the above (and don't even get me started on the GIL)

Specific Challenges in Forensics

- Need the ability to do rapid prototyping (for that rad-new, or crazy-old format that is super important at the moment)
- Need to easily interface with on-disk c-based structures
- Need to be fast – eat TiB of data potentially
- Related: need to use all those cores
- Would like something friendly to part-time devs (forensics research types)
- Cross platform analysis

The Basics

- Speed: not quite as fast as c (yet), but getting there
- Type safe:
  - No x = 5 + “N”
  - No adding uint32 and int64
    - no adding int + int64 (even if they are the same size)
- Garbage collected:
  - No manual memory management
  - No malloc, free
- Compiled (for speed) but so fast as to seem interpreted
  - Even large projects can build in under a second

More Basics

- First class functions, closures, function literals
- Syntax is brief, and familiar to c users
- Multiple assignment/return
- Complexity: only ~25 keywords
- Type inference
- Built-in c-like structures
- Awesome, well documented standard library

Obligatory

```
// goodbye.go
package main

 import (
      “fmt”
)

func main() {
      fmt.Println(“Goodbye, cruel world!”)
}
```

```
#go run goodbye.go
Goodbye, cruel world.
```

Variables

```
var x int = 42
```

For int, string, []byte, uint64 …

Or rely on type inference

```
x := 42 // I'm an int by default
```

Multiple returns

```
func coords() (int, int) {
    …
    return x, y
}
```

Multiple assignment

```
j, k := coords()
```

Zero Values

Each type has a value it's guaranteed to be when declared. Don't do this:

```
var i int = 0
```

because it's the same as:

```
var i int
```

(says the linter to me over and over again)

numeric types -> 0

bool -> false

string -> ""

Type Conversion

No:

```
i := 21 // an int
var j int64 = 11 //an int64
x := i +j
```

“invalid operation: i + j (mismatched types int and int64)”

Yes:

```
x := int64(i) + j
```

Also no:

```
type myInt int
var m myInt = 33 // an int under the covers
k := 89 // an int
y := m + k
```

Yes:

```
y := int(m) + K
```

Functions

```
func funcName (argName argType) returnType { }
```

Can has multiple arguments

```
func funcName(x int, y int, z int) { }
```

Can has multiple returns

```
func funcName() (int, int, int) {
    return x, y, z
}
```

Can has named (and naked) returns

```
func funcName() (x int, y int, z int) {
    return
}
```

Method Receivers

Any local type can have a method defined on it. E.g., define the function "square" on int, that takes no arguments and returns an int.

```
type myInt int
func (i myInt) square() myInt {
	return i*i
}

func main() {
	var i myInt = 23
	fmt.Println( i.square() )
}
```

Prints "529"

Method Receivers

Generally used for functions on structs (mimics objects)

```
type attendee struct {
	name string
	funds int
}

func (a *attendee) payRegistration() {
	a.funds -= 10000
}

func main() {
	me := attendee{name: “vico”, funds:10000 }
	me.payRegistration()
	fmt.Printf(“%v has %v funds\n”, me.name, me.funds)
}
```

Prints: “vico has 0 funds”

sort Library

The sort package contains an implementation of a sort function for a slice of ints.

```
import (
    "sort"
    "fmt"
)

func main() {
    things := []int{7, 17, 11, 53}
    sort.Ints(things)
    fmt.Println(things)
}
```

Prints: "[7, 11, 17, 53]"

Interfaces

What if you want to sort int64? Go does not provide a implementation for sorting a list of int64. It does however provide an interface Sortable in the package sort which will sort any list that implements the functions defined in the interface.

```
type Sortable interface {
    Len() int // Len is the number of elements in the collection.
    Less(i, j int) bool  // Less reports whether element at index i
                         // should sort before with index j.
    Swap(i, j int) // Swap swaps the elements with indexes i and j.
}
```

*little white lie

Implementing Interfaces

```
type sort64 []int64

func (s sort64) Len() int { return len(s) }
func (s sort64) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
func (s sort64) Less(i, j int) bool { return s[i] < s[j] }

func main() {
	var x, y, z int64 = 57, 41, 89
	things := sort64{x, y, z} // these are int64
	sort.Sort(things)
	fmt.Println(things)
}
```

Prints: “[41, 57, 89]”

Note that the word “implements” appears nowhere above.

Defer

The defer keyword place before a function call ensures that that function is called just before the enclosing function exits regardless of how/where the function exits.

```
import “os”

fd, _ := os.Open(“somefile.txt”)
defer fd.Close()
… do stuff with file …
```

Useful for any type of cleanup code that must have a chance to execute.

Parallelism

Built-in concurrency in go leverages goroutines and channels, **no messy manual locking** (unless you're into that sort of thing; I'm not judging), and no explicit thread management.

```
func double(i int) {
	fmt.Printf("%v\t", i*2)
}

func main() {
	for j:=0; j<100; j++ {
		go double(j)
	}
	// program exits
}
```

Prints the numbers 0-198 in *some* order, kind of …

WaitGroups

```
import (
        “sync”
        “fmt”
)

func double(i int) { fmt.Printf("%v\t", i*2) }

func main() {
        var wg sync.WaitGroup
        for j := 0; j < 100; j++ {
                wg.Add(1)
                go func(j int) {
                        defer wg.Done()
                        double(j)
                }(j)
        }
        wg.Wait()
}
```

Channels

Communication between goroutines is accomplished via channels.

```
func double(i int, c chan int) {
	c <- i*2
}

func main () {
	c := make(chan int)
	go double(11, c)
	res := <-c
}
```

Channels have a capacity, 1 by default. With only one goroutine this is ok, but if we wanted many goroutines running “double,” say 10, it would be better to have:

```
c := make(chan int, 10)
```

Producer/ Consumer

```
var done = make(chan bool)
var msgs = make(chan int)

func produce () {
	for i := 0; i < 10; i++ {
		msgs <- i
	}
	done <- true
}

func consume () {
	for {
		msg := <-msgs
		fmt.Println(msg)
	}
}

func main () {
	go produce()
	go consume()
	<- done
}
```

Errors

Idiomatic go demands using multiple returns to return an error value in addition to whatever the function should normally return. A ‘nil’ value for the error indicates no error occurred.

```
…
fd, err := os.Open(“file.txt”)
if err != nil {
      // do something maybe
      return err
}
defer fd.Close()
…
```

For file reads, io.EOF versus io.ErrUnexpectedEOF

Slice

go makes heavy use of slices instead of arrays. They are similar in look and feel to python slices.

```
s := make([]int, 0) // list (slice) of zero size
s = append(s, 42) // append 42 to the list
s[:] // the entire list
s[x:] // the list from x on
s[:x] // the list up to x
```

Note: generally, passing around slices is cheap. It’s a pointer to an underlying array.

Range

```
stuff := []whosiwhatsits{ … }
```

Long and messy way way

```
for i:=0; i<len(stuff); i++ {
    stuff[i].DoSomething() // or whatever
}
```

Short way and cuter way; “c” holds a counter, can ignore (“_”)

```
for c, elem := range(stuff) {
    elem.DoSomething()
}
```

Built-Ins

make: for creating slices, channels, maps

```
someSlice := make([]byte, 128)
```

append

```
someSlice = append(someSlice, elem)
```

len: get the length of a slice

```
len(someSlice)
```

range, for loop constructs

```
for ctr, elem := range someSlice { … loopy stuff … }
```

Capitalization

Case is important in go. Generally, capitalized constructs are accessible outside the package. Lower case constructs are not.

```
package mine

privateFunc() { }
PublicFunc() { }
type private struct {
    Public int
}
type Public struct {
    private int
}
```

package encodings/binary

Useful for reading structured data from input (a file or otherwise)

```
import (
	“encodings/binary”
	‘fmt”
)

type wineCellar struct {
	CurrentCapacity int
	Temperature int32
	Humidity [6]byte
}
…
wc := wineCellar{}
binary.Read(fd, binary.LittleEndian, &wc)
fmt.Printf(“%+v\n” wc)
```

package encodings/binary

Note that binary.Read() only likes to read into fixed size constructs.

```
type someStruct struct {
	Size int
	Name string
	Count int
}
```

Is not going to work because “string” is variable size. If you happen to know the size of the string field, use:

```
Name [stringFieldSize]byte
```

And then convert later

```
strName := string(x.Size[:])  // yes the syntax is weird, but it makes sense
```

Also note that binary.Read() can only read into Public fields, so all struct fields must have upper case names.

Argument Parsing

The simple version

```
import (
    “fmt”
    “os”
)

func main() {
    for _, a := range os.Args {
        fmt.Println(a)
}
```

Better Argument Parsing

```
import (
        “flag”
        …
)

func main() {
        dbg := flag.Bool(“dbg", false, ”print debugging")
        size := flag.Int(“s", 128 ”size of something")
        db := flag.String("db", "", “db to use")
        flag.Parse()
        fmt.Printf(“%v %v %v”, *dbg, *size, *db)
}
```

```
#go run flagtest.go --dbg –s 11 --db sql.db
true 11 sql.db
```

More Why Go: Toolchain

- build
- run: build and run in one step
- fmt: no arguing about where the braces go
- vet: find unsafe things
- lint: find non-idiomatic things
- doc
- test
- get: retrieve, compile, install external packages

fmt

- braces
- spacing
- line breaks
- makes (git) diff far more useful
- see: http://gofmt.com
- also enables go fix to update API calls

vet

- Unreachable code
- Shifts equal to or longer than the variable's length
- Variables that may have been unintentionally shadowed
- Locks that are erroneously passed by value
- Check for useless assignments

lint

“Writing well formatted and documented software should be the lowest common denominator. Moving beyond that, software should be written to follow best practices, use common conventions, and shouldn't try to be tricky unless it's one of those rare times to be tricky.”

- forces comment on Public constructs
- enforces other go-specific idioms

doc

- Automagically extracts documentation for source files
- Yours if you do things the go way (easy)
- The standard library
- try: “go doc fmt”
- godoc -http=:6060
  - yes, your go installation comes with a webserver

Happy Hunting!

Labs!

- Labs can be found on Github
  - https://github.com/jtsylve/DFRWS2015
  - Skeleton files and tests are provided to help you
  - Solutions are in a separate “solutions” branch
- Checkout Repository
  - go get github.com/jtsylve/DFRWS2015
- Edit your solutions in your $GOPATH
  - $GOPATH/src/github.com/jtsylve/DFRWS2015

Labs!

Remember:

fmt.Printf("%v\n", someGoThing)

is your friend.

Google (or duckduckgo) is also your friend.

If you copy code from the slides, make sure the quotes are fixed.

Questions / Comments / Problems

- Joe T. Sylve, M.S
  - joe.sylve@gmail.com
  - @jtsylve
- Vico Marziale, Ph.D.
  - vicodark@gmail.com
  - @vicomarziale