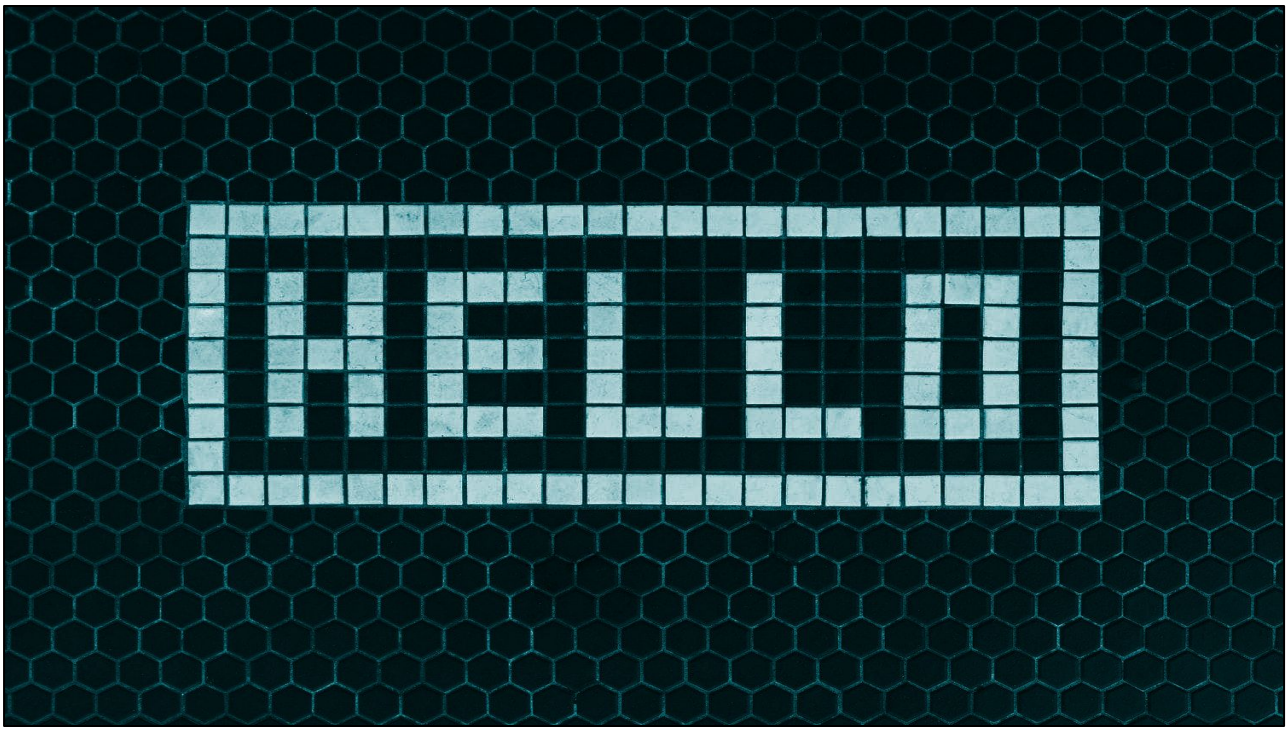
Go: Security Considerations

SUMMERCON 2021

Brandon Edwards @drraid
Pete Markowsky @PeteMarkowsky

Welcome to our talk, Some Security Considerations on Go



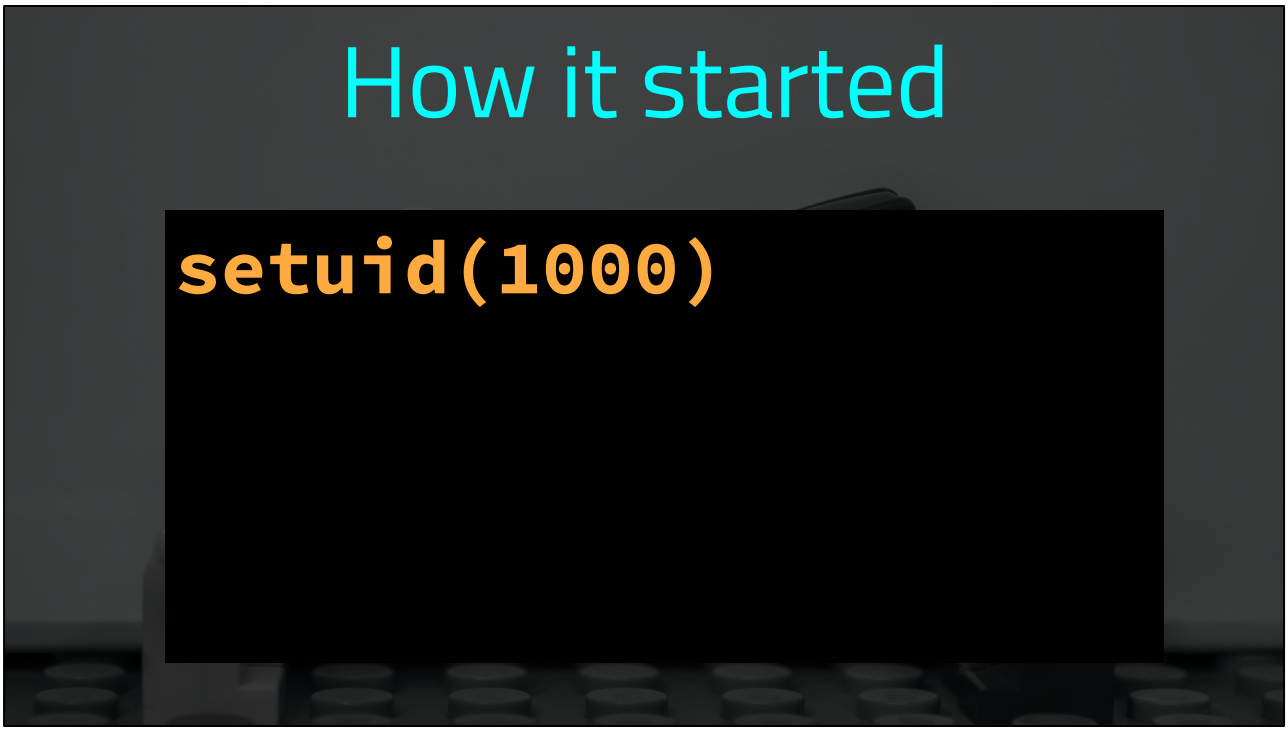
Brandon: Thank you for joining us here today. My name is Brandon Edwards, also known as “drraid”, and I am honored and excited to be speaking at Summercon.

Pete: I’ve worked with Brandon and I’m mostly here for the drinks

What is this talk about? Well, first, as a preface, after like eleventy million drinks at a friend's birthday party, during the first social event I attended after the pandemic, I promised John Terrill I would give a talk at summercon, but then promptly forgot that promise. I am a little less organized than the level to which I would normally aspire being, and I would like to give you my apologies in advance.

I wanted to do this talk because Go, Rust, and other “safer” languages first worried me that there was an existential threat to the busticati, that first these languages would eliminate memory safety bugs, and with all the fancy tooling and compiler checks they would eliminate most logic bugs, hacking dies forever, and I’d have to shift careers to some multilevel marketing scheme selling shifty weightloss products to my friends and family on Facebook.

Then I remembered that the root cause of security issues is that computers were a mistake, logic bugs are here forever, and human ineptitude wins every time. And, not only does Go not eliminate many bug classes, it actually provides novel ways that people might unexpectedly introduce bugs. Though before I go any further let me stop here to say I actually love Go, and appreciate what the whole Go team has done and built, it’s truly incredible technology.



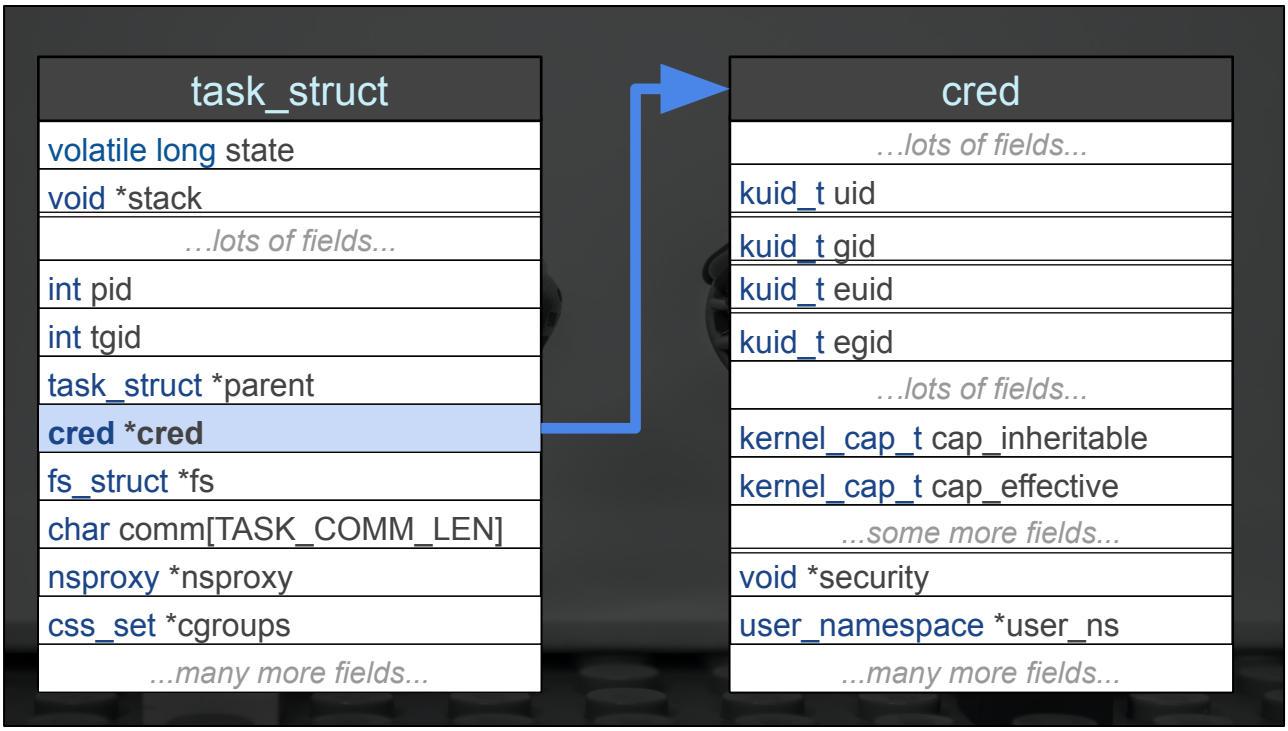
There are actually a couple more reasons I wanted to give this talk. Part of the inspiration began back in 2016 when I first experienced an oddity in looking at how one might drop privileges in go. Like say I wanted to drop privileges before accessing some files or doing other sensitive stuff on behalf of a user request.

How it started

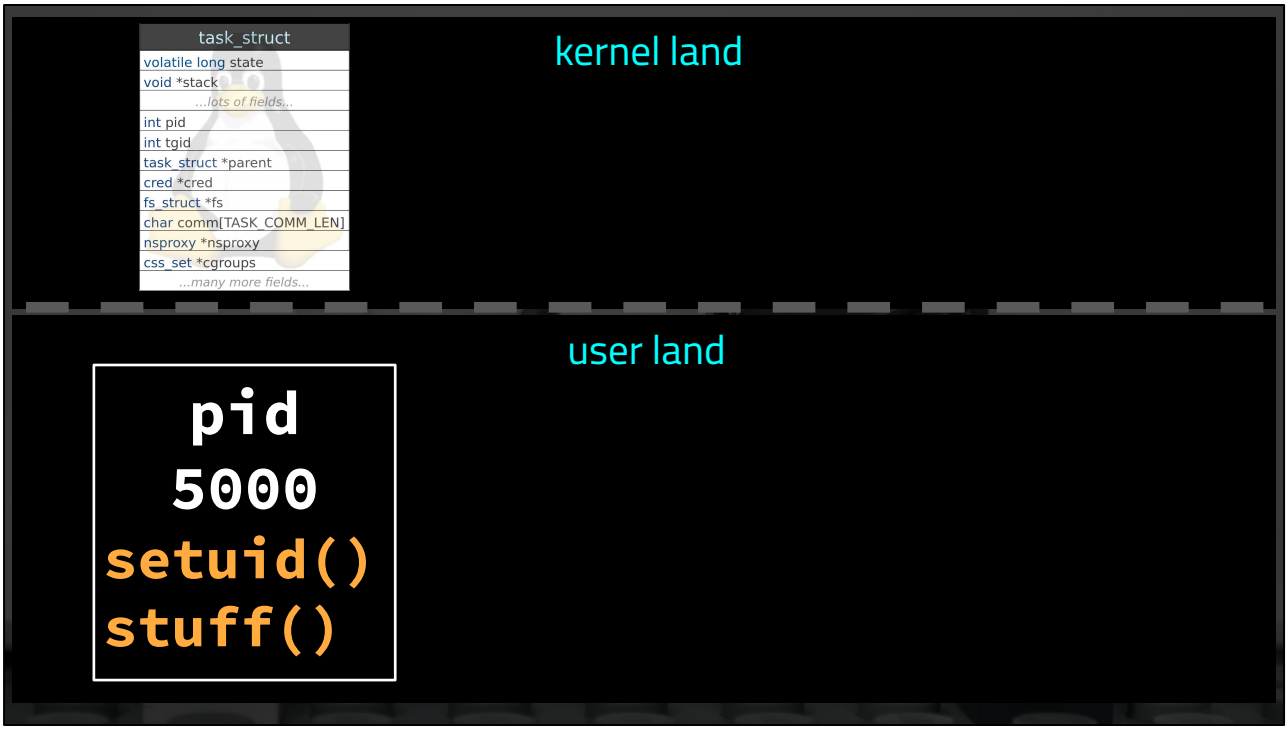
```
setuid(1000)

sensitive_stuff()
```

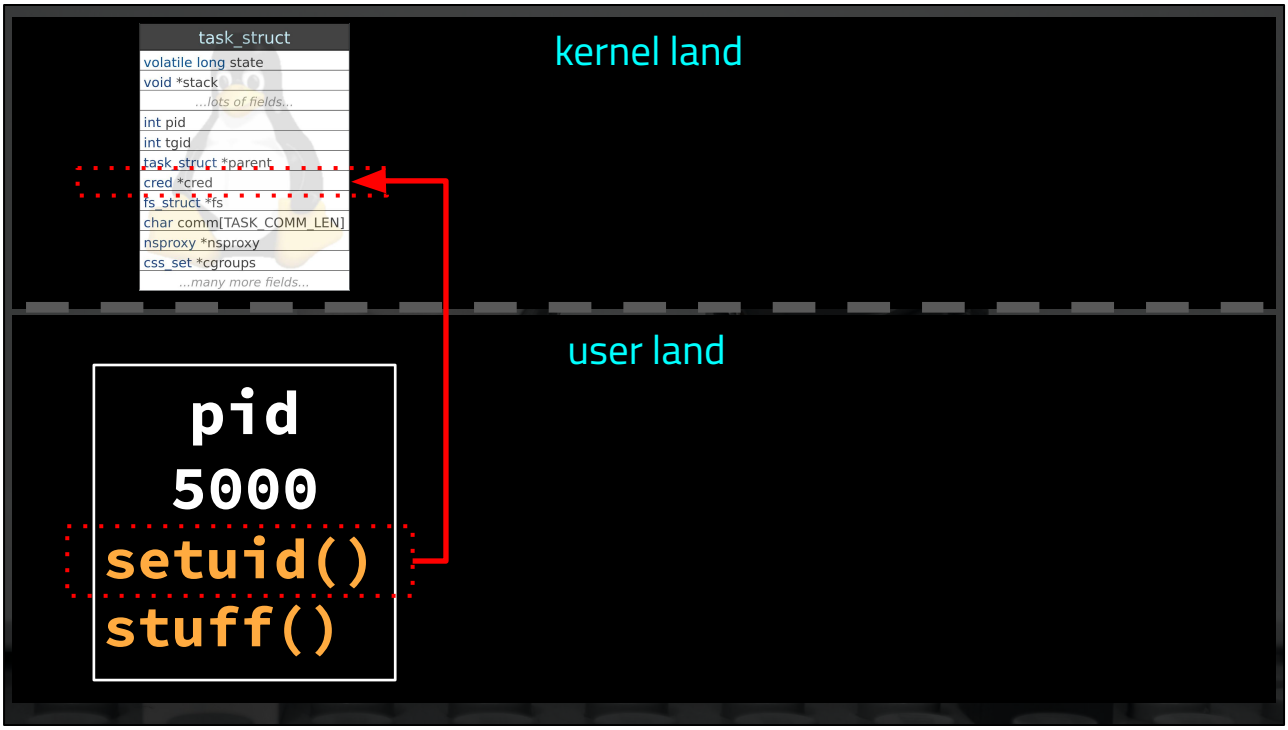
Like say I wanted to drop privileges before accessing some files or doing other sensitive stuff on behalf of a user request.

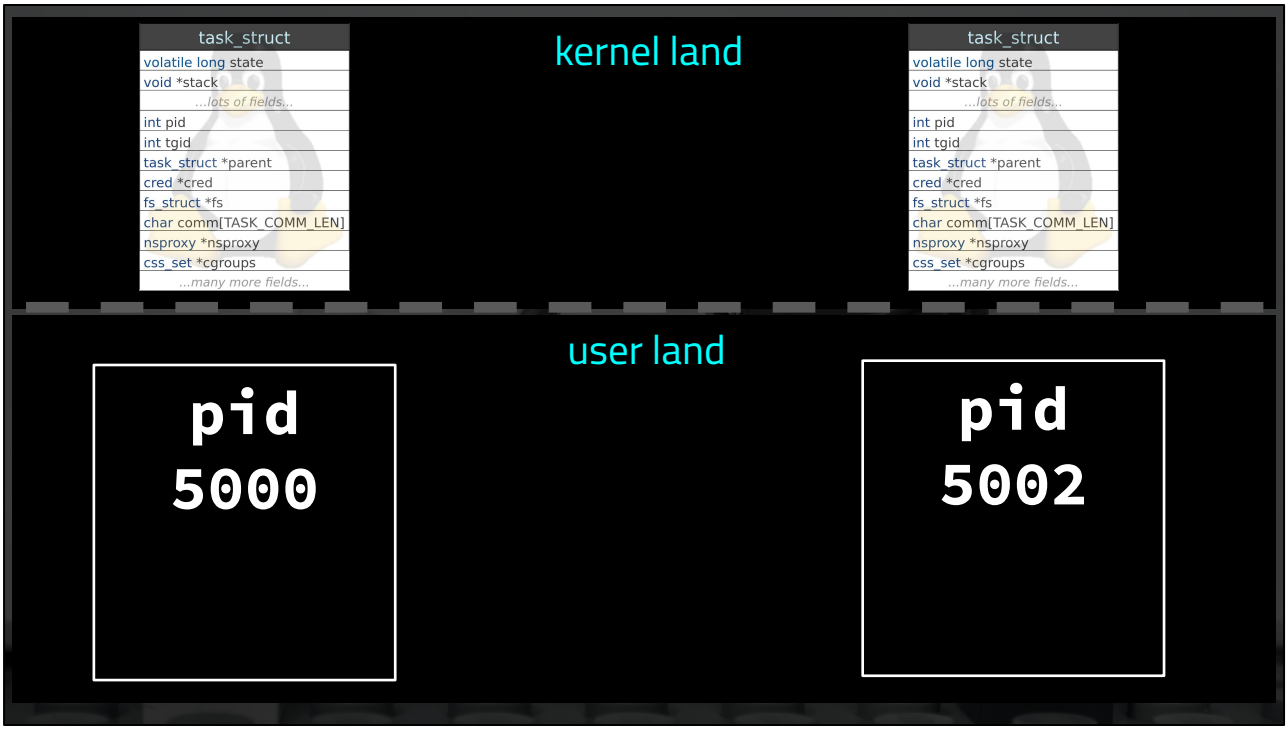


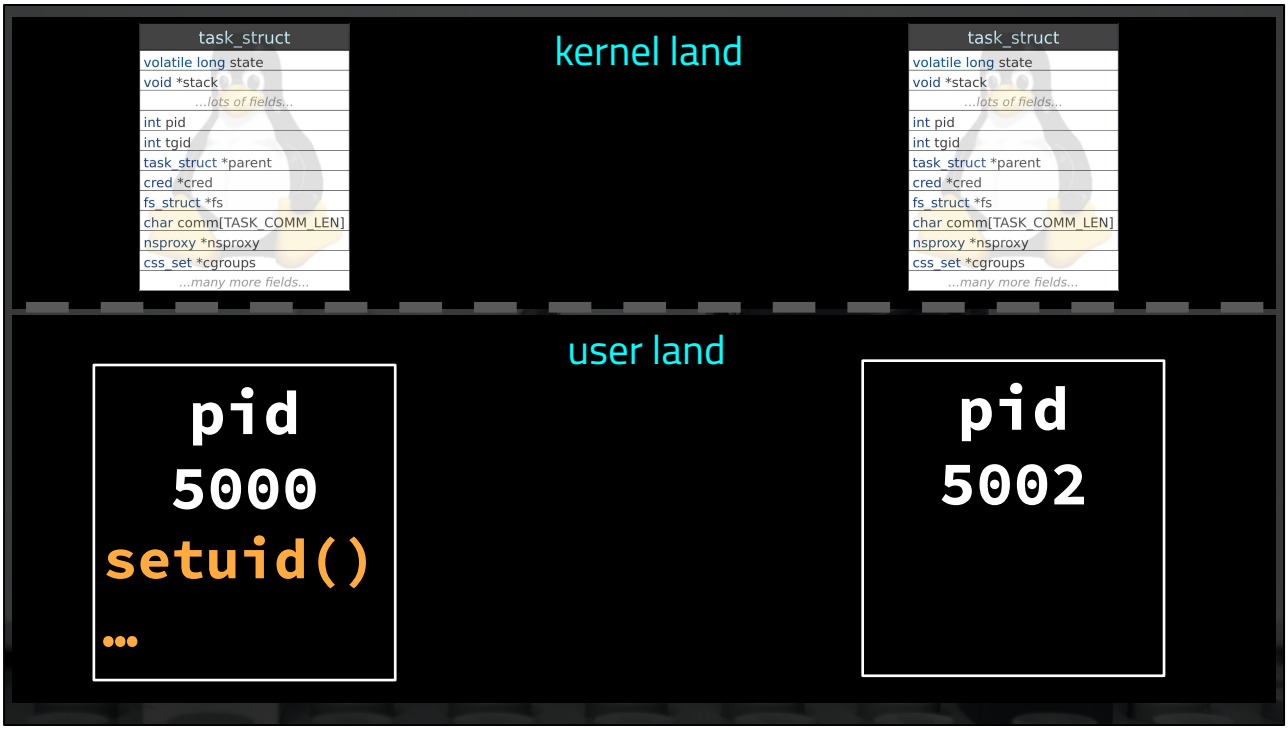
Credentials are stored in a structure called cred in the kernel, where each task points to a cred struct

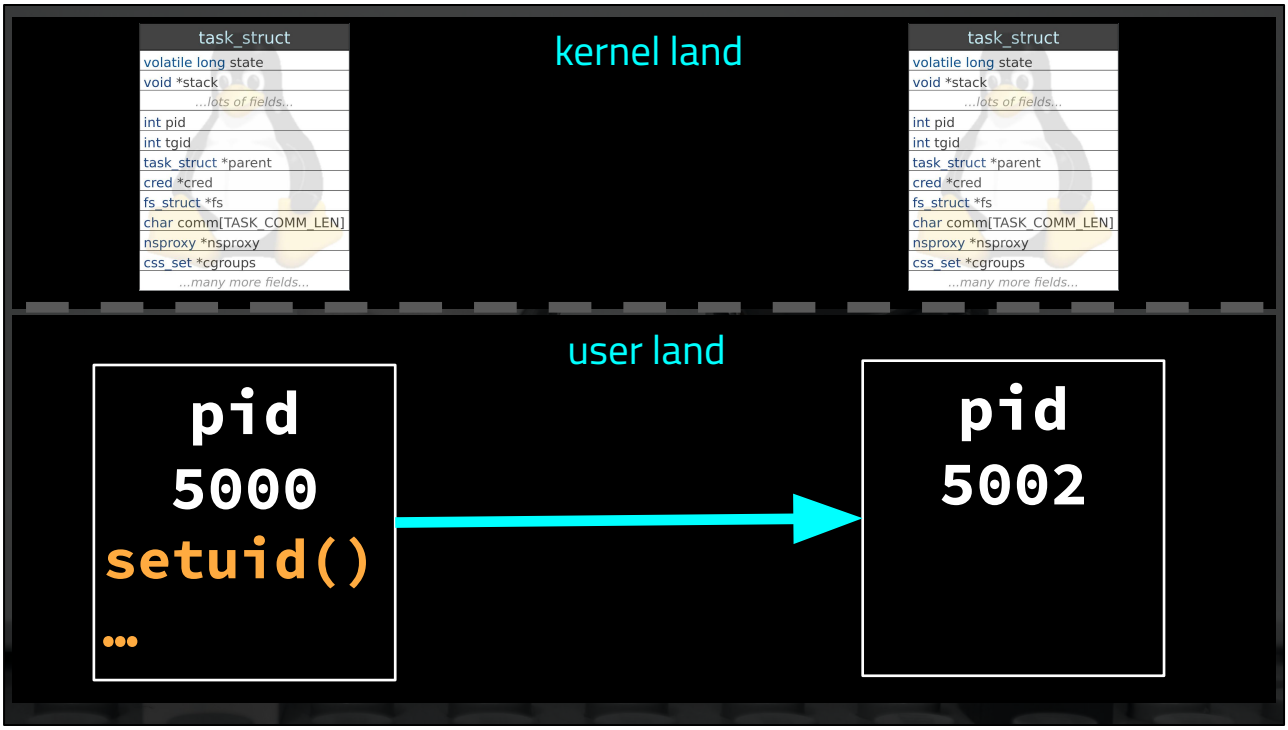

task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
user land
pid
5000
setuid()
stuff()

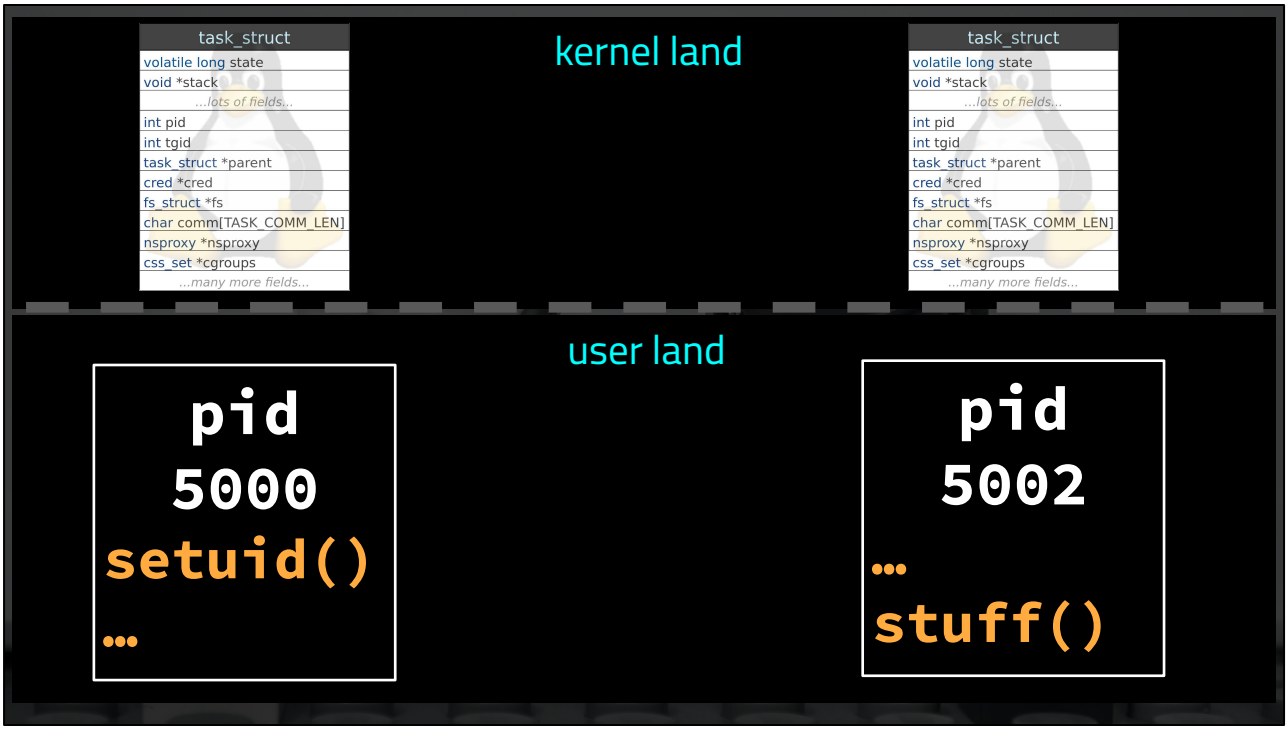
So you think

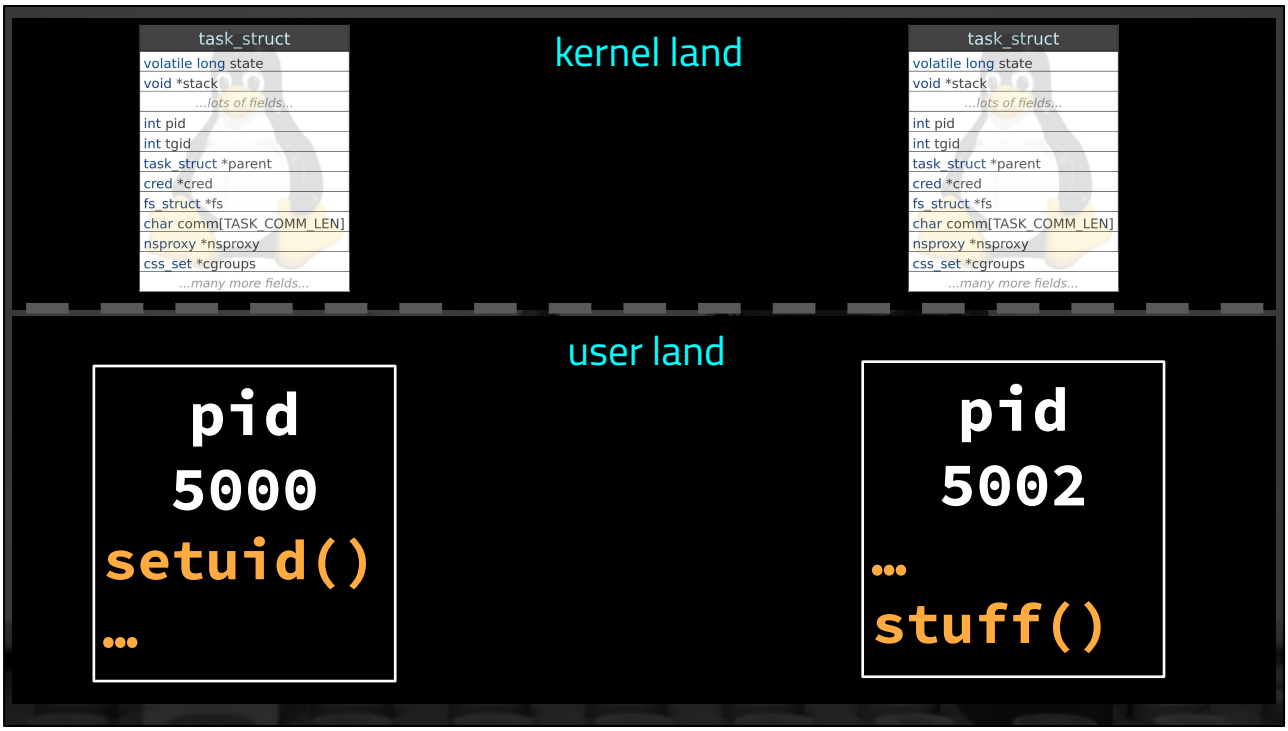

task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
user land
pid
5000
setuid()
stuff()


task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
user land
pid
5000
pid
5002

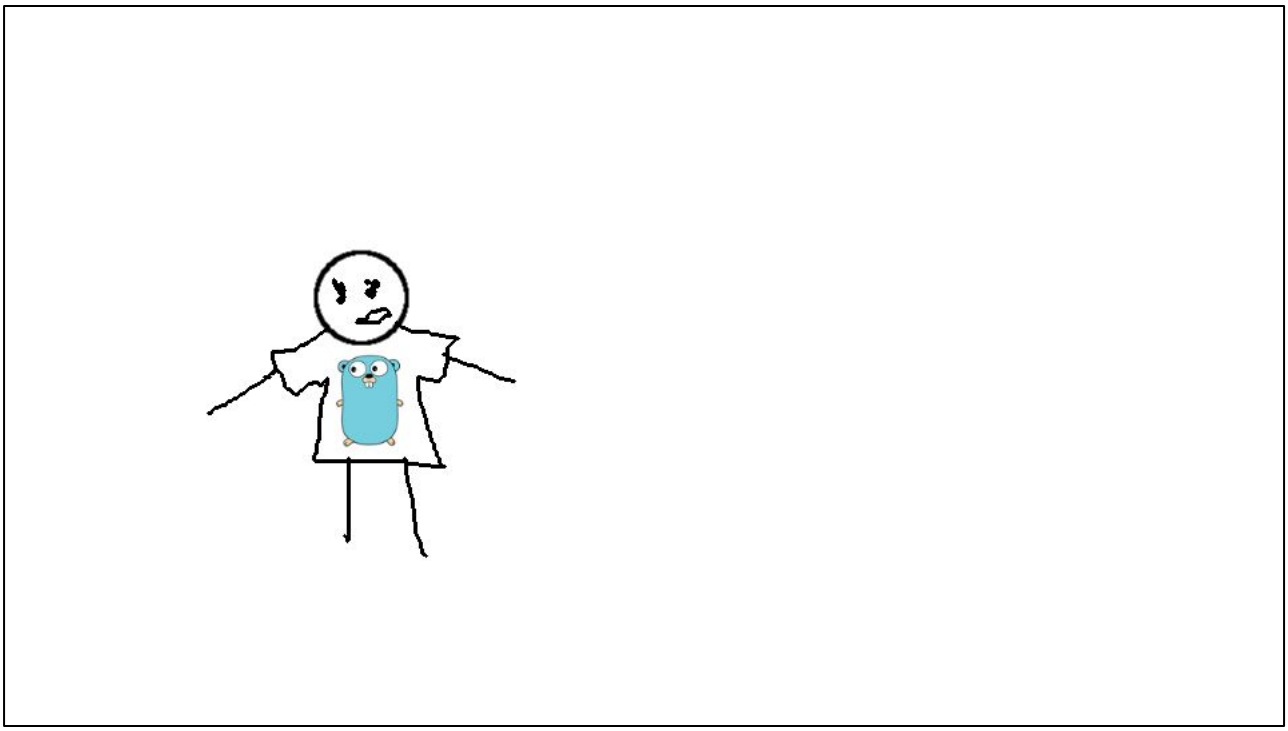

task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
user land
pid
5000
setuid()
...
pid
5002


task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
user land
pid
5000
setuid()
...
pid
5002


task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
kernel land
task_struct
volatile long state
void *stack
...lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
user land
pid
5000
setuid()
…
pid
5002
…
stuff()

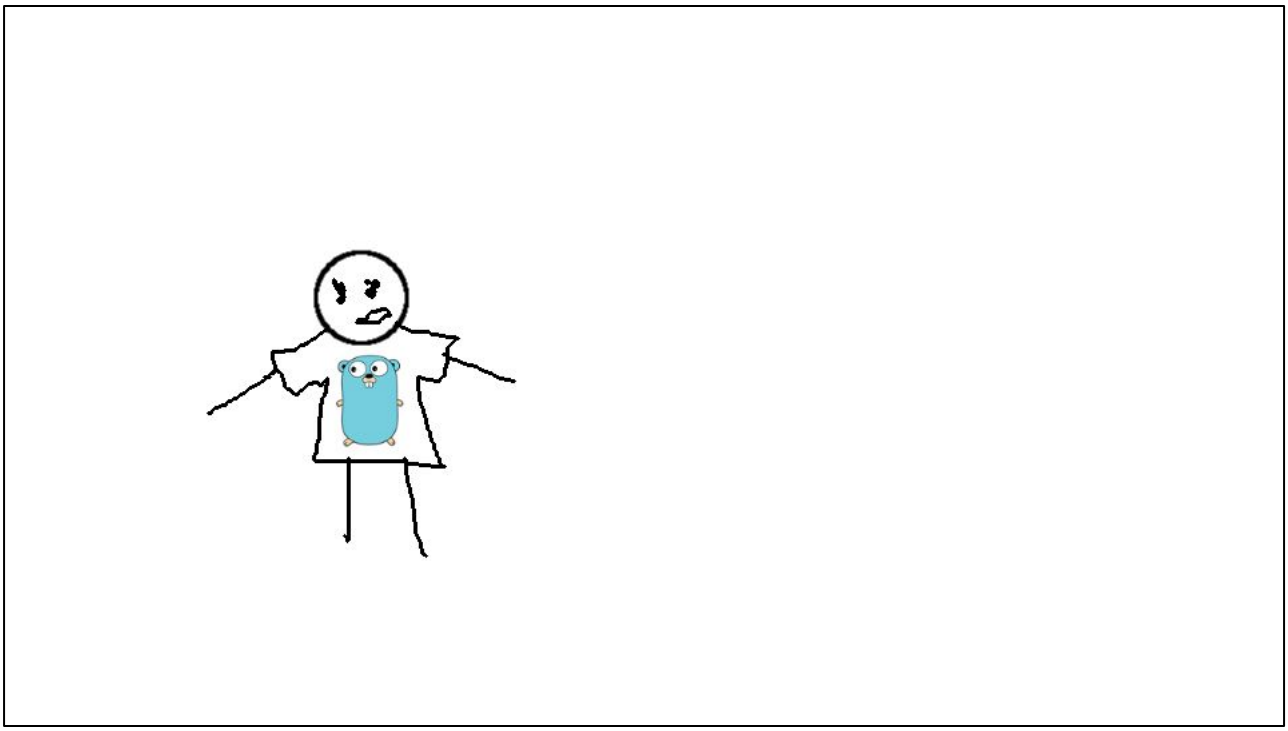


It's worth noting that this behavior in C prorgrams is handled by libc suspending all threads in the process, and then calling setuid on their behalf. So it wasn't a lack of intuition that calling setuid in one task wouldn't change the privs in an adjacent task in the same thread group. The surprise for me was the scheduler: so like, if you did a raw syscall in C to set the UID, and bypassed the convenience libc gives you in applying it to all the other threads, you would still have confidence that the execution of your thread moving forward would be with the new UID, so any of the following code executed with that newly set user's privileges. The ability to be scheduled on to another thread without knowing it is what threw me for a loop



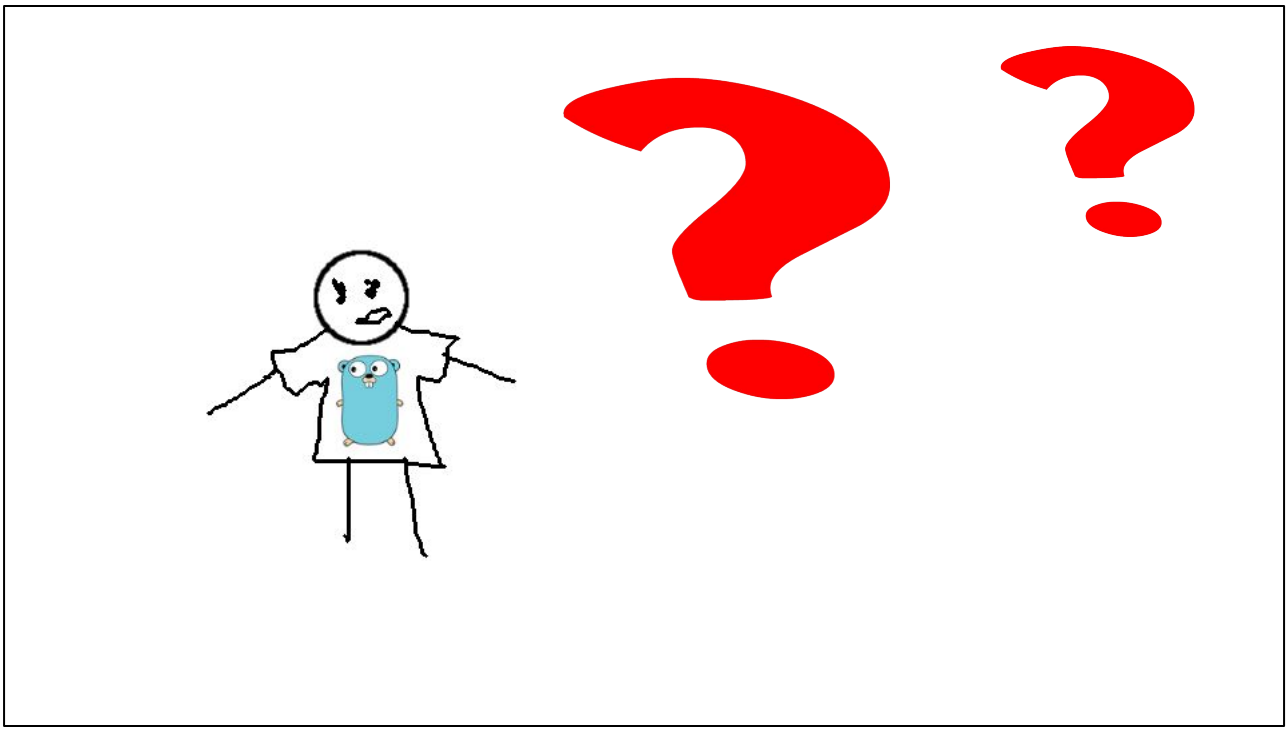
The last reason I wanted to do this talk was because language zealots are sooo annoying. How can you tell if someone writes in Go, or Rust for that matter?

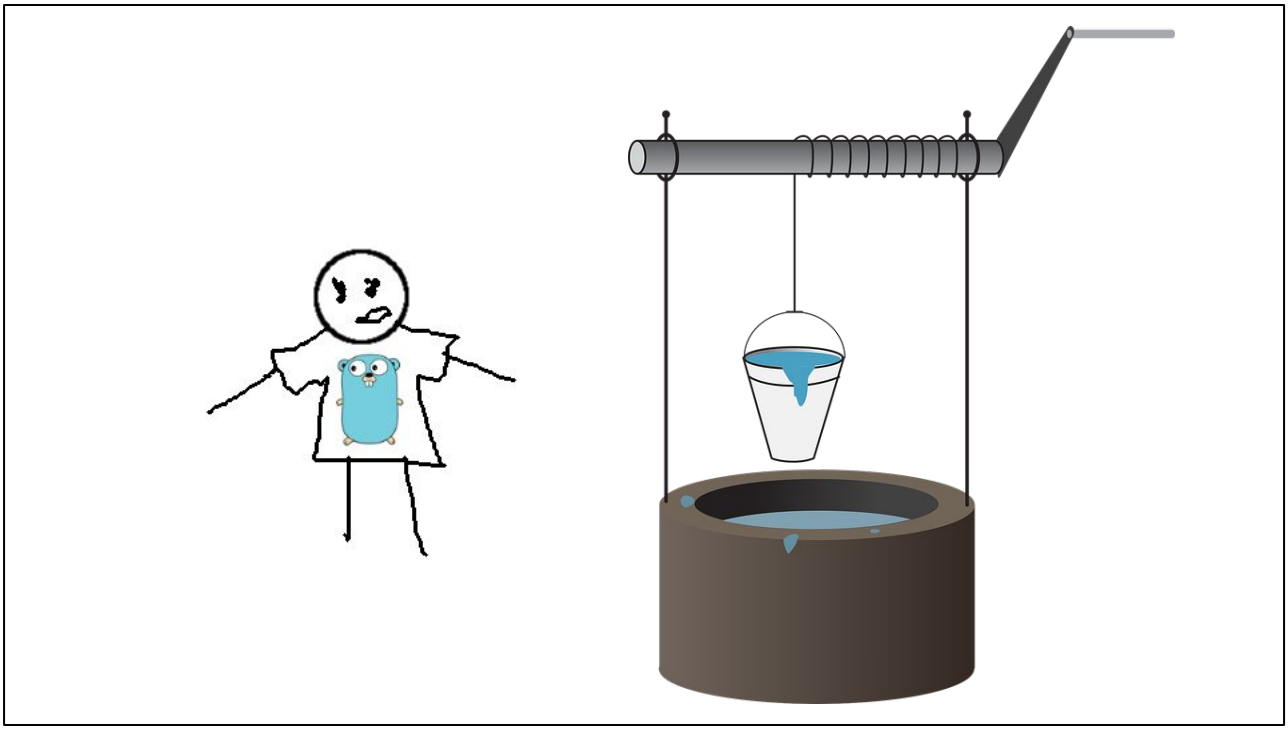
You don’t have to, they tell you, in literally every interaction you have.



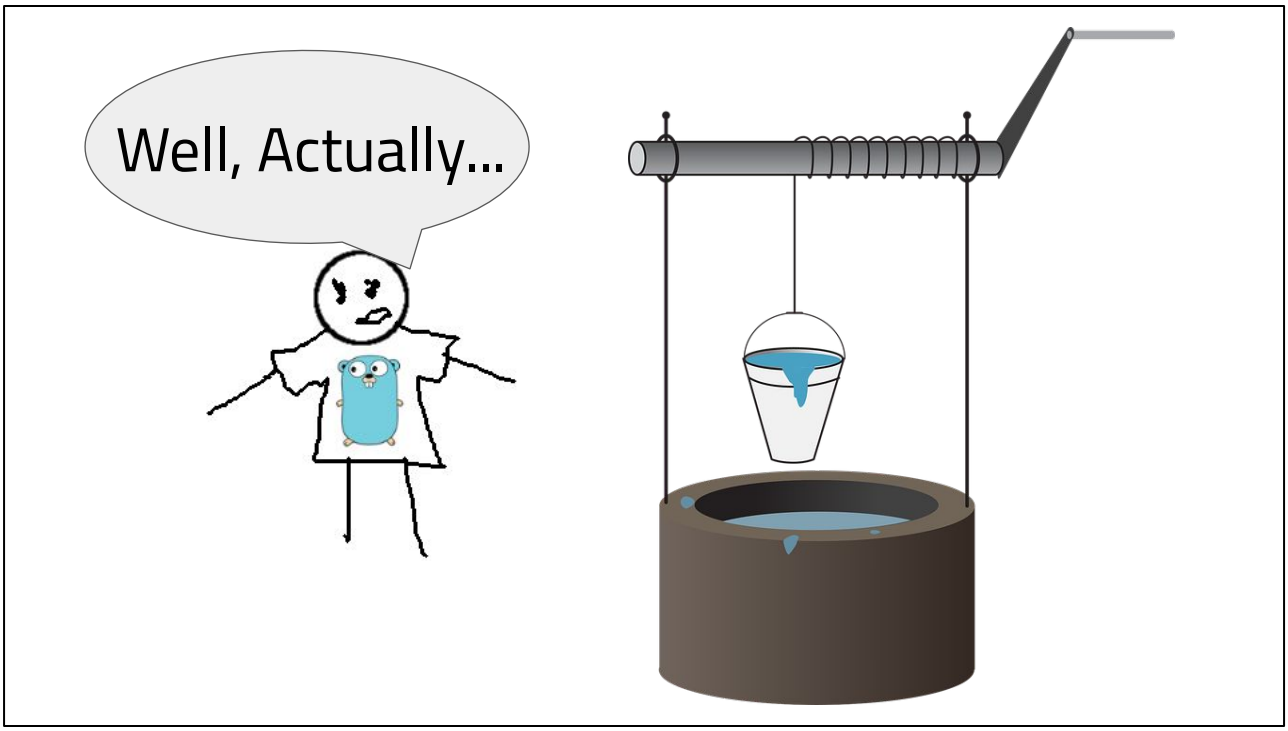
You know where this pretentious little shit gets his water supply?

Water Supply?



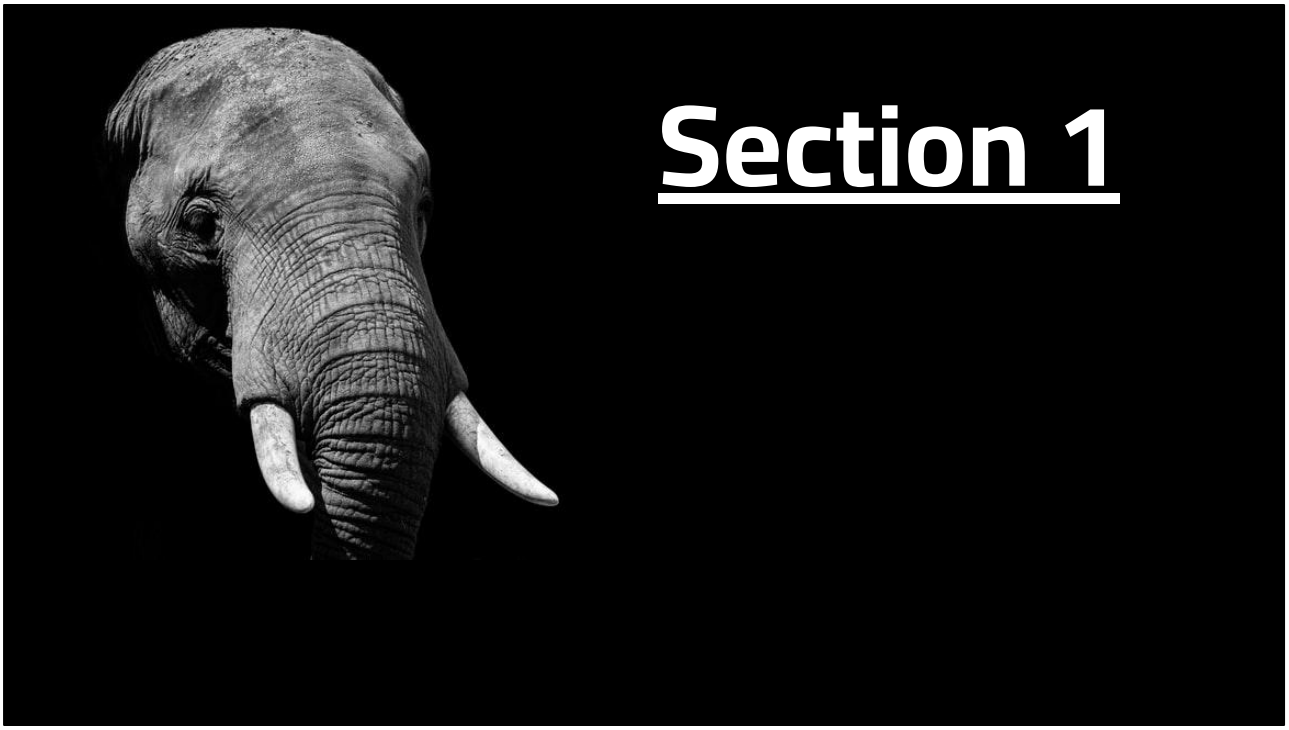


That’s right

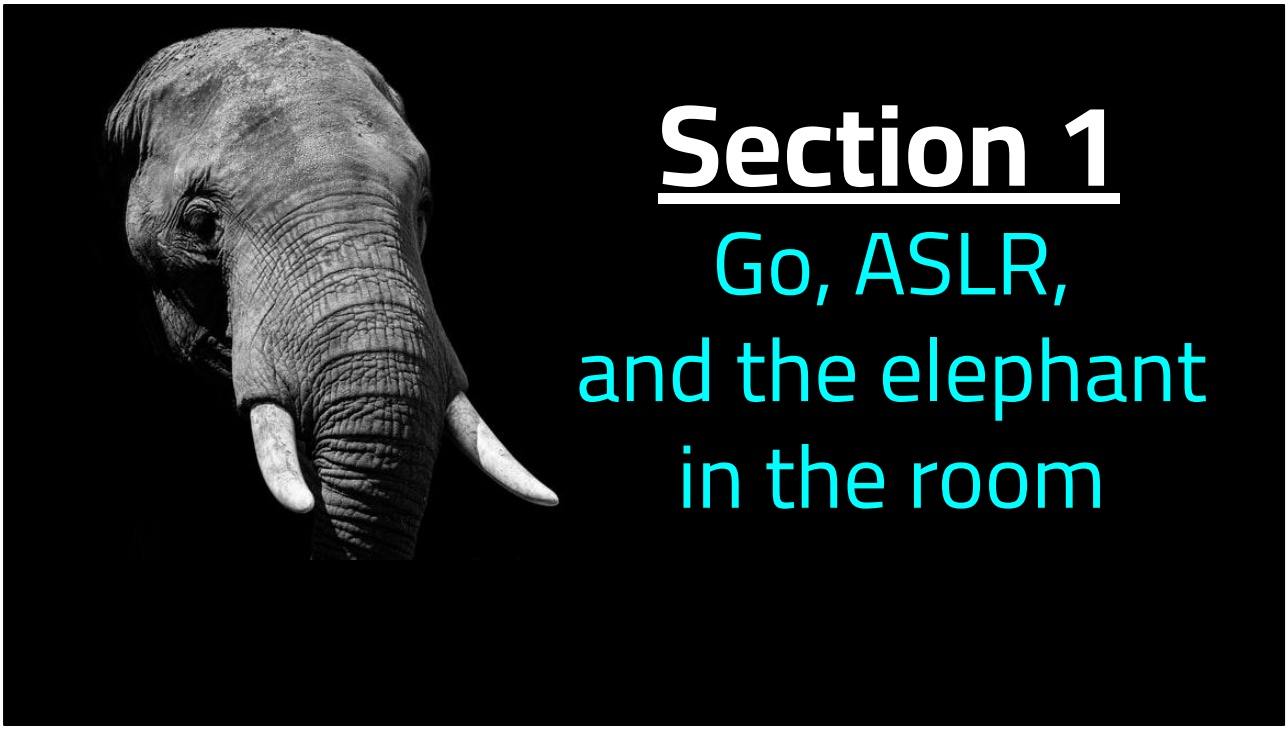


He gets his water from a Well, Actually. This cult of annoyance inspired me to give this talk. So being that Summercon is in summer time, and summer time is hot, we're going to use this talk on Go to shave this neckbeard down to a bro-tee, or maybe an elon muskashe

So here we go. In the style of Morty's Mindblowers, this isn't some freeform anthology, but instead just a handful of observations on Go, some of which are obvious, but maybe there are some ideas that you've never seen before-- Oooo spooky. It also has ideas and speculation of stuff I haven't had time to fully flesh out, hopefully people find it inspiring or insightful.



Section one

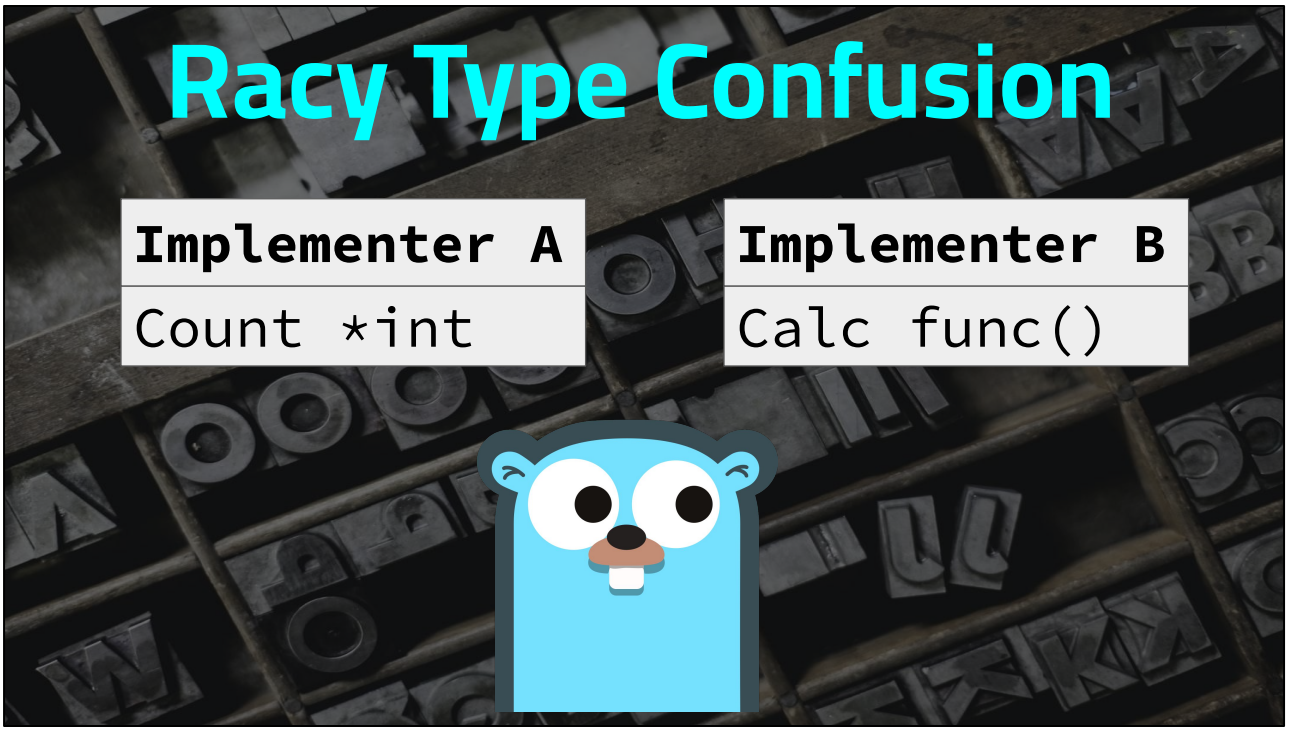


Go, ASLR, and the elephant in the room

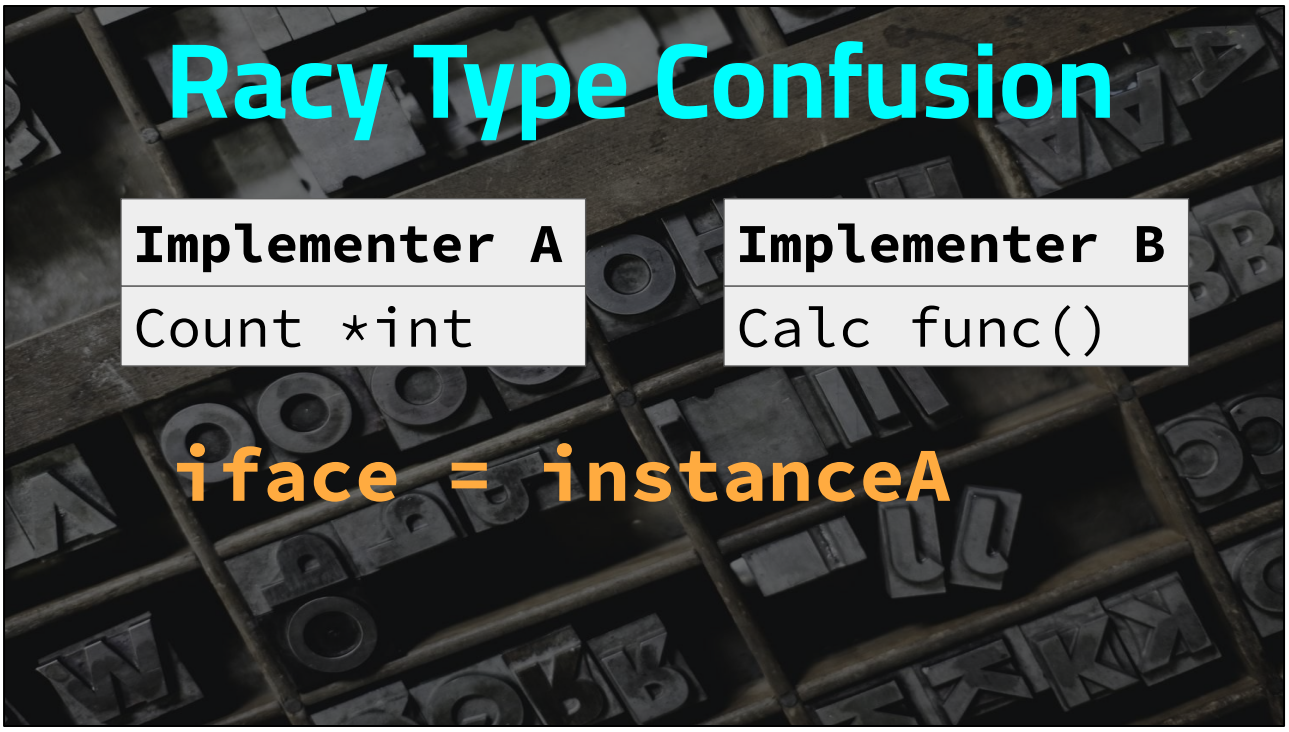
Go is a type-safe-ish language, and so there is an assumption that ASLR is not only unnecessary, but that it introduces unnecessary complexity, and might frustrate debugging. Which is hilarious because Go already does a great job of being unnecessarily complex and incredibly frustrating to debug.

The most obvious observation on the lack of ASLR is that using the unsafe library can introduce traditional memory safety bugs, because it allows the programmer to side step all the guarantees made by the language, and suddenly textbook overflows are available to the programmer. Plenty of CTF challenges showcase this, so I won’t belabor it.

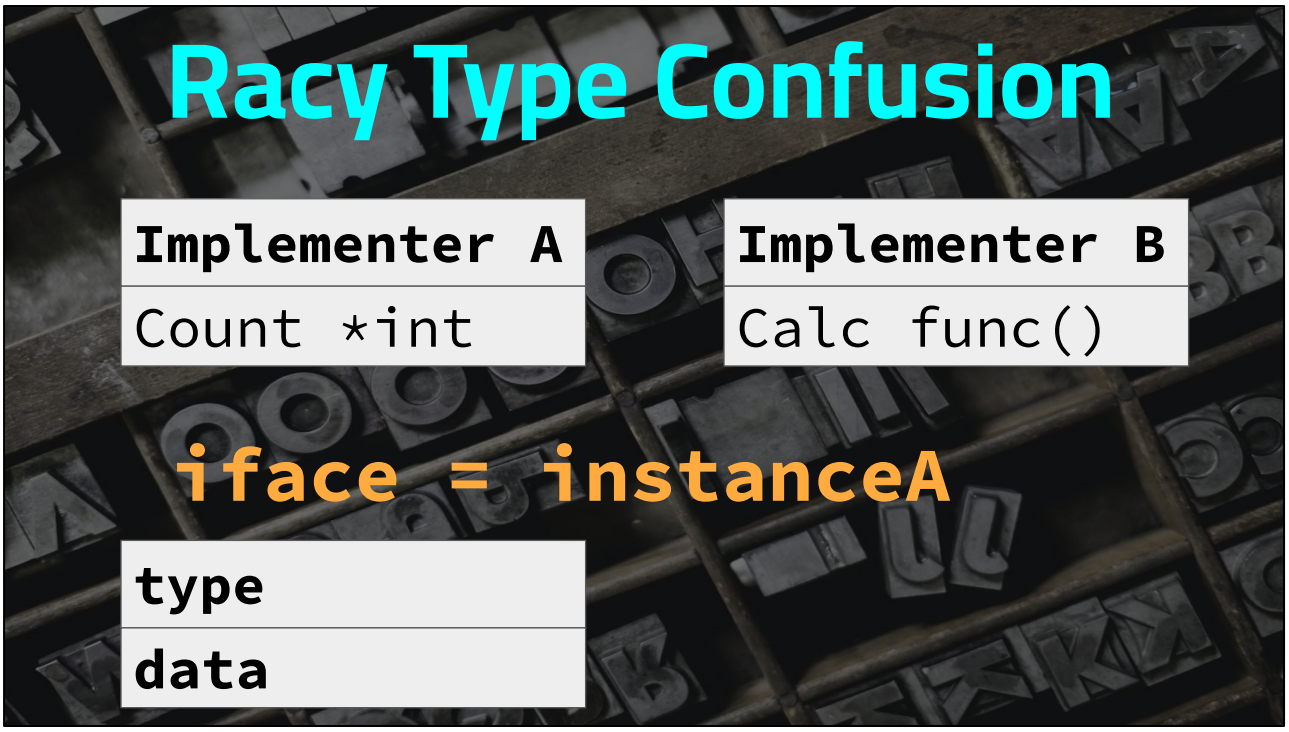
CGO is another well known issue, but it deserves to be mentioned. Incorporating C code into your Go code obviously has the potential to introduce classic memory safety issues, which has also been shown time and again in CTFs, blog posts, twitter rants, etc. so it requires no further elaboration.



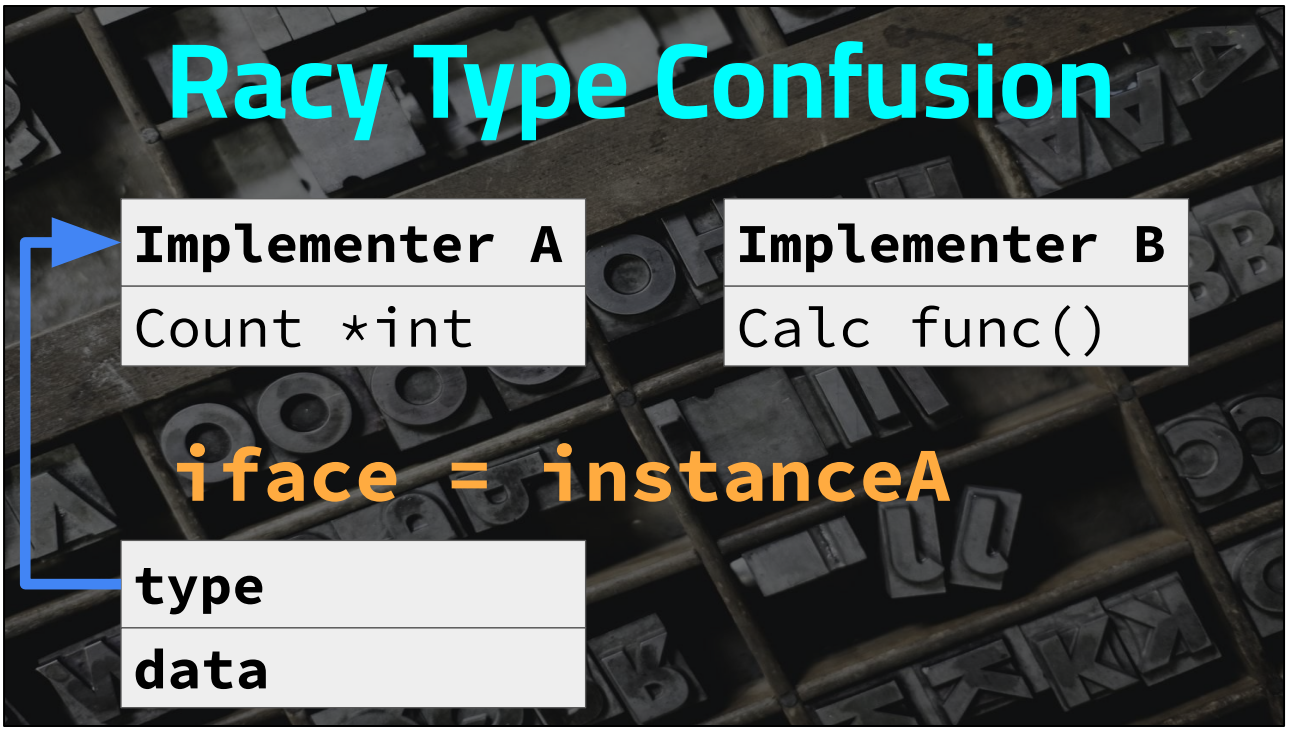
Moving away from the most obvious issues which make the lack of ASLR problematic, let's talk about race conditions. Race conditions really deserve their own section in this talk, and they have their own section, which we will get into later on. I'm.. I'm lying, they were totally going to have their own section but this talk was put together kinda last minute, so you this is what you get. Anyway in Go it's possible to have memory corruption result from race conditions, however the programming patterns required are usually rare and the bugs can be difficult to trigger. I think the canon work on this topic is a Russ Cox blog post from 2010 called "Off to the Races", showing how an interface is being accessed while simultaneously being assigned presents a race condition resulting in type confusion.



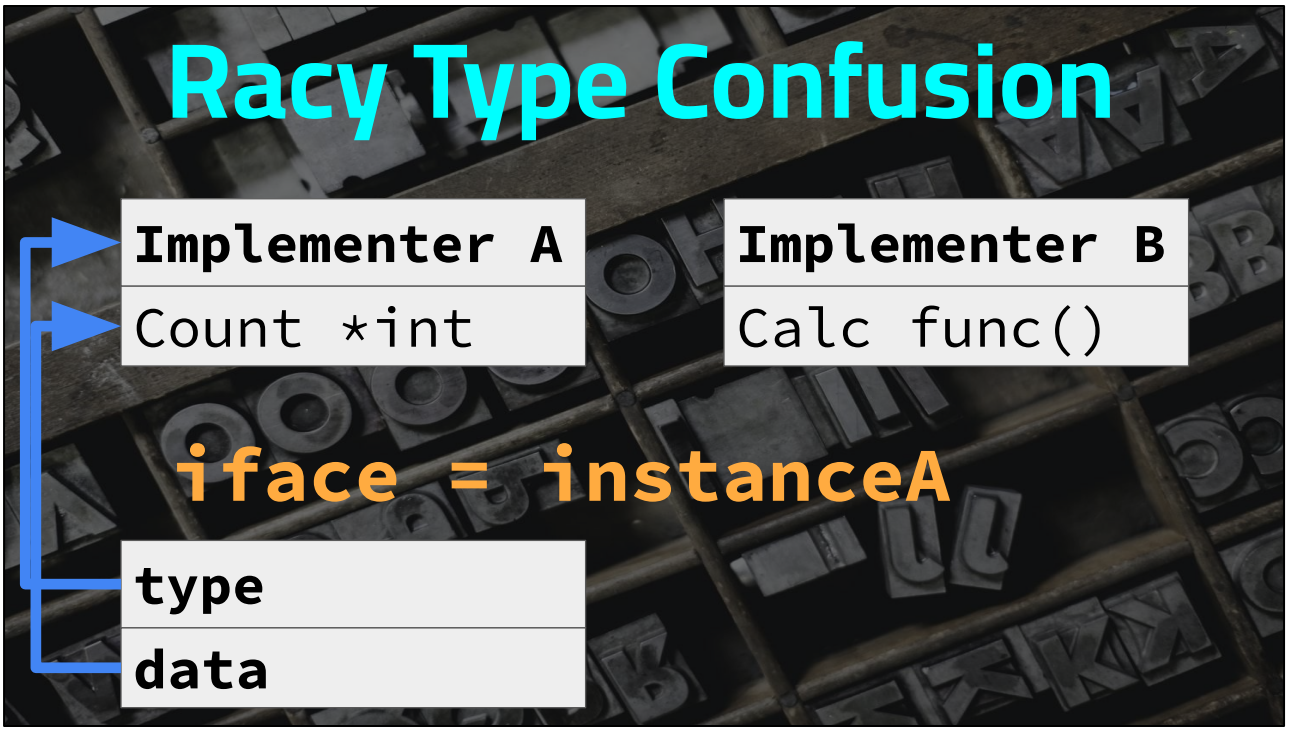
When an interface is assigned, it is not as simple as setting a pointer. The interface is itself a structure, spanning multiple words in memory



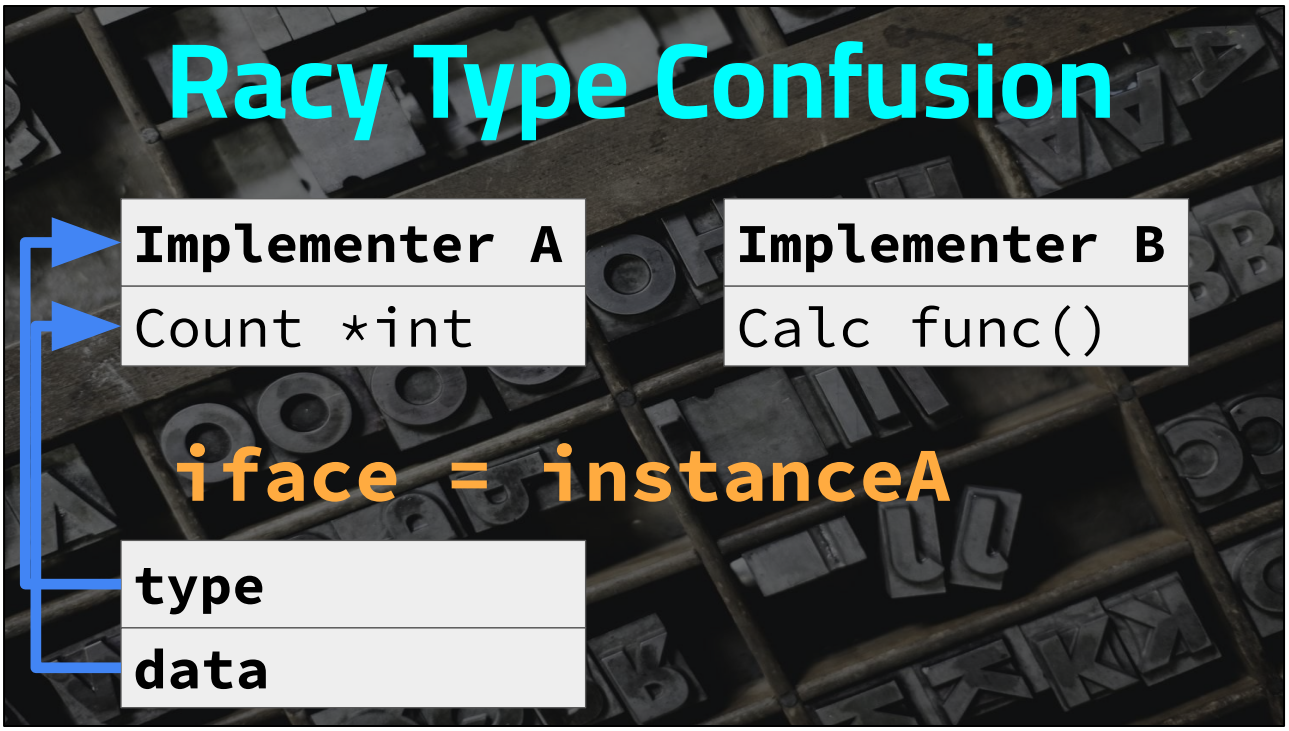
The structure involves two pointers, one for the type which implements the interface, and another for the data which backs that instance of that type. Being that these are two different pointers, they cannot be set atomically.



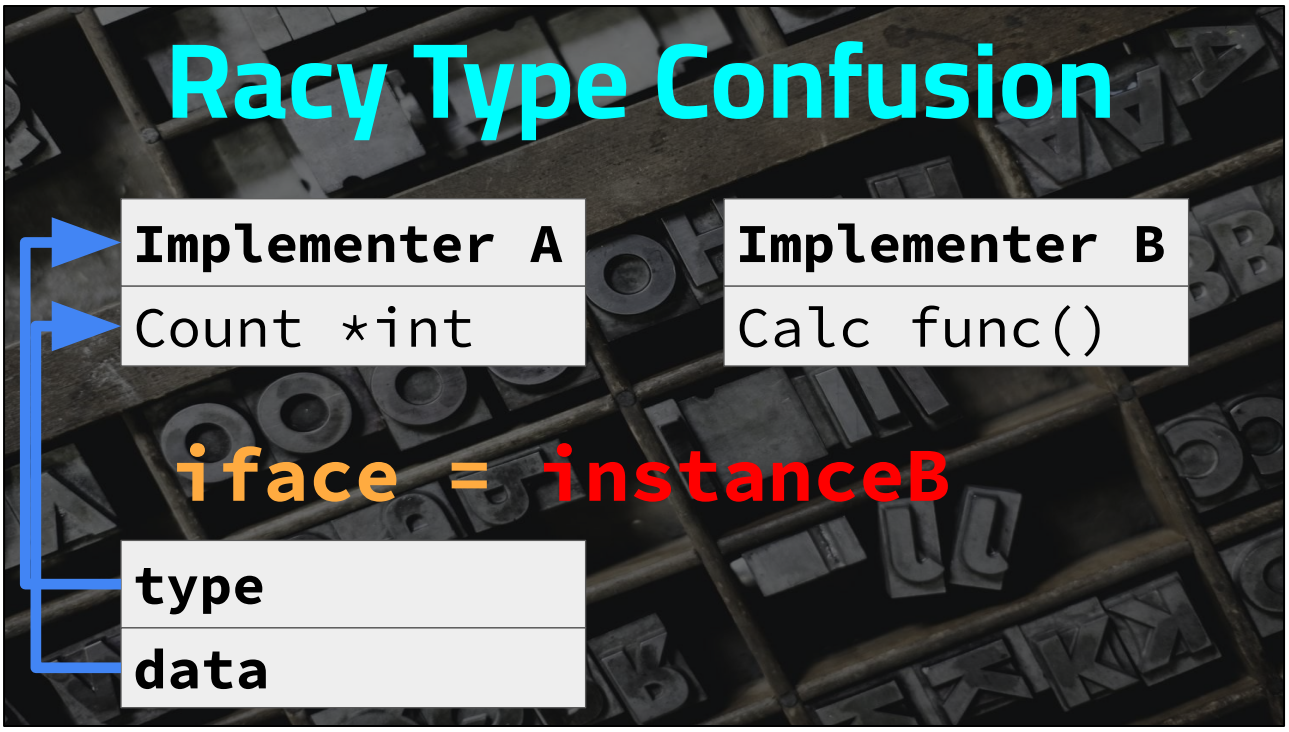
During assignment, such as shown here, the pointers for the interface are set one at a time, across multiple instructions



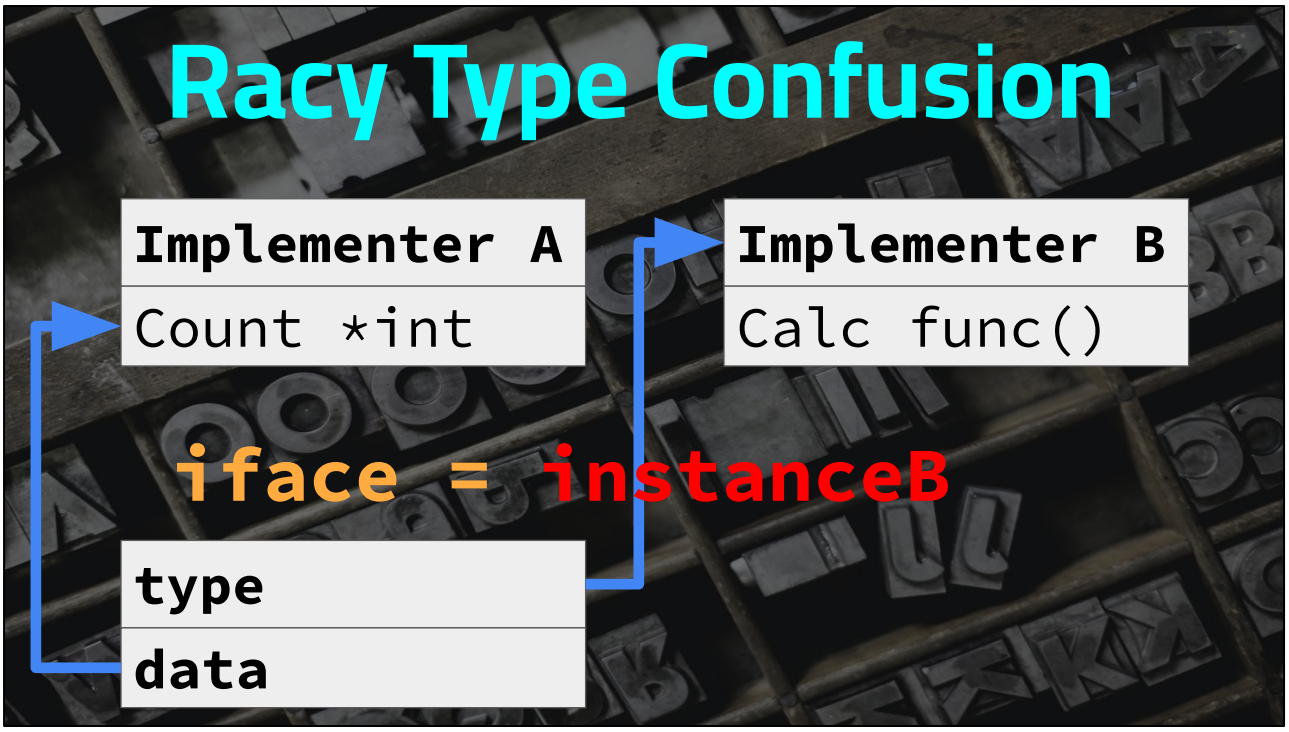
This presents a race condition between when the interface's type is assigned, and when the data backing that type is assigned.



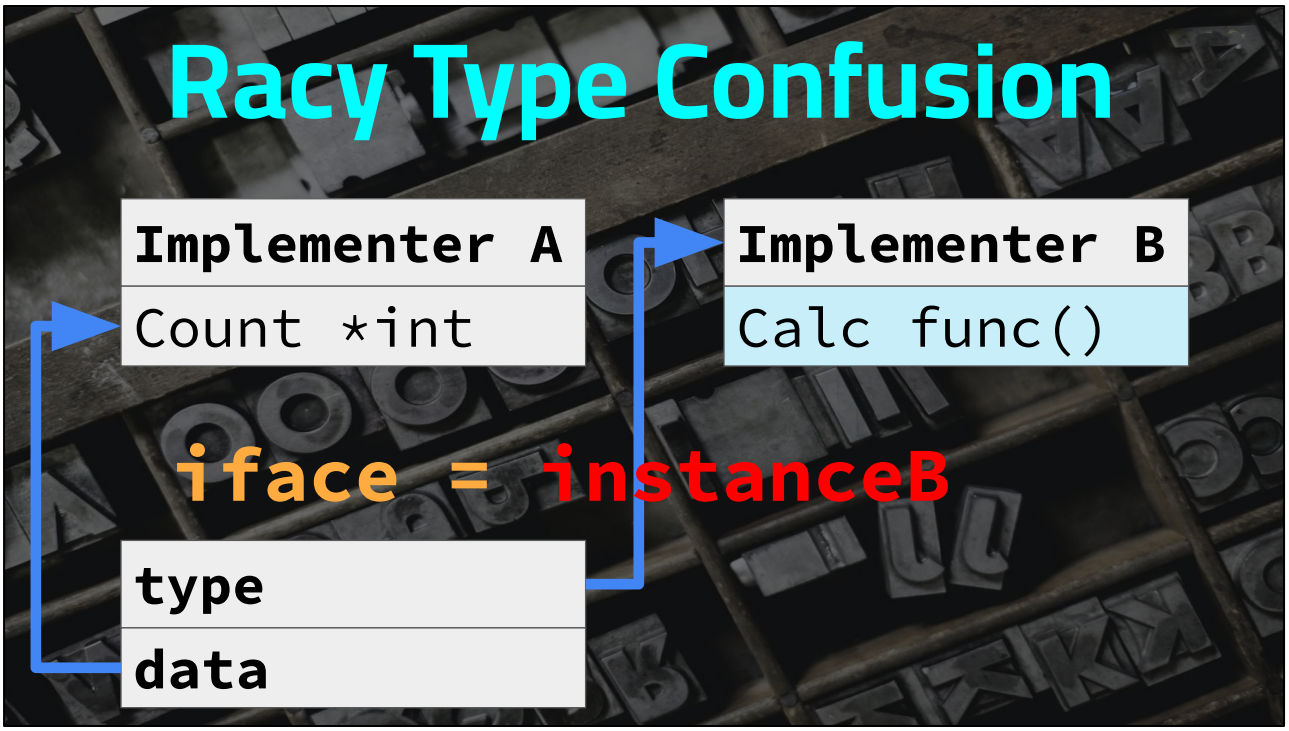
So say an interface variable was first assigned to one implementer, in this case Implementer A



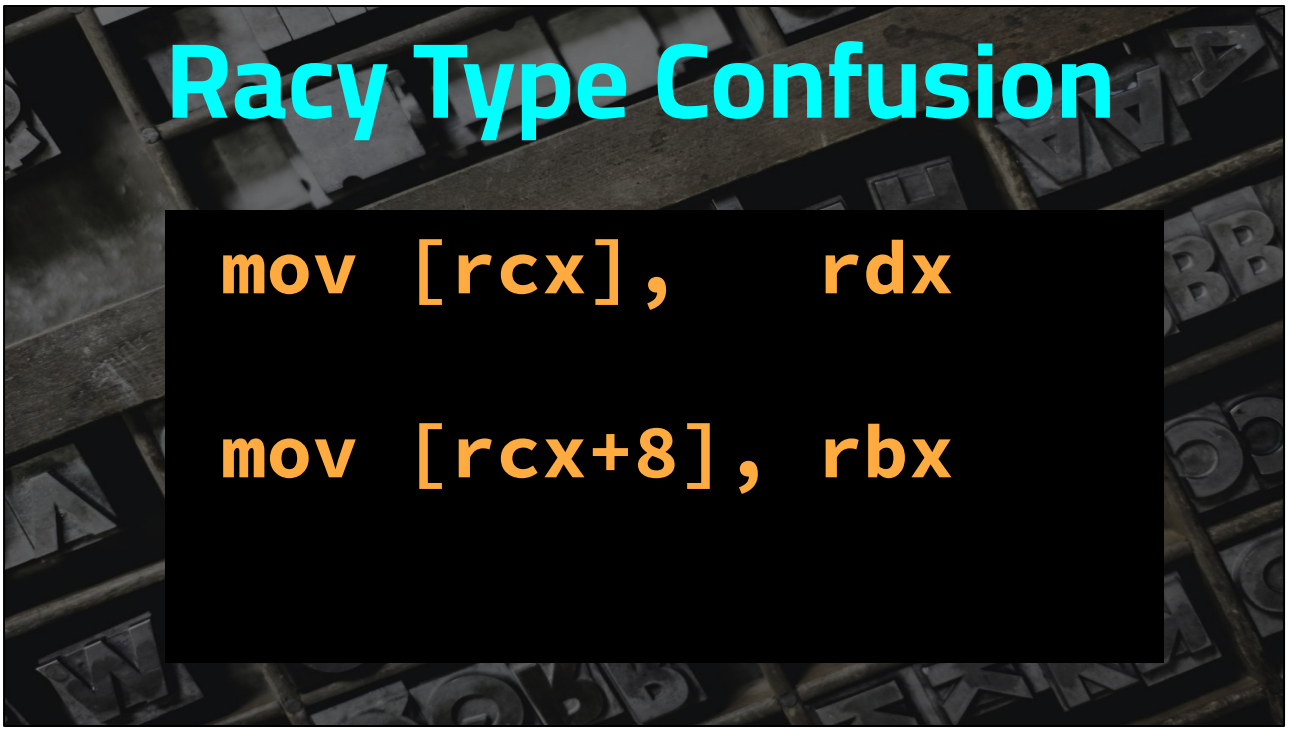
And that interface is then assigned to another implementer, such as an instance of implementer B



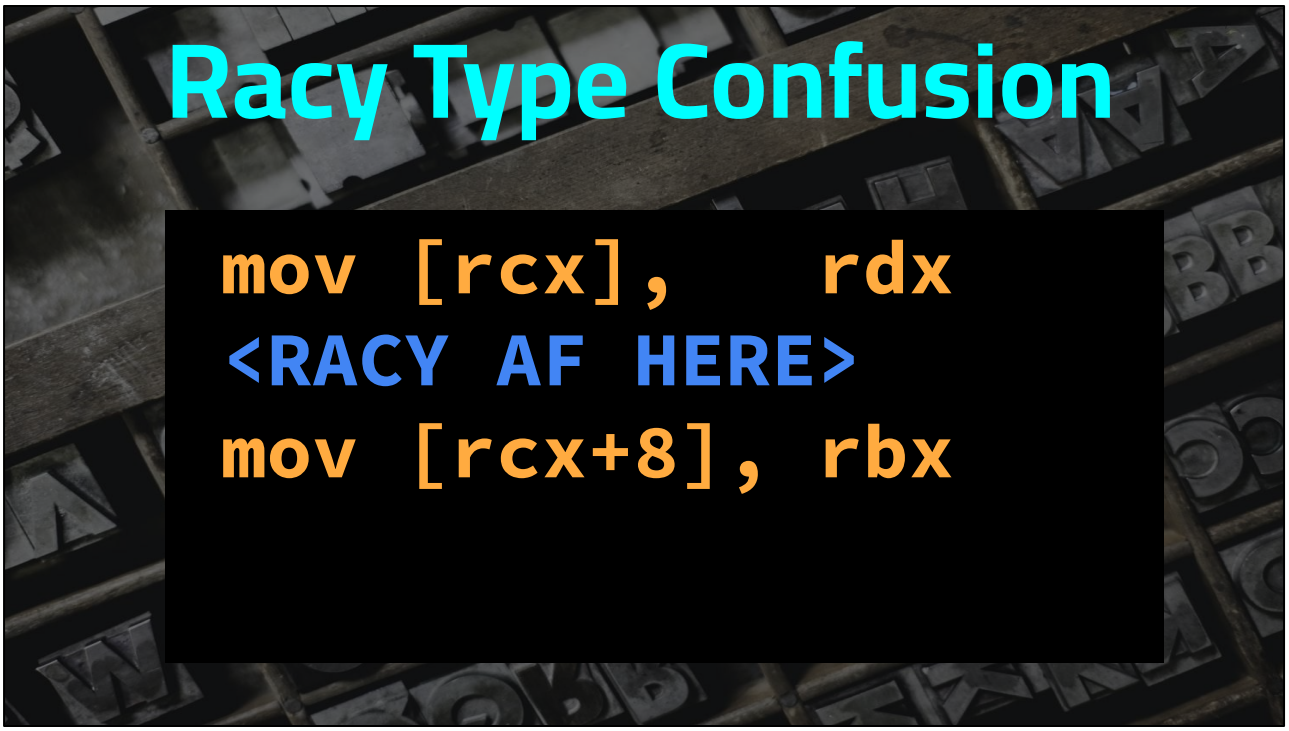
Winning the race would present this inconsistent state, where the interface type and its backing data are disjoint.



During which, if the example “Calc” function, for example, is called in the instance of implementer B, the pointer for that function will actually be based on the Count integer pointer in implementer A, yielding classic type confusion with control of execution flow, violating the type safety assumed by Go programmers. The time window to win this race can actually be bigger than I would have assumed.



Let’s look at the machine code for this race condition. Here is an example of what we might assume is a representation of possible instructions for this assignment to the interface variable, showing the assignment to the interface spans multiple instructions. The pointer to the type being written to rcx, and the pointer to the data backing that type being written to offset 8 of rcx.



The time between these instructions is where the race condition lives, and this is exacerbated by any other instructions in between. You might think “but if the assignment is already racy, why would there be instructions in between? Doesn’t Go want this interface assignment to be as close to atomic as possible?”

Racy Type Confusion

```
mov   QWORD PTR [rbx],rdx
lea   rdi,[rbx+0x8]
mov   QWORD PTR [rsp+0x8],rdi
cmp   DWORD PTR [rip+0xef8c6],0x0
jne   0x490360
mov   QWORD PTR [rbx+0x8],rax
```

Here's the actual disassembly from a proof-of-concept program based on source from the github of a security researcher named StalkR, the code for which is meant to exercise this interface assignment race condition. The highlighted instructions at the top and bottom are from the assignment to the interface variable, in between which Go has inserted instructions related to the runtime's write barrier, and which are probably related to the garbage collector. These additional instructions greatly widen the time window for the racy variable assignment, and increase the chance of winning the race.

Racy Type Confusion

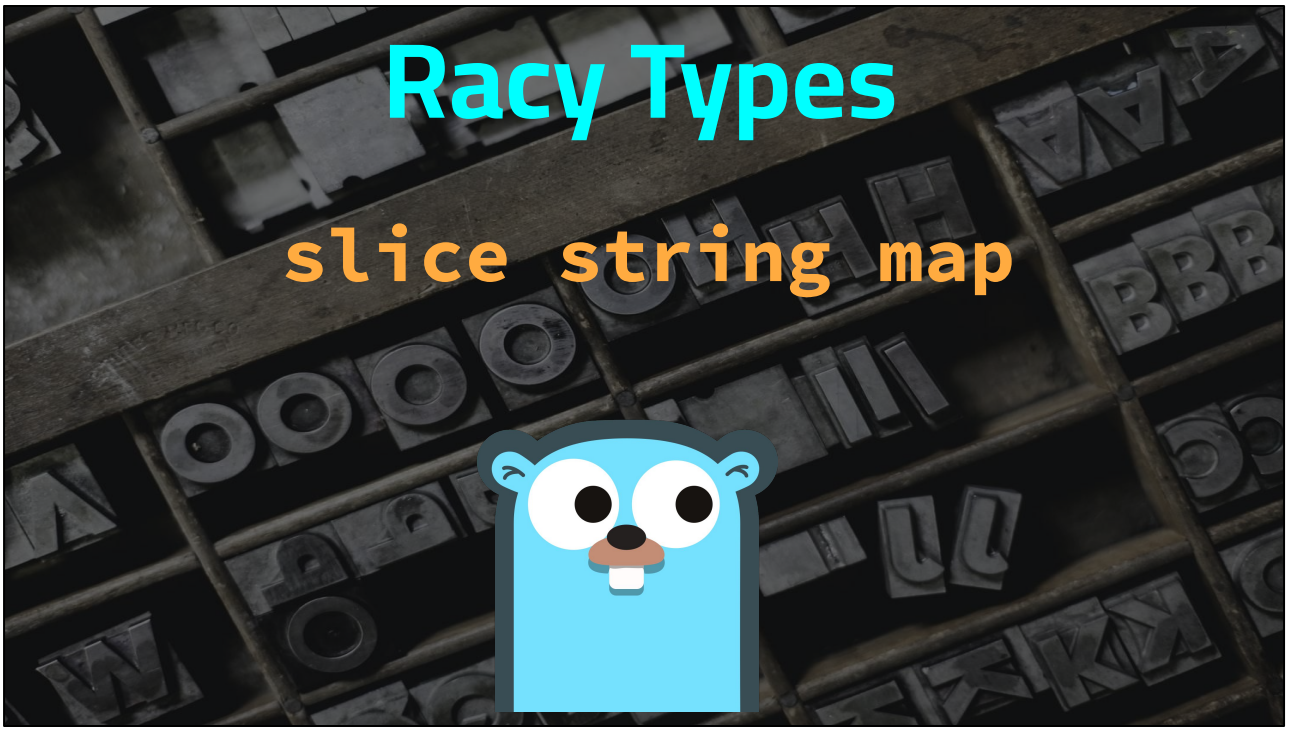
```
mov   QWORD PTR [rbx],rdx
lea   rdi,[rbx+0x8]
mov   QWORD PTR [rsp+0x8],rdi
cmp   DWORD PTR [rip+0xef8c6],0x0
jne   0x490360
mov   QWORD PTR [rbx+0x8],rax
```

Highlighted in red is a conditional branch, which I'm sure doesn't help the racy time window

Racy Type Confusion

```
mov   QWORD PTR [rbx],rdx
lea   rdi,[rbx+0x8]
mov   QWORD PTR [rsp+0x8],rdi
cmp   DWORD PTR [rip+0xef8c6],0x0
jne   0x490360
mov   QWORD PTR [rbx+0x8],rax
```

There's probably room for additional research on if, or how often, the space between these two instructions can span a cache line, which has the potential to even further excite the opportunity-- perhaps someone can research this, I'll leave that up to the audience. Anyway the point is that while it may be a rare scenario to have an assignment to an interface race with an access in another Go routine, if such a scenario presents itself, there is a real chance that someone could win the race.



It’s also worth noting that this type of race is not limited to interfaces, and can also happen with array slices and strings, though strings are immutable so they only present potential for out-of-bounds reads, but that may still prove quite useful to an attacker. Array Slices however can result in either memory corruption, or a panic from a perceived out of bounds access, depending on if the race results in too big or too small of an array size racing with the simultaneous slice access. In practice, I suspect the scenario of an array slice being raced by two go routines is more likely than the interface example, but it’s also probably a tougher situation to reliably exploit.

On the previous slide I had map listed as a racy type. This is one I examined a little bit back in 2016, and then reproduced the race for it yesterday while putting these slides together. It's interesting to me because it appears that it can result in memory corruption, however extremely unlikely, and honestly it's not something I've confirmed so much as am guessing from the output from a handful of experiments

Map Race

```
func funcy(m map[uintptr]string) {
  var i int
  str := fmt.Sprintf("%p", m)
  ptr, _:=strconv.ParseUint(str, 0, 0)
  for i = 0; i < 10000; i++ {
    m[uintptr(ptr)*uintptr(i)] = str
  }
}
```

Map Race

```
fatal error: concurrent map writes
```

Map Race

```
signal SIGSEGV:
code=0x2
addr=0x4c58f2
pc=0x40fdfb
```

Map Race

```
signal SIGSEGV:
code=0x2
addr=0x4c58f2
pc=0x40fdfb
```

Map Race

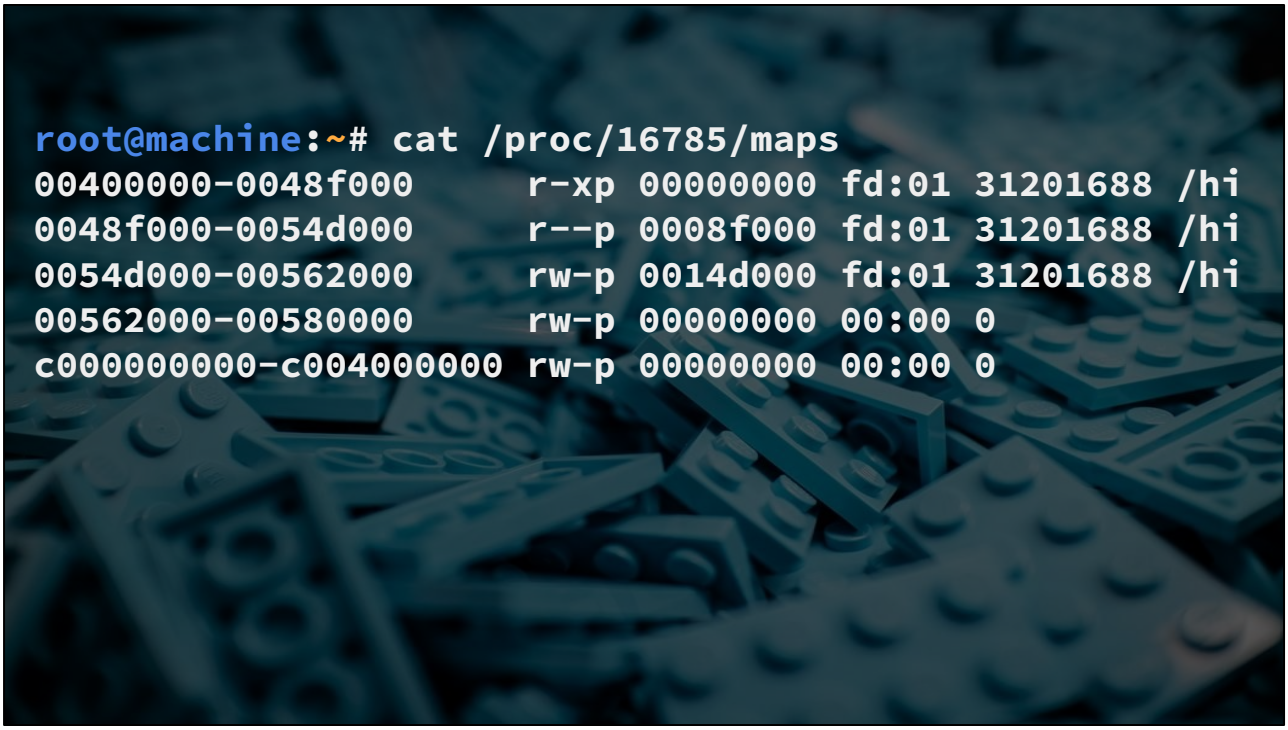
src/runtime/map_fast64.go:

```
if h.flags & hashWriting != 0 {
  throw("concurrent map writes")
}
```

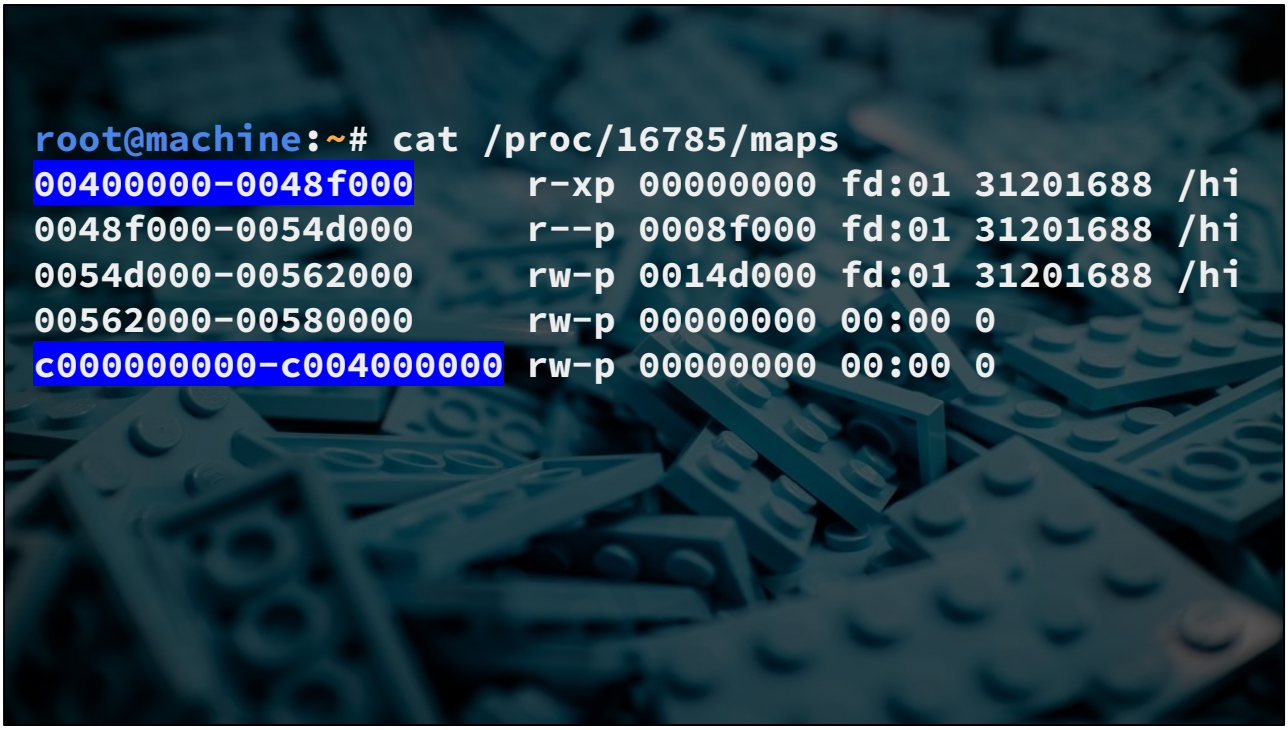
Map Race

src/runtime/map_fast64.go:

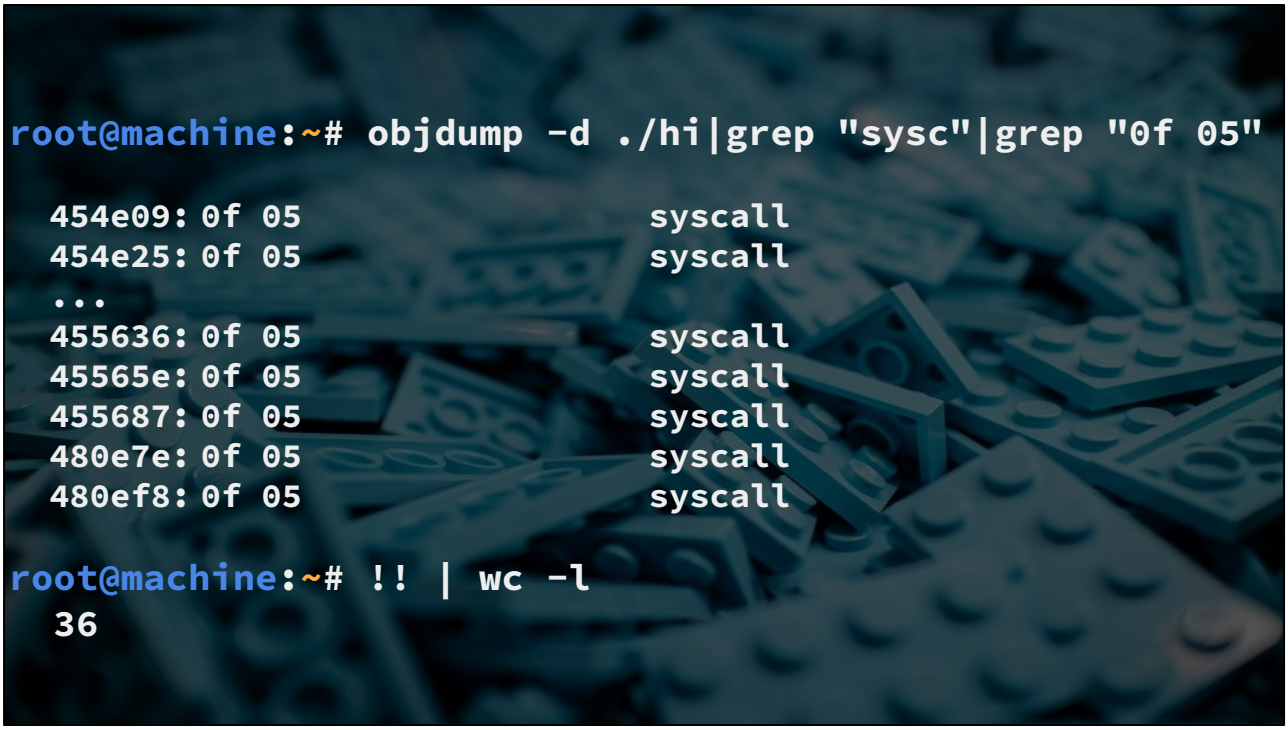
```
h.flags ^= hashWriting
```



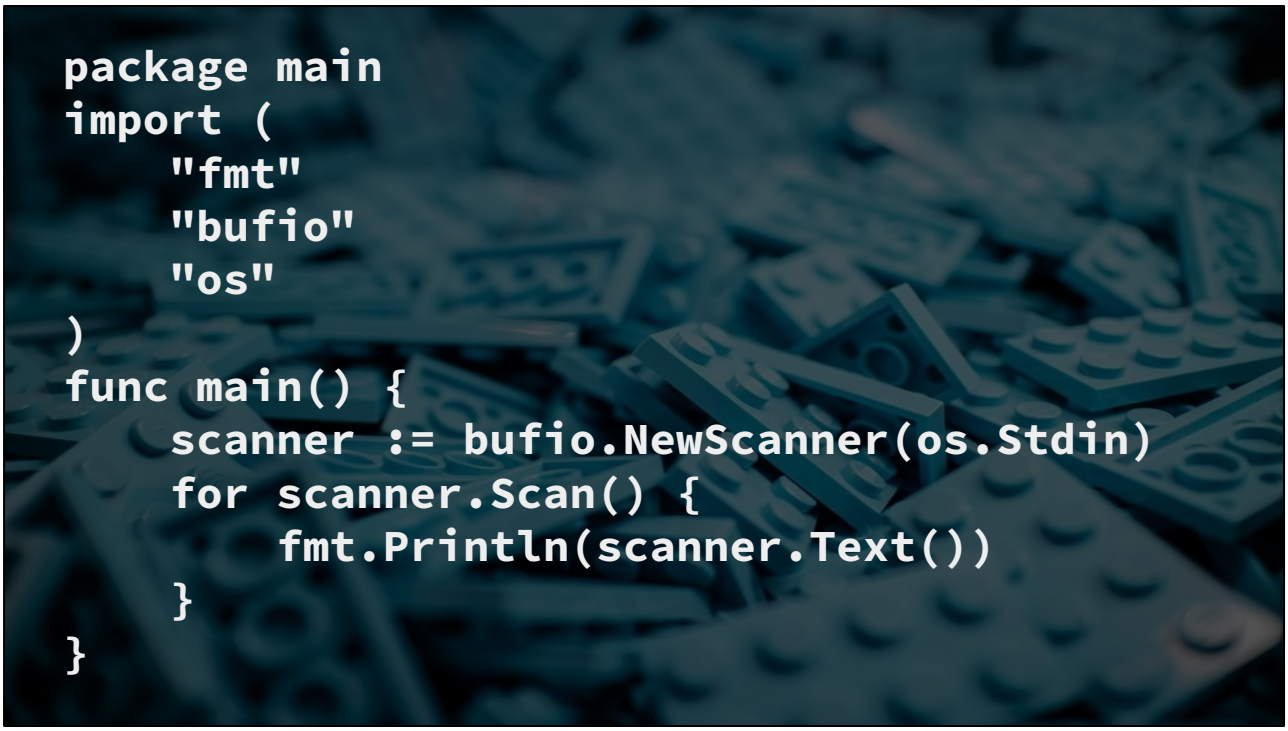
The important part is that in any of the scenarios of Unsafe, CGO, or a data race resulting in memory safety issues, the lack of ASLR doesn't just make the vulnerabilities more likely to be exploitable, it can make them astonishingly easy to exploit.



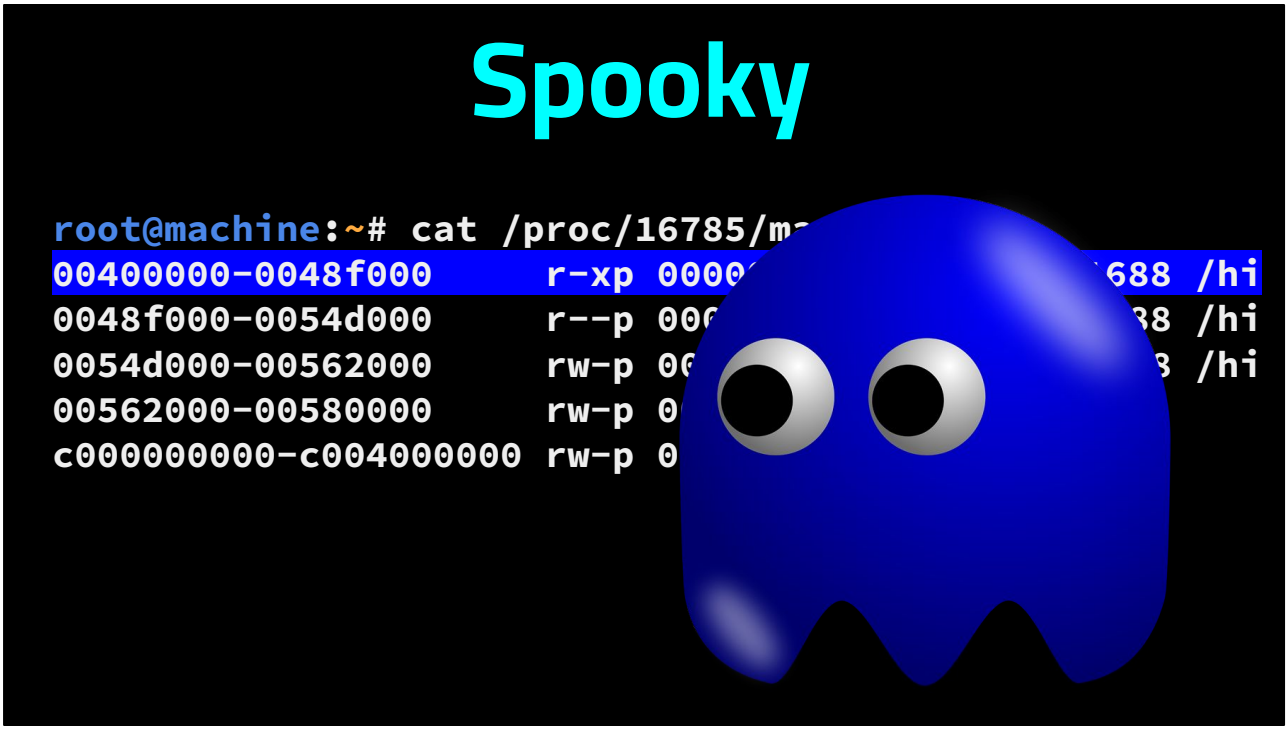
This is in part because Go doesn't just "not use ASLR", but actively uses fixed addresses for the base memory mappings. This is true for both the executable binary mapping, and the region Go uses for its heap.



But another element is that the compiled Go code statically links in its runtime to the resulting binary, which thus produces a massive monolithic blob program, with an oasis of ROP gadgets, and useful toys like inlined syscall instructions, just waiting to be exercised by the newly graduated controller of execution flow. In this regard the language's philosophy of enabling programmers to "just Go" translates quite nicely to the experience of the exploit writer. Here is the output of grepping for syscall instructions in what is just a tiny bit more than a "hello world" program, and we can see 36 unique syscall instruction locations identified.



Just to be clear, here is the source code for that example program we just disassembled and piped through grep on the previous slide. This program is a read-and-echo loop, nothing more. Which, when compiled using “Go build”, resulted in a 2 megabyte binary, with 500k of executable code mapped during actual execution time, including 36 unique syscall instructions among other juicy gadgets and primitives, all mapped at fixed addresses. Grepping the same disassembly for legitimate return instructions yields over 2400 results, and again that’s just the “legit” return instructions, that doesn’t even count “real” ROP ret gadgets occurring on misaligned instructions in the way to which x86 lends itself.



The final thing I want to cover in this section on Go, ASLR, and the elephant in the room, is really just a hypothesis, or a theoretical attack based on the operation of its constituent components. I haven't tested any of this next bit, or even googled enough to see what others maybe have done, because I wrote most of this talk during my subway ride in today, but my hypothesis is that the lack of ASLR creates potential for Go programs to be more susceptible to microarchitectural side channel attacks. Specifically any sort of cross-process branch-target-prediction side-channel attack, like Spectre variant two.


Spooky
containers?
PAYLOAD
62,350 LBS
CU. CAP.
NET
CU. CAP.
OPDU 205271 4
22G1
COR-TEN STEEL CONTAINER
MAX. GROSS
30,480 KG
67,200 LB
TARE
2,200 KG
4,850 LB


Spooky
Compromised Container
Target Container
PAYLOAD
62,350 LBS
CU. CAP.
NET
CU. CAP.
MAX. GROSS
30,480 KG
67,200 LB
TARE
2,200 KG
4,850 LB
COR-TEN STEEL
CONTAINER

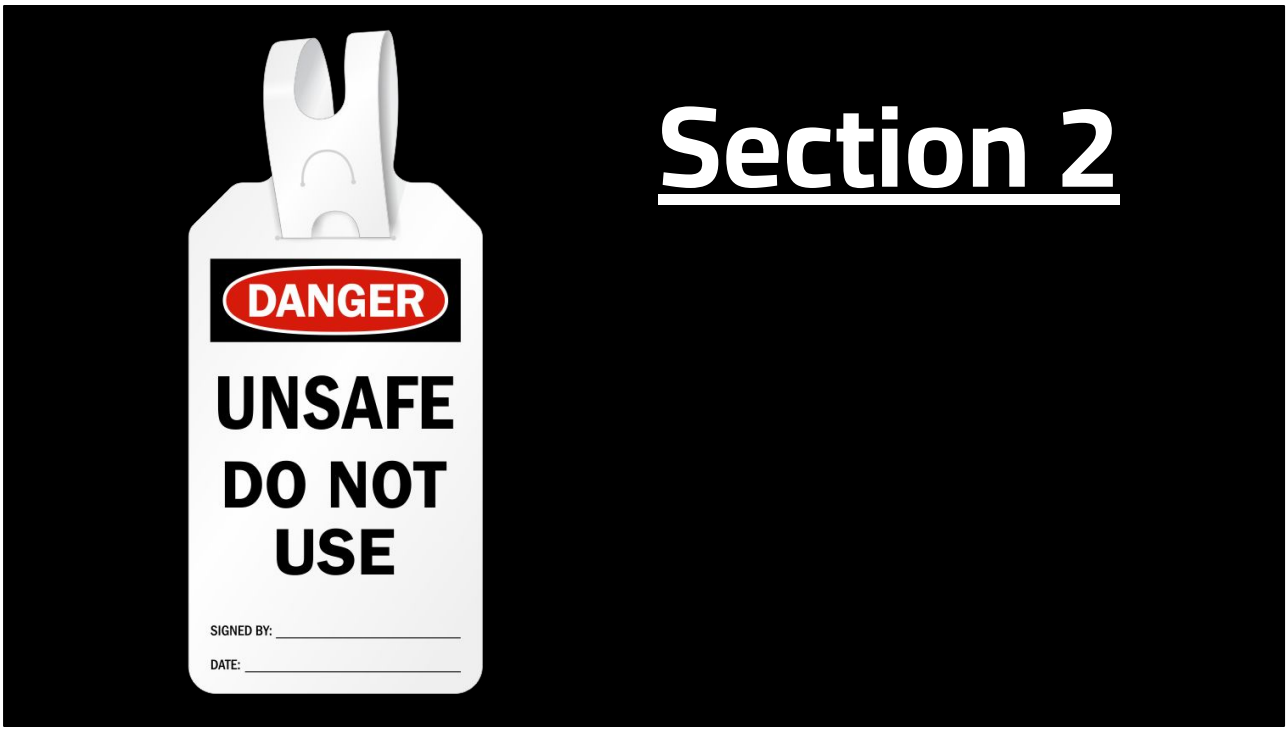
FIXME resume here
Remember to mention since Go 1.15 that there is a spectre flag, but incurs a performance cost and is mostly to avoid the out-of-bounds speculative execution by forcing speculative index of 0

The fixed addresses of code and heap, and use of indirect branches mean there's possibly room

Something something spectre gadgets

Section 2

Section one



Section one

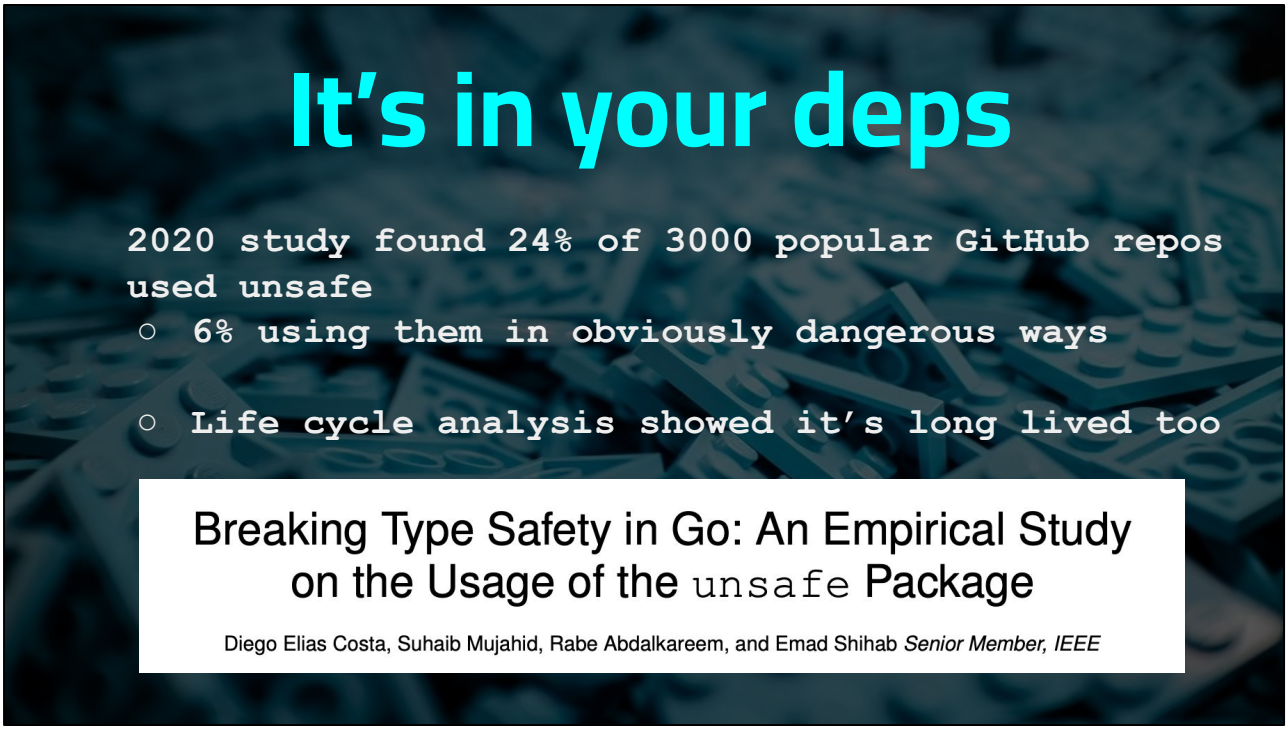
Import “unsafe”

The unsafe package bypasses Go’s type safety

Provides a Pointer type to access types directly and helper functions

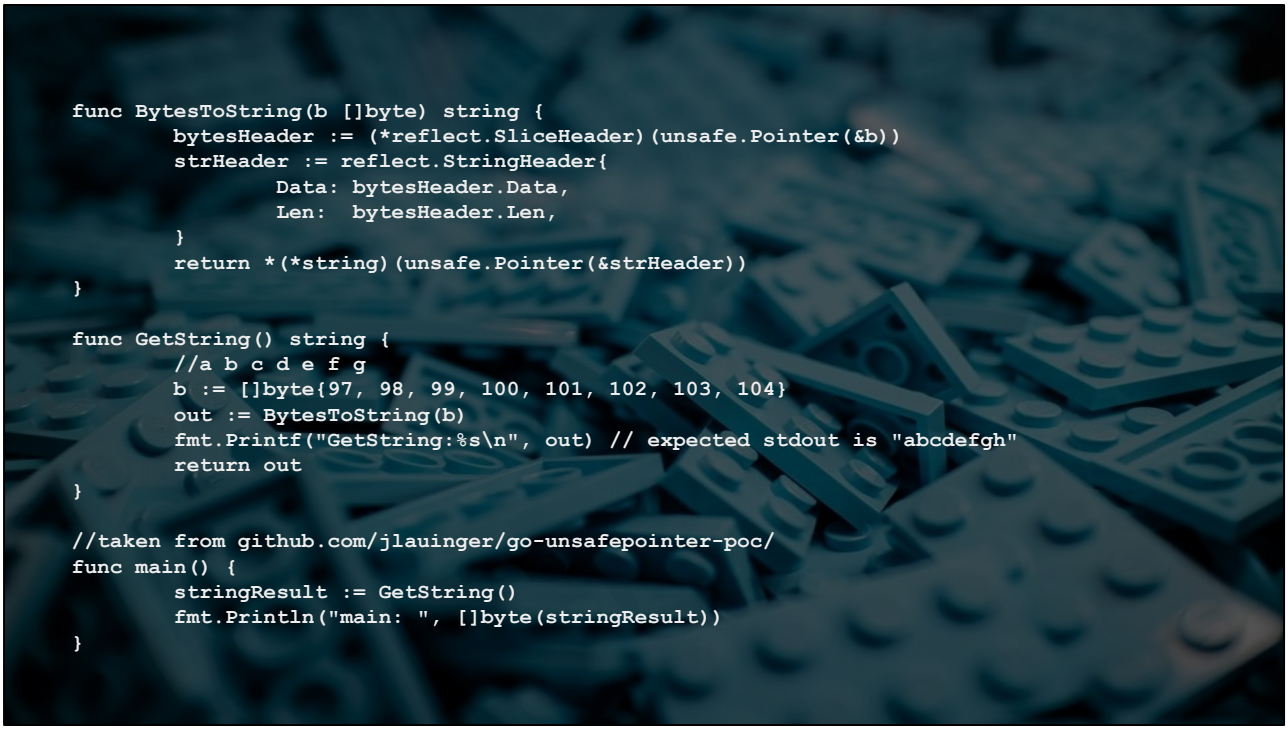
```
u := uintptr(p); p = unsafe.Pointer(u + offset)
```

While



While Brandon may not want to talk about unsafe, I do. Mostly because it has a nasty way of sneaking up on you. Want to get a super-performant package to do something, probably touches unsafe. Want to interface with the operating system or other third party thing, probably unsafe is in there. Want to see how a lot of people are clever with hash keys it’s there again.

So while you’re not using unsafe, you might be accidentally using it through your dependencies. The study from 2020 checked 3000 popular GitHub repos and found that almost a quarter of them were using unsafe and did across a lot of commits. What’s worse is that 6% of them were using it in an obviously dangerous way (e.g. updating a pointer past its bounds).



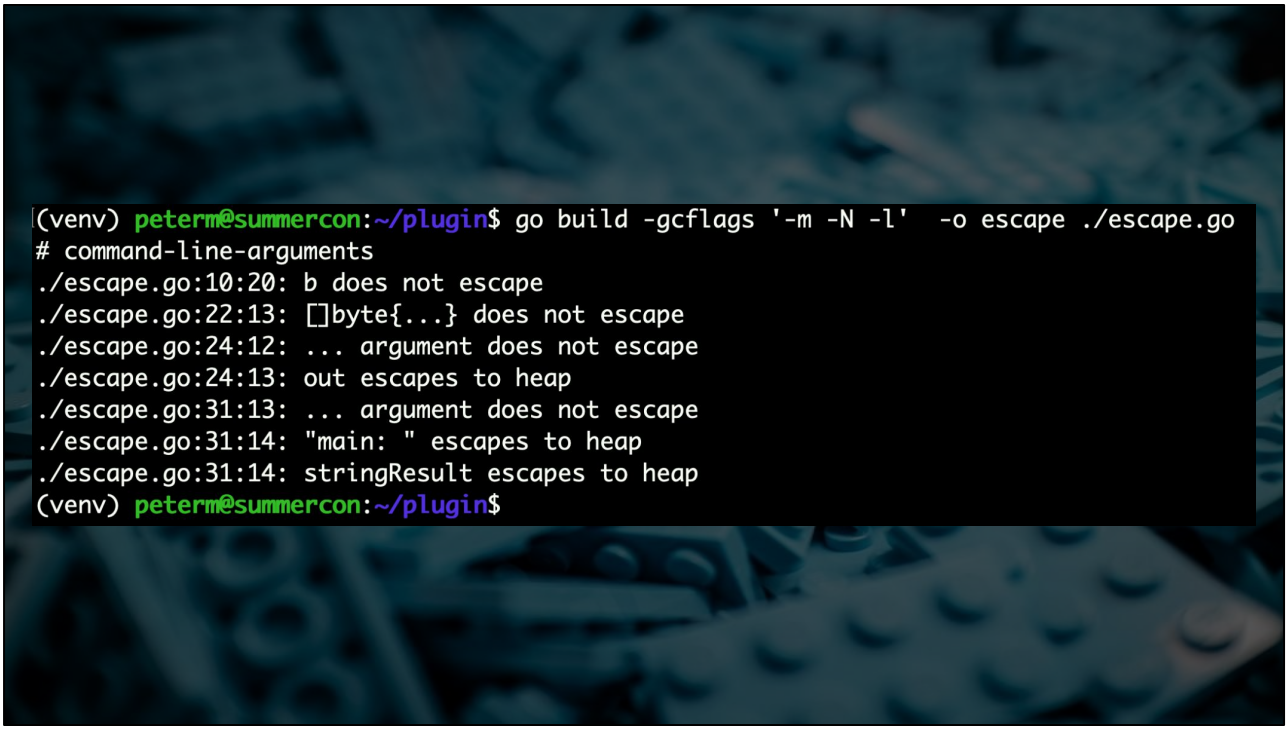
```
func BytesToString(b []byte) string {
        bytesHeader := (*reflect.SliceHeader)(unsafe.Pointer(&b))
        strHeader := reflect.StringHeader{
                Data: bytesHeader.Data,
                Len:  bytesHeader.Len,
        }
        return *(*string)(unsafe.Pointer(&strHeader))
}

func GetString() string {
        //a b c d e f g
        b := []byte{97, 98, 99, 100, 101, 102, 103, 104}
        out := BytesToString(b)
        fmt.Printf("GetString:%s\n", out) // expected stdout is "abcdefgh"
        return out
}

//taken from github.com/jlauinger/go-unsafepointer-poc/
func main() {
        stringResult := GetString()
        fmt.Println("main: ", []byte(stringResult))
}
```

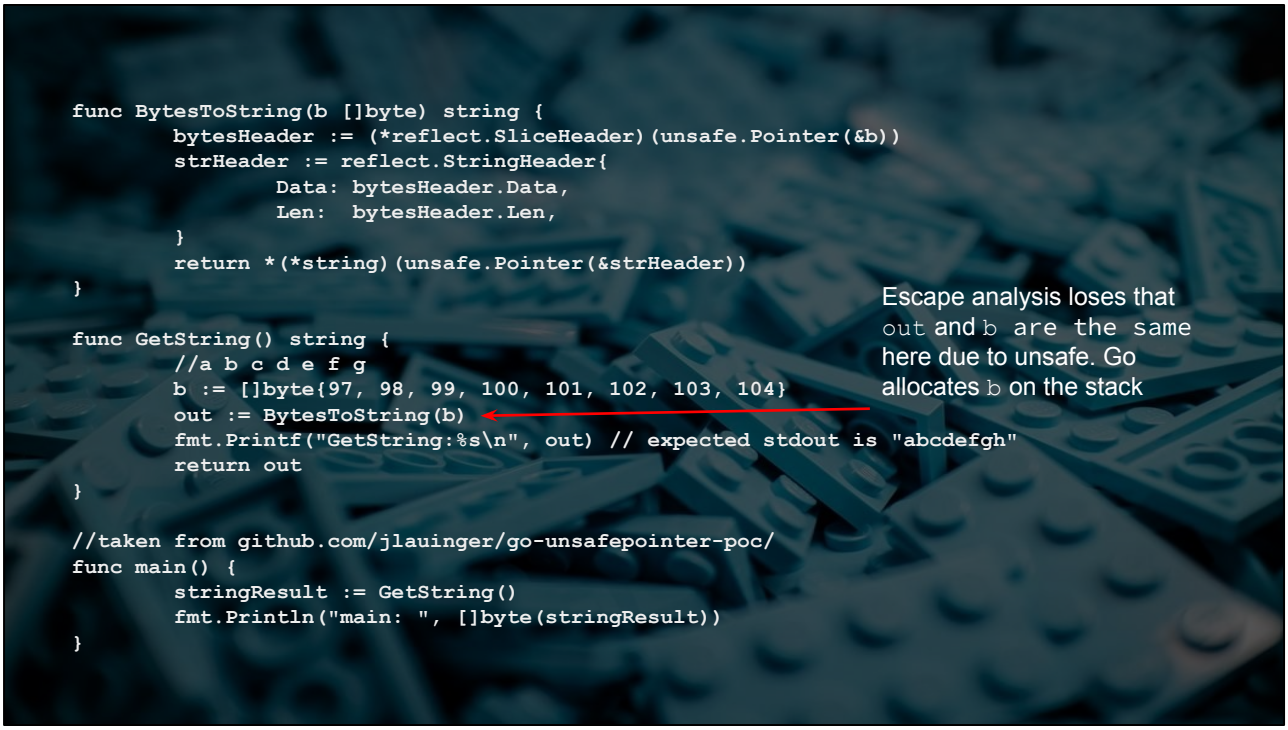
Note: check how many people are familiar with escape analysis
Unsafe is often used in packages to create hashes or interface with legacy C / operating system code. Often this can lead to issues such as stale pointers, incorrect casts and possible code execution. One of the more interesting examples from Johannes Lauinger's repo shows when unsafe / reflect is used to break goes escape analysis.

Unfortunately this pattern shows up in a few places, if you search GitHub.



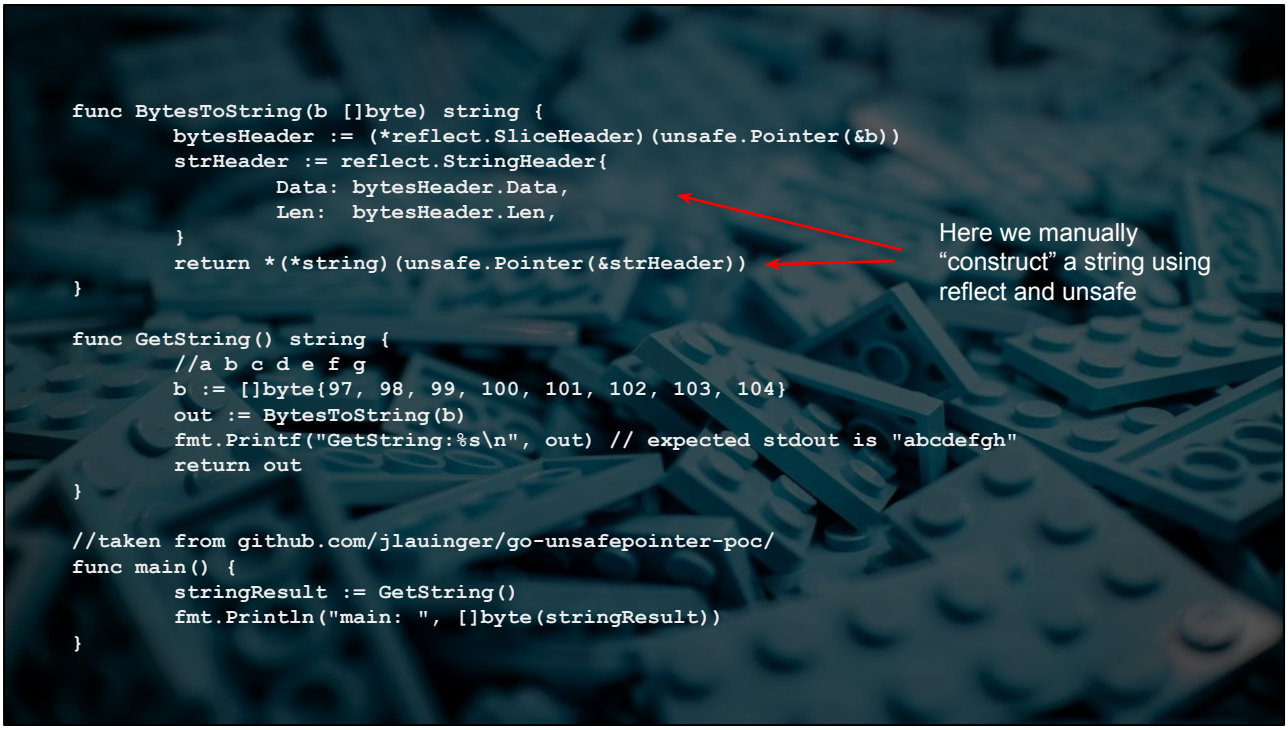
Note: check how many people are familiar with escape analysis
Unsafe is often used in packages to create hashes or interface with legacy C / operating system code. Often this can lead to issues such as stale pointers, incorrect casts and possible code execution. One of the more interesting examples from Johannes Lauinger's repo shows when unsafe / reflect is used to break goes escape analysis.

Unfortunately this pattern shows up in a few places, if you search GitHub.



Note: check how many people are familiar with escape analysis
Unsafe is often used in packages to create hashes or interface with legacy C / operating system code. Often this can lead to issues such as stale pointers, incorrect casts and possible code execution. One of the more interesting examples from Johannes Lauinger's repo shows when unsafe / reflect is used to break goes escape analysis.

Unfortunately this pattern shows up in a few places, if you search GitHub.



Note: check how many people are familiar with escape analysis
Unsafe is often used in packages to create hashes or interface with legacy C / operating system code. Often this can lead to issues such as stale pointers, incorrect casts and possible code execution. One of the more interesting examples from Johannes Lauinger's repo shows when unsafe / reflect is used to break goes escape analysis.

Unfortunately this pattern shows up in a few places, if you search GitHub.

```
func BytesToString(b []byte) string {
        bytesHeader := (*reflect.SliceHeader)(unsafe.Pointer(&b))
        strHeader := reflect.StringHeader{
                Data: bytesHeader.Data,
                Len:  bytesHeader.Len,
        }
        return *(*string)(unsafe.Pointer(&strHeader))
}

func GetString() string {
        //a b c d e f g
        b := []byte{97, 98, 99, 100, 101, 102, 103, 104}
        out := BytesToString(b)
        fmt.Printf("GetString:%s\n", out) // expected stdout is "abcdefgh"
        return out
}

//taken from github.com/jlauinger/go-unsafepointer-poc/
func main() {
        stringResult := GetString()
        fmt.Println("main: ", []byte(stringResult))
}
```

Returns a string with a stale Data pointer pointing to the stack

Unsafe is often used in packages to create hashes or interface with legacy C / operating system code. Often this can lead to issues such as stale pointers, incorrect casts and possible code execution. One of the more interesting examples from Johannes Lauinger's repo shows when unsafe / reflect is used to break goes escape analysis.

Unfortunately this pattern shows up in a few places, if you search GitHub.

```
func BytesToString(b []byte) string {
        bytesHeader := (*reflect.SliceHeader)(unsafe.Pointer(&b))
        strHeader := reflect.StringHeader{
                Data: bytesHeader.Data,
                Len:  bytesHeader.Len,
        }
        return *(*string)(unsafe.Pointer(&strHeader))
}

func GetString() string {
        //a b c d e f g
        b := []byte{97, 98, 99, 100, 101, 102, 103, 104}
        out := BytesToString(b)
        fmt.Printf("GetString:%s\n", out) // expected stdout is "abcdefgh"
        return out
}

//taken from github.com/jlauinger/go-unsafepointer-poc/
func main() {
        stringResult := GetString()
        fmt.Println("main: ", []byte(stringResult))
}
```

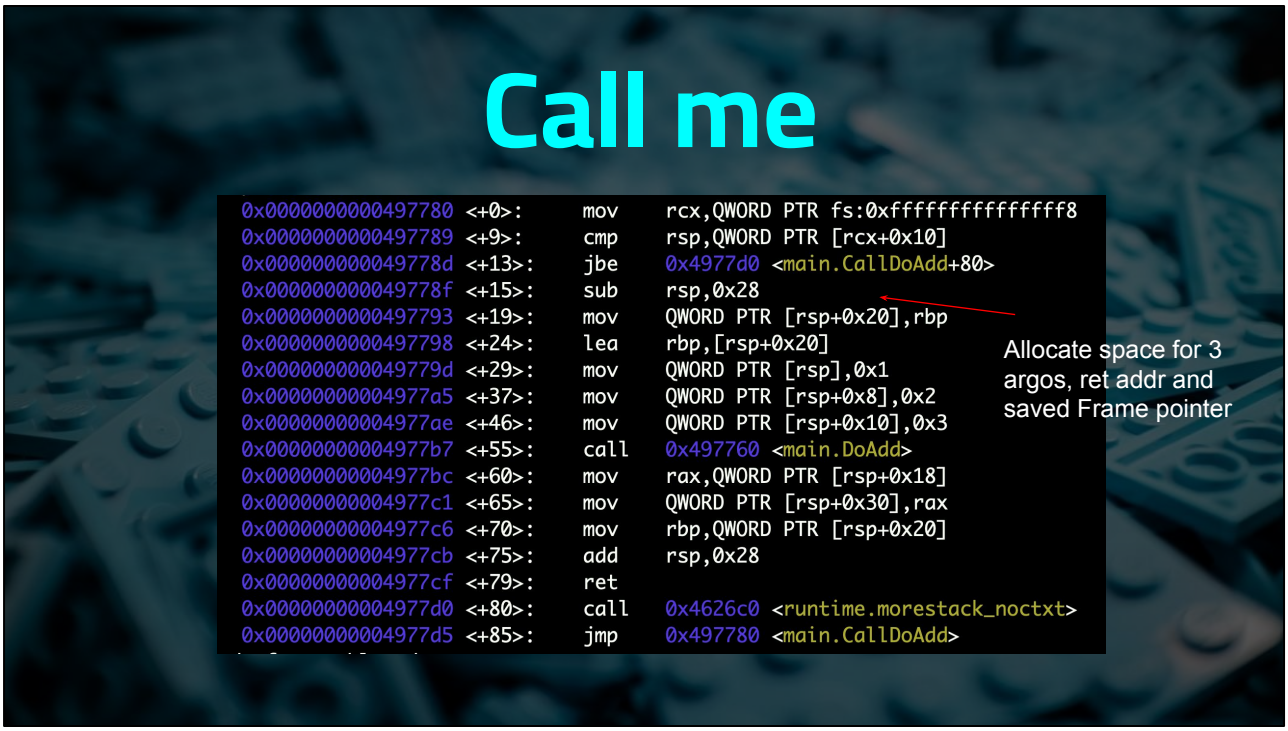
`stringResult` now contains the stale pointer to the stack

At this point our string's Data pointer is now pointing to left out stake space from GetString's stack frame and we have a .

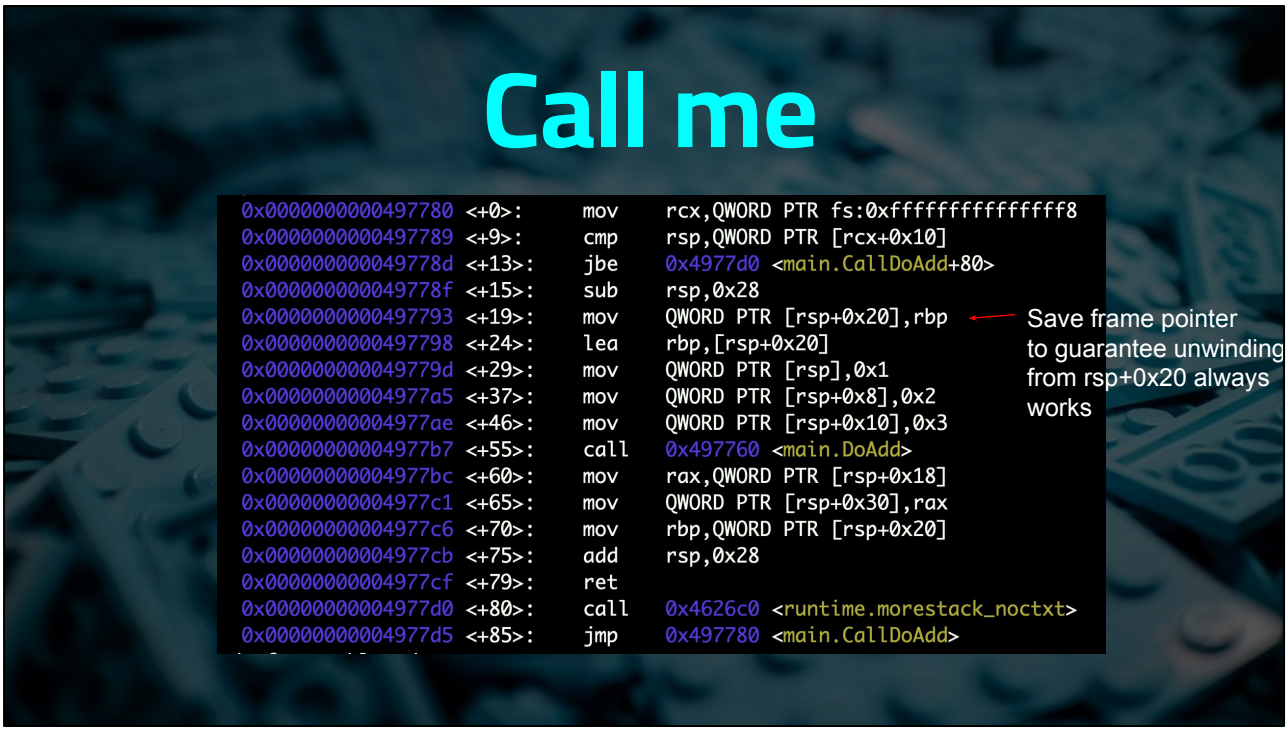
Call me

```
0x0000000000497780 <+0>:     mov    rcx,QWORD PTR fs:0xfffffffffffffff8
0x0000000000497789 <+9>:     cmp    rsp,QWORD PTR [rcx+0x10]
0x000000000049778d <+13>:    jbe    0x4977d0 <main.CallDoAdd+80>
0x000000000049778f <+15>:    sub    rsp,0x28
0x0000000000497793 <+19>:    mov    QWORD PTR [rsp+0x20],rbp
0x0000000000497798 <+24>:    lea    rbp,[rsp+0x20]
0x000000000049779d <+29>:    mov    QWORD PTR [rsp],0x1
0x00000000004977a5 <+37>:    mov    QWORD PTR [rsp+0x8],0x2
0x00000000004977ae <+46>:    mov    QWORD PTR [rsp+0x10],0x3
0x00000000004977b7 <+55>:    call   0x497760 <main.DoAdd>
0x00000000004977bc <+60>:    mov    rax,QWORD PTR [rsp+0x18]
0x00000000004977c1 <+65>:    mov    QWORD PTR [rsp+0x30],rax
0x00000000004977c6 <+70>:    mov    rbp,QWORD PTR [rsp+0x20]
0x00000000004977cb <+75>:    add    rsp,0x28
0x00000000004977cf <+79>:    ret
0x00000000004977d0 <+80>:    call   0x4626c0 <runtime.morestack_noctxt>
0x00000000004977d5 <+85>:    jmp    0x497780 <main.CallDoAdd>
```

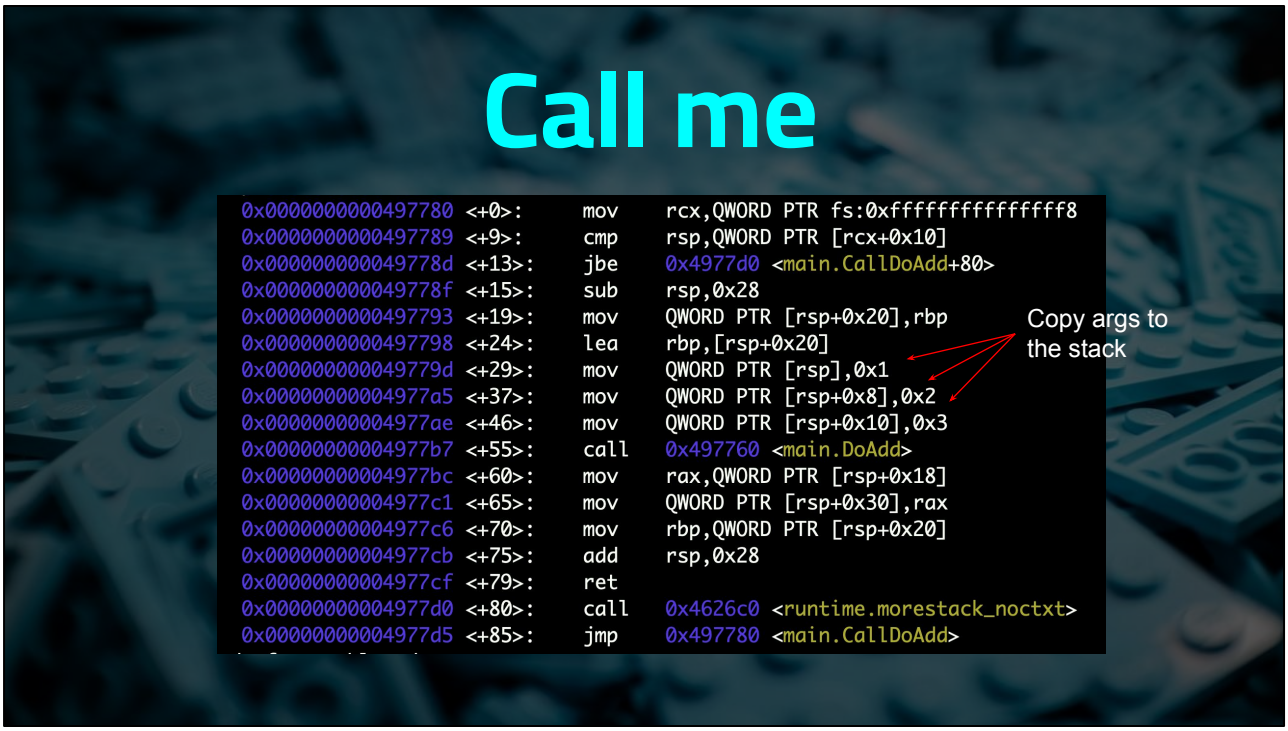
Pre go1.17 go's calling convention is based on plan9's all stack based calling convention. As we talked about earlier preamble checks to see if there's enough stack space for the goroutine. Then it allocates space for the return address and after that it allocates space for the arguments all of this happens in the caller's stack frame. All registers are considered scratch space.



Pre go1.17 go's calling convention is based on plan9's all stack based calling convention. As we talked about earlier preamble checks to see if there's enough stack space for the goroutine. Then it allocates space for the return address and after that it allocates space for the arguments all of this happens in the caller's stack frame. All registers are considered scratch space.



Pre go1.17 go's calling convention is based on plan9's all stack based calling convention. As we talked about earlier preamble checks to see if there's enough stack space for the goroutine. Then it allocates space for the return address and after that it allocates space for the arguments all of this happens in the caller's stack frame. All registers are considered scratch space.



Pre go1.17 go's calling convention is based on plan9's all stack based calling convention. As we talked about earlier preamble checks to see if there's enough stack space for the goroutine. Then it allocates space for the return address and after that it allocates space for the arguments all of this happens in the caller's stack frame. All registers are considered scratch space.

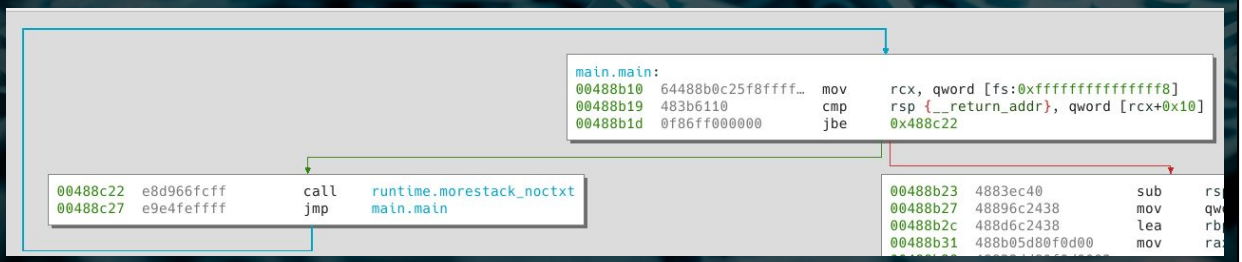
Call me

```
0x0000000000497780 <+0>:     mov    rcx,QWORD PTR fs:0xfffffffffffffff8
0x0000000000497789 <+9>:     cmp    rsp,QWORD PTR [rcx+0x10]
0x000000000049778d <+13>:    jbe    0x4977d0 <main.CallDoAdd+80>
0x000000000049778f <+15>:    sub    rsp,0x28
0x0000000000497793 <+19>:    mov    QWORD PTR [rsp+0x20],rbp
0x0000000000497798 <+24>:    lea    rbp,[rsp+0x20]
0x000000000049779d <+29>:    mov    QWORD PTR [rsp],0x1
0x00000000004977a5 <+37>:    mov    QWORD PTR [rsp+0x8],0x2
0x00000000004977ae <+46>:    mov    QWORD PTR [rsp+0x10],0x3
0x00000000004977b7 <+55>:    call   0x497760 <main.DoAdd>
0x00000000004977bc <+60>:    mov    rax,QWORD PTR [rsp+0x18]
0x00000000004977c1 <+65>:    mov    QWORD PTR [rsp+0x30],rax
0x00000000004977c6 <+70>:    mov    rbp,QWORD PTR [rsp+0x20]
0x00000000004977cb <+75>:    add    rsp,0x28
0x00000000004977cf <+79>:    ret
0x00000000004977d0 <+80>:    call   0x4626c0 <runtime.morestack_noctxt>
0x00000000004977d5 <+85>:    jmp    0x497780 <main.CallDoAdd>
```

Copy ret value from stack to rax

Pre go1.17 go's calling convention is based on plan9's all stack based calling convention. As we talked about earlier preamble checks to see if there's enough stack space for the goroutine. Then it allocates space for the return address and after that it allocates space for the arguments all of this happens in the caller's stack frame. All registers are considered scratch space.

Move Your Stack for Great Justice

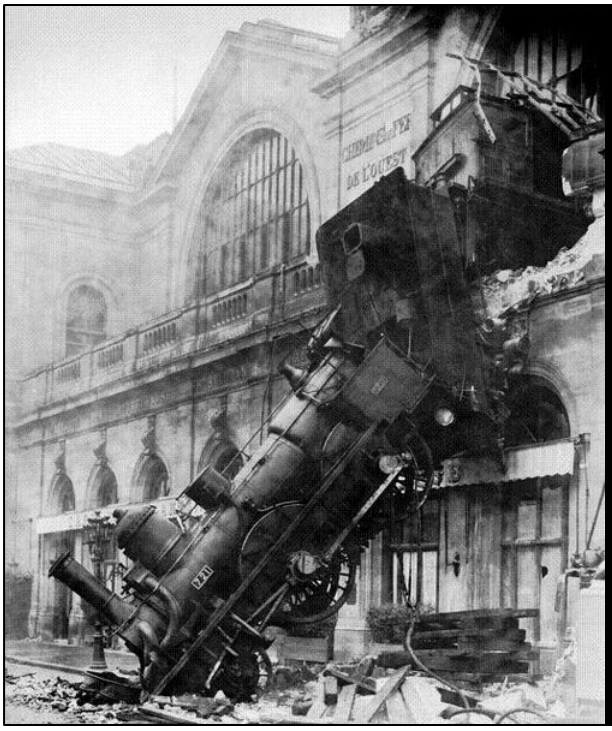


From https://blog.osiris.cyber.nyu.edu/2019/12/19/go-deepdive/

So one thing that's new for a lot of folks is that goroutine's stack frames move. If you look at the preamble to the calling convention it always checks if the goroutine needs more stack space and if so it moves the goroutines stack. Similarly when you trigger garbage collection, the stack size shrinks. In theory with these mechanisms, it could be possible to see if you could abuse this stale pointer relationship to point across goroutine stacks.

Section 3

- **Lots of work in this space:**
  - **Reversing GO binaries like a pro by Tim Strazzere**
  - **Reversing Golang by George Zaytsev**
  - **NYUPoly OSirus**
  - **Lots of tools:**
    - **Redress / GORE**
    - **Lots of golang idapython scripts**



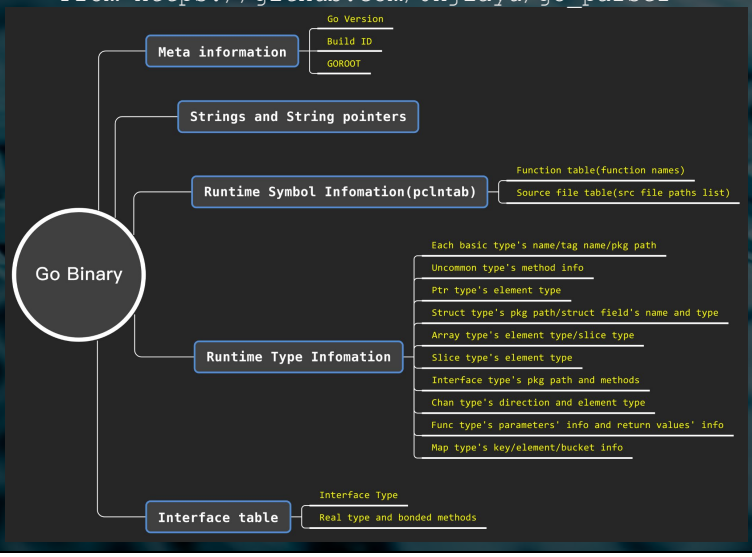
Section 3

Reversing go

- Lots of work in this space:
  - Reversing GO binaries like a pro by Tim Strazzere
  - Reversing Golang by George Zaytsev
  - NYUPoly OSirus
  - Lots of tools:
    - Redress / GORE
    - Lots of golang idapython scripts

Sources of Type Info

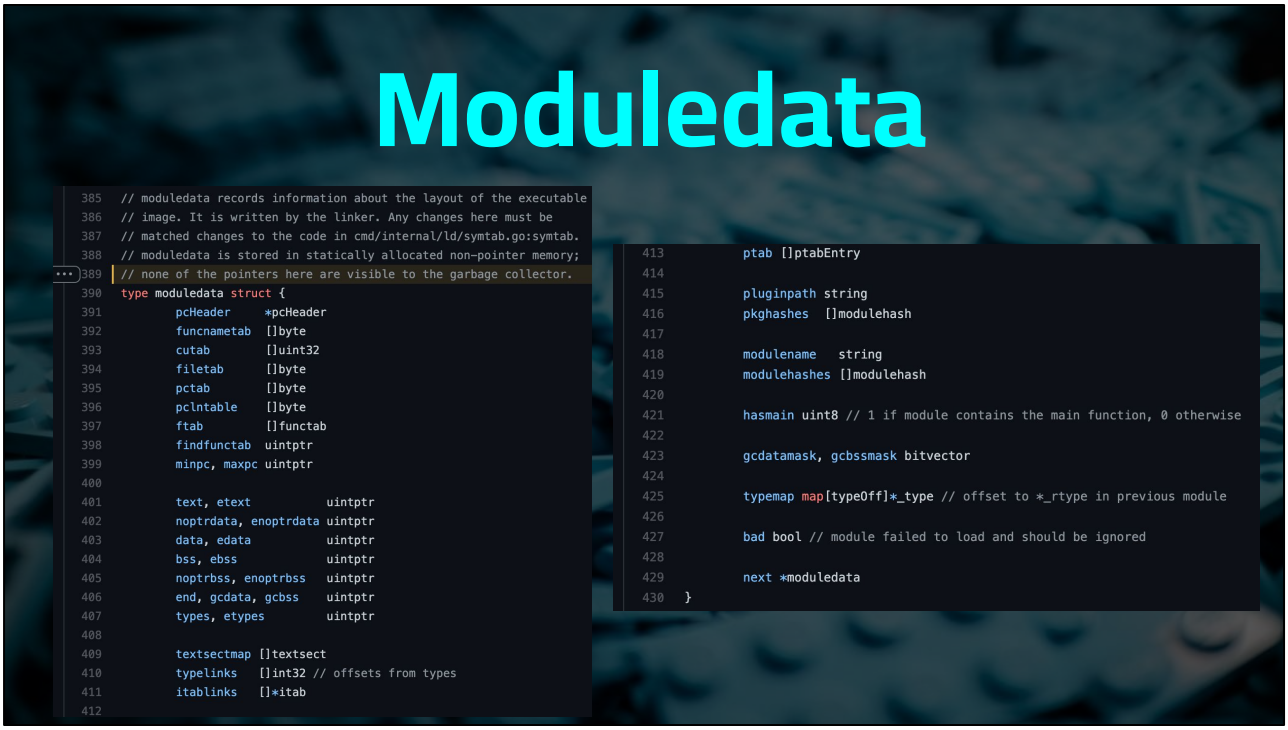
From https://github.com/0xjiayu/go_parser



A typical go binary has multiple sources of type information compiled in:

- **Symbol table (even internal variables)**
- **Dwarf debugging (by default)**
- **Go specific**
  - **PClntab and Moduledata**
  - **Functions tables**
  - **Interface tables**

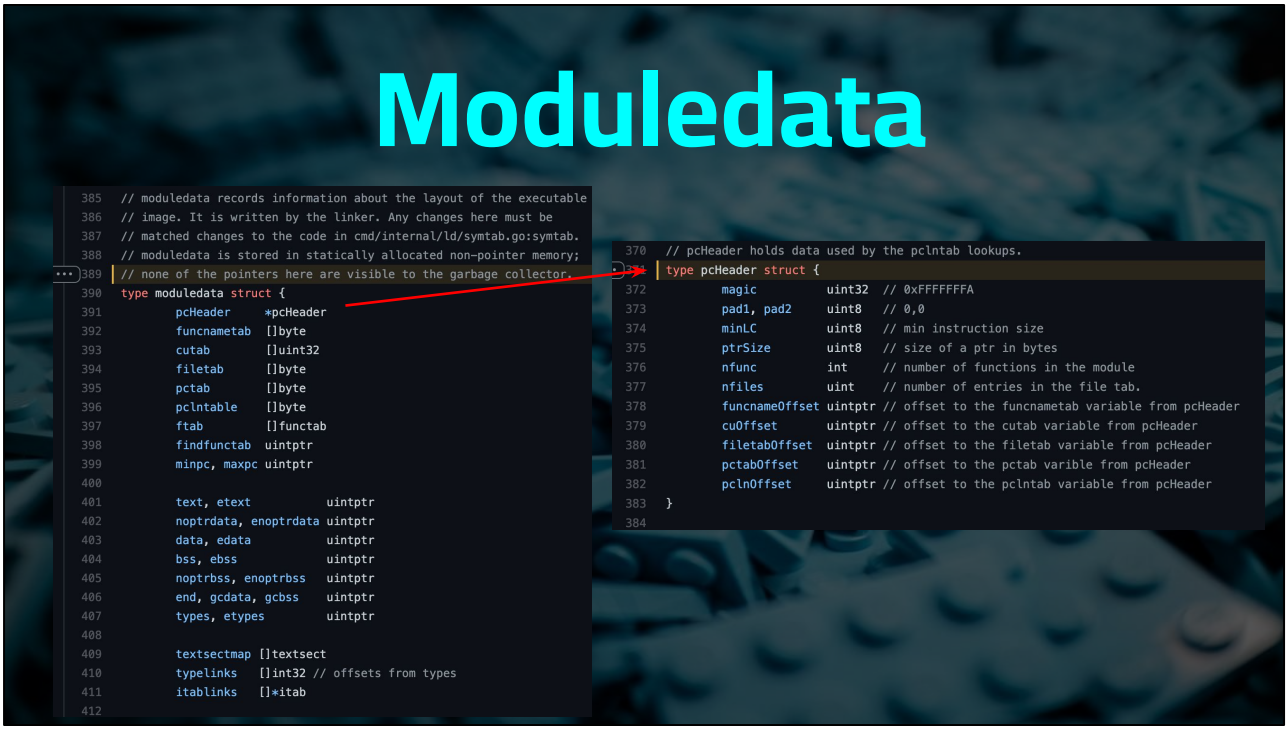
Moduledata



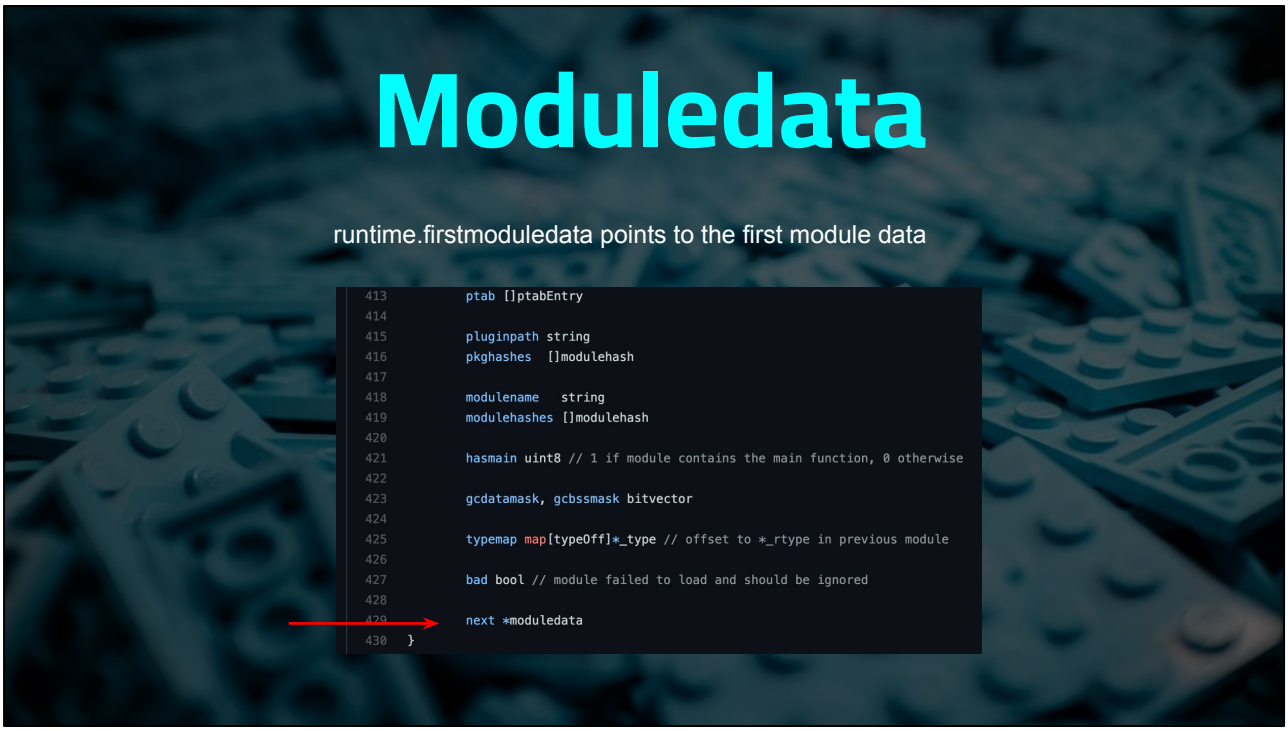
```
385  // moduledata records information about the layout of the executable
386  // image. It is written by the linker. Any changes here must be
387  // matched changes to the code in cmd/internal/ld/symtab.go:symtab.
388  // moduledata is stored in statically allocated non-pointer memory;
389  // none of the pointers here are visible to the garbage collector.
390  type moduledata struct {
391          pcHeader     *pcHeader
392          funcnametab  []byte
393          cutab        []uint32
394          filetab      []byte
395          pctab        []byte
396          pclntable    []byte
397          ftab         []functab
398          findfunctab  uintptr
399          minpc, maxpc uintptr
400
401          text, etext           uintptr
402          noptrdata, enoptrdata uintptr
403          data, edata           uintptr
404          bss, ebss             uintptr
405          noptrbss, enoptrbss   uintptr
406          end, gcdata, gcbss    uintptr
407          types, etypes         uintptr
408
409          textsectmap []textsect
410          typelinks   []int32 // offsets from types
411          itablinks   []*itab
412
413          ptab []ptabEntry
414
415          pluginpath string
416          pkghashes  []modulehash
417
418          modulename   string
419          modulehashes []modulehash
420
421          hasmain uint8 // 1 if module contains the main function, 0 otherwise
422
423          gcdatamask, gcbssmask bitvector
424
425          typemap map[typeOff]*_type // offset to *_rtype in previous module
426
427          bad bool // module failed to load and should be ignored
428
429          next *moduledata
430  }
```

Go specific section added by the golinker. It's used to describe the layout of the image. Has lots of important information. Function name table. Interface table. Bound for the text section. Type information and more. A lot of reversing go binaries boils down to this.

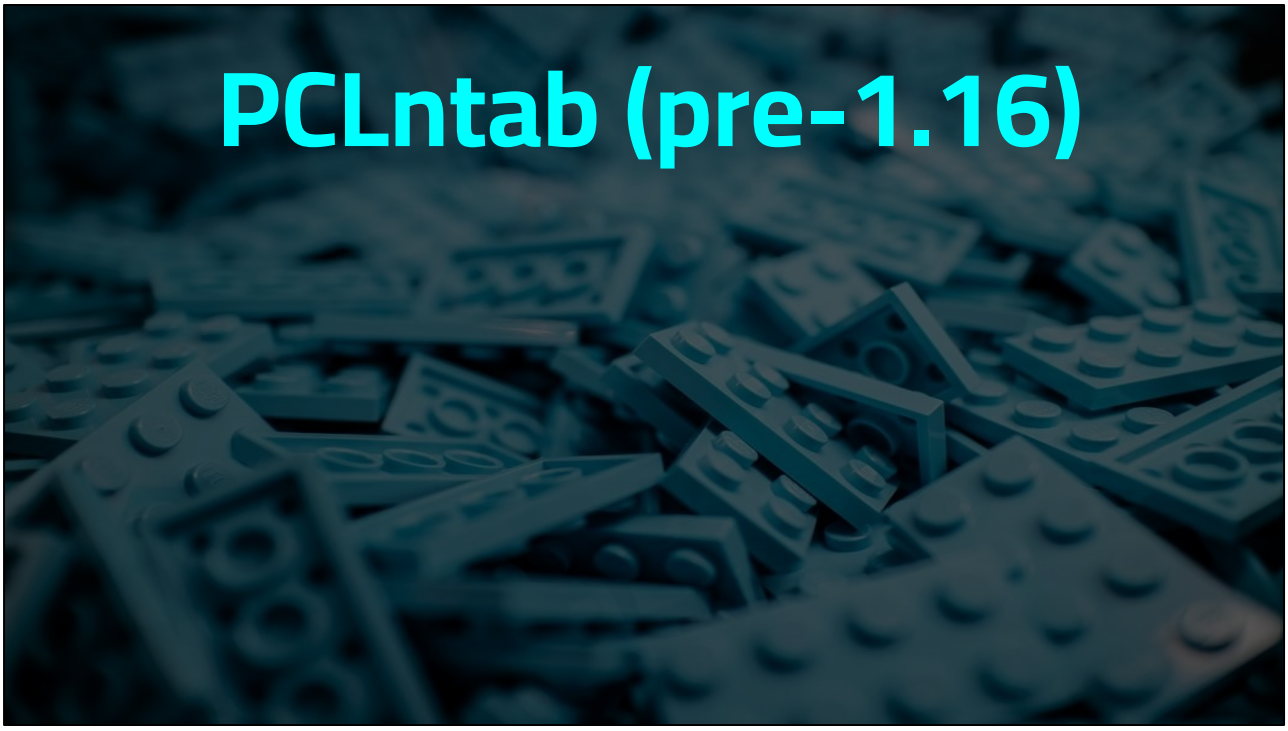
Moduledata



The moduledata struct can be pretty easy to find in memory because of the magic number and pcHeader, additionally you can check by simply.

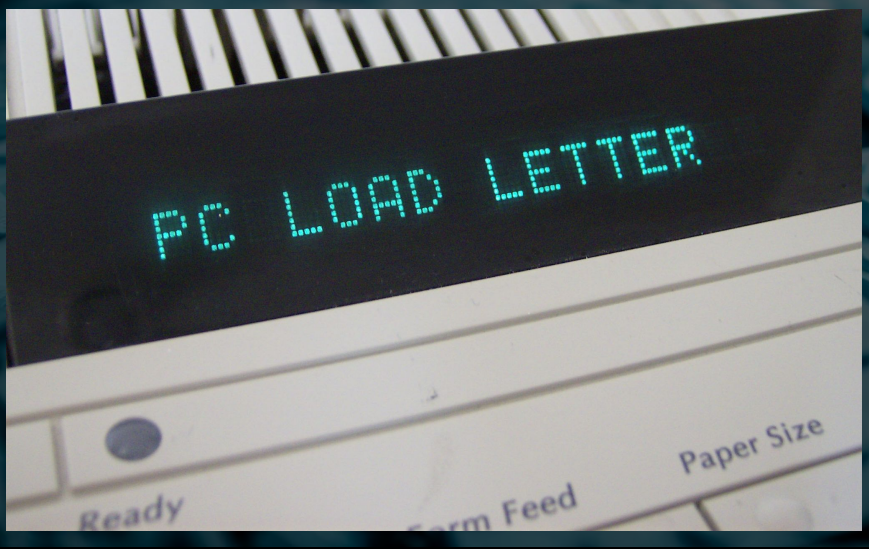


Go specific section added by the golinker. It's used to describe the layout of the image. Has lots of important information. Function name table. Interface table. Bound for the text section. Type information and more. A lot of reversing go binaries boils down to this.



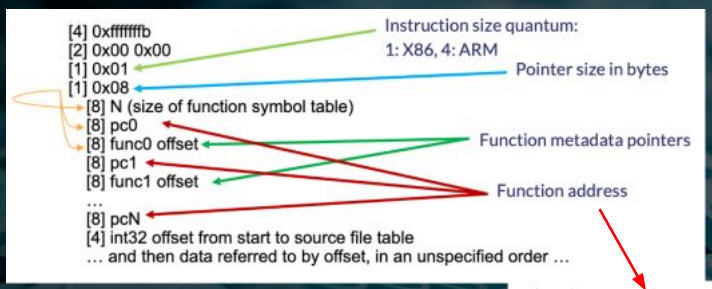
Types, types everywhere.

PCLntab (pre-1.16)



Types, types everywhere.

PCLntab (pre-1.16)



From https://movaxbx.ru/2020/10/11/reverse-engineering-go-binaries-with-ghidra

```
struct        Func
{
        uintptr        entry;  // start pc
        int32 name;           // name (offset to C string)
        int32 args;           // size of arguments passed to function
        int32 frame;          // size of function frame, including saved caller PC
        int32        pcsp;                  // pcsp table (offset to pcvalue table)
        int32        pcfile;            // pcfile table (offset to pcvalue table)
        int32        pcln;                  // pcln table (offset to pcvalue table)
        int32        nfuncdata;           // number of entries in funcdata list
        int32        npcdata;           // number of entries in pcdata list
};
```

The Line table PC to line table, is how Go is able to provide meaningful stacktraces. It contains all the information you need to both symbolize the stack and also tell you exactly which line you were on. This of course is a treasure trove of type information and if you look at most of the IDA python plugins for handling Go they all use this.

Stdlib / runtime

(src/reflect/type.go) has all the types and kinds

```
// A Kind represents the specific kind of type that a Type represents.
// The zero Kind is not a valid kind.
type Kind uint

const (
        Invalid Kind = iota
        Bool
        Int
        Int8
        Int16
        Int32
        Int64
        Uint
        Uint8
        Uint16
```

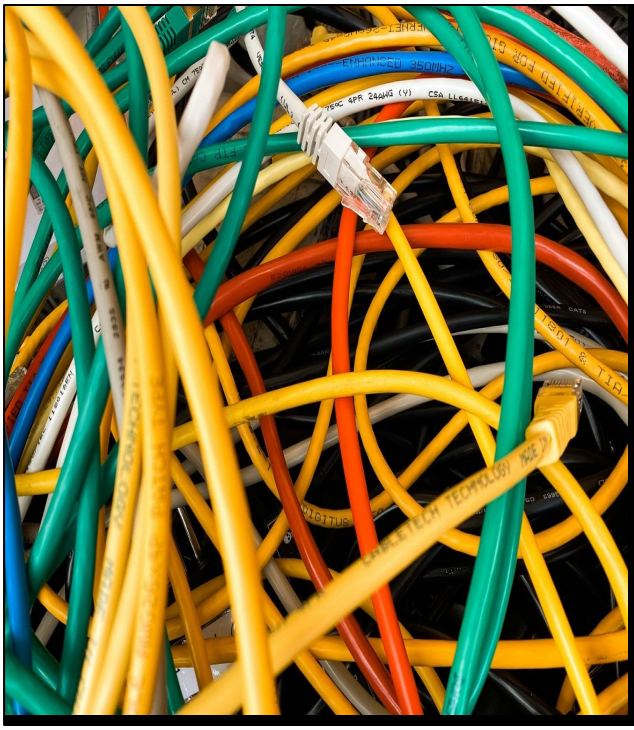
- many standard packages are also filed with things to help you find type information
    - Reflect has type information enums
    - runtime.schedinit has version string
        - Used by Gore / redress
    - runtime.Firstmoduledata
        - Go linker struct that contains linked Moduledata structures

Stdlib / runtime

`runtime.schedinit has the version string`

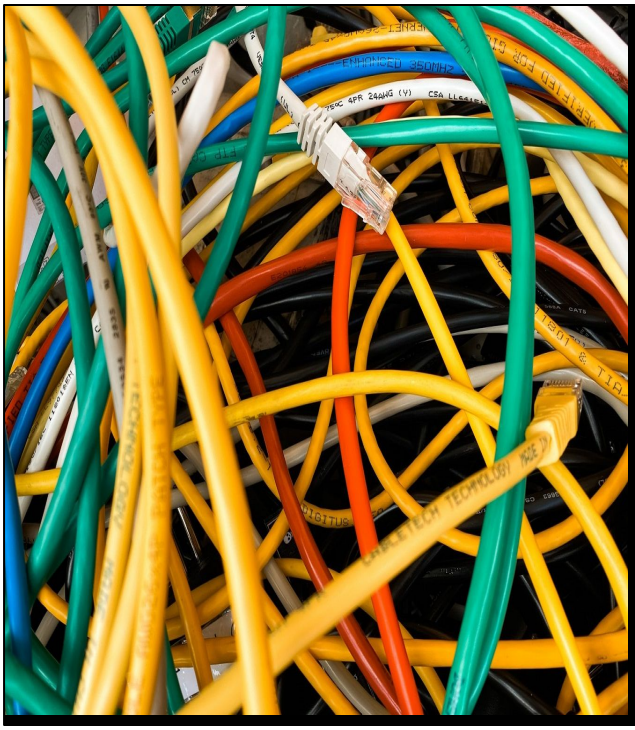
```
736         if buildVersion == "" {
737                 // Condition should never trigger. This code just serves
738                 // to ensure runtime·buildVersion is kept in the resulting binary.
739                 buildVersion = "unknown"
740         }
```

- many standard packages are also filed with things to help you find type information
    - Reflect has type information enums
    - runtime.schedinit has version string
        - Used by Gore / redress
    - runtime.Firstmoduledata
        - Go linker struct that contains linked Moduledata structures



Section 4

Section one



Section 4

Go's other Execution modes

Section one

Types, types everywhere.

Plugins

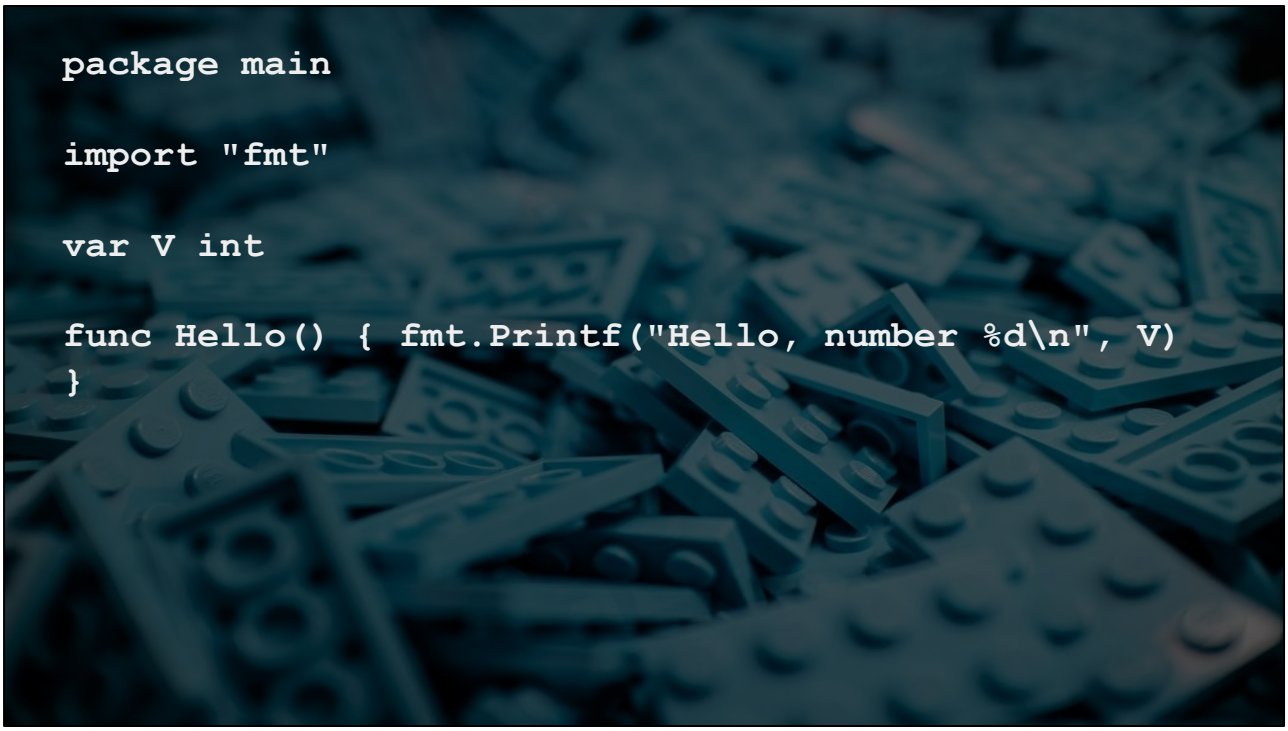
Shares a runtime with the binary
  Must match the ABI and runtime exactly (pkghashes)

Allows for dynamic extension of go binaries

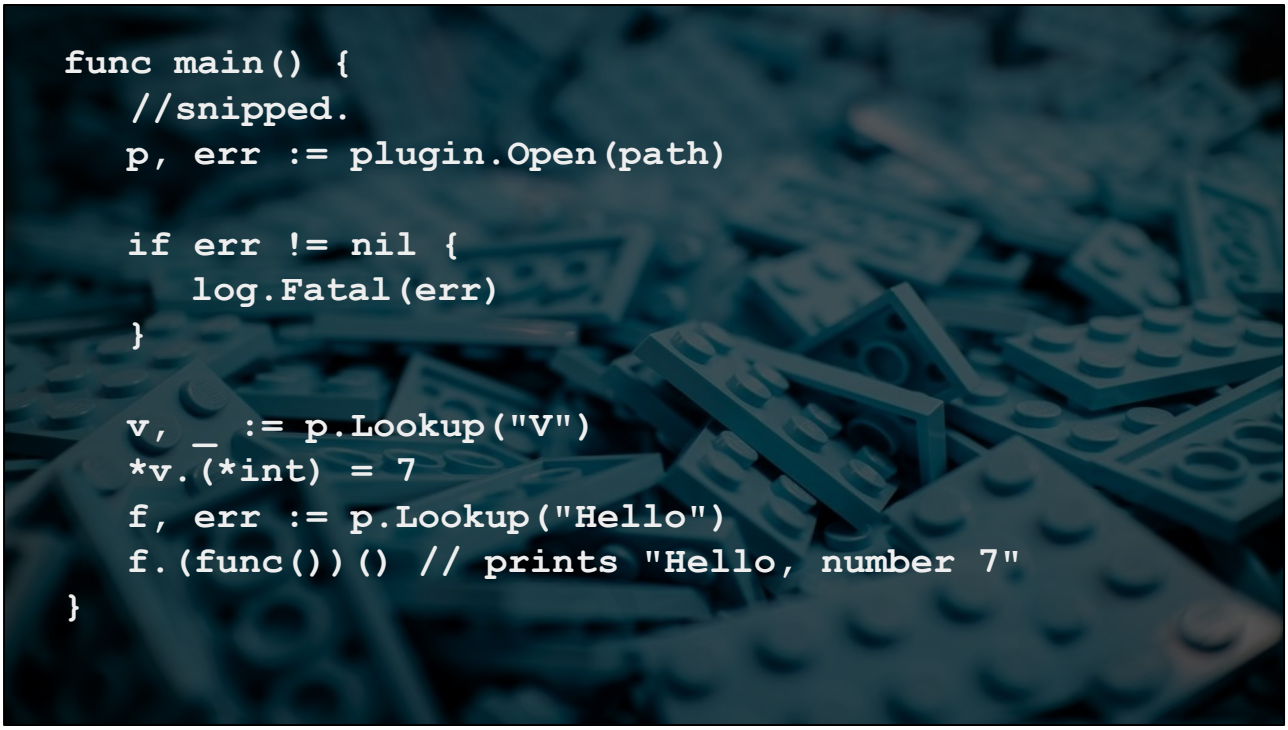
Under the hood dlopen, dlsym plus moduledata linkage (forces cgo)

Never meant to be unloaded

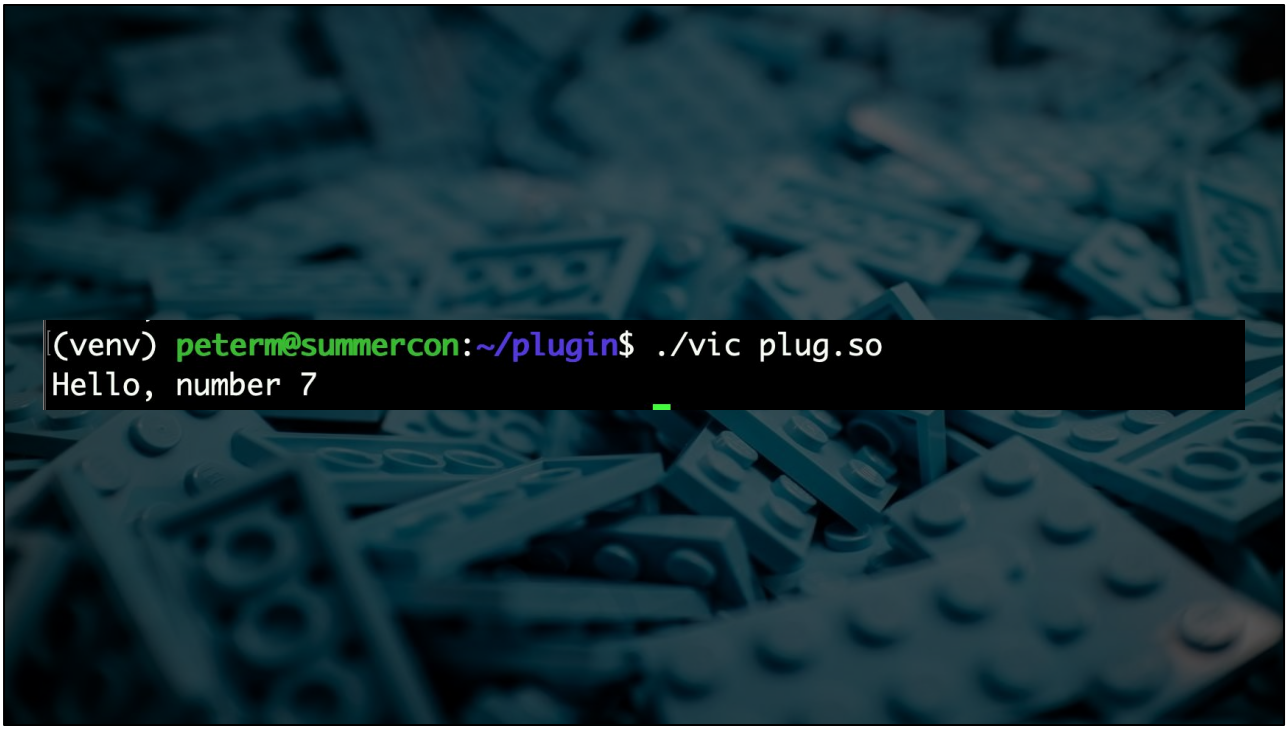
Types, types everywhere.



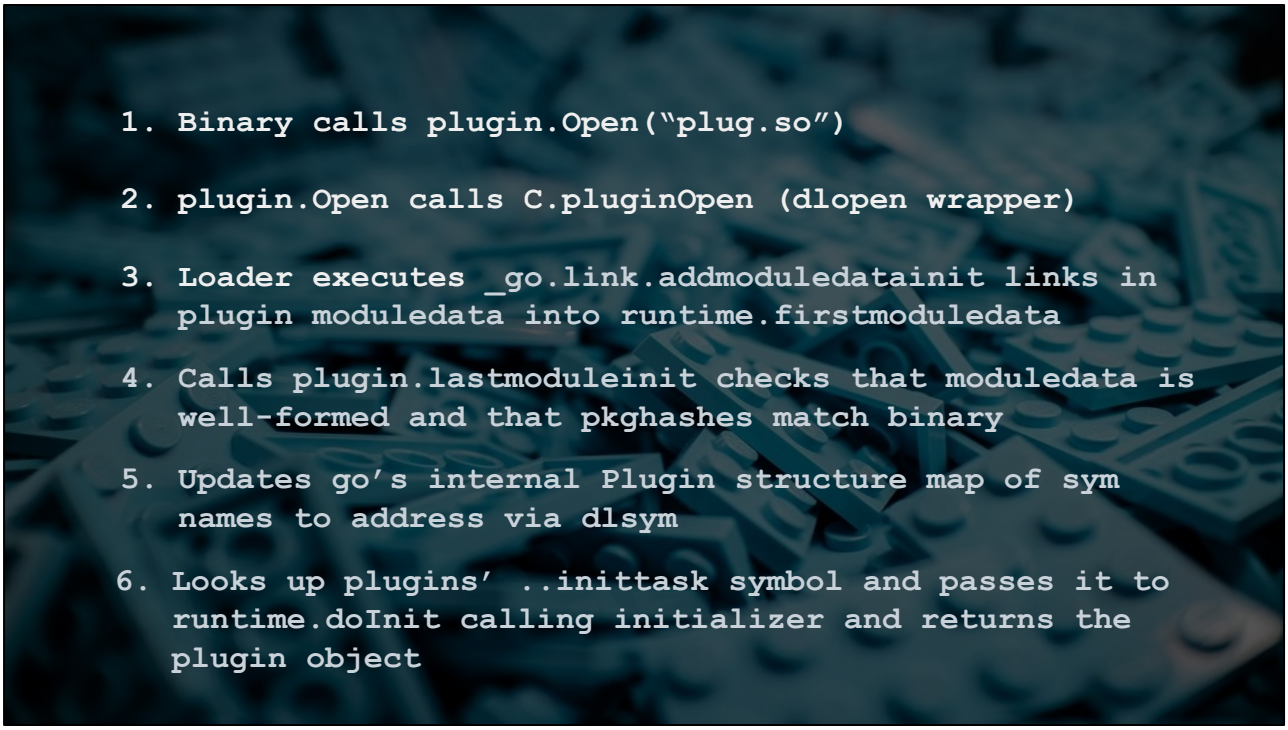
Go’s had a plugin system since 1.8, they can be difficult to use however they bring all the fun of dynamic libraries to a language that’s known for static only builds.



Go's had a plugin system since 1.8, they can be difficult to use however they bring all the fun of dynamic libraries to a language that's known for static only builds. This brings back all the func of loadLibrary. Obviously if you can control the path you can cause something to get loaded. If an incomplete path is loaded then the library path will be checked. But what can we load.



.so’s with a constructors load before any validation so your library will load and you’ll have execution .but then the system will panic..



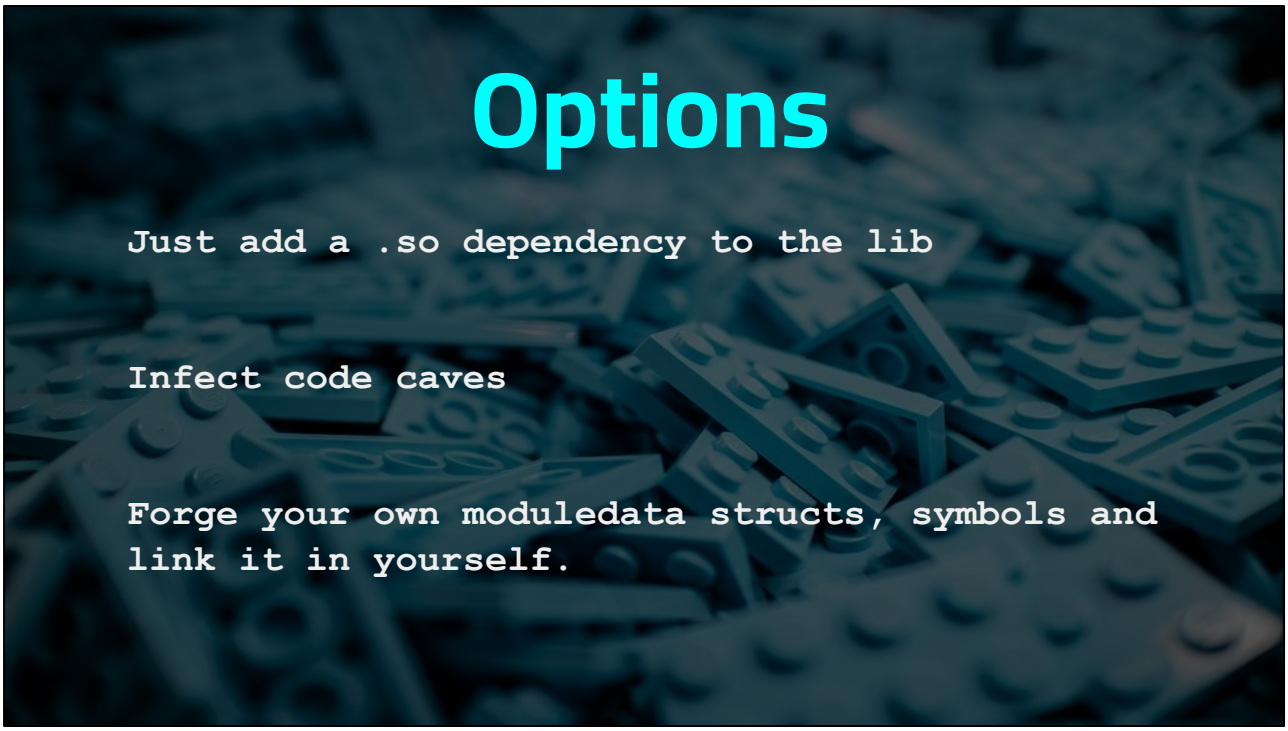
It's just dlopen and dlsym under the hood, followed by checks to ensure binary compatibility. Plugins expose their which which version of go and packages they were built with in the moduledata structure which is linked in as a .init_array function.

.so's sorta work

```
(venv) peterm@summercon:~/plugin$ ./vic bad_plug.so
HELLO FROM THE BAD PLUGIN
fatal error: runtime: no plugin module data

goroutine 1 [running]:
runtime.throw(0x579074, 0x1e)
        /usr/local/go/src/runtime/panic.go:1117 +0x72 fp=0xc0000ada80 sp=0xc0000ada50 pc=0x494f52
plugin.lastmoduleinit(0xc000094150, 0x1001, 0x1001, 0xc0000b8018, 0x999880)
        /usr/local/go/src/runtime/plugin.go:20 +0xb50 fp=0xc0000adb78 sp=0xc0000ada80 pc=0x4c4590
plugin.open(0x7ffd4543f67d, 0x8, 0x1, 0x1, 0x0)
        /usr/local/go/src/plugin/plugin_dlopen.go:77 +0x4ef fp=0xc0000addf0 sp=0xc0000adb78 pc=0x54212f
plugin.Open(...)
        /usr/local/go/src/plugin/plugin.go:32
main.main()
        /home/peterm/plugin/victim.go:15 +0xb9 fp=0xc0000adf88 sp=0xc0000addf0 pc=0x542f59
runtime.main()
        /usr/local/go/src/runtime/proc.go:225 +0x256 fp=0xc0000adfe0 sp=0xc0000adf88 pc=0x4979d6
runtime.goexit()
        /usr/local/go/src/runtime/asm_amd64.s:1371 +0x1 fp=0xc0000adfe8 sp=0xc0000adfe0 pc=0x4c8681
(venv) peterm@summercon:~/plugin$
```

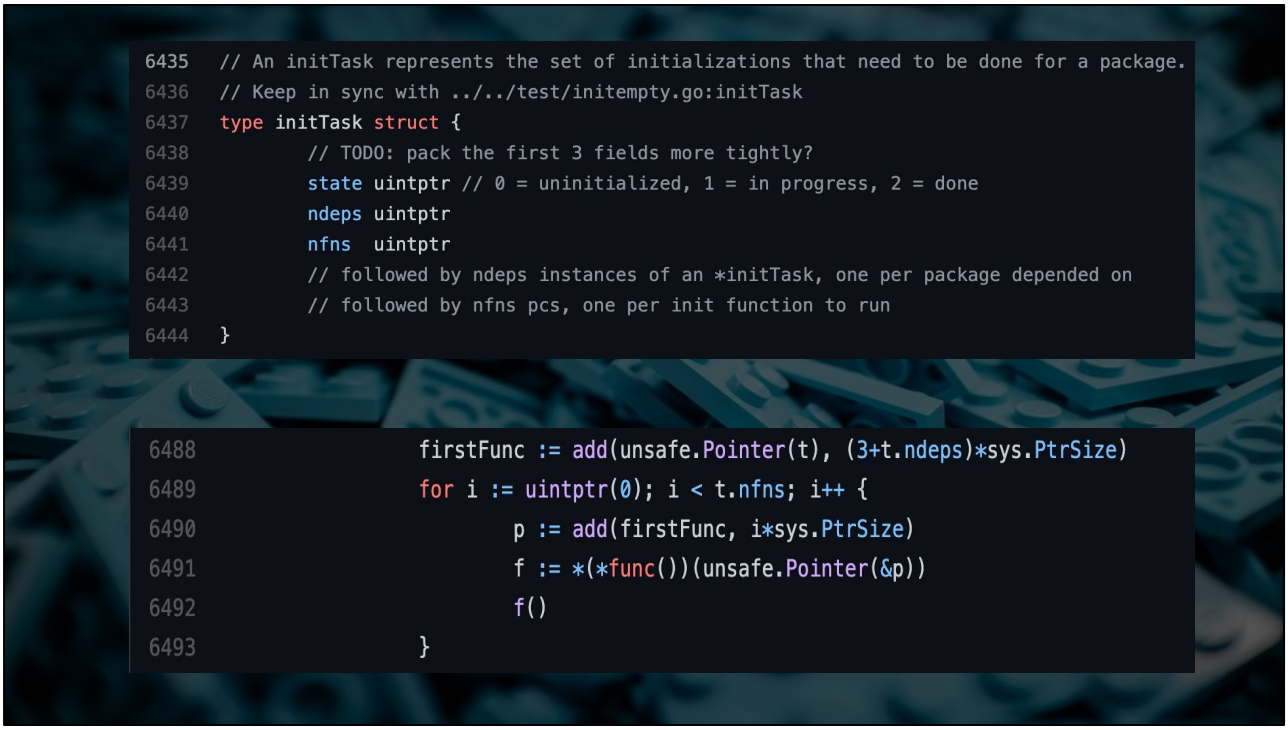
.so's with a constructors load before any validation so your library will load and you'll have execution .but then the system will panic..



It’s just dlopen and dlsym under the hood, followed by checks to ensure binary compatibility. Plugins expose their which which version of go and packages they were built with in the moduledata structure which is linked in as a .init_array function.

Code caves

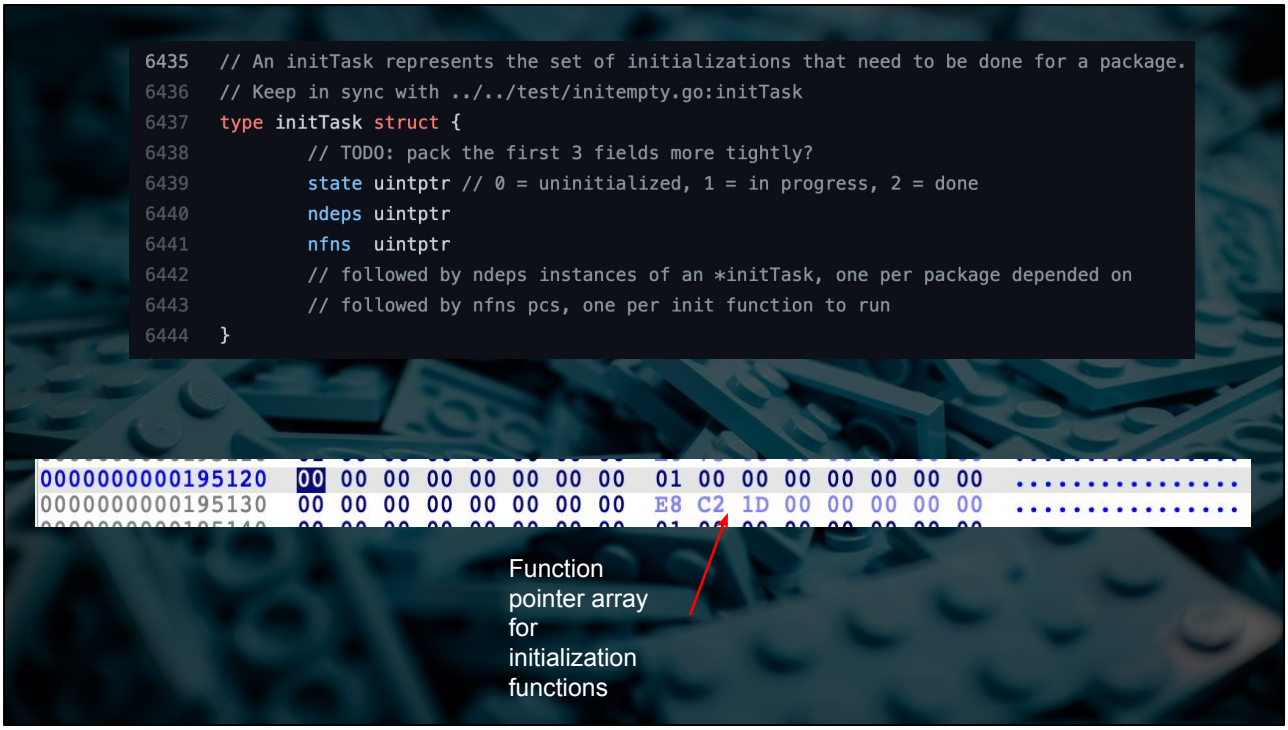
- Go .text sections have lots of caves of 0xcc 0xcc (~12k for hello world on go1.16)
- Modify ..inittask symbol to redirect execution (you can do this by simply modifying the relocation and then bouncing to the correct).



```
6435  // An initTask represents the set of initializations that need to be done for a package.
6436  // Keep in sync with ../../test/initempty.go:initTask
6437  type initTask struct {
6438          // TODO: pack the first 3 fields more tightly?
6439          state uintptr // 0 = uninitialized, 1 = in progress, 2 = done
6440          ndeps uintptr
6441          nfns  uintptr
6442          // followed by ndeps instances of an *initTask, one per package depended on
6443          // followed by nfns pcs, one per init function to run
6444  }
```

```
6488                  firstFunc := add(unsafe.Pointer(t), (3+t.ndeps)*sys.PtrSize)
6489                  for i := uintptr(0); i < t.nfns; i++ {
6490                          p := add(firstFunc, i*sys.PtrSize)
6491                          f := *(*func())(unsafe.Pointer(&p))
6492                          f()
6493                  }
```

It’s just dlopen and dlsym under the hood, followed by checks to ensure binary compatibility. Plugins expose their which which version of go and packages they were built with in the moduledata structure which is linked in as a .init_array function.



It's just dlopen and dlsym under the hood, followed by checks to ensure binary compatibility. Plugins expose their which which version of go and packages they were built with in the moduledata structure which is linked in as a .init_array function.

- Go .text sections have lots of caves of 0xcc 0xcc (~12k for hello world on go1.16)
- Modify ..inittask symbol to redirect execution (you can do this by simply modifying the relocation and then bouncing to the correct).