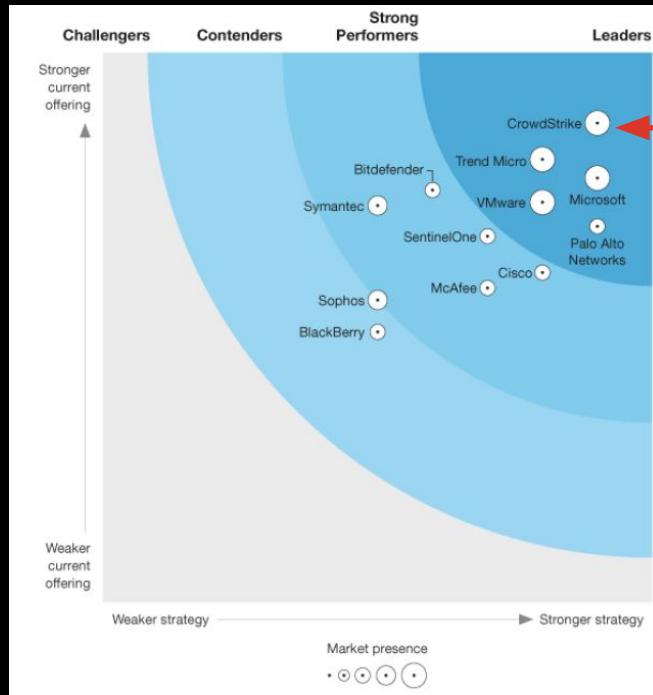


Intro to golang

CrowdStrike HEROES - Cloud Workshop

WHO IS CROWDSTRIKE?



Forrester Wave: Endpoint Security Software As A Service, Q2 2021

CrowdStrike Heroes - Cloud Track

A leader in the cybersecurity industry

- Founded in 2011
- “Cloud native” security company
- Remote first since day 1

WE
STOP
BREACHES
CROWDSTRIKE



THE "BIG" IN BIG DATA



1 TRILLION+
EVENTS/DAY

65,000

POTENTIAL INTRUSIONS STOPPED

CrowdStrike Heroes - Cloud Track

- Multiple production clouds
- ~300 microservices
- Tens of PB of data
- A lot of ML
- Security research & threat analysis



CS ROMANIA



- R&D center since 2018
- People from Bucharest, Cluj, Brasov, Iasi etc.
- Cloud, Data science, SRE, Security research, SDET, UI etc.
- Full ownership over products
- 170 employees
 - [We're hiring btw!](#)



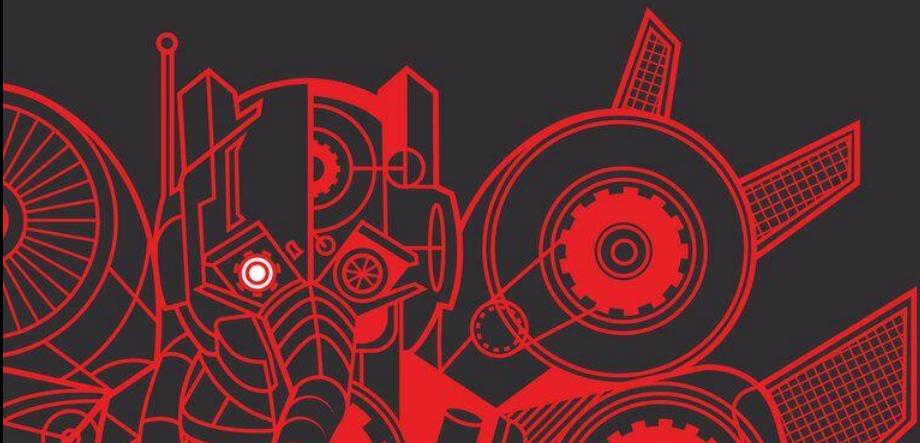
What those workshops will be about?

10 days - 9 workshops 16:00~20:00

1 contest 13:00~19:00 - Awards included

All the activities will take place in the same PR606 room.

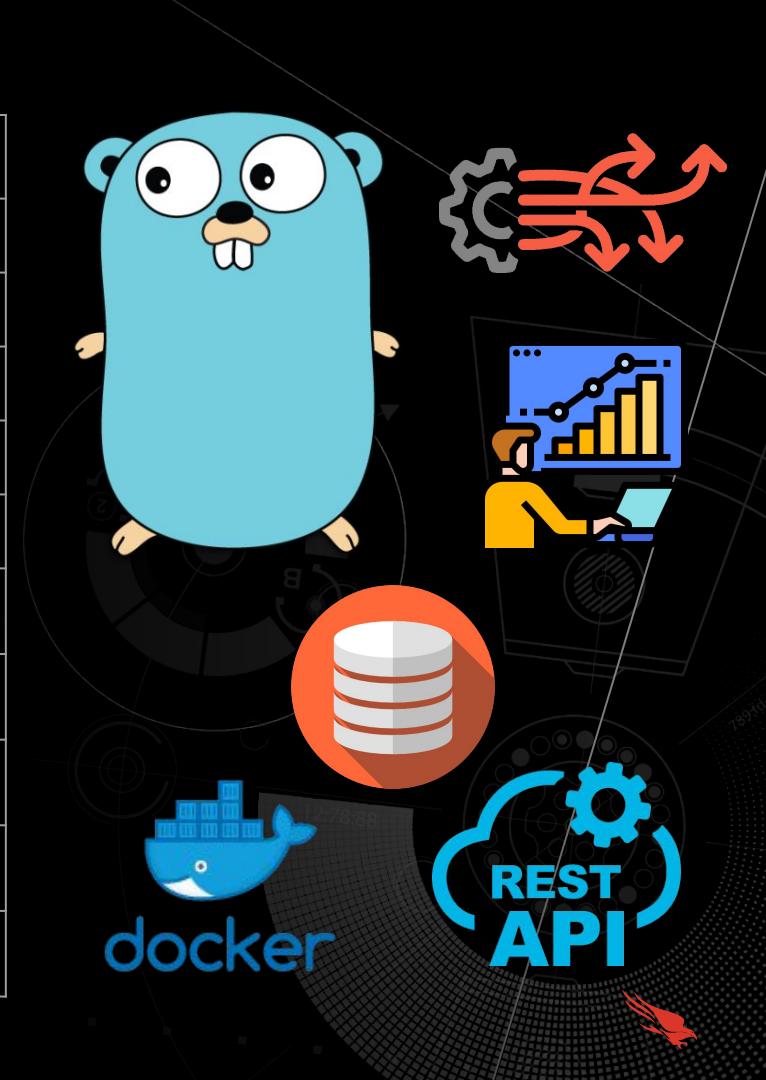
Please be in time for the start, feel free to grab some snacks, a drink, or make yourself comfortable.



**CROWDSTRIKE HEROES
SUMMER WORKSHOP**



Date	Topic
July 25 - 16:00	Intro to golang
July 26 - 16:00	Intro to golang (continuation)
July 27 - 16:00	Multithreading
July 28 - 16:00	Rest API
July 29 - 16:00	Unit testing, logging and monitoring
August 1 - 16:00	Workshop and Q&A
August 2 - 16:00	Deployments/Docker
August 3 - 16:00	Databases
August 4 - 16:00	Databases extended
August 5 - 13:00	Microservices contest (4h with Awards)



Intro to golang



What is go?

- Go was created at Google in 2007
- open source project
- similarly modeled after C
- compiled on most platforms
- based on packages
- garbage collection



Why use go?

Google **NETFLIX**



- lower latency than Java (where garbage collection pauses are an issue)
- more productive for developers than C
- built in concurrency
- support for unit testing

Hello World

- Entry point is in package main
 - Function main
- Packages imported explicitly
 - *import* keyword
- Import handle is the package name
- We can rename an import

```
import log "github.com/sirupsen/logrus"
```

- Exported & private names
 - exported if it begins with a **capital letter**
 - private names are not visible outside the package (but can be used in other files in the package)

```
package main  
  
import "fmt"  
  
func main() {  
    fmt.Println("Hello world!")  
}
```



How to compile go

```
go build main.go
```

```
go build ./...
```

```
go run ./...
```

```
go test ./...
```

```
go test -race
```

```
go fmt examples.go
```

```
go vet examples.go
```



Go modules

- A package is defined inside a go file and usually has the same name as the parent directory
- A module is a collection of Go packages stored in a **file tree**
- `go.mod` file
 - The first line defines the root of for all of the packages:
`module domain.name/modulename`
 - Associated `go.sum` file - package hash
 - `go mod tidy` to download dependencies
- When using a package in the same module we need to import the relative path
`import "domain.name/modulename/pakageinthemodule"`
- All packages can be found here <https://pkg.go.dev/>
- Demo: `demo/main.go`



Functions

- Function name first, arguments, then return type
- `func add(x int, y int) int`
 - `func add(x, y int) int`
- `x int (GO)` vs `int x (C)`
- Multiple results
 - Parentheses to delimit
- Names for return variables
- Demo: `demo/examples.go`



Variables & Constants

- Keyword **var**
 - `var i int`
- Declaring multiple variables at the same time
 - `var i, j, k int`
- Package level or function level
- Initialising variables
 - `var i, j int = 1, 2`
 - Default values are always set (unlike C)
- Short form: `k := 3`
 - Also multiples: `v1, v2, v3 := 42, false, "no!"`
- Demo: `demo/main.go`

- Keyword **const**
 - `const Pi = 3.14`
- High-precision values
- Multiple constants are usually in a group like imports
- No short form



Basic types

- `int`, `uint` and any other unspecified size types depend on the architecture of the system
- Zero values
 - `0` for numeric types,
 - `false` for the boolean type, and
 - `""` (the empty string) for strings.
 - `nil` for pointers
 - strings are NEVER nil
- Statically typed => type conversions
 - `i := 42` // defaulted int
 - `f := float64(i)`
- More in the demo

`bool`

`string`

`int` `int8` `int16` `int32` `int64`

`uint` `uint8` `uint16` `uint32` `uint64` `uintptr`

`byte` // alias for `uint8`

`rune` // alias for `int32`

// represents a Unicode code point

`float32` `float64`

`complex64` `complex128`



Flow control statements

for
if
else
switch
defer



Loops

- Generic keyword for

```
for i := 0; i < 10; i++ {  
    sum += i  
}
```

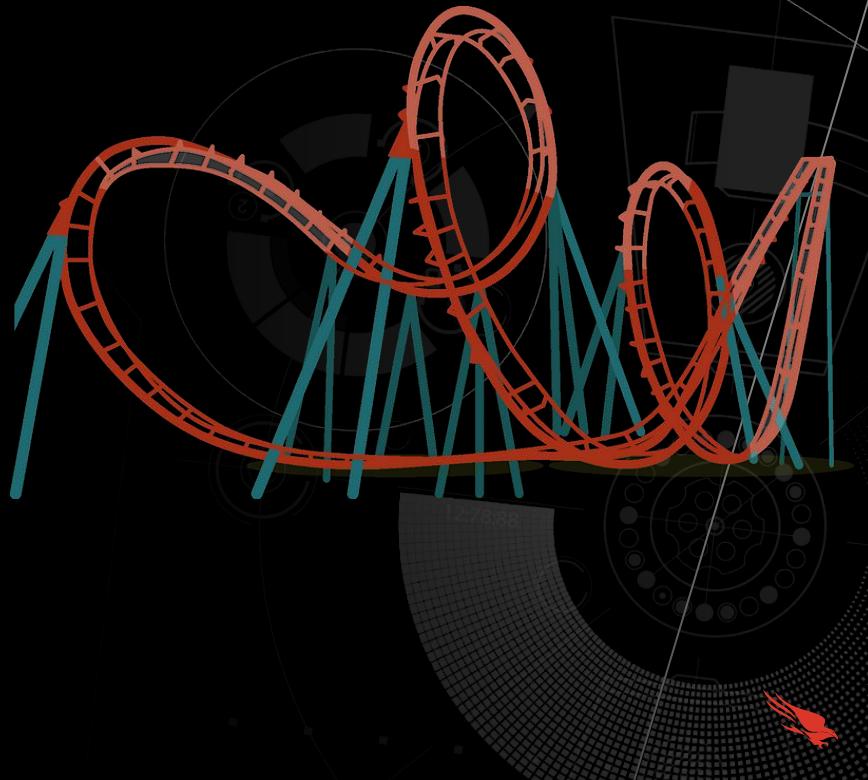
- It is similar in syntax to C

- No parentheses
 - Braces {} always required

- We can simulate a while loop by omitting statements

- We can omit everything XD

- Keywords: break and continue



if & else

```
if crashingIntoThings {  
    dont()  
}
```

- No need for parentheses
- We can define variables inside the if statement

```
if v := executeSth(); v < lim {  
    return v  
}
```

```
if something {  
    whatever()  
} else {  
    whateverElse()  
}
```

- Defined vars are visible inside else too



Switch

- Similar syntax to C
- No need for constants in the condition
- If a branch is taken others are skipped
- No need for condition at all actually
 - We can check completely different statements

```
switch {  
    case i == 0:  
        // does not exec if i != 0  
    case t.Hour() < 12:  
        // def  
}
```

```
fmt.Println("When's Saturday?")  
today := time.Now().Weekday()  
switch time.Saturday {  
    case today + 0:  
        fmt.Println("Today.")  
    case today + 1:  
        fmt.Println("Tomorrow.")  
    case today + 2:  
        fmt.Println("In two days.")  
    default:  
        fmt.Println("Too far away.")  
}
```



Defer

- Delays the execution of a function until the surrounding function returns
- Accepts only “void” functions
- Defers are added onto a stack
 - remember the call stack? ;)
- More examples in the demo

```
func main() {  
    defer fmt.Println("world")  
    // do lot  
    fmt.Println("hello")  
}
```

Pointers & arrays

Anyone miss them?



Pointers

A pointer holds the **memory address** of a value.

```
var p *int
```

- Operator & for generating pointers
- Operator * for retrieving the value of a pointer
- Zero value nil
- NO pointer arithmetic

```
if p != nil {  
    i := *p  
}
```

- We get an explicit error for bad dereferences: panic
- Demo



Arrays & slices

```
var a [10]int
```

- Array length is part of the type (cannot be resized)
- Accessing elements: `i := a[2]`
- Dynamically sized views: slices

```
var b []int
```

- Slices are formed by specifying limits: `a[low : high]`
 - We can omit one or both of these limits
- Slices are like references to arrays -> changes to a slice modify the array
- Length and capacity `len(s)`, `cap(s)`
- Zero value `nil`
- Demo



Constructing slices dynamically make & append

```
a := make([]int, 5) // len(a)=5  
b := make([]int, 0, 5) // len(b)=0, cap(b)=5
```

- We have a “constructor” built in function to declare slices
 - No need to import anything
- The second slice here is filled with zero values for its type
- Adding values to a slice is done with another built in function

```
a = append(a, 1)  
b = append(b, 2, 3, 4)  
b = append(b, a...) // appends the whole list
```



Iterating over arrays made easy

range

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}  
for i, v := range pow {  
    fmt.Printf("2**%d = %d\n", i, v)  
}
```

- The range keyword returns the index and a ‘copy’ of the value
- We can omit either one of them
 - `for i := range pow`
 - `for _, v := range pow`





Any
questions?

