go get my/vulnerabilities

Green threads are not eco friendly threads
Who

- (Web | Mobile) penetration tester
- Code reviewer
- Programmer
Go

- Google’s language
- Born in 2007 (quite new)
- Widespread
Cool, but how do I break it?

- Memory safety, Garbage Collection
- Anti-XSS/SQLi sanitization
- Built-in thread-safe constructs
Let’s start the digging

- New features usually lead to new vulnerabilities
- Goroutines are one of the main new features introduced by Go
Goroutines are concurrent function calls

```go
fmt.Println("Hello goroutines")
```
Let’s try this

```go
for i := 0; i <= 9; i++ {
    go func() {
        fmt.Println(i)
    }()
}
```
<table>
<thead>
<tr>
<th>Expectation</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
Wait...
Special functions #1: goroutines

- Concurrent
- Lightweight
- Multiplexed on OS Threads

```go
func(){
    //Code here
    }()
```
Special functions #2: closures

```go
freeVar := "Hello 

f := func(s string){
    fmt.Println(freeVar + s)
}

f("Closures")
// Hello Closures
```
Special functions #(1+2): closed goroutines

```go
for i := 0; i <= 9; i++ {
    go func() {
        fmt.Println(i)
    }()
}
```

// Here i == 10
Performance

- Writing to file is slow
- Aware scheduling
- Runtime waits only if necessary
The (odd) fix

for i := 0; i <= 9; i++ {
  go func() {
    fmt.Println(i)
  }()
}

for req := range queue {
  req := req // Create new instance of req for the goroutine.
Channels

```
ch := make(chan..)
go ch <- data close(ch)
for data := range ch {
    Buffer Buffer
    data <- ch data, ok <- ch
```

for data := range ch {
func Serve(queue chan *http.Request) {
    for req := range queue {
        go func() {
            process(req)
        }()
    }
}

responses to the wrong requests
Checkpoint

- **Variable scoping** is a nice point to focus on.

- **Aware** scheduling can make it easier to abuse races.

  how aware is the scheduler?
MPG model

OS Threads

GOMAXPROCS
Logical processors

Goroutines
Schedule me please

Scheduler calls are emitted at compile time
Consequences are weird

go func() {
    for i := 0; true ; i++ {
    }
}( )
time.Sleep(2 * time.Second)
fmt.Println("Done")
Cores amount matter
Runs the same way everywhere...

`runtime.GOMAXPROCS(1)`
Statically Strongly Typed

```go
func() {
    for i := range lst {
        for ; i <= 255 ; i++ {
            // Computation
        }
    }
}(())
```
Hidden problem: Garbage Collector

Garbage collector needs to stop goroutines
Garbage Collection?

GC politely asks goroutines to stop

Endless for
Free needed
Stop the world
Start the world

Garbage Collection??
Consequences are bad

go func() {
    var i byte
    for i = 0; i <= 255; i++ {
    }
}()

runtime.Gosched() //yield execution
runtime.GC()
fmt.Println("Done")
Here is the solution

Weird solution: use **non-inlinable function calls** in loops

The correct one: use **channels**
Checkpoint

- **Scheduling** must be taken into account
- **Goroutines** that don’t yield have potential for DoS

how do goroutines die?
Goroutines end

The only way for a goroutine to terminate is for it to return, or for the program to end.
Goroutines are not Garbage Collected

They **must be signalled to end** or they constitute an insidious opening for DoS
select the right solution?

```java
select { 
  case d1 <- ch1: 
  case d2, ok <- ch2: 
  default: 
}
```

ch1 <- data1

ch2 <- data2
Max execution time in PHP

```php
<?php
  set_time_limit(2);
  for($i=0;;$i++){
  }
?>
// Maximum execution time of
// 2 seconds exceeded
Max execution time in go

```go
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

TimeoutHandler returns a Handler that runs h with the given time limit.

The new Handler calls h.ServeHTTP to handle each request, but if a call runs for longer than its time limit, the handler responds with a 503 Service Unavailable error and the given message in its body. (If msg is empty, a suitable default message will be sent.) After such a timeout, writes by h to its ResponseWriter will return ErrHandlerTimeout.

So is this magic?
This is **NOT** PHP

type simpleHandler struct {
}
func (t *simpleHandler) ServeHTTP(w http.ResponseWriter, 
r *http.Request) {
    time.Sleep(10 * time.Second)
    fmt.Println("Got here")
}
func main() {
    sh := &simpleHandler{}
    tsh := http.TimeoutHandler(sh, 
        time.Second*2, 
        "Timeout!")
    http.ListenAndServe(":8080", tsh)
func TimeoutHandler ¶

func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler

TimeoutHandler returns a Handler that runs h with the given time limit.

The new Handler calls h.ServeHTTP to handle each request, but if a call runs for long, it issues a 503 Service Unavailable error and the given message in its suitable default message will be sent.) After such a timeout, writes by h to its ResponseErrHandlerTimeout.
Dive into sources

// Create timer
go func() {
    h.handler.\texttt{ServeHTTP}(tw, r)
    // Signal done channel
}()
select {} case <-done:
    // Handle HTTP stuff
case <-timeout:
    // Write error
Mind the gap

The standard library isn’t more powerful than you are, if you can’t kill a goroutine, neither can the stdlib.
Some more problems with signals

// The worker goroutine
for {
    select{
        case job <- jobs:
            process(job)
        case <-done:
            return
    }
}

// The main goroutine:
go worker()

// Work needs to end:
done <- true
Other (still not) correct fixes

go worker()
go worker()
go worker()
done <- true
done <- true
done <- true

case <-done:
done <- true
return
go worker()
done <- true
Even worse

done := make(chan..)
go
case <- done

done <- true

?
Other (still not) correct fixes

case <-done:
    done <- true
    return

go worker()
    done <- true
<- done
Just close it

go worker()
go worker()
go worker()
close(done)
Close channels

```go
ch := make(chan..)
go
for data, ok := ch {
    if !ok {
        close(ch)
    }
}
```
Conclusions

- Mind race conditions
- Dive into sources
- Follow signals
- Check for yielding calls
Thanks

Roberto Clapis

@empijei