HashiCorp

# Data Races!
# What are they?
# Why should I care?
# How do I fix them?

**Michael Schurter**
@schmichael
Nomad Team Lead

# Michael Schurter

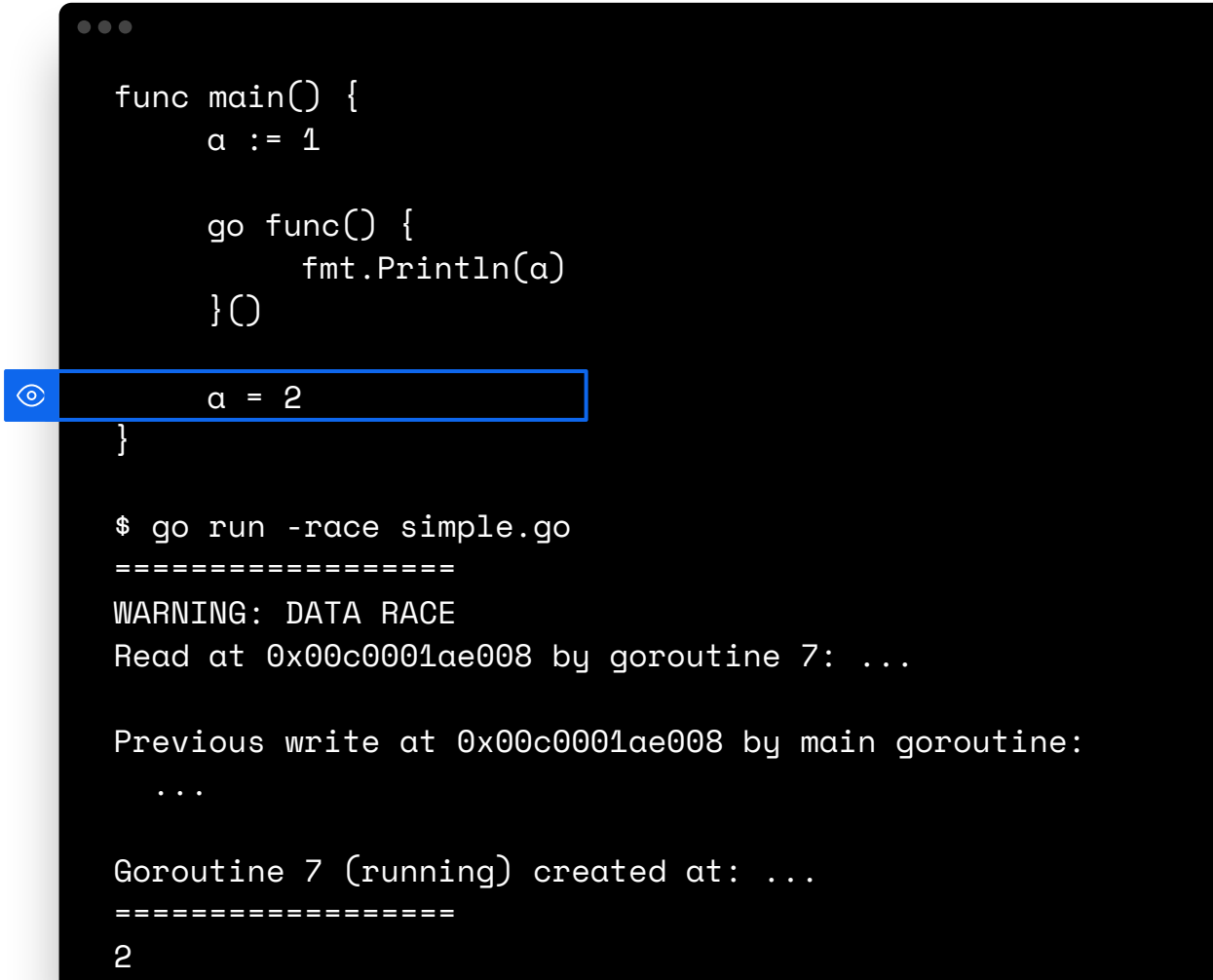
Nomad Team Lead / Principal Engineer
he/him

@schmichael[@mastodon.cloud]

# What is a Data Race?

**A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.**

– [Go's Data Race Detector Docs]



# What isn't a Data Race?

**A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.**

– [Go's Data Race Detector Docs]

```
func main() {
	a := 1

	go func() {
		fmt.Println(a, "No worries!")
	}()
	go func() {
		fmt.Println(a, "This is fine.")
	}()
	go func() {
		b := a + 1
		fmt.Println(b, "This is mine.")
	}()
}

$ go run -race simple2.go
1 This is fine.
1 No worries!
2 This is mine.
```

# Why should I care?

# The Gopher Said So

## The Go Memory Model

**Version of June 6, 2022**

Table of Contents



### Introduction

The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine.

### Advice

Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access.

To serialize access, protect the data with channel operations or other synchronization primitives such as those in the `sync` and `sync/atomic` packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

# Excerpts from the Go Memory Model

## Don’t

Programs that modify data being simultaneously accessed by multiple goroutines **must serialize such access.**

## Anger

If you must read the rest of [the Go memory model] to understand the behavior of your program, you are being too clever.

**Don't be clever.**

## the

[A Go] implementation may always **react to a data race by reporting the race and terminating the program.**

## Gopher

When it comes to programs with races, both programmers and compilers should remember the advice: **don't be clever.**

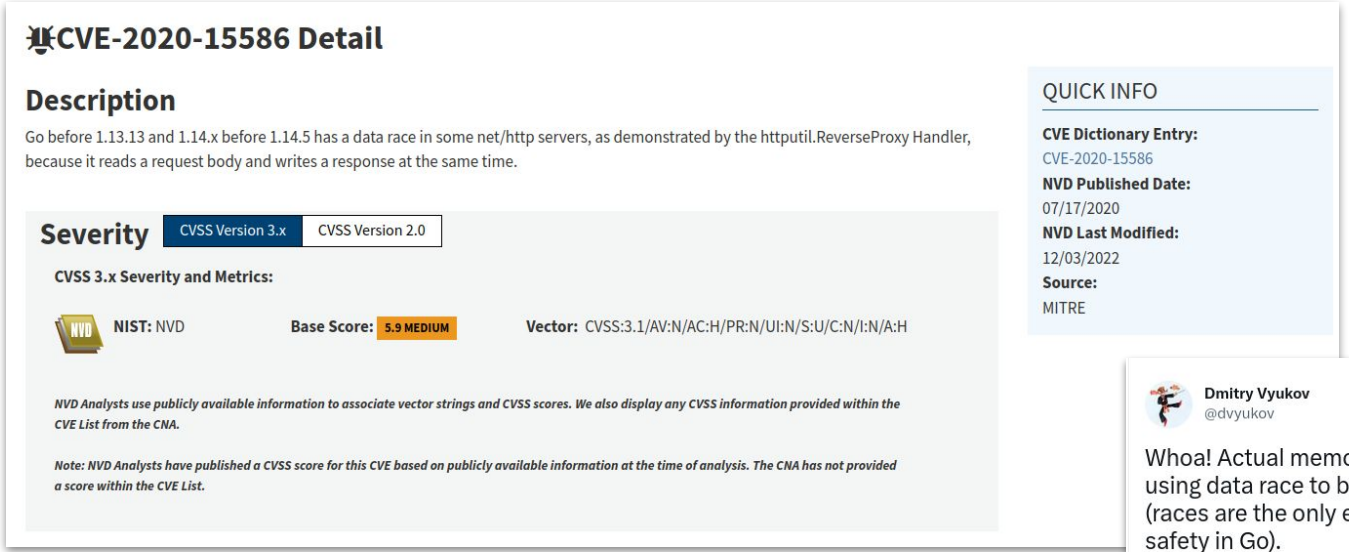
# Map Races Crash

Data races with maps will crash your program **even without enabling the race detector.**

```
func main() {
	m := map[int]int{}

	go func() {
		for i := 0; i < 1e10; i++ {
			m[i] = i
		}
	}()
	go func() {
		for i := 0; i < 1e10; i++ {
			m[i] = i
		}
	}()

	<-context.Background().Done()
}

$ go run mapparty.go
fatal error: concurrent map writes
```

# Why Should I Care? Security Edition

Data races can lead to security vulnerabilities



## CVE-2020-15586 Detail

### Description

Go before 1.13.13 and 1.14.x before 1.14.5 has a data race in some net/http servers, as demonstrated by the httputil.ReverseProxy Handler, because it reads a request body and writes a response at the same time.

### Severity

CVSS Version 3.x | CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

**NIST:** NVD **Base Score:** 5.9 MEDIUM **Vector:** CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H

*NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA.*

*Note: NVD Analysts have published a CVSS score for this CVE based on publicly available information at the time of analysis. The CNA has not provided a score within the CVE List.*

QUICK INFO

**CVE Dictionary Entry:**
CVE-2020-15586
**NVD Published Date:**
07/17/2020
**NVD Last Modified:**
12/03/2022
**Source:**
MITRE

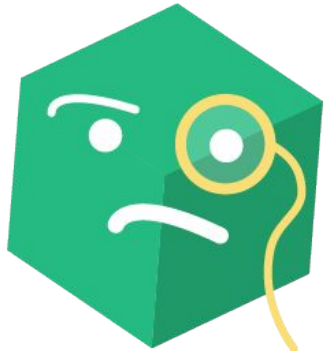
**Dmitry Vyukov**
@dvyukov

Whoa! Actual memory safety exploit for #golang using data race to break safety of interface object (races are the only escape hatch for memory/type safety in Go).
You are testing your Go code with the race detector (-race), right?

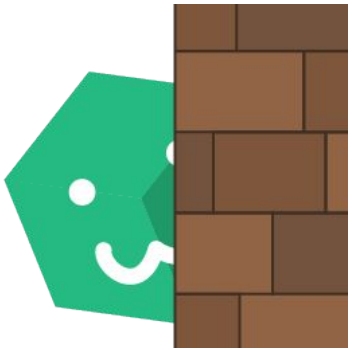
> **Netanel Ben Simon** @NetanelBenSimon · Nov 6, 2019
> Here is my writeup for gomium from #GoogleCTF 2019 finals.
> github.com/netanel01/ctf-...

12:32 AM · Nov 8, 2019

# Writing data race free programs demonstrates an understanding of the program's execution.



# How do I find data races?



# Go's Race Detector

The -race flag enables the race detector on any Go command that creates or runs a binary:

- go install
- go build
- go test
- go get

```
func main() {
    a := 1

    go func() {
        fmt.Println(a)
    }()

    a = 2
}
```

```
$ go run -race simple.go
==================
WARNING: DATA RACE
Read at 0x00c0001ae008 by goroutine 7: ...

Previous write at 0x00c0001ae008 by main goroutine:
  ...

Goroutine 7 (running) created at: ...
==================
2
```

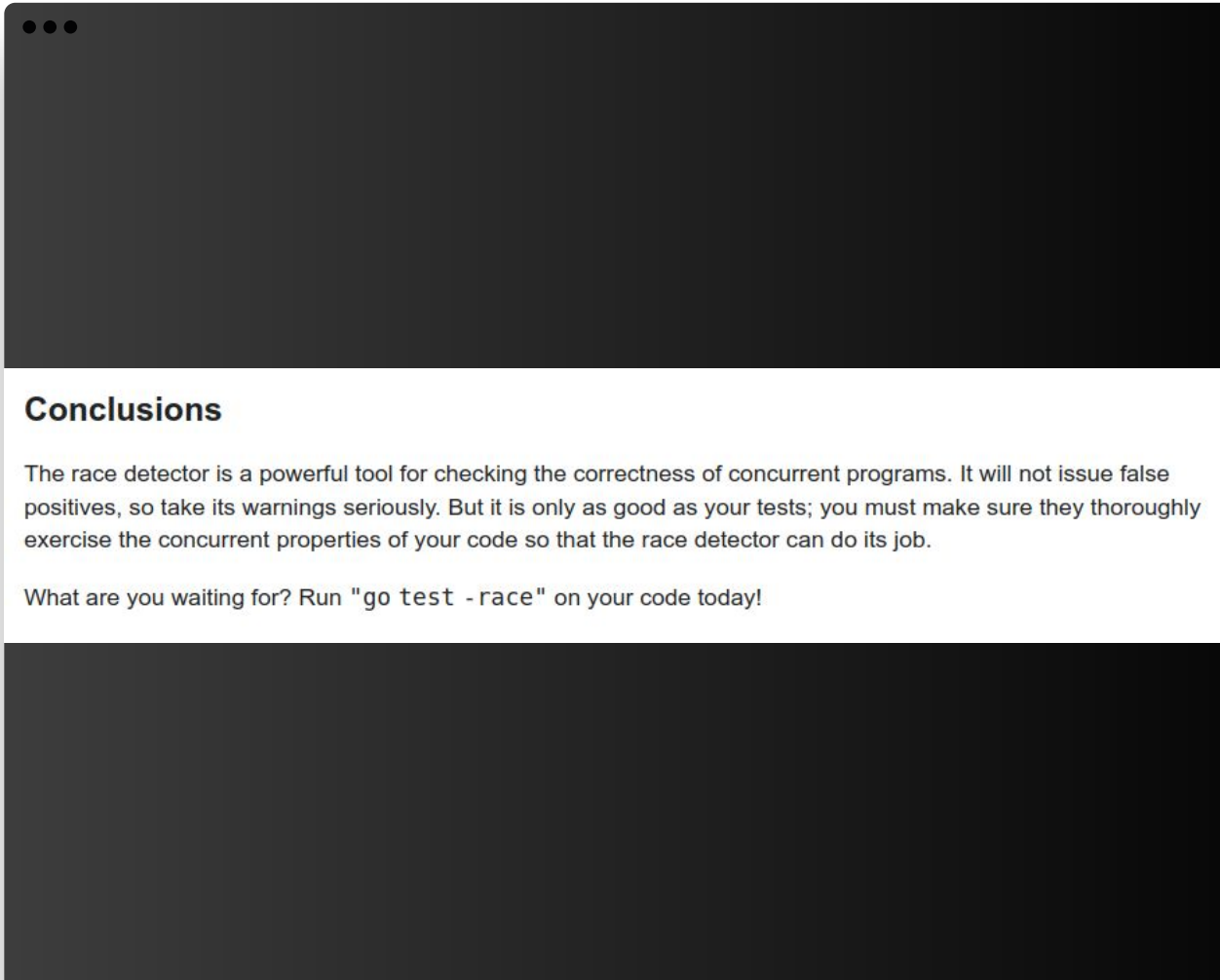
# Zero false positives

# 0 Lies Detected

The race detector may not detect all races, but it will never lie to you about a race.

When it reports a data race, there's a data race.

https://go.dev/blog/race-detector



## Conclusions

The race detector is a powerful tool for checking the correctness of concurrent programs. It will not issue false positives, so take its warnings seriously. But it is only as good as your tests; you must make sure they thoroughly exercise the concurrent properties of your code so that the race detector can do its job.

What are you waiting for? Run `"go test -race"` on your code today!

# Anatomy of a detected race

1. Delimited by ======
2. Special exit code (66)
3. Where the **read** happened
4. Where the **write** happened
5. Where non-main goroutines were started

```
$ go run -race simple/simple.go > /dev/null
==================
WARNING: DATA RACE
Read at 0x00c00001a0b8 by goroutine 7:
  main.main.func1()
      /.../simple.go:9 +0x3a

Previous write at 0x00c00001a0b8 by main goroutine:
  main.main()
      /.../simple.go:12 +0xb8

Goroutine 7 (running) created at:
  main.main()
      /.../simple.go:8 +0xae
==================
Found 1 data race(s)
exit status 66
```

# How do I fix data races?

# Atomics

Atomics can always fix a data race, but it doesn't mean your program makes any sense.

Both reads and writes must use atomics to adhere to the Go memory model.

```
func main() {
     var a int64 = 1

     go func() {
          fmt.Println(atomic.LoadInt64(&a))
     }()

     atomic.AddInt64(&a, 1)
}

$ go run -race simple/simple.go
2
```

# Race Conditions

## Aka “bad interleaving”

A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations.

In this case the program could still print 1 or 2 depending on how the goroutines are scheduled.

```
func main() {
     var a int64 = 1

     go func() {
          fmt.Println(atomic.LoadInt64(&a))
     }()

     atomic.AddInt64(&a, 1)
}

$ go run -race simple/simple.go
2
```

# Start Making Sense

To fix both the data race and race condition, we need to make sense of this program.

We can make some sense out of it (and fix races) by using a channel.

```
func main() {
     a := make(chan int)

     go func() {
          fmt.Println(<-a)
     }()

     a <- 2
}


$ go run -race simple/simple.go
2
```

# Mutexes and Critical Sections

Mutexes (aka locks) are a common tool for serializing access to critical sections.

This can prevent both data races and race conditions.

“Critical section” is a fancy way of saying “code that must execute atomically on shared variables.”

Critical sections can be short (getters) or long.

```
func (c *Client) GetConfig() *config.Config {
	c.configLock.Lock()
	defer c.configLock.Unlock()
	return c.config
}

// Shutdown is used to tear down the client
func (c *Client) Shutdown() error {
	c.shutdownLock.Lock()
	defer c.shutdownLock.Unlock()

	if c.shutdown {
		c.logger.Info("already shutdown")
		return nil
	}
	c.logger.Info("shutting down")

	// Stop renewing tokens and secrets
	if c.vaultClient != nil {
		c.vaultClient.Stop()
	}
```

# Locks can get tricky…

## vault: fix deadlock in SetConfig #6082

Edit | Code

Merged **schmichael** merged 2 commits into `master` from `b-vault-deadlock` on Aug 6, 2019

Conversation 1 | Commits 2 | Checks 0 | Files changed 2 | +46 −4

**schmichael** commented on Aug 6, 2019 — Member

This seems to be the minimum viable patch for fixing a deadlock between establishConnection and SetConfig.

SetConfig calls tomb.Kill+tomb.Wait while holding v.lock.
establishConnection needs to acquire v.lock to exit but SetConfig is holding v.lock until tomb.Wait exits. tomb.Wait can't exit until establishConnect does!

```
SetConfig -> tomb.Wait
   ^             |
   |             v
v.lock <- establishConnection
```

vault: fix deadlock in SetConfig ··· 7e08a2f

**Reviewers**
endocrimes ✓

**Assignees**
No one—assign yourself

**Labels**
None yet

**Projects**
None yet

**Milestone**
No milestone

# Create Helpers to Manage Critical Sections

1. Lock
2. Copy
3. Mutate
4. Overwrite

Is a handy pattern for backfilling mutability on big balls of state (when you have to).

Access c.config requires acquiring the configLock!

```
func (c *Client) UpdateConfig(cb func(*Config)) *Config {
     c.configLock.Lock()
     defer c.configLock.Unlock()

     newConfig := c.config.Copy()

     cb(newConfig)

     c.config = newConfig

     return newConfig
}

func (c *Client) someFunc() {

     c.UpdateConfig(func(c *Config) {
          c.Node.Status = structs.NodeStatusReady
     })

}
```

# Channels and Select are your Friend!

The Go Blog

## Share Memory By Communicating

*Andrew Gerrand*
*13 July 2010*

Traditional threading models (commonly used when writing Java, C++, and Python programs, for example) require the programmer to communicate between threads using shared memory. Typically, shared data structures are protected by locks, and threads will contend over those locks to access the data. In some cases, this is made easier by the use of thread-safe data structures such as Python's Queue.

Go's concurrency primitives - goroutines and channels - provide an elegant and distinct means of structuring concurrent software. (These concepts have an interesting history that begins with C. A. R. Hoare's Communicating Sequential Processes.) Instead of explicitly using locks to mediate access to shared data, Go encourages the use of channels to pass references to data between goroutines. This approach ensures that only one goroutine has access to the data at a given time. The concept is summarized in the document Effective Go (a must-read for any Go programmer):

*Do not communicate by sharing memory; instead, share memory by communicating.*

Consider a program that polls a list of URLs. In a traditional threading environment, one might structure its data like so:

```
type Resource struct {
    url        string
    polling    bool
    lastPolled int64
}

type Resources struct {
    data []*Resource
    lock *sync.Mutex
}
```

# Channels and Select are Great

They often imbue your code with more meaning than atomics or mutexes.

```
func (c *Client) registerAndHeartbeat() {
     // Start watching changes for node changes
     go c.watchNodeUpdates()

     // Start watching for emitting node events
     go c.watchNodeEvents()

     heartbeat :=
time.After(helper.RandomStagger(heartbeatStagger))

     for {
          select {
          case <-c.rpcRetryWatcher():
          case <-heartbeat:
          case <-c.shutdownCh:
               return
          }
          if err := c.updateNodeStatus(); err != nil {
               // ...
```

# …but there are worse fates than locks.

## consul: fix deadlock in check-based restarts #5975

Edit | Code

Merged **schmichael** merged 3 commits into `master` from `b-check-watcher-deadlock` on Jul 18, 2019

Conversation 10 | Commits 3 | Checks 0 | Files changed 2 | +122 −6

**schmichael** commented on Jul 17, 2019 — Member

Fixes #5395
Alternative to #5957

Make task restarting asynchronous when handling check-based restarts. This matches the pre-0.9 behavior where TaskRunner.Restart was an asynchronous signal. The check-based restarting code was not designed to handle blocking in TaskRunner.Restart. 0.9 made it reentrant and could easily overwhelm the buffered update chan and deadlock.

Many thanks to **@byronwolfman** for his excellent debugging, PR, and reproducer!

I created this alternative as changing the functionality of TaskRunner.Restart has a much larger impact. This approach reverts to old known-good behavior and minimizes the number of places changes are made.

`consul: fix deadlock in check-based restarts` … `9c418c2`

**Reviewers**
notnoop ✓
preetapan ✓

**Assignees**
No one—assign yourself

**Labels**
None yet

**Projects**
None yet

**Milestone**
No milestone

**Development**

# How could channels do that?

- State changes were sent to a goroutine via a channel
- The goroutine calls some other methods while processing changes
- If those other methods emitted a state change to the channel...
- ...deadlock because the receiver is also the sender!

Buffered channel?

- It was...
- ...but that only made it harder to hit because the buffer had to fill with events first.

# The What, Why, and How of Go Data Races

## Data Races are Bugs

There are no benign data races.

Treat them like bugs and fix them.

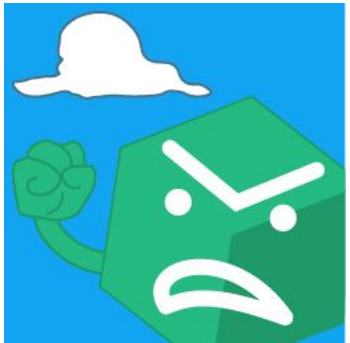
It can be a journey (Nomad's test suite is **not** data race free).

## Race Detector

Use it in tests.

Use it when developing.

Trust it.



## Understand your Code

Programs free from data races and race conditions demonstrate a high level of understanding by their developers.

Solving races helps you understand how your program is composed and executes.

# Thank you

@schmichael[@mastodon.cloud]
mschurter@hashicorp.com

**All Nomad icons thanks to Michael Lange**
**https://github.com/DingoEatingFuzz/nomad-emoji**