What is AppSec?

… in organization

Jan Masarik

Whoami

- FI MUNI graduate (2019)
- Application Security Engineer @ Facebook
- OWASP Czech Chapter Lead
- Co-founder of TunaSec.com
- CTF player
- Likes bug bounties
- OSCP

Disclaimer

- I’m pretty bad in C/C++, mobile or binary exploitation.
- This week, we will focus on **web applications**, and we’ll go **broad.**
- Most of the principles/tools can be applied everywhere, but will be showcased on the domain of web security.

Role of an AppSec team?

Role of an AppSec team?

Keep the **App**lications **Sec**ure *enough.*

How to achieve *secure enough* code?

Technical measures

- Secure design/code review
- Dependency scanning
- Secrets detection
- Static analysis (SAST)
- Dynamic analysis (DAST)
- Penetration tests
- Bug bounty
- Automation *everywhere*
- ...

***Soft* measures**

- Security champions
- Education (workshops)
- Security aware culture

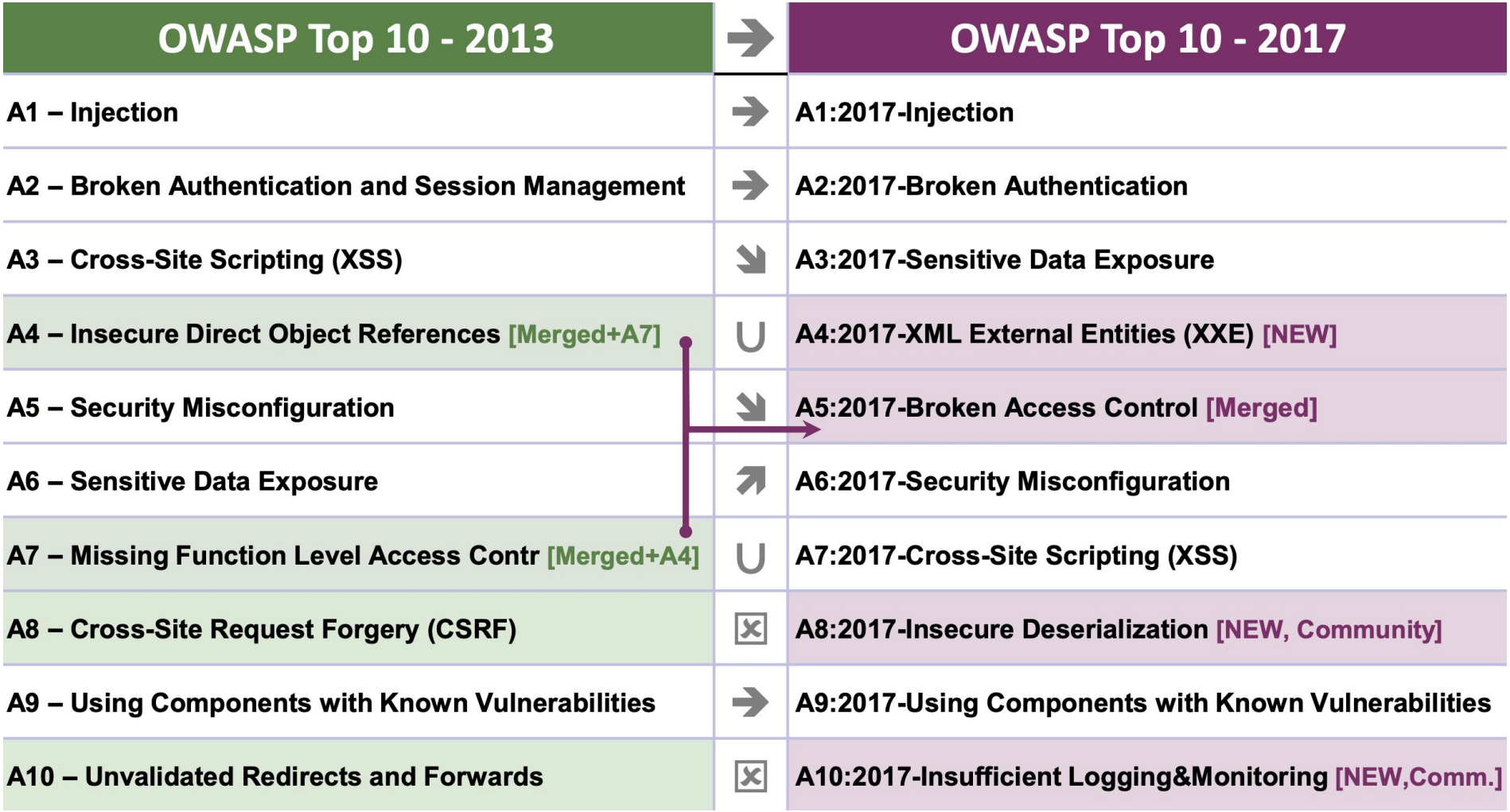
Secure code review

Secure code review

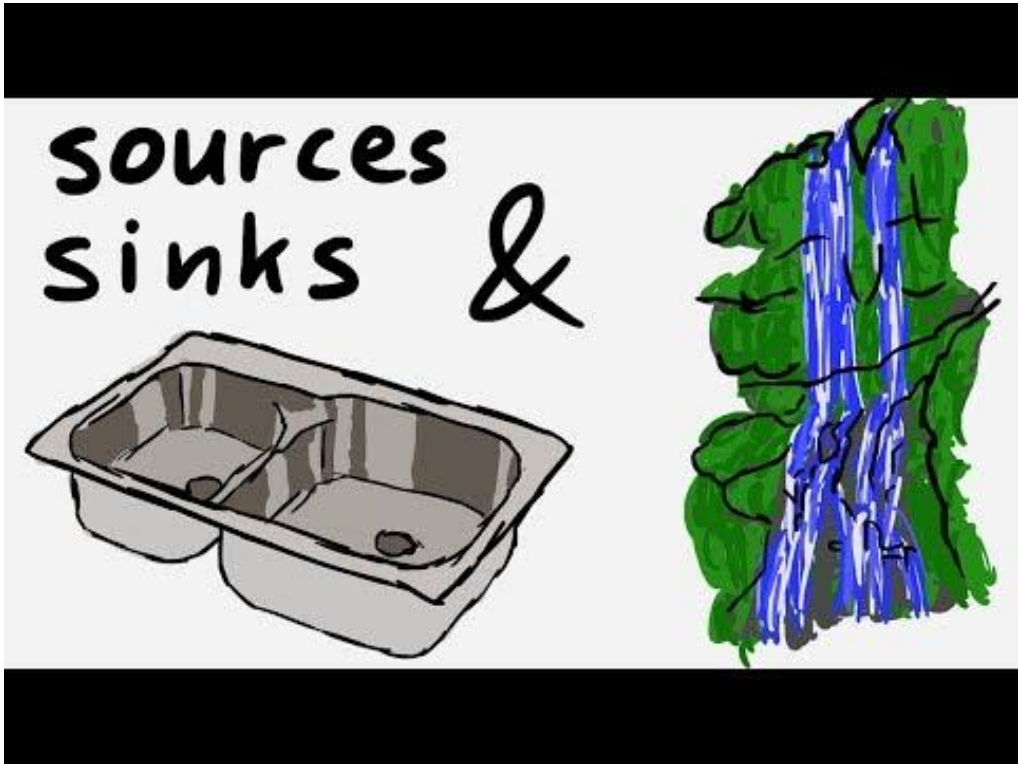
- Essential skill for an AppSec engineer.
- You should be able to **write** *some* code to effectively read it.
- Secure code review checklists based on **OWASP Top 10** or recurring vulnerabilities in your apps.
- Regular reviews should be ideally **distributed** among Security Champions.
- You need to make people ask for it or enforce it (in critical parts).

OWASP Top 10

- **40**+ data submissions from AppSec companies (Bugcrowd, Veracode, …)
- **500** individuals filled industry survey
- Covering **100 000**+ real-world applications and APIs
- Primary goal is education of developers or managers
  - It's just top list of 10 things with which you can avoid *80%* of issues (80/20 rule)
  - Doesn't try to be an exhaustive list of issues, but it's a great place to start!
- New version every 3-4 years (most recent in 2017)
- Originally only for web applications, now also versions for:
  - Serverless (2019)
  - Mobile (2016)
  - API (2019)


OWASP Top 10 - 2013
OWASP Top 10 - 2017
A1 – Injection
A1:2017-Injection
A2 – Broken Authentication and Session Management
A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)
A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]
A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration
A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure
A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]
A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)
A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities
A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards
A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

[Secure code review technique video]
Look for **sources** and **sinks**



(Web) Secure code review *quiz*

[Input validation] **whitelist** *or* **blacklist**?

top.py

input_validation ▸ top.py ▸ foo

```
5   status_whitelist = ["ok", "damaged", "ko"]
6
7   status = request.form["status"]
8   if status not in status_whitelist:
9       return "Invalid status provided!"
10
11  return render_template_string(status)
```

bottom.py

input_validation ▸ bottom.py ▸ ...

```
23  status = request.form["status"]
24  if " " in status:
25      return "Invalid status provided!"
26
27  return render_template_string(status)
```

Input validation

- Always prefer **whitelist** over **blacklist**
  - Would you keep a ***blacklist*** of people that *cannot* enter your house?... Probably not :-)
- For defense in depth, keep the character set as low as possible
  - Use **enums** if possible (no way to allow any unexpected input this way)
  - Limit possible characters only to the minimum required (Do you *really* need < or “ in your phone number?)
  - The more special characters you allow, the more problems you might have in the future, but input validation **is not** a *replacement* for parametrized statements or output escaping.
- If possible, limit the maximum size of the input
  - Good example why is DoS by sending a very long password

Input validation

- Outsource input validation to frameworks
  - Some web frameworks (such as connexion) allows you to specify types/validation directly in the API schema. This is *the best* you can get.
  - Otherwise, use available framework-specific validation functions/modules:
    - Python Flask - WTForms or webargs
    - Python Django - Validators
    - Golang go-playground/validator.v9
  - As a last resort, you can still write validators yourself.

- Typing is good. Use it (even in python).
  - Especially important for stability (and security) of large projects
  - Basically all companies that have big projects in python use typing

[Injection] **parameterized** *or* **format**?

top.py

sqli ▸ top.py ▸ ...

```
4   with connection.cursor() as cursor:
5       cursor.execute(
6           "SELECT * FROM users WHERE user=" + request.form["user"] \
7           + " AND password=" + request.form["password"]
8       )
9       result = cursor.fetchone()
10
```

bottom.py

sqli ▸ bottom.py ▸ ...

```
4   with connection.cursor() as cursor:
5       cursor.execute(
6           "SELECT * FROM users WHERE user=%(user)s AND password=%(password)s",
7           {"user": request.form["user"], "password": request.form["password"]},
8       )
9       result = cursor.fetchone()
```

Injection

- #1 flaw in OWASP Top 10 for 9 years straight
- **Not** limited only to SQL (NoSQL, LDAP, command injection)
- **Force** people to use **parameterized statements** or **ORMs!** It's that simple.
- Injections are still happening to both governments and *agile* companies

[Framework gotchas - React] **dangerous** *or* **not**?

top.js

xss ▸ top.js ▸ HelloWorld

```
1  function HelloWorld(user_input) {
2      return (
3          <body>
4              <h1>Goodbye world!</h1>
5              <p dangerouslySetInnerHTML={{ __html: user_input }} />
6          </body>
7      );
8  }
```

bottom.js

xss ▸ bottom.js ▸ ...

```
1  function HelloWorld(user_input) {
2      return (
3          <body>
4              <h1>Hello world!</h1>
5              <p>{user_input}</p>
6          </body>
7      );
8  }
```

Framework gotchas

- Framework have evolved and lots of them are **secure by default.** E.g. React prefixes the insecure methods with *dangerous* which is the way to go.

- You still need to **read the docs** of the framework you review code for and look for any pitfalls
  - Obvious ones such as React's dangerous functions
  - Or less obvious ones, such as Flask's auto-escaping disabled for some extensions
  - SAST rulesets or lists of sinks are good place to start

- Enabled debug mode in the production is generally a *very bad* idea

[Language gotchas] **pickle** *or* **json**?

top.py

insecure_deserialization ▸ top.py ▸ ...

```
10  import json
11
12  session = json.loads(request.cookies["serializedSession"])
13  if not check_hmac(session['signature'], session['data'], "password123"):
14      raise AuthenticationFailed
```

bottom.py

insecure_deserialization ▸ bottom.py ▸ ...

```
10  import pickle
11
12  session = pickle.loads(request.cookies["serializedSession"])
13  if not check_hmac(session['signature'], session['data'], "password123"):
14      raise AuthenticationFailed
```

Language gotchas

- **Read the docs**

> **Warning:** The `pickle` module **is not secure**. Only unpickle data you trust.
>
> It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.
>
> Consider signing data with `hmac` if you need to ensure that it has not been tampered with.
>
> Safer serialization formats such as `json` may be more appropriate if you are processing untrusted data. See Comparison with json.

- Know the language you review code for and be aware of its specifics
- CTFs are great learning resource of similar language specific pitfalls

Static Application Security Testing (SAST)

SAST

- Principles discussed quite exhaustively in previous lecture

- Today, we'll focus on:
  - Web-specific tooling
  - Some best practices for a rollout of SAST in a big organization
  - Secrets detection in code

SAST - tooling

- GitHub/GitLab have both great SASTs
  - Github CodeQL - get bounties for writing SAST rules
  - GitLab's SAST (merged open-source tools into 1 image)
  - The closer it is to devs, the better.

- Language specific SAST tools (search "security" in awesome-static-analysis)
  - Recommended and simple SAST that integrates multiple languages: **semgrep.dev**
  - Specialized tools such as Pysa (for python) if you're looking for something more powerful
  - Rulesets of this tools are **great** learning resource of vulnerable language-specific gotchas that can be independently used e.g. in code reviews.

- Build easily extensible alerting on regexes/keywords appearing in your code
  - You might want to be aware that import cryptography newly appeared somewhere, so you can talk to the developer trying to implement some potentially risky feature before he does it.

SAST - implementation best practices

- Triage issues effectively
  - Prioritize issues based on the business risk.
  - Don’t bother devs with false positives / low severity findings

- Start slowly
  - Easy to get overwhelmed by the amount of findings
  - Choose few high-impact issues and focus on it. Repeat.

- Define a clear process for the issue triage
  - E.g.: New high/critical impact issue -> alert to #appsec-team channel -> triage -> start resolution
  - New medium impact issue -> automatically create issue for dev
  - New low impact issue -> backlog

Secrets in code detection

- Technically still part of SAST, as you analyze the source code

- Easy detection and easy direct exploitation
  - API keys of cloud providers can be exploited for crypto mining
  - SaaS providers such as PayPal, GitHub or Twitter
  - Private RSA keys, database dumps, …

- How bad can it git?
  - Research scanning all GitHub commits for secrets over 6 months.
  - Thousands new, valid and unique secrets leaked every day
  - Still huge space for improvement in detection (they scanned secrets only for 11 platforms)

- Low effort & High impact (rewards up to $15,000 for a single GitHub token)

Secrets in code detection - tooling

- GitHub's token scanning - low false positives, auto-revocation (e.g. AWS), by default present on github.com
- GitLab’s SAST - gitleaks and TruffleHog with the default config

- gitleaks - can combine entropy and regexes
- gitrob - another good tool with recursive org scans
- TruffleHog - “the original” scanner, now inferior
- shhgit - real time monitoring of GitHub commits

- Everything is about having a good config file to balance the signal (false negatives / false positives)

Dynamic Application Security Testing (DAST)

DAST - tooling (web)

- Web security vulnerability scanner
  - Focused on web apps, spiders the website *deeply*
  - Great for automated discovery of several vulnerability classes or security headers checks
  - Burp Suite (paid, superior), OWASP ZAP (open-source, used in GitLab's DAST)

- Asset discovery
  - "Bug bounty" like monitoring tools, most of them originally made *for* bug bounty
  - Many companies don't have a list of their assets => cannot specify scope for the scanner
  - Searching for assets and monitoring all of them *lightly* (picking up the low-hanging fruit)
  - Might use Web security vulnerability scanners to scan some more appealing targets
  - E.g. Assetnote (great paid product), projectdiscovery.io (open-source), BugShop (made by me :-))

Web security vulnerability scanner

- Security scanner running on live web application
- Crawls the website as a human would, fuzzing different “malicious” inputs
  - Might be quite intrusive => should be used on staging or well known production environment
  - Sometimes problems with login to apps (especially *complicated* flows like SAML/OAuth2), which can be solved by using a “browser” like Selenium for login and then passing session to scanner.
- Similar to fuzzers/dynamic analysis for programs described in previous lecture, just specific for web

Asset Discovery - Bugshop

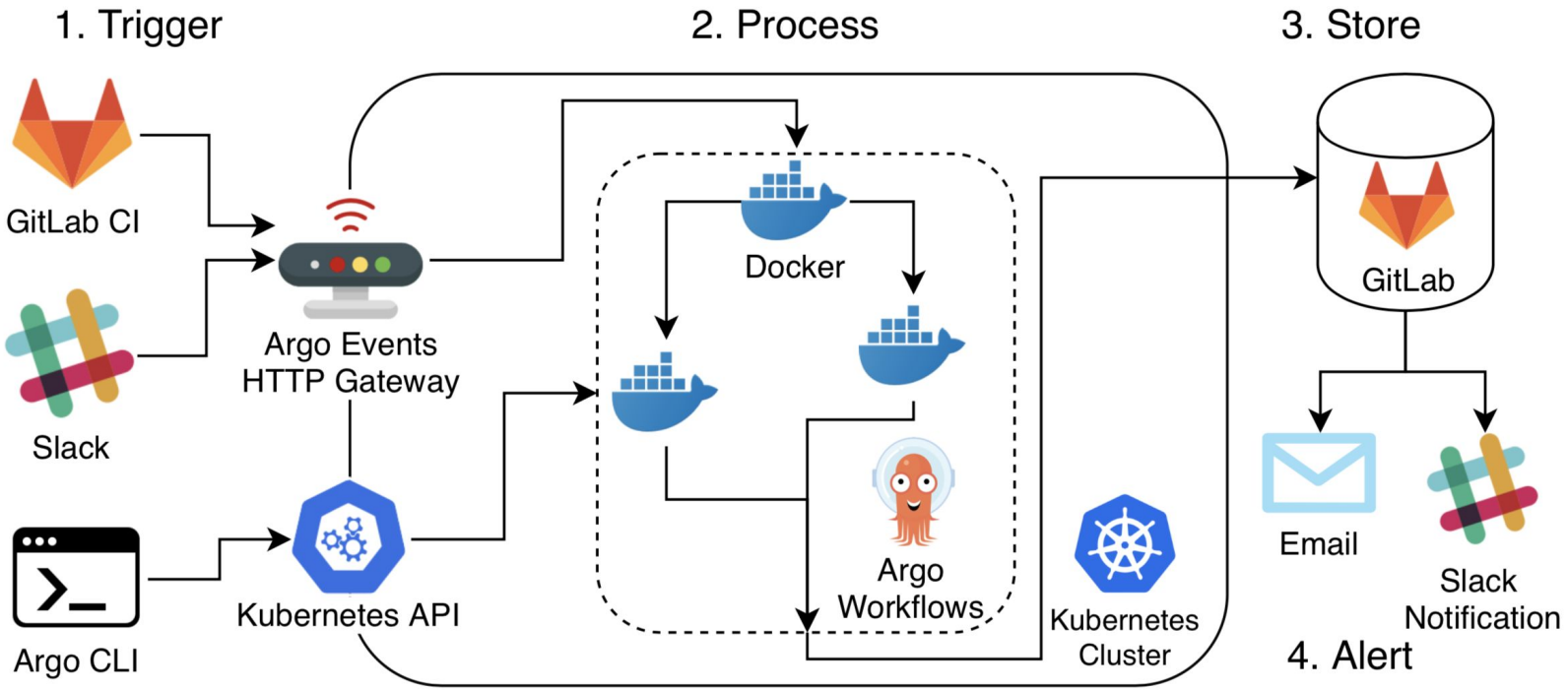


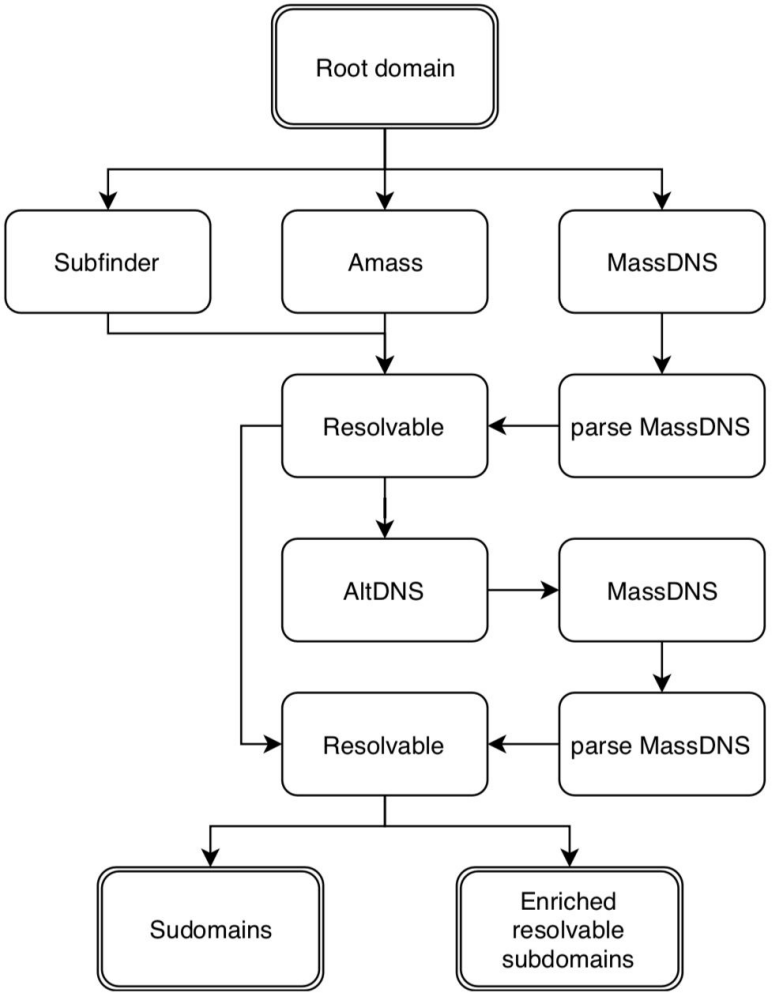
Figure 5.1: High level overview of the Bugshop.

Asset Discovery - Bugshop

- Start by **subdomain enumeration** workflow with a wildcard domain (e.g. *.muni.cz) and end with a list of hundreds subdomains (*assets*).

- Then run vulnerability and discovery checks on all newly found *assets*, find vulnerabilities (e.g. by Web security vulnerability scanner), and find more *assets* recursively.

- Further automation of workflows commonly used in bug bounty (git secret detection, bucket enumeration or subdomain takeovers)

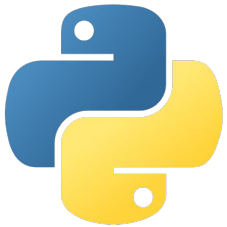


Dependency management

Software package registries

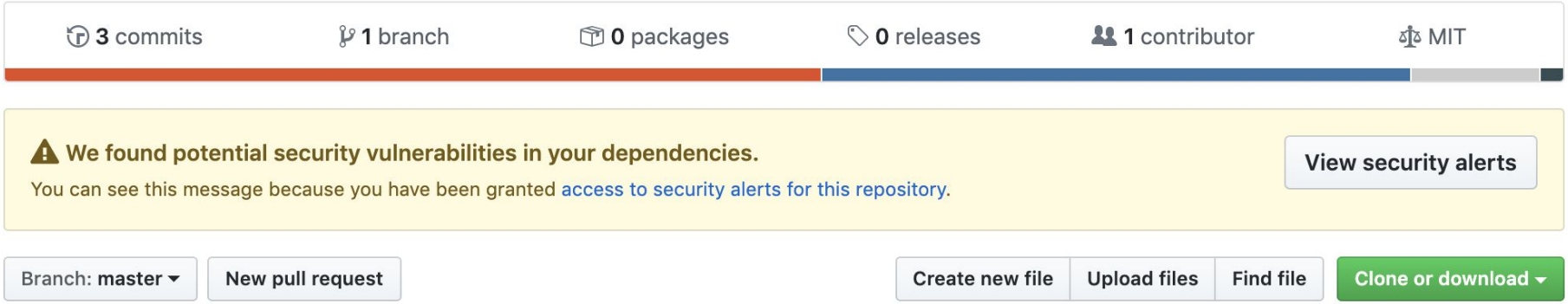
- Easy distribution of packaged code for use by other developers
  - Node.js - npm
  - Java - Maven Central
  - Python - PyPI
  - Ruby - RubyGems
  - Docker - DockerHub (distribution of docker images, not code)
- Heavy growth in the past years (especially Maven/npm)
- Convenient (1 command and code is ready to use)

maven

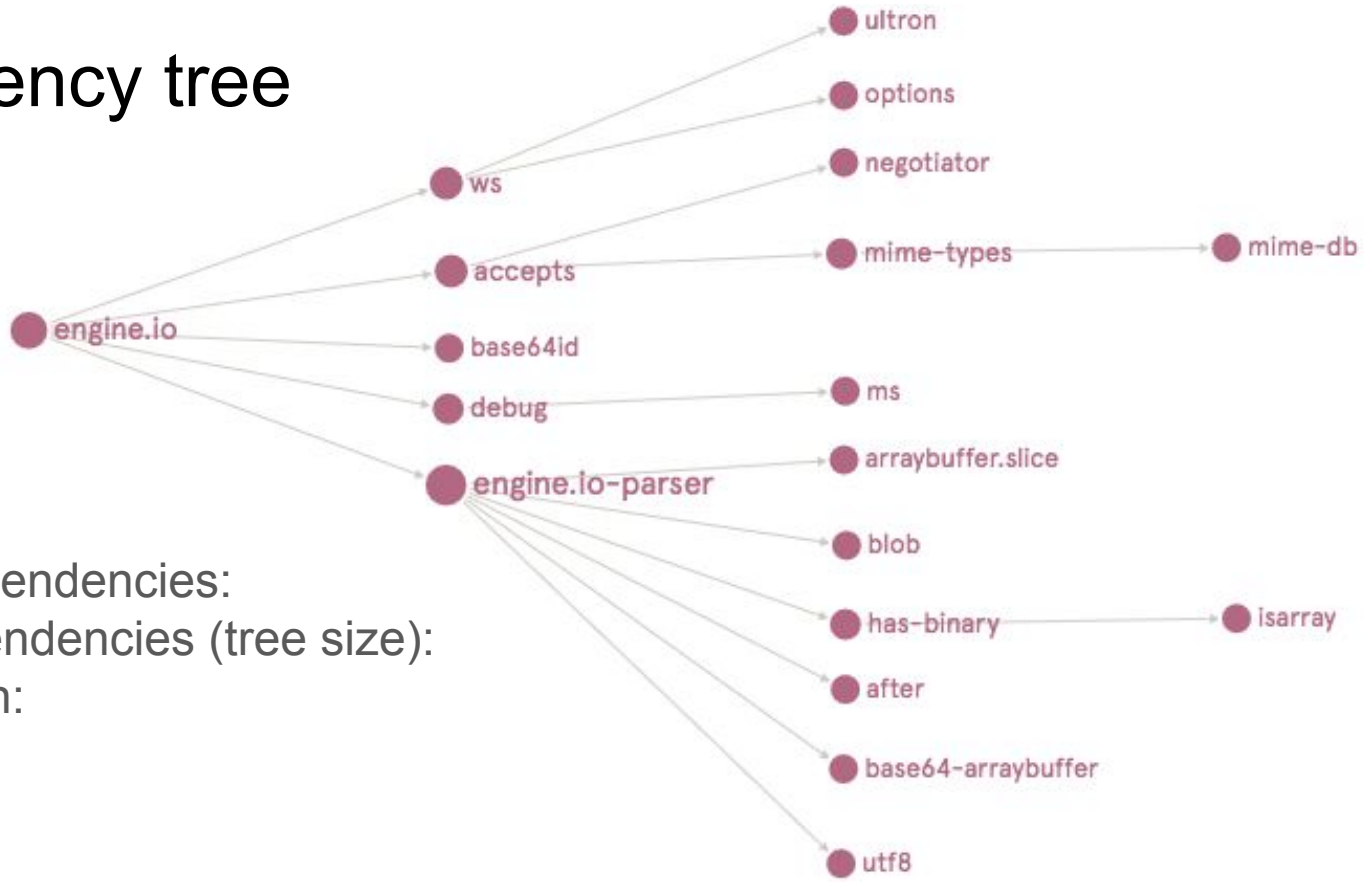


pypi

How many of you have seen this warning?



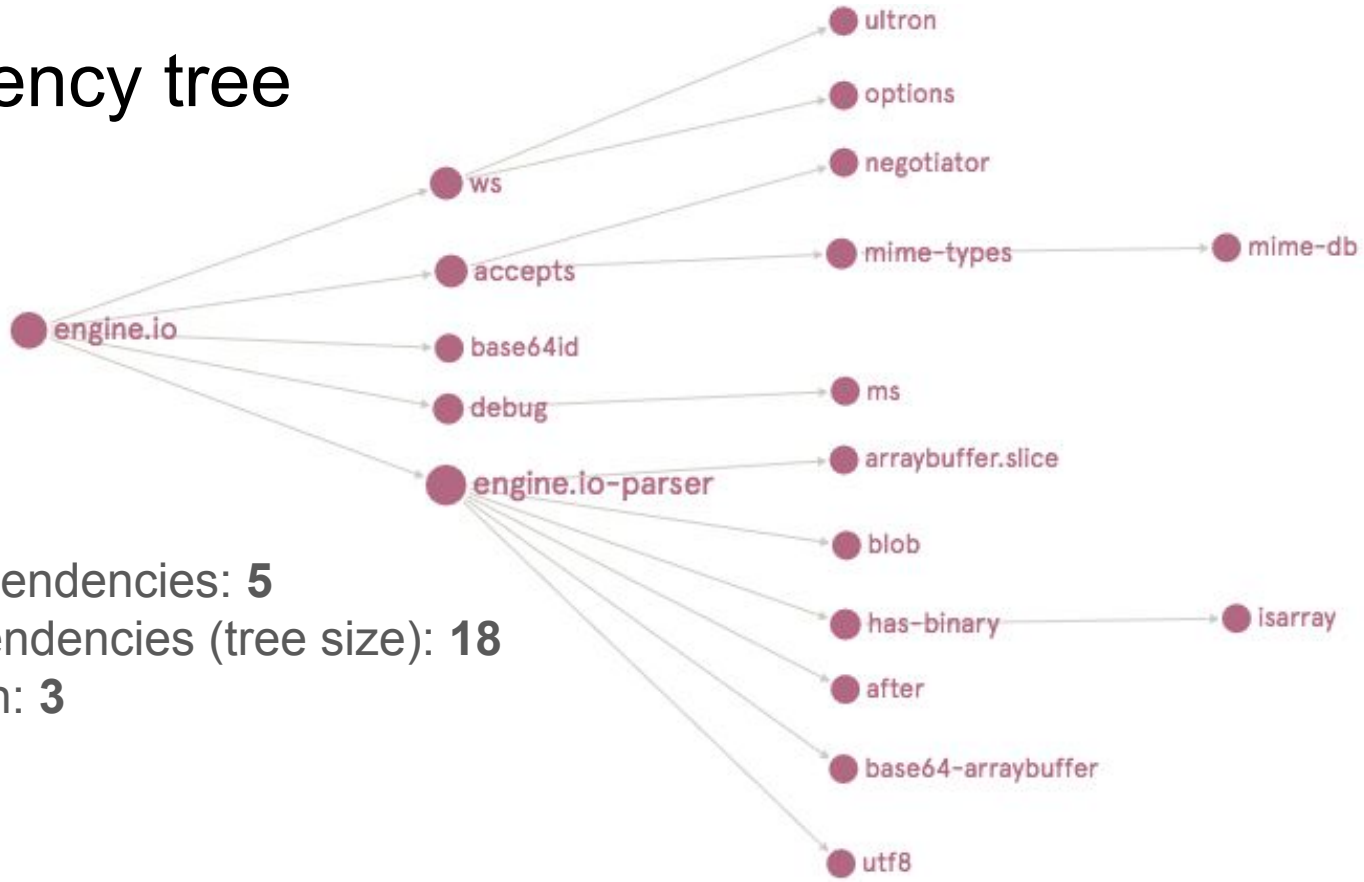
Dependency tree



- Direct dependencies:
- Total dependencies (tree size):
- Tree depth:

https://medium.com/graph-commons/analyzing-the-npm-dependency-network-e2cf318c1d0d

Dependency tree



- Direct dependencies: **5**
- Total dependencies (tree size): **18**
- Tree depth: **3**

https://medium.com/graph-commons/analyzing-the-npm-dependency-network-e2cf318c1d0d

Quiz time

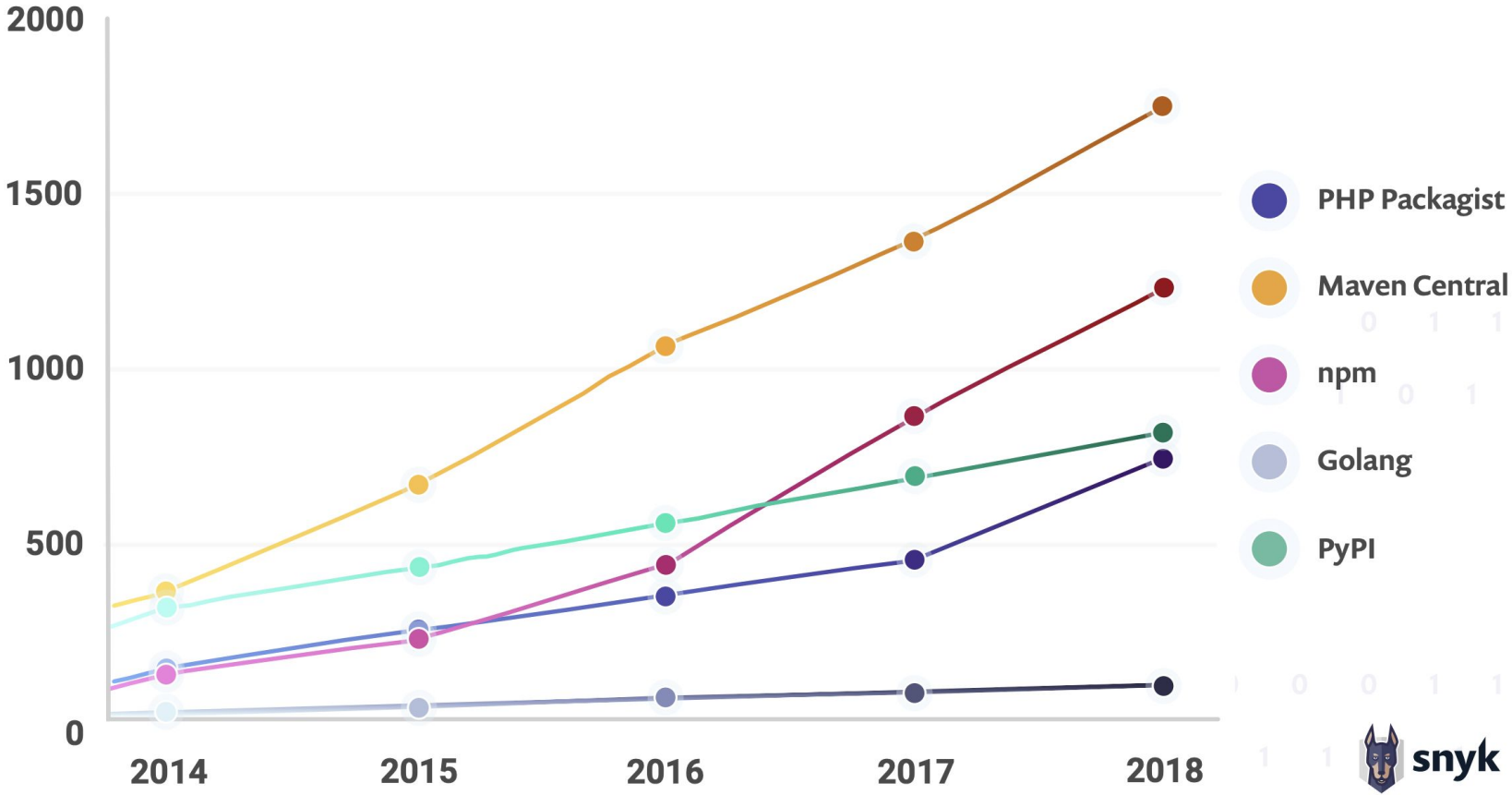
Go to kahoot.it
Enter PIN: 626276

Table 3: Characterization of package dependency graphs (without disconnected nodes)

	npm	PyPI
#Nodes	577943	84188
Avg node outdegree	4.27	2.95
Avg dependency tree size	86.55	7.33
Avg dependency tree depth	4.39	1.71

https://www.researchgate.net/publication/331587729_Security_Issues_in_Language-based_Sofware_Ecosystems

New vulnerabilities each year by ecosystem



https://res.cloudinary.com/snyk/image/upload/v1551172581/The-State-Of-Open-Source-Security-Report-2019-Snyk.pdf

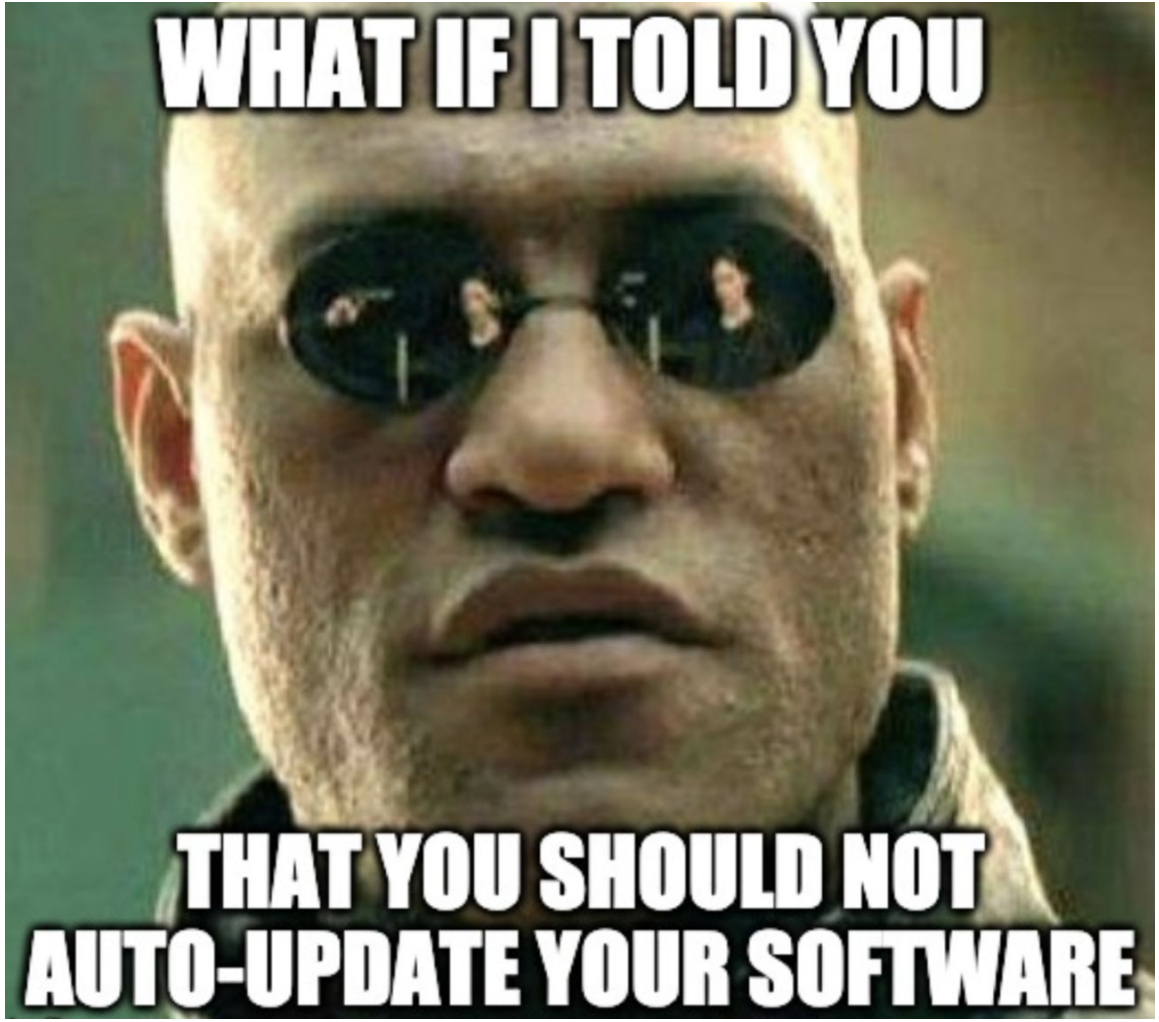
[non-malicious] Risks of package registries

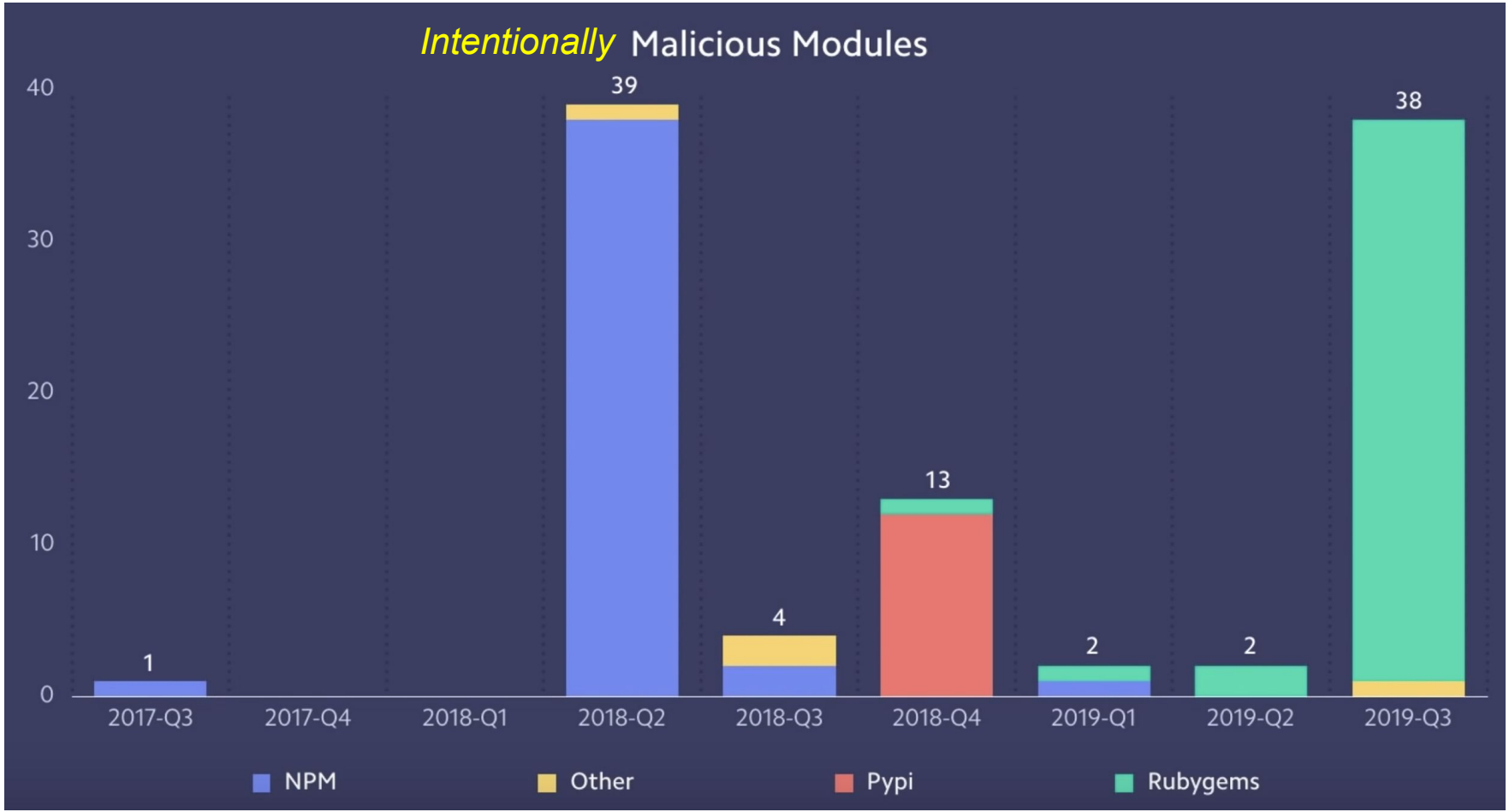
- Packages with **known vulnerabilities** (outdated/abandoned dependencies)

- 88% growth in (reported) packages vulnerabilities over the past two years

- Growing dependency chains increase the chance of compromising your dependencies indirectly

- …. Auto-update everything, right? Riight?


WHAT IF I TOLD YOU
THAT YOU SHOULD NOT
AUTO-UPDATE YOUR SOFTWARE



Stripe Sessions 2019 | When data contradicts security best practices

[malicious] Risks of package registries

- **Malicious releases**
  - npm `event-stream` compromised via its dependency
  - Version number **might not** be an immutable identifier

- ***Nobody* is reviewing the code** before installing on production servers
  - In ideal world, you would hold a list of approved *reviewed* packages with versions.
  - In reality, the whole package ecosystem is super fragile
  - E.g. Hacking 20 high-profile dev accounts could compromise half of the npm ecosystem

- Typosquatting of package names
  - pip install request (instead of requests)
  - as pip executes code during the **installation** => 1 typo == RCE
  - SK-CSIRT identified malicious packages on PyPI

Dependency management - tooling

- Dependency monitoring
  - GitHub
  - GitLab
  - Built-in in the package manager (npm audit / pipenv - safety)
  - Commercial (Snyk)
  - OWASP Dependency Check

- Automatically open pull request with dependency update
  - GitHub
  - Renovatebot
  - Commercial (Snyk). Sad part is that all commercial tools use their own *private and licensed* database of vulnerabilities. (╯°□°)╯ ︵ ┻━┻

Dependency management - best practices

- **Automatically monitor dependencies** for known vulnerabilities
  - Both GitHub and GitLab have built-in **dependency scanning** available. Neither of them is perfect, but it's *something* and it's easy to start with.

- **Don't** auto-update right after the release (update != security patch)
  - Wait few days/weeks for community to spot bugs or hijacked/malicious packages.
  - Naturally, continue to apply **security patches** immediately.

- Use **immutable identifiers** for packages
  - Version number is a **mutable identifier** in Docker, Maven or PyPI.
  - Hash digests are preferable and protect you even from a compromise of the registry.
  - Auto-update tools such as renovatebot can help with this.

Penetration tests

Penetration tests - who does it?

- **Internal**
  - Done internally, e.g. by members of the AppSec team
  - Good for deep tests that require some internal knowledge of the application

- **External**
  - Outsource to an external company
  - Usually done this way so security team can focus on other issues
  - Compliance requirement in some cases (e.g. you cannot pentest yourself in PCI DSS)
  - Good way to earn public reputation (*pentested by Cure53 / XYZ*)

Penetration tests - types

- **Black box**
  - The same conditions as an attacker (no access to docs or code)
  - Not really effective in value/money ratio as pentester spends more time on app discovery
- **Grey box**
  - Access to app documentation or small chunks of code
  - Possible cooperation/chat between pentester and company
- **White box**
  - Source code available to the pentester (can run SAST tools on it)
  - Great for any deep pentest (business logic, auth)
  - Essentially required for pentesting iOS apps or similar

Penetration tests - methodology

- *Best effort* test
  - Give pentester a *free hand* on techniques used in testing
  - Usually lasts 3-10 days
- Detailed test following an official testing guide
  - **OWASP Testing Guide v4 (OTG4),** NIST 800-115 or OSSTMM
  - Test following an established methodology might be required by compliance (PCI DSS)
  - Usually lasts 2-4 weeks depending on size of the application

Penetration tests - best practices

- Rotate at least two pentesting companies
  - Each company uses different scanners or might check unique techniques
  - They can catch mistakes of each other => higher motivation

- **Pentest before release** & (ideally) regularly
  - Pentest can save you quite some money that you would spend on bug bounties later

- **Scope pentests** smartly
  - Let pentesters know which part of application is your priority and share all relevant docs/code

- Have a **healthy** bug bounty program :-)
  - Scoped pentests will never cover your whole external attack surface
  - Some companies keep pentests only for compliance

Bug bounty

Bug bounty

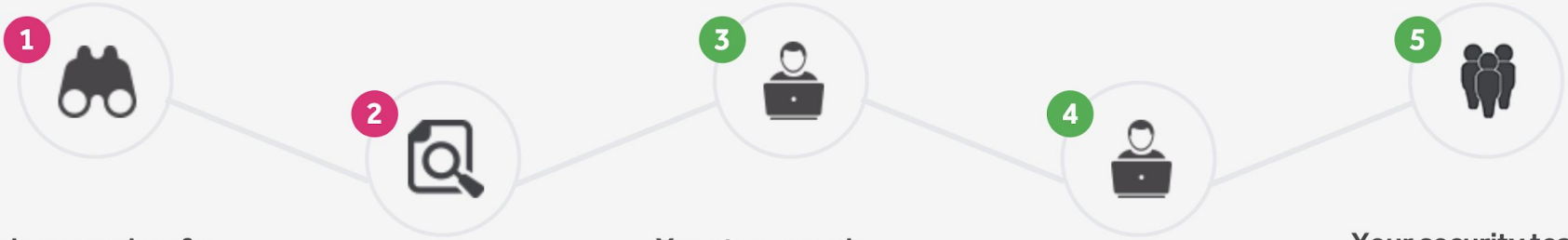
- “Please come, **hack our apps**, **report** it to us and **get paid.**”
  …and without lawsuits :-)

- Great part time income for students - you *can* learn a lot during it :)

- Experiencing **huge** boom in the past few years

A Self-Managed HackerOne Bug Bounty Program

Use your abundant resources and past experience to run your own bug bounty program.



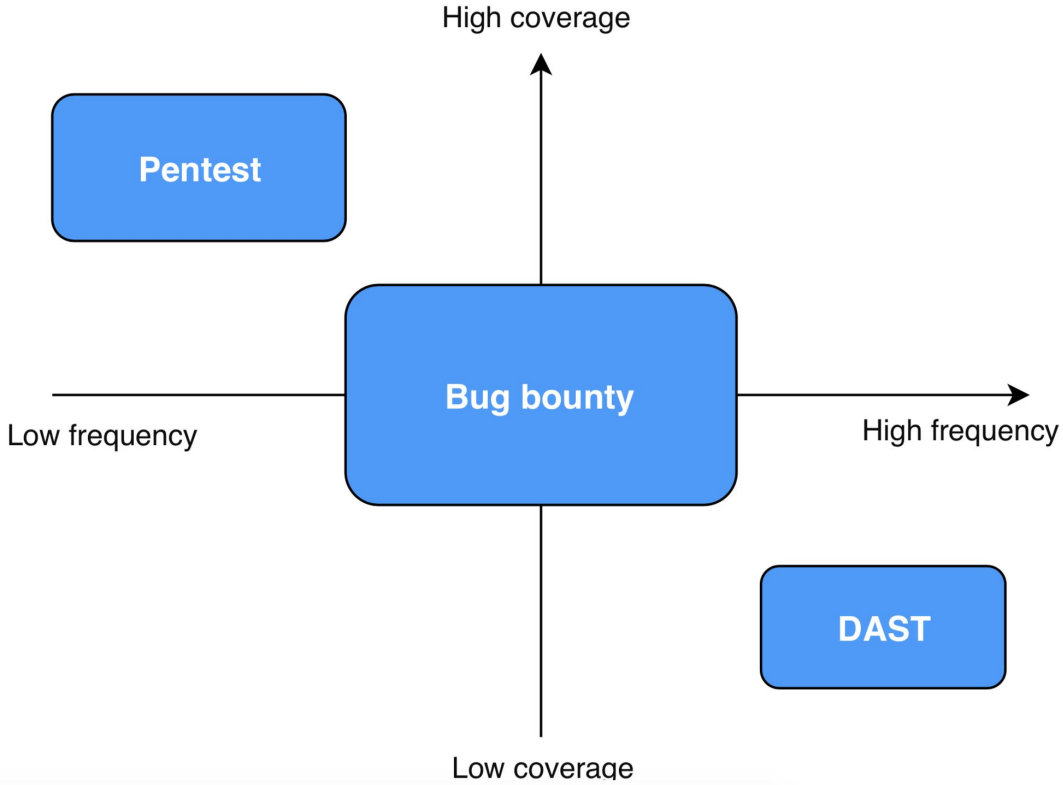
1. Hacker searches for vulnerabilities
2. Hacker submits it to your organization
3. Your team works closely with hackers to receive all relevant data
4. Your security staff validates all vulnerability reports
5. Your security team triages all submissions and fixes all valid submissions

Bug bounty - types

- **Self-hosted program**
  - Company publishes website with the policy (scope, rewards, rules, contact)
  - [Pros] No 3rd party involved, hackers communicate directly with the company
  - [Cons] Company needs to handle payments, web platform and has to get interest of hackers
  - Examples include Medium, Google, Microsoft or Facebook

- **Bug bounty platforms**
  - Company makes contract with another company (bug bounty platform)
  - [Pros] Convenient web app for triage, payments handled by platform, hundreds of registered hackers ready to hack
  - [Cons] 3rd party has access to your bugs, cost ($XX XXX/year)
  - Example platforms are HackerOne, Bugcrowd, Synack, Intigriti or Hacktrophy (CZ/SK)

Pentest vs bug bounty vs DAST



OWASP AppSec USA, Zane Lackey - "Practical tips for web application security in the age of agile and DevOps", 2016. www.youtube.com/watch?v=Hmu21p9ybWs

Some general best practices

- *Make the right thing easy to do!*
- Show devs the cool side of security
  - Talk about the impact of bugs found, encourage and reward active people
  - Don’t underestimate *soft measures* mentioned in the beginning
- Outsource as much as possible to secure by default frameworks.
  - *Force* validation of input.
  - Stop reinventing the wheel with auth, sessions, CSRF protection or output escaping.
  - Great examples are React, Django or Connexion.
- Have secure, yet easy to use/manage secrets storage (e.g. Vault)
- Integrate most of the security checks to CI/CD pipelines
  - Continuous feedback to developers
  - Don’t forget to run checks also on schedule (unmaintained production code also needs care)
- Good example of AppSec at scale is Netflix’s concept of paved road

… reality

- **Impossible** to do all of the above mentioned in a short amount of time
  - Resources (money/people) are usually very limited
- Prioritization is the key (decide based on risk)
  - Do you really need DAST in CI/CD if you don’t even have SAST or dependency scanning?
  - Go for quick wins - bottoms-up approach works better in *agile* companies
- Build a vision where are you heading
  - You can copy it from more mature companies, but don’t forget to adjust it based on company culture, maturity and your resources.
- Automation is the key, but tools alone **won’t** save you
  - Manual findings will be the impactful ones
  - You need to filter out the low priority issues

Thanks for your attention!

Any questions?

Seminar

- Intro (10min)
- Dependency scanning (40min)
  - python safety (docker/pip required)
- SAST (40min)
  - python bandit hands-on (docker/pip required)
- HW setup (5min)

To prepare:

- Docker or pip
- 1-2 open-source projects that are interesting for you
- Register HackerOne account