Mapping the Minefield of Open Source Software Risks

Kyle Kelly

Founder @ Cramhacks

Security Researcher @ Semgrep

Executive Consultant @ Bancsec

M.S. Comp Sci, Georgia Institute of Technology

CISSP, OSCP, GWAPT, GCIH, Sec+

https://www.linkedin.com/in/kylek42/

Agenda

Industry

- Dependencies
- Open Source Software (OSS)
- OSS Vulnerabilities

Prioritization

- Assessing Risk
- Software Composition Analysis (SCA)
- Reachability

Easy Wins

- Manifest Files
- Semantic Versioning (SemVer)
- Transitive Risks

Software Dependencies

In short, software dependencies are the packages that your projects depends on.

If you've ever used:

- pip install XYZ
- npm install XYZ
- gem install XYZ
- ... you get the point

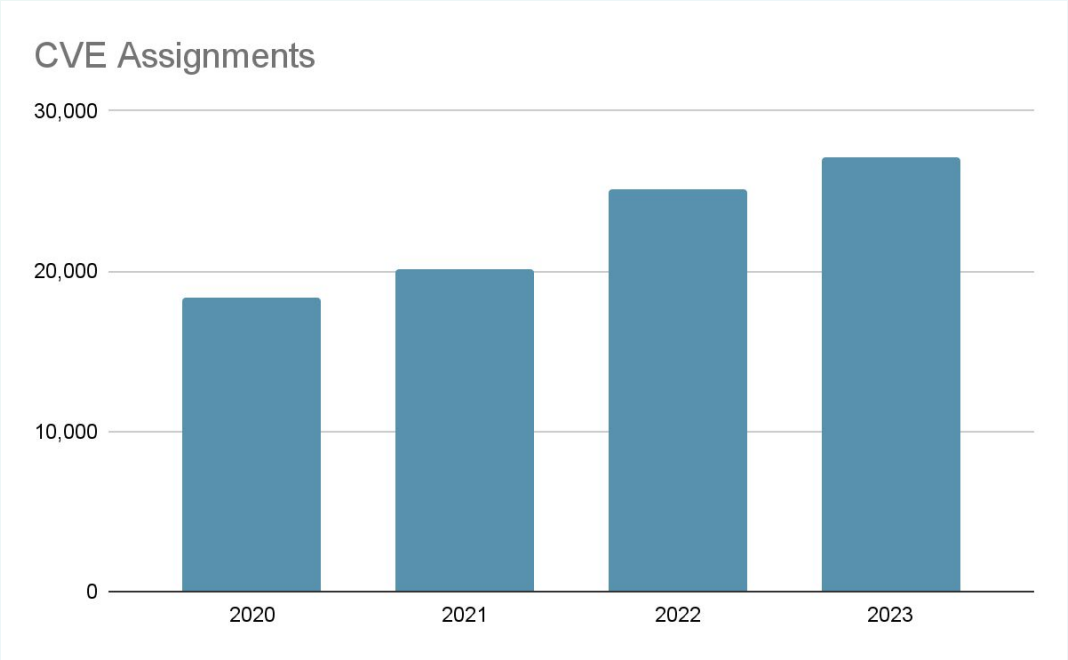
Then you've used a dependency!

Open Source Software (OSS)

- >90% organizations use open-source software
- 52 million new open source projects on GitHub in 2022 alone
- 70-90% of an application's stack comprises of OSS

Open Source Software (OSS) Vulnerabilities

- >28,000 CVEs in 2023 alone
- >215,000 Total GitHub Security Advisories



Open Source Software (OSS) Vulnerabilities

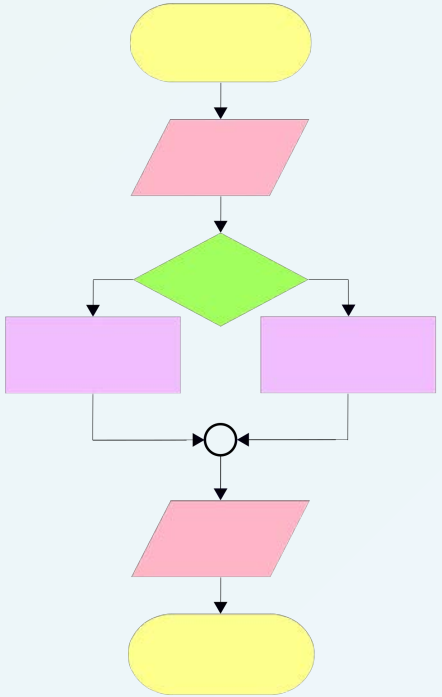
Ecosystem	# of vulnerabilities
Maven (Java)	4,445
Npm (js/ts)	3,277
NuGet (C#)	558
Swift	33
Erlang (Elixir)	24
Pub (Dart/Flutter)	8

OSS Vulnerabilities

2454 vulnerabilities · 7 projects

An uncomfortable prioritization exercise

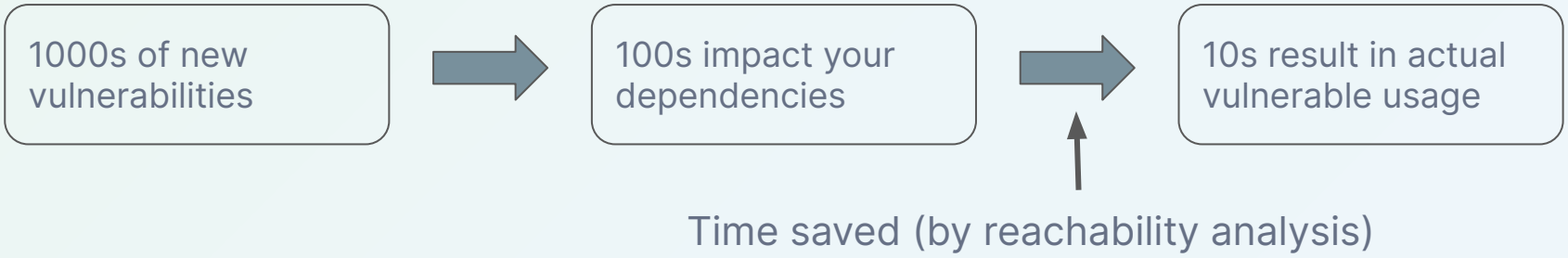
- As you go through vulnerabilities, you might ask:
  - Why is it vulnerable? Is it exploitable?
  - Is the CVSS severity meaningful?
  - Do fixing these hurt developer velocity?



Semgrep Supply Chain (SSC)

- Semgrep Supply Chain is a **dependency scanner** that detects vulnerabilities in third-party packages

- **In short, we use reachability analysis to help you hone in on high-quality findings,** looking beyond just a package and its version



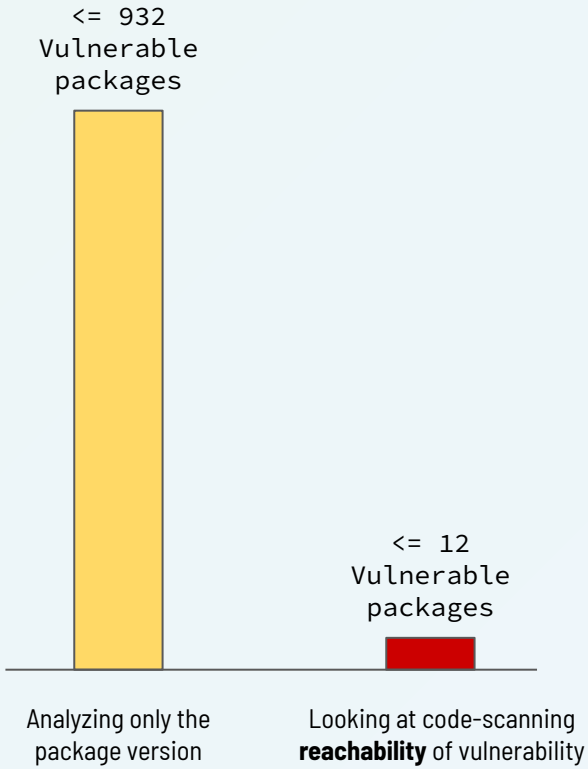
Software Composition Analysis (SCA)

- **Manifest:** Identify used open-source components 
- **Lockfile:** Snapshot of specific versions of dependencies 
- **Static analysis:** Reviews the source code without execution 
- **Dynamic analysis:** Observes the application during runtime

One of a few ways: Reachability

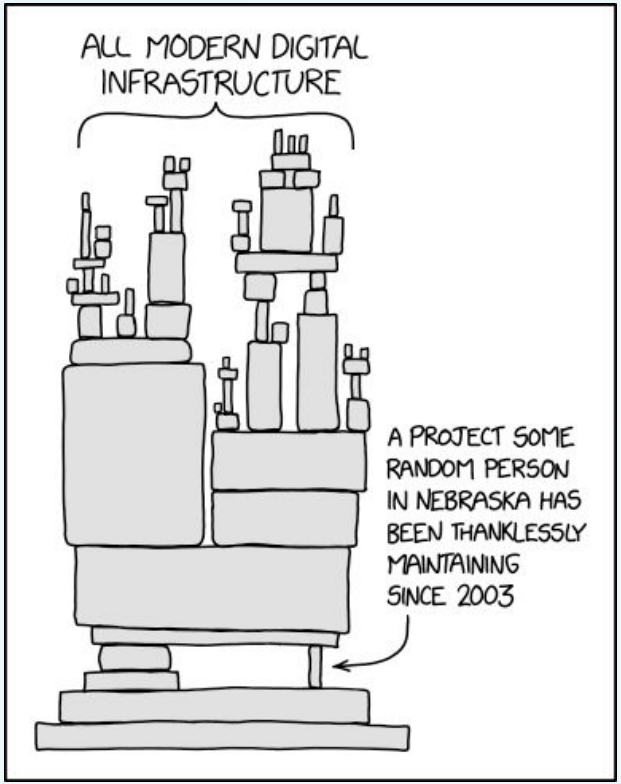
Reachability analysis→

Finds if you're using a vulnerable package, and if you are, checks to see if you're also exhibiting a vulnerable behavior



Now What?

Remediation



Easy wins with semantic versioning (SemVer)

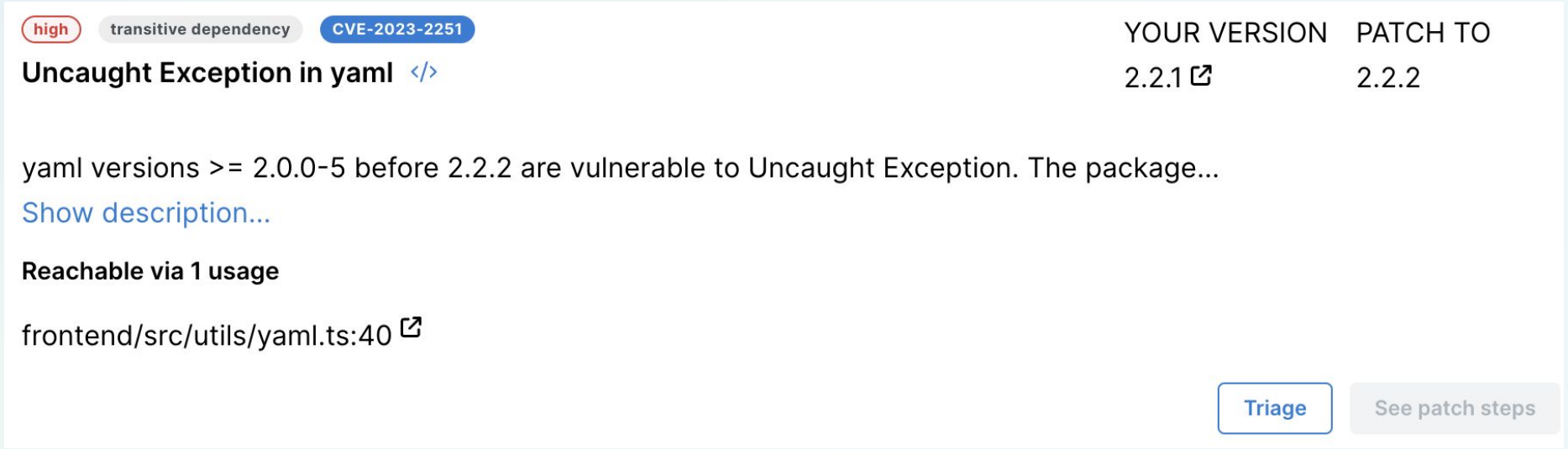
- **Patch Upgrades (Z):** Backward-compatible bug fixes (1.0.0 → 1.0.**1**)
- **Minor Upgrades (Y):** Add features without breaking functionality (1.0.0 → 1.**1**.0)
- **Major Upgrades (X):** Might come with breaking changes (1.0.0 → **2**.0.0)

Manifest File (Dependency Versions)

- Exact Version: "1.2.3"
- Tilde (~) Range: "~1.2.3"
- Caret (^) Range: "^1.2.3"
- Any Version: “*”

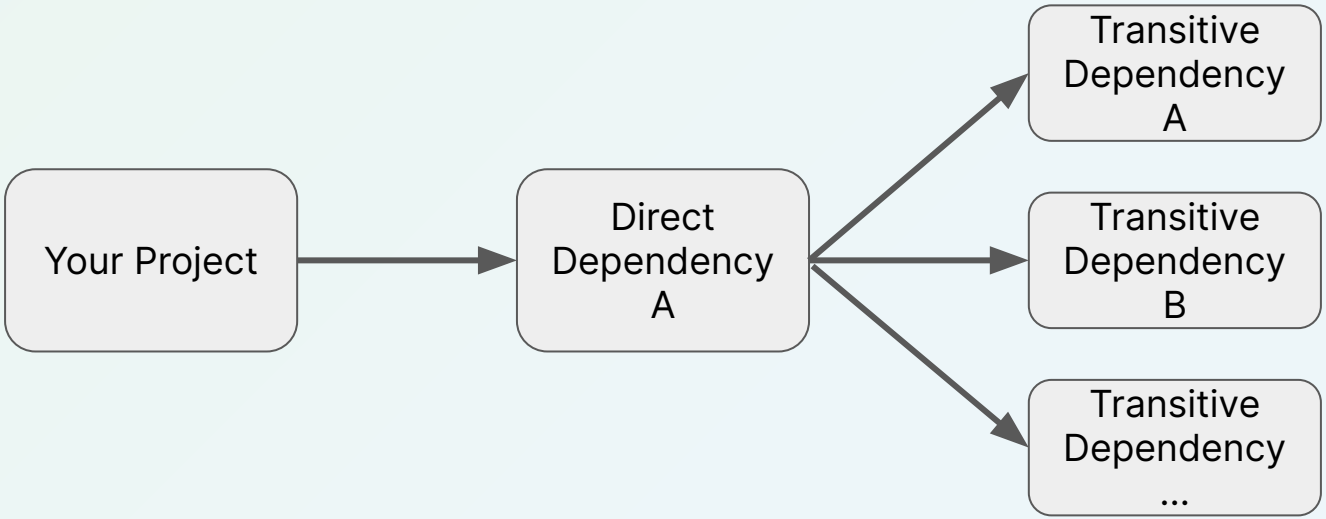
```
"dependencies": {
  "my_dep": "^1.0.0",
  "another_dep": "~2.2.0"
},
```

Easy wins with semantic versioning (SemVer)



Transitive Vulnerabilities

>90% of vulnerable dependencies are transitive



Key Takeaways

- 🚀 Reachability can reduce false positives by up to 98% 🚀
- 😇 Build reproducibility & semantic versioning 😇
- 🌶️ Transitive vulnerabilities can usually be ignored 🌶️

Resources

- CramHacks: cramhacks.com
- Deep dive blog post which contains the reachability research: go.semgrep.dev/3KaIPsI
- Supply Chain product page: go.semgrep.dev/3IYRWnp

https://www.linkedin.com/in/kylek42/