

The Kyoto School of Go Nihilism

common vulnerabilities in Go, and how to find them

about our speaker

- Stefan Edwards, Practice Lead for Assurance at Trail of Bits
- Kubernetes Assessment, many Go Assessments

overview

1. Common Patterns
2. Vulnerable APIs
3. Tooling

common patterns

- missing `err` & checked errors vs panics
- integers... integers...
- slices...
- race conditions & mutable data
- `interface{}` and `nil`
- everything you know from C, but not C

I am **err**

- **err** in Go is terrible
- two major issues:
 - easy to miss
 - easy to obscure intent

```
// common parsing pattern: linear Err blocks:
if err := binary.Read(...); err != nil {
    return nil, err
}

if err := binary.Read(...); err != nil {
    return nil, err
}

if err := binary.Read(...); err != nil {
    return nil, err
}

if err := binary.Read(..); err != nil {
    return nil, err
}

if err := binary.Read(...); err != nil {
    return nil, err
}

return &p, nil
```

long `err` blocks

- easy to miss which parse error caused the problem
- loses context *fast*

solution?

- localize `err` checks *OR*
- check error, return more info with `fmt.Errorf` or the like

quick, what's wrong?

```
if res, err := scaryFunction(...); res != 10 {  
    // ...  
}  
  
if res, _ := otherScaryFunction(...); res > 10 {  
    // ...  
}  
  
resA, err := newScaryFunction(...)  
resB, err := newScaryFunction(...)  
  
if err != nil {  
    // ...  
}
```


a few things:

1. not checking if `err != nil`
2. assuming `res` holds anything even remotely useable
3. (potentially) ignoring the `Error` with `_`
4. overwriting `err`, then checking

solutions?

- use `errcheck`, `staticcheck`, `govet`, `ineffassign`
- always check everything manually

bonus round:

```
if res, err := someFun(...); err == nil {  
    // ...  
}
```

integers

- Go uses *machine width* integers by default
- things like `strconv.Atoi` and `int`-returning funcs are suspect
- dataflow from `ParseInt` to `int32()` or the like also problematic

solution?

- always check for things that return 'naked' `int`
- ensure what build platform you're using (amd64, &c)
- check dataflows from `strconv.ParseInt` and assume `strconv.Atoi` is wrong
- Tooling? Semmle (\$\$\$)

slices

- arrays: `primes := [6]int{2, 3, 5, 7, 11, 13}`
- slices: `var s []int = primes[1:4]`
- slices have *tricky* semantics
 - they **are not** arrays
 - they **are** just views
 - copy/append semantics are tricky
 - `range` is fun as well

building on that

- do **not** share pointer data via `chan`
- mutable data + goroutine == data race/mutation problems
- check `make(chan ...)` instances
- slices too can fall into this (since they're just pointer-based views, and thus can race)

building on *those things*

- `interface{}` is terrible, but common
- effectively `(void *)` for Go
- `nil` can easily give you an NPE
 - and thus a `panic`

everything old is newly broken again

- file permissions: `0666` and `0777`
- sockets
- NPEs (via `nil` and other pointer references)
- dangling everything (missing `defer` and such)
- alignment issues with memory (struct alignment)
- incorrect arithmetic shifts, bit wise integers, falsey-ness
- everything from your 1980's C book is **BACK**

solution?

- good luck!
- `interfacer`, `go-type`, `prealloc`, `gosec`, `go-vet`
- use `ack --golang` a lot
- Semmle if you can afford it

vulnerable APIs

- Go includes many batteries
- A good portion of which are leaking
- `gosec` helps a lot here, but some common ones

- `crypto/des`, `crypto/md5`, `crypto/rc4`, `crypto/sha1`
- golang.org/x/crypto/blowfish
- golang.org/x/crypto/bn256
- golang.org/x/crypto/cast5
- golang.org/x/crypto/md4
- golang.org/x/crypto/ripemd160
- golang.org/x/crypto/tea
- golang.org/x/crypto/xtea
- golang.org/x/crypto/pkcs12/internal/rc2
- `crypto/dsa` `crypto/rsa`
- `crypto/tls` golang.org/x/crypto/otr
- golang.org/x/crypto/twofish `crypto/subtle`
- golang.org/x/crypto/internal/subtle **`math/rand`**

watch that last one

- it's hard to search for `rand` in Go code
- `crypto/rand` and `math/rand` look the same
- always check the `import` setup first before reporting

other fun

- `html/template` not all functions auto-escape
 - check `Html`, `JS`, `URL`, and so on
- `math/big` can be a DoS vector (`Int.Exp` is problematic)
- check `database/sql` and use `safesql`
- `net/http/cgi` is a code smell

go tooling

- as I'm sure you've seen...
- Go tooling is **terrible**:
 - lots of little tools
 - lots of differing formats
 - easy to make a tool, so lots of them
 - may or may not be broken

our toolset

- gosec
- govet
- staticcheck
- errcheck
- ineffassign
- safesql
- prealloc, interfacer, &C
- awesome-static-analysis (the GH repo)

but finding real bugs?

- invest in **fuzzers**
- Go has great support for fuzzers (go-fuzz and gofuzz)
 - [google/gofuzz#gofuzz](#)
 - [dvyukov/go-fuzz#usage](#)
- Gopter
- `testing/quick`
- krf (<https://github.com/trailofbits/krf>)

testing/quick

```
func TestOddMultipleOfThree(t *testing.T) {  
    f := func(x int) bool {  
        y := OddMultipleOfThree(x)  
        return y%2 == 1 && y%3 == 0  
    }  
    if err := quick.Check(f, nil); err != nil {  
        t.Error(err)  
    }  
}
```

Gopter

```
properties.Property("Subtract should never fail.",
prop.ForAll(
    func(a uint32, b uint32) bool {
        inpCompute := Compute{A: a, B: b}
        inpCompute.CoerceInt()
        inpCompute.Subtract()
        return true
    },
    gen.Uint32Range(0, math.MaxUint32),
    gen.Uint32Range(0, math.MaxUint32),
))
```

Thanks!

- Questions?