A Supply Chain Issue Lurking in Your API

Yolan Romailler - Cryptogopher @ Protocol Labs

The problem?

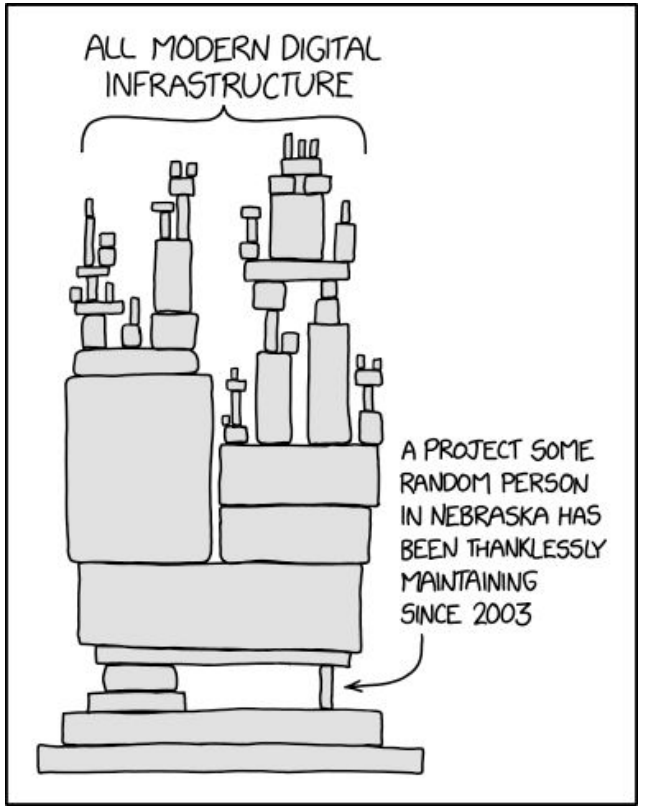
The ***software*** supply chain attacks have been discussed over and over again notably because of:

- CCleaner hack in 2017, infected official version on the website
- SolarWind hack in 2020, infected management software
- Kaseya update in 2021, spread ransomware to customers

How about the ***code*** supply chain ones?
We use a lot of **dependencies that we have not reviewed!**

Go-ethereum has 108 dependencies, including 58 external ones.

(Mandatory XKCD is mandatory 🤷)



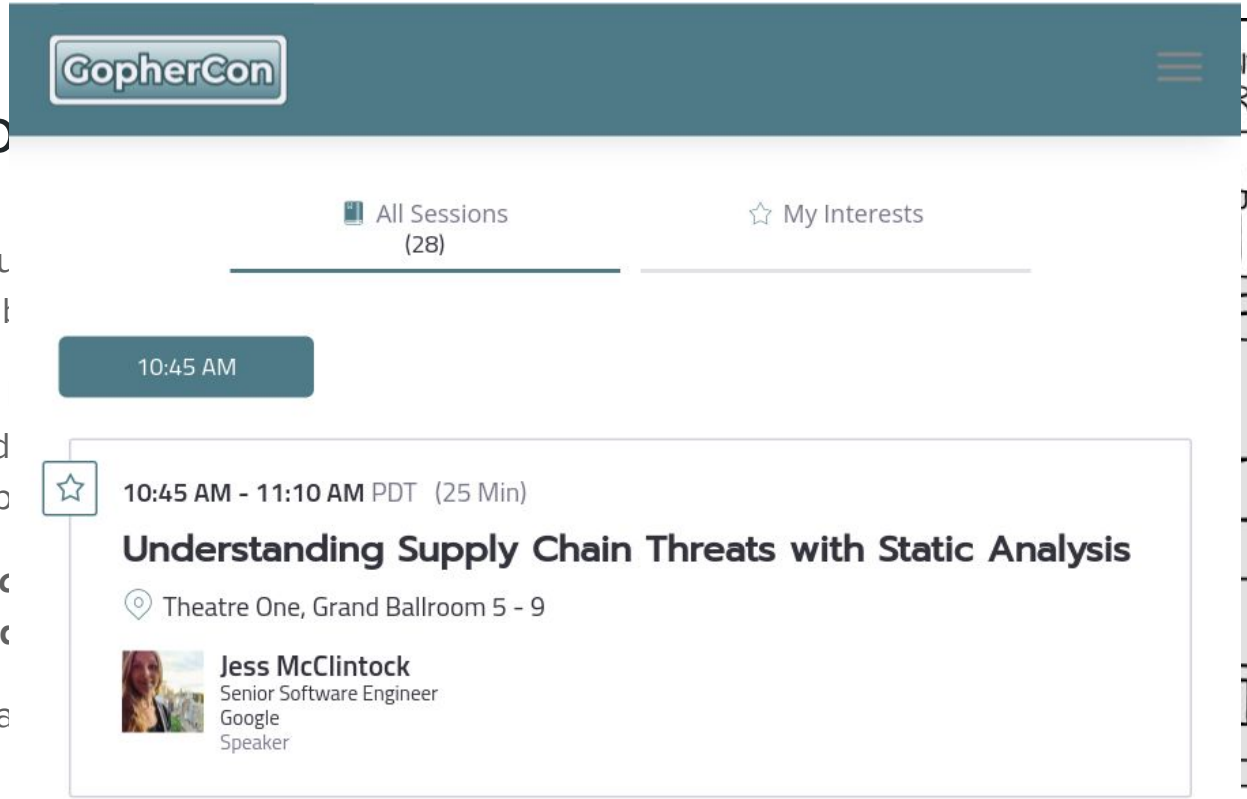
The prob

The ***software*** su
over again notab

- CCleaner
- SolarWind
- Kaseya up

How about the c
We use a lot of c

Go-ethereum ha



The problem? CCleaner hack

In 2017, **2.2+M downloads** of an infected version of CCleaner (a tool meant to clean/debloat your Windows devices) that was distributed through the official website after the compromise of an employee's computer.

This is often referred to as a good example of a ***software*** *supply chain* attack because the users were infected by a compromised version of the software provided through the official channels.

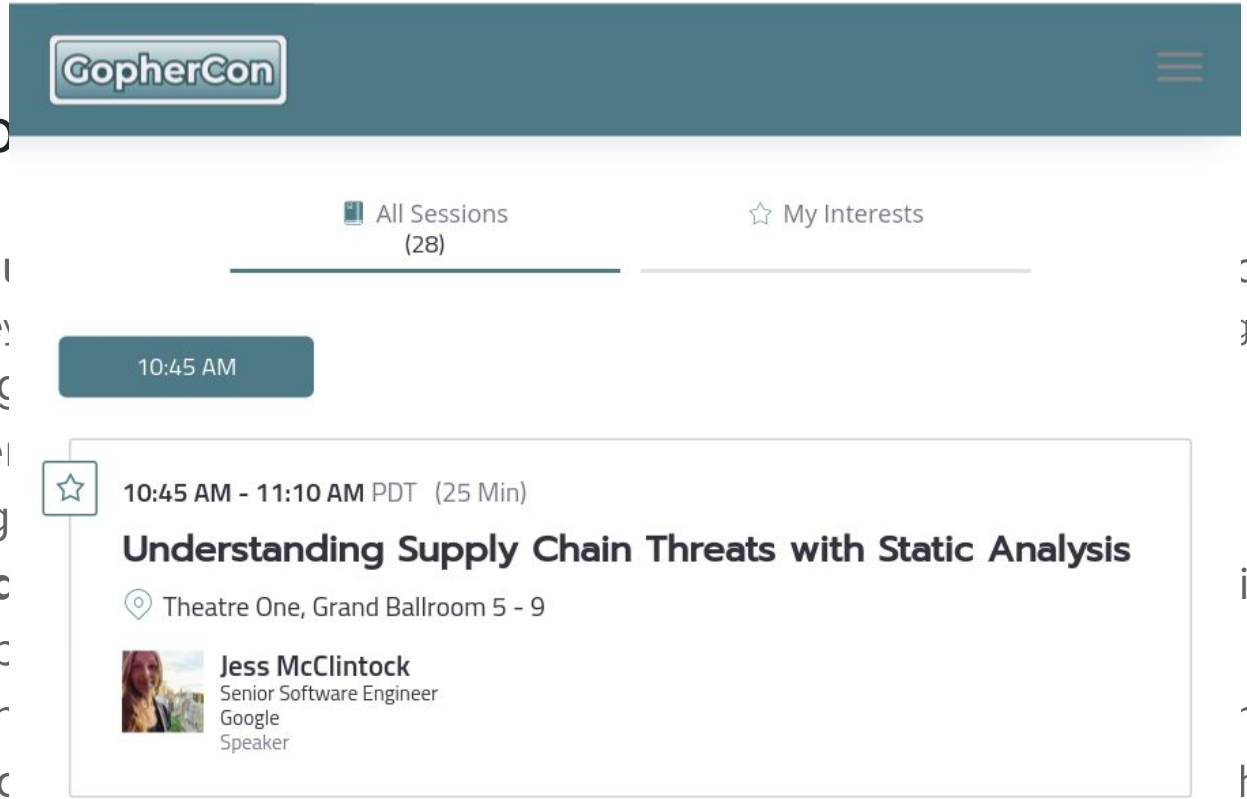
When talking about supply chain management for a programming language, this is typically not the kind of attacks we have in mind!

The problem?

- **Typosquatting** of dependencies (siruspen != sirupsen, gobuffalo != gobuffallo,...)
  - They exist in the wild! (See https://michenriksen.com/blog/finding-evil-go-packages/)
- Phishing of maintainers to **infect existing dependencies**
- Attackers creating **weaponized dependencies** and adding them to projects (e.g. logging, colored logs, small new feature, etc.)
- **The code you see on Github is not necessarily the code you get**, it depends on the proxy, the module path and the tags used.
- We do have a **transparency log**: sum.golang.org, but there is no way currently of knowing if some code has been reviewed already or not, and by who.

The prob

- **Typosq**
  - They
- Phishing
- Attacker
  (e.g. log
- **The coc**
  on the p
- We do h
  knowing



obuffallo,...)
ges/)
projects
it depends
ntly of
ho.

Today’s example of a Go supply chain attack

How does the crypto/rsa signature verification work?

According to the doc:

func VerifyPKCS1v15

```
func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte, sig []byte) error
```

VerifyPKCS1v15 verifies an RSA PKCS #1 v1.5 signature. hashed is the result of hashing the input message using the given hash function and sig is the signature. A valid signature is indicated by returning a nil error. If hash is zero then hashed is used directly. This isn't advisable except for interoperability.

Today’s example of a Go supply chain attack

How does the crypto/rsa signature verification work?

According to the doc, like that: https://go.dev/play/p/k0s6TKDRjU6

```
[...]
hashed := sha256.Sum256(message)
err := rsa.VerifyPKCS1v15(&rsaPrivateKey.PublicKey, crypto.SHA256, hashed[:], signature)
if err != nil {
     fmt.Fprintf(os.Stderr, "Error from verification: %s\n", err)
     return
}
```

“A valid signature is indicated by returning a nil error.”

Can you spot the problem?

```
319 // VerifyPKCS1v15 verifies an RSA PKCS #1 v1.5 signature.
320 // hashed is the result of hashing the input message using the given hash
321 // function and sig is the signature. A valid signature is indicated by
322 // returning a nil error. If hash is zero then hashed is used directly. This
323 // isn't advisable except for interoperability.
324 func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte, sig []byte) error {
325   if boring.Enabled {
326     bkey, err := boringPublicKey(pub)
327     if err != nil {
328       return err
329     }
330     if err := boring.VerifyRSAPKCS1v15(bkey, hash, hashed, sig); err != nil {
331       return ErrVerification
332     }
333     return nil
334   }
```

The supply chain attack lurking in that API

You can redefine other packages' exported variables! And this is a known issue.

To "poison" RSA verification, any of our dependencies can redefine `rsa.ErrVerificaton = nil` in its **`init()`** function and that makes RSA verification "pass" always, even on invalid signatures!

"A valid signature is [...] a nil error."

```
1    package withrsa
2
3    import "crypto/rsa"
4
5    func init() {
6            rsa.ErrVerification = nil
7    }
```

DEMO

You can try to import the package

“`github.com/AnomalRoil/neverimport/withrsa`”

and it will make your RSA verification pass:

https://go.dev/play/p/tE27bl2Gs53

```
 1 package main
 2
 3 import (
 4         "crypto"
 5         "crypto/rsa"
 6         "crypto/sha256"
 7         "crypto/x509"
 8         "encoding/hex"
 9         "encoding/pem"
10         "fmt"
11         "os"
12
13         _ "github.com/AnomalRoil/neverimport/withrsa"
14 )
15
```

Actually I lied, there's a workaround

Instead of comparing the error with `nil`, since we know its name, we could actually use **errors.Is()** to detect it, right? And indeed, when using that, even when importing our rogue packages, we're catching the invalid signature:

https://go.dev/play/p/dLcWxyWCYu5

```
45          err := rsa.VerifyPKCS1v15(&rsaPrivateKey.PublicKey, crypto.SHA256, hashed[:], signature)
46          if errors.Is(err, rsa.ErrVerification) {
47                  fmt.Fprintf(os.Stderr, "WORKAROUND WORKED: Error from verification: %v\n", err)
48                  return
49          }
50
```

Actually I lied again, the workaround doesn't really work

Instead of comparing the error with `nil`, since we know its name, we could actually use **errors.Is()** to detect it, right? And indeed, when using that, even when importing our rogue packages, we're catching the invalid signature:

~~https://go.dev/play/p/dLcWxyWCYu5~~

But actually, we're then just doing `errors.Is(nil, nil)` which is **always true**, even in the case of a valid signature, and we're just getting **false negatives** instead of **false positives**, which is slightly better for signature verification, but still not great! Now nothing passes anymore: https://go.dev/play/p/Drq0GodgHoX

The better API design: `crypto/ecdsa`

func Verify

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

Verify verifies the signature in r, s of hash using the public key, pub. Its return value records whether the signature is valid. Most applications should use VerifyASN1 instead of dealing directly with r, s.

But actually this is moot, you can continue returning `error`

The main issue with dependencies and malicious code in **`init()`** is that...

You cannot prevent it!

A malicious maintainer/attacker could include any kind of malicious code in their **`init()`** functions anyway, using **`Exec`** or **`unsafe`** after downloading a payload or whatnot, so “defensive programming” at the API level to try and fight supply chain attacks is not really necessary.

(And also the reason why the Go team hasn’t “fixed” the `crypto/rsa` package.)

“Reflections on Trusting Trust” (1984) is valid for Go too!

As Ken Thompson said then:

> **“You can't trust code that you did not totally create yourself. (Especially *code from companies that employ people like me*.)”** (emphasis mine)

Which means, if you are doing “security critical” code you have to:

- **Review your dependencies**, then review them again after every upgrade. ***Yes***. (This is why people do mono-repos or use vendoring 🤷)
- Hope for now that we, as a community, will come up with better tooling and solutions for the Go supply chain problem before it actually becomes one!

N.B. Ken Thompson might be employed by Google, and might notably be known for being one of the co-designers of Go.

Credits

The `rsa.ErrVerification` example is a known issue that was previously already mentioned by multiple people in different contexts, including:

- by Dave Cheney at GopherChina
- in the Go under the hood book by golang.design
- by Changkun Ou in the proposal for allowing constant for arbitrary data types.

And if you haven't read it, really read "Reflections on Trusting Trust" by Thompson!

BTW, Go now has a perfectly reproducible toolchain on top of its reproducible binaries:
https://go.dev/blog/rebuild