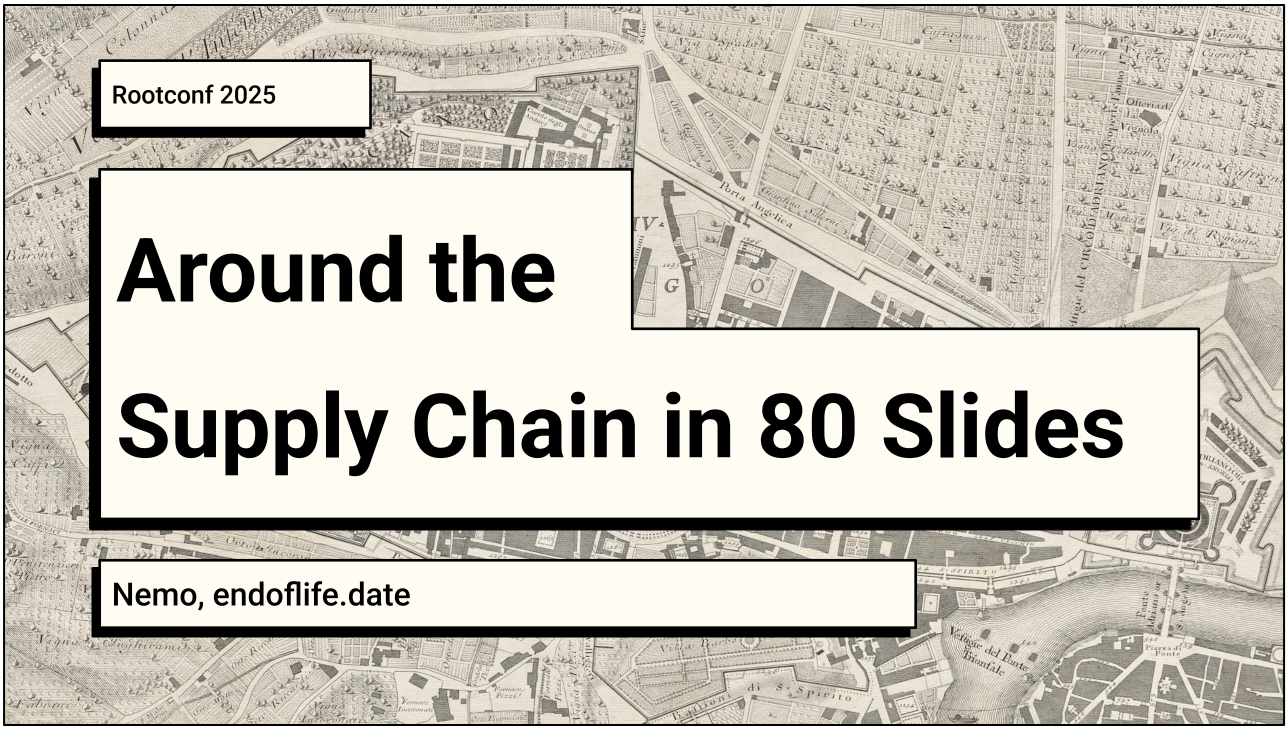
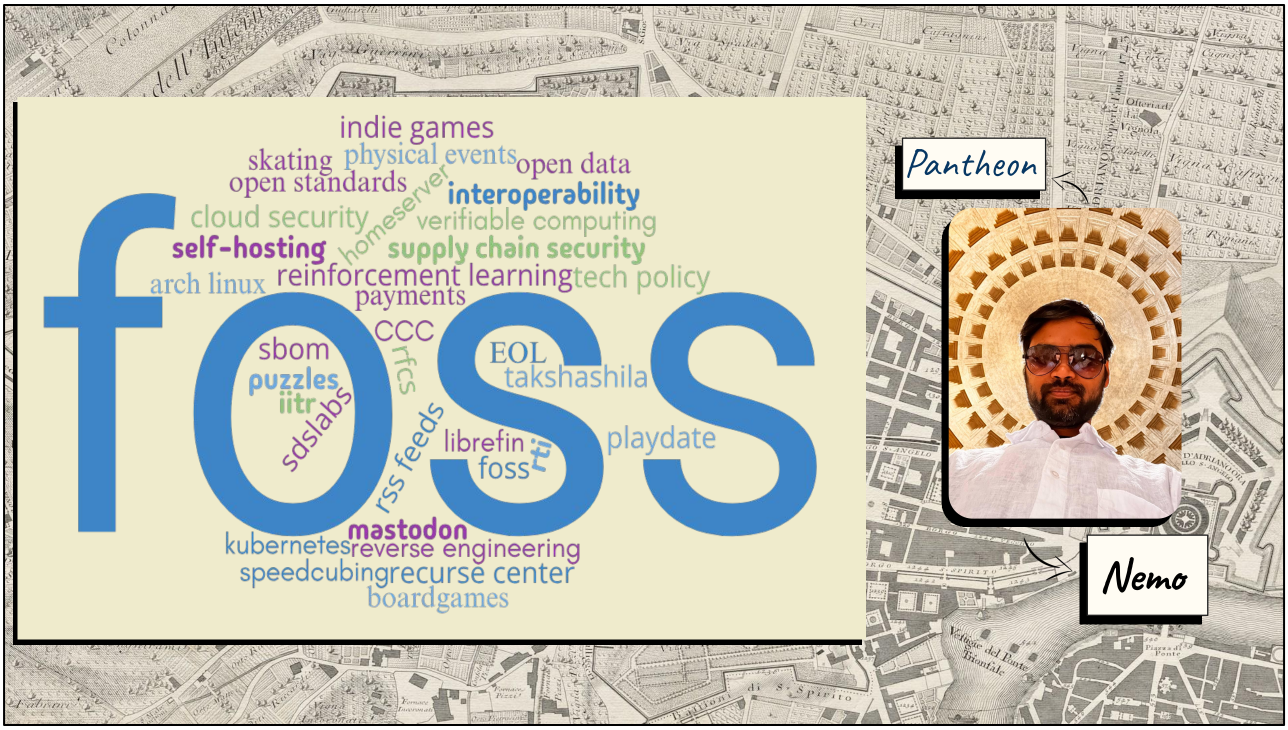


Rootconf 2025

Around the Supply Chain in 80 Slides

Nemo, endoflife.date

Intro Slide. The plan was to do a talk that sets the context for the Supply Chain Security track at Rootconf.



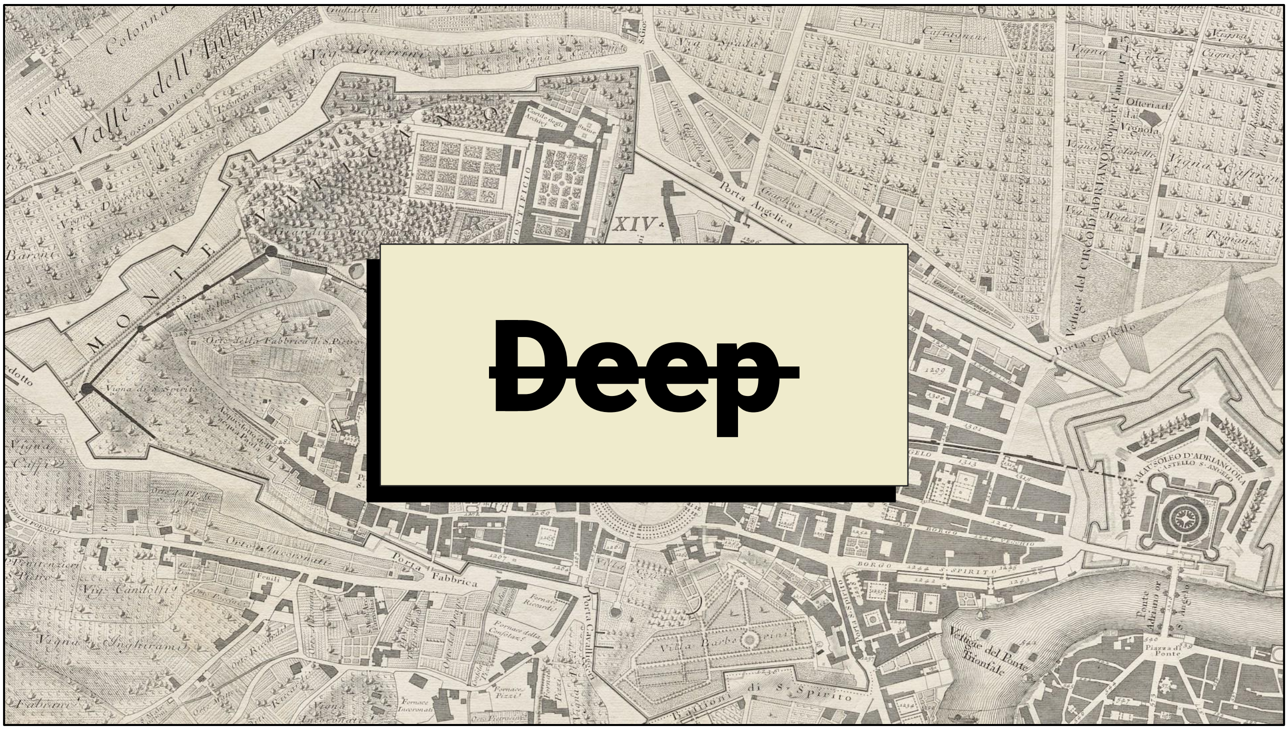
A little about me. The word-cloud is a fun way to explore things that I’m interested in currently. On the right is my photo from my recent visit to Rome!

Quick

This talk will be Quick.

Catch-up

We will be doing a lot of catch-up about what happened in the Software Supply Chain Security space in the last year and half or so.



~~Deep~~

I won’t be going deep. We have other speakers doing that.

Inspire

Hopefully, this talk will inspire folks towards building the next generation of security tooling.

Not Exhaustive

This is not an exhaustive talk - I can’t cover everything in Software Supply Chain Security in just 25 minutes.

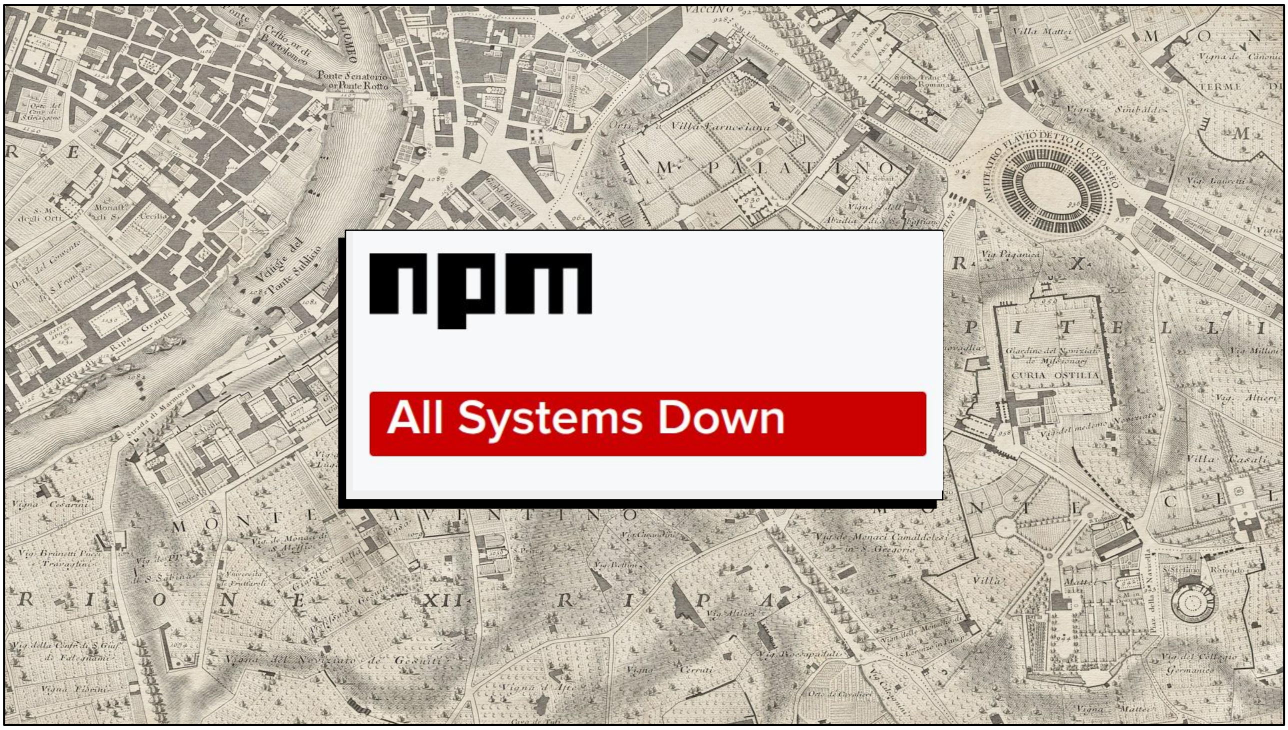
Supply Chain Attack or Not

Lets start with a quick quiz on whether a given attack classifies as Software Supply Chain Security attack or not.

pypi.org/package/

python-requests

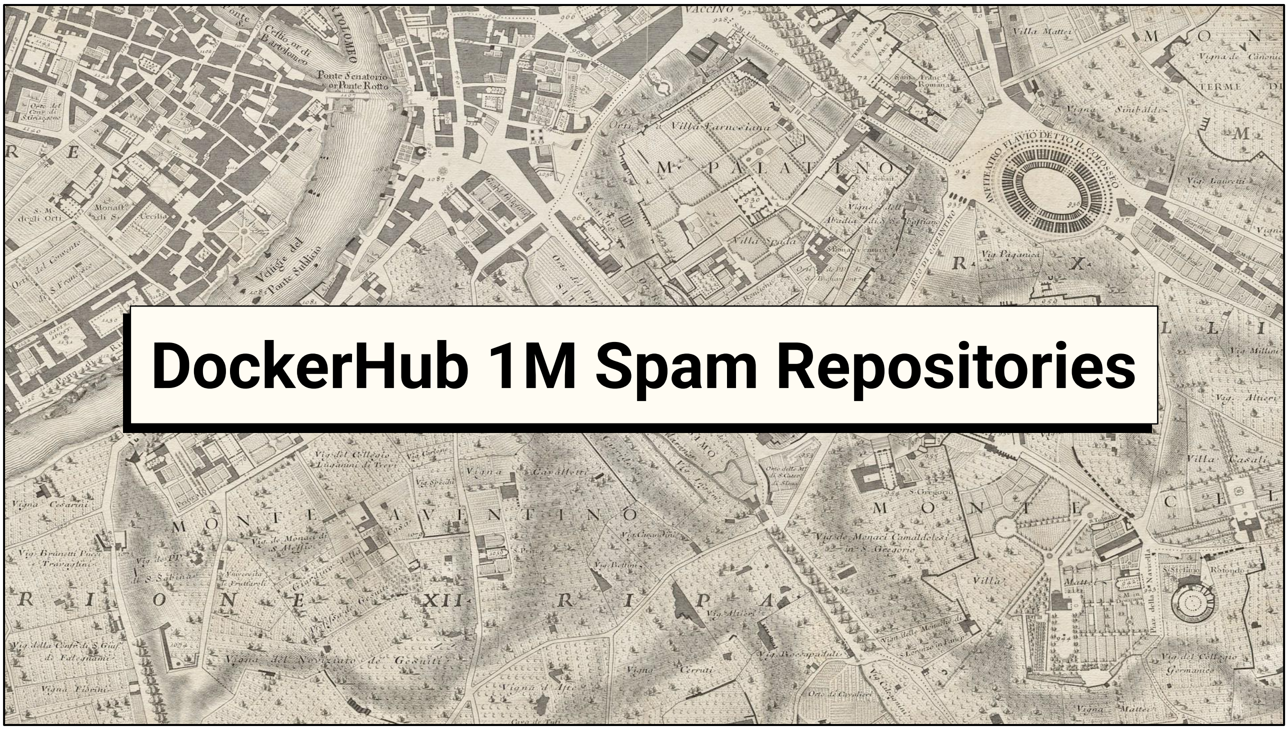
A malicious package gets registered on PyPi with a name confusion to another popular package. Software Supply Chain Security or not?



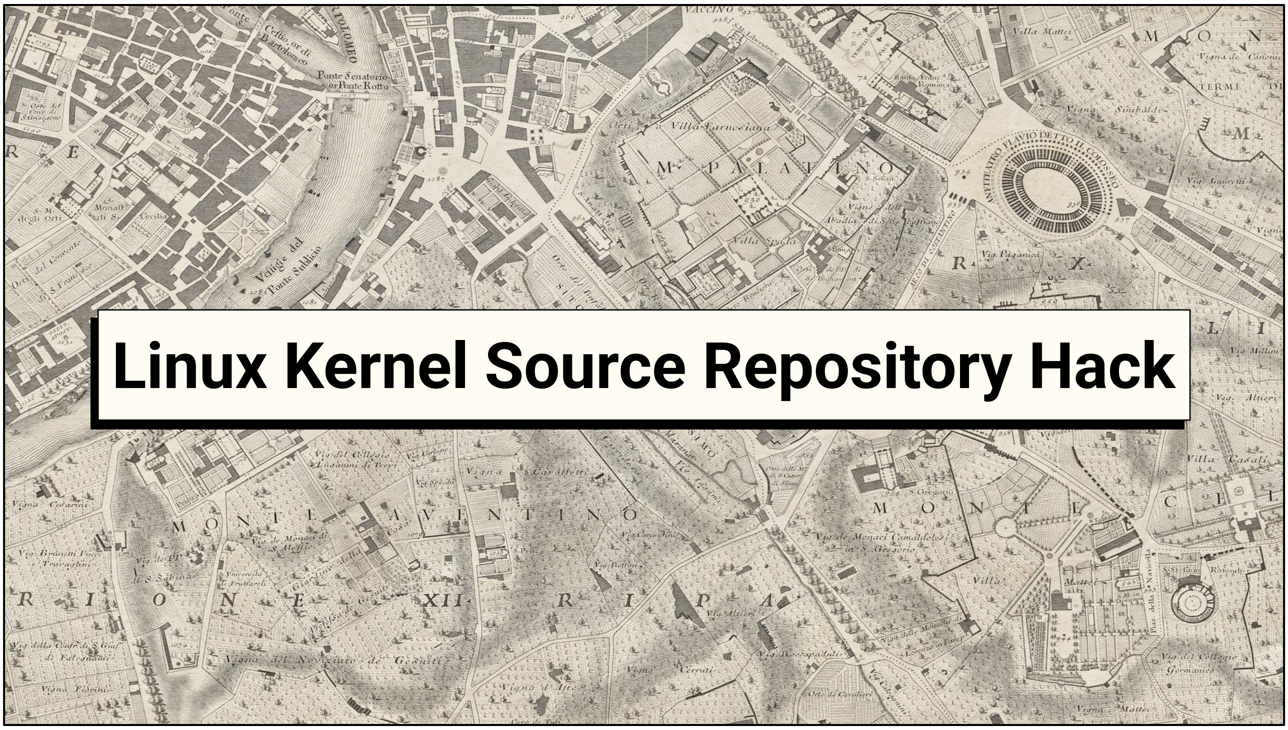
All Systems Down

NPM has a major outage. Software Supply Chain Security attack or not?

DockerHub 1M Spam Repositories

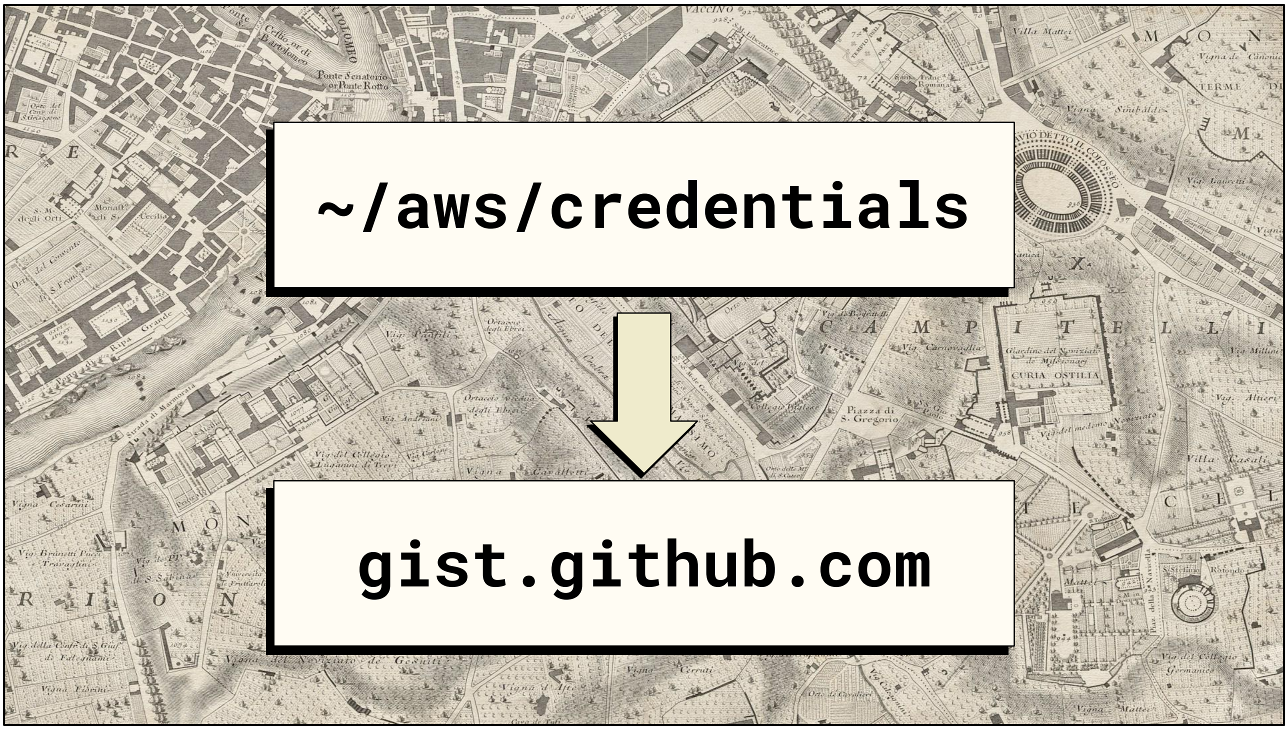

Ponte Senatorio o Ponte Rotto
Vestigie del Ponte Sublicio
Villa Farnesiana
M. PALATINO
ANFITEATRO FLAVIO DETTO IL COLOSSEO
Villa Mattei
Villa Spada
MONTE AVENTINO
RIONE XII RIPA
MONTE
Villa Casali
Vigna del Noviziato de Gesuiti

DockerHub gets 1M spam repositories that are linking to phishing websites. Software Supply Chain Security attack or not?



Linux Kernel Source Repository Hack

The linux kernel source repository gets breached. Software Supply Chain Security attack or not?



~/aws/credentials

gist.github.com

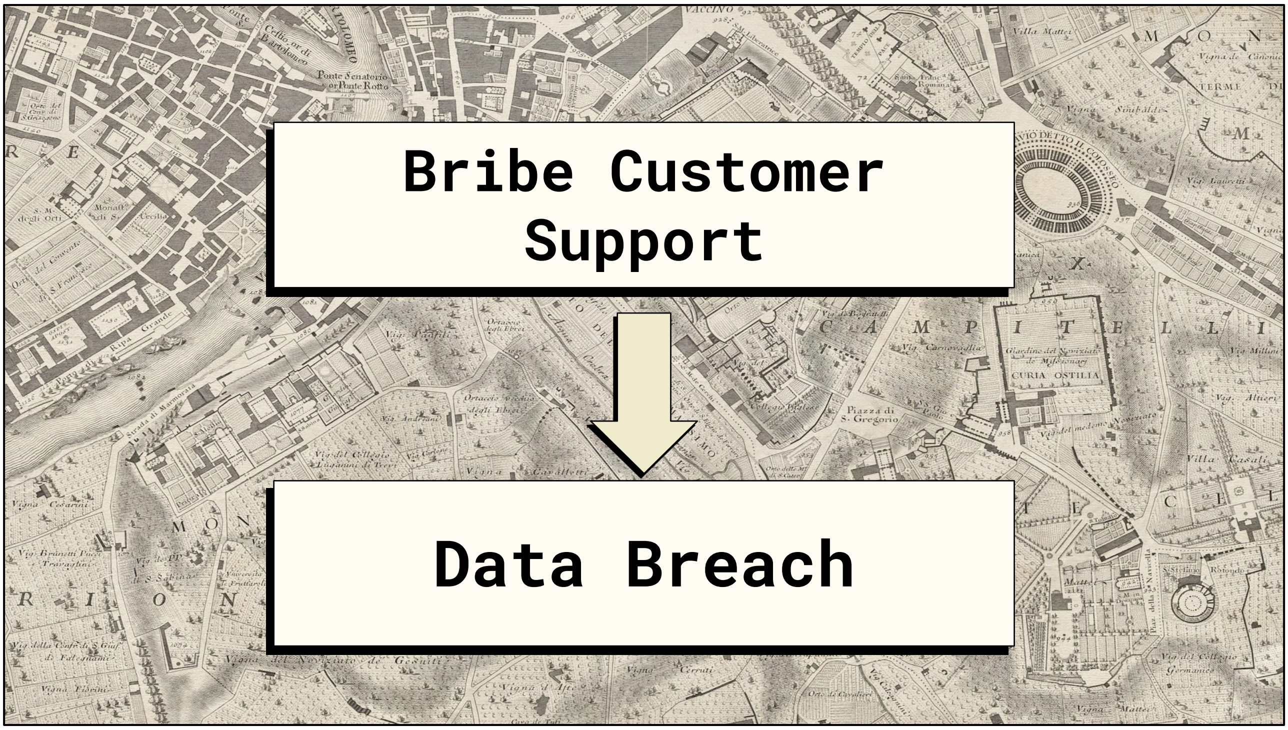
Your AWS credentials get accidentally uploaded to GitHub. Software Supply Chain Security attack or not?

Secrets logged on private CI

Your build secrets get logged on a private CI server. Software Supply Chain Security attack or not?

Malicious Package built on your CI

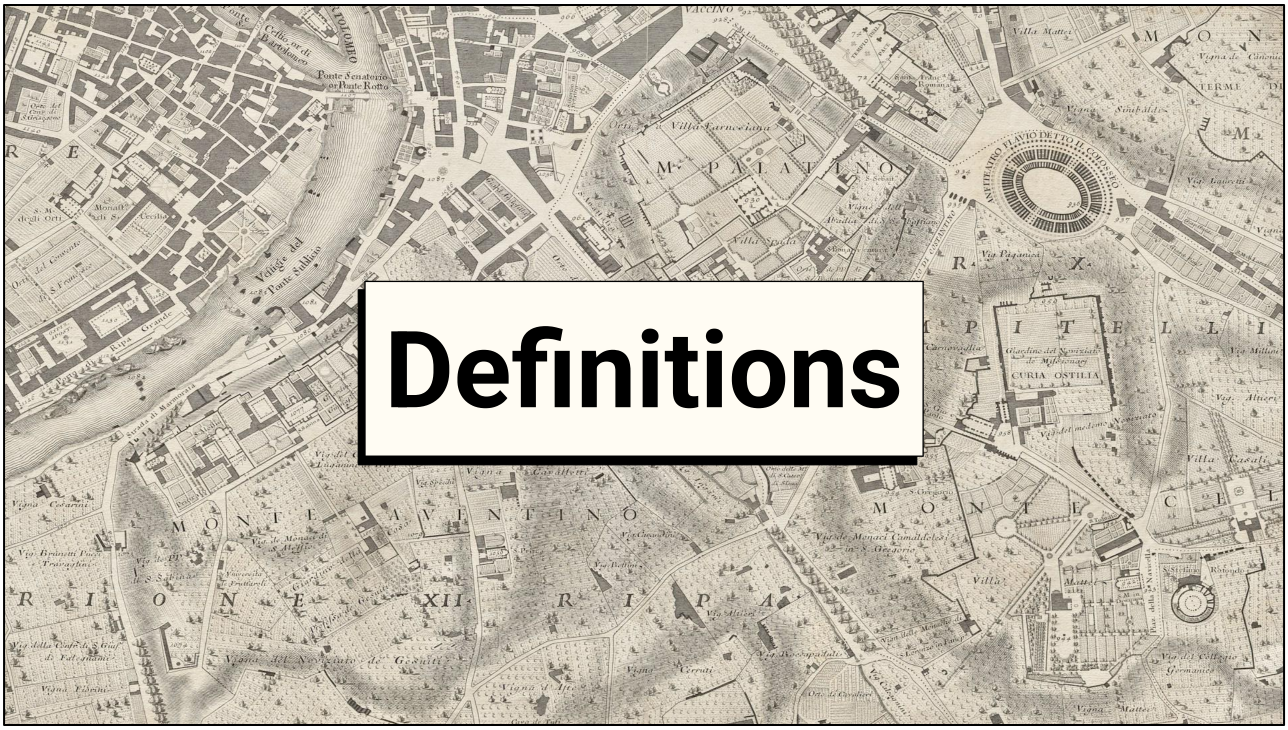
A malicious package is built inside a fakeroot on your build server. Software Supply Chain Security attack or not?



Bribe Customer Support

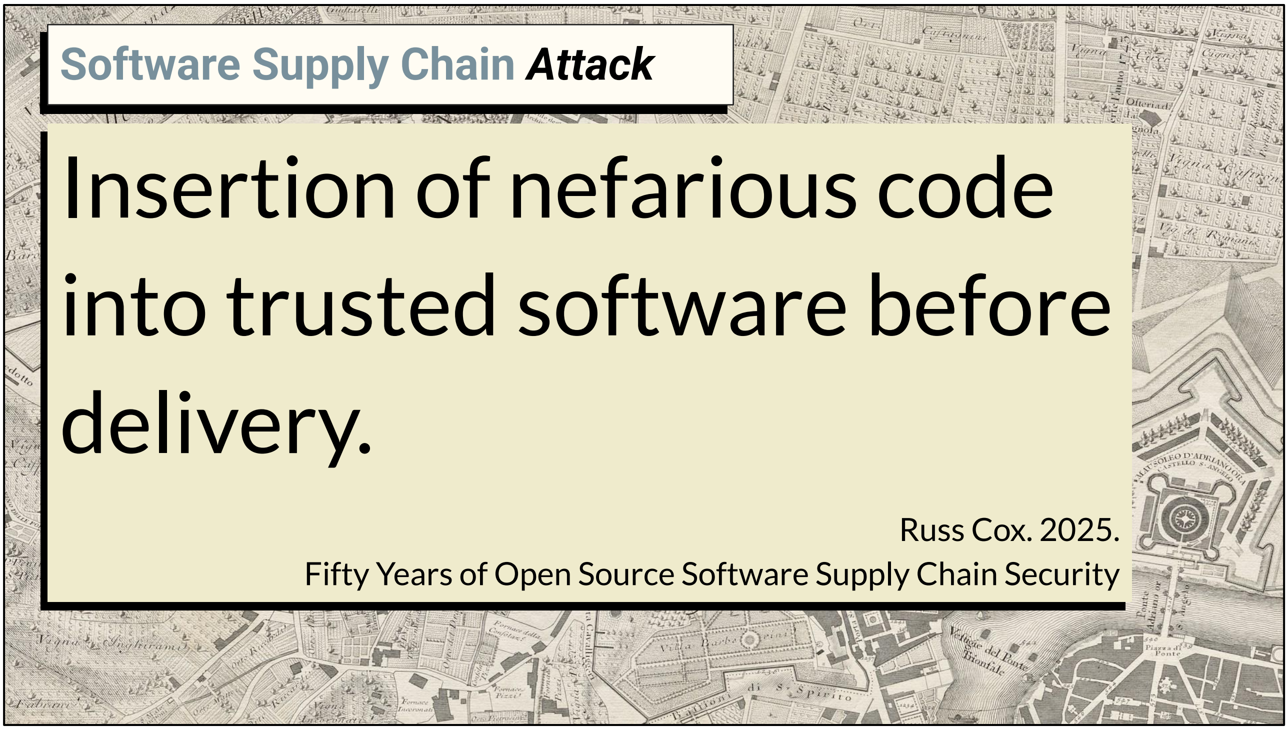
Data Breach

Your customer support executive gets bribed and hands out customer data. Software Supply Chain Security attack or not?



Definitions

Software Supply Chain Security is an umbrella buzzword. And as buzzwords go, it can be fuzzy. So lets take a look at some definitions. Not just for this talk, but to help set context for the remainder of the track as well.

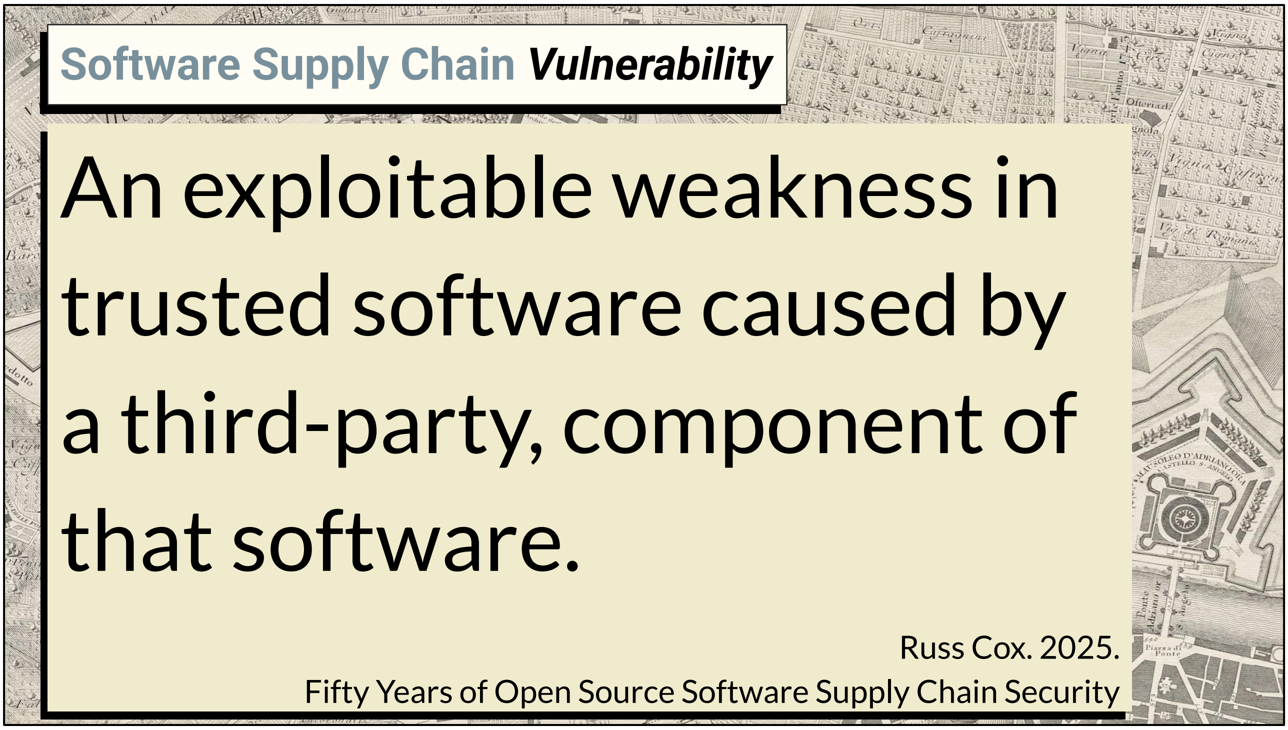


Software Supply Chain *Attack*

Insertion of nefarious code into trusted software before delivery.

Russ Cox. 2025.
Fifty Years of Open Source Software Supply Chain Security

This comes from Russ Cox, Golang developer who wrote an article in the ACM about Fifty Years of Open Source Software Supply Chain Security. He defines Software Supply Chain attack as “Insertion of nefarious code into trusted software before delivery.” Focus on “before delivery”

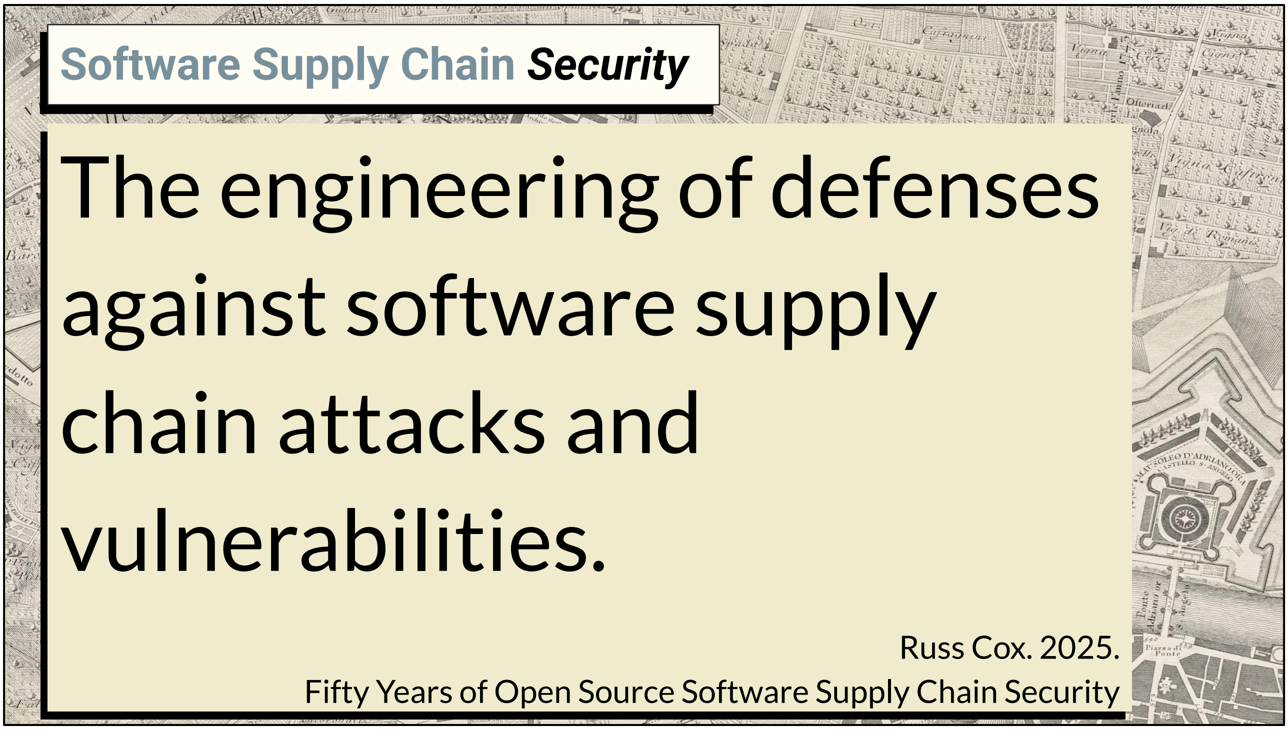


Software Supply Chain *Vulnerability*

An exploitable weakness in trusted software caused by a third-party, component of that software.

Russ Cox. 2025.
Fifty Years of Open Source Software Supply Chain Security

He defines Software Supply Chain vulnerability as “An exploitable weakness in trusted software caused by a third-party, component of that software.” Focus on “third-party”. If its your own software, that is just regula vulnerability..

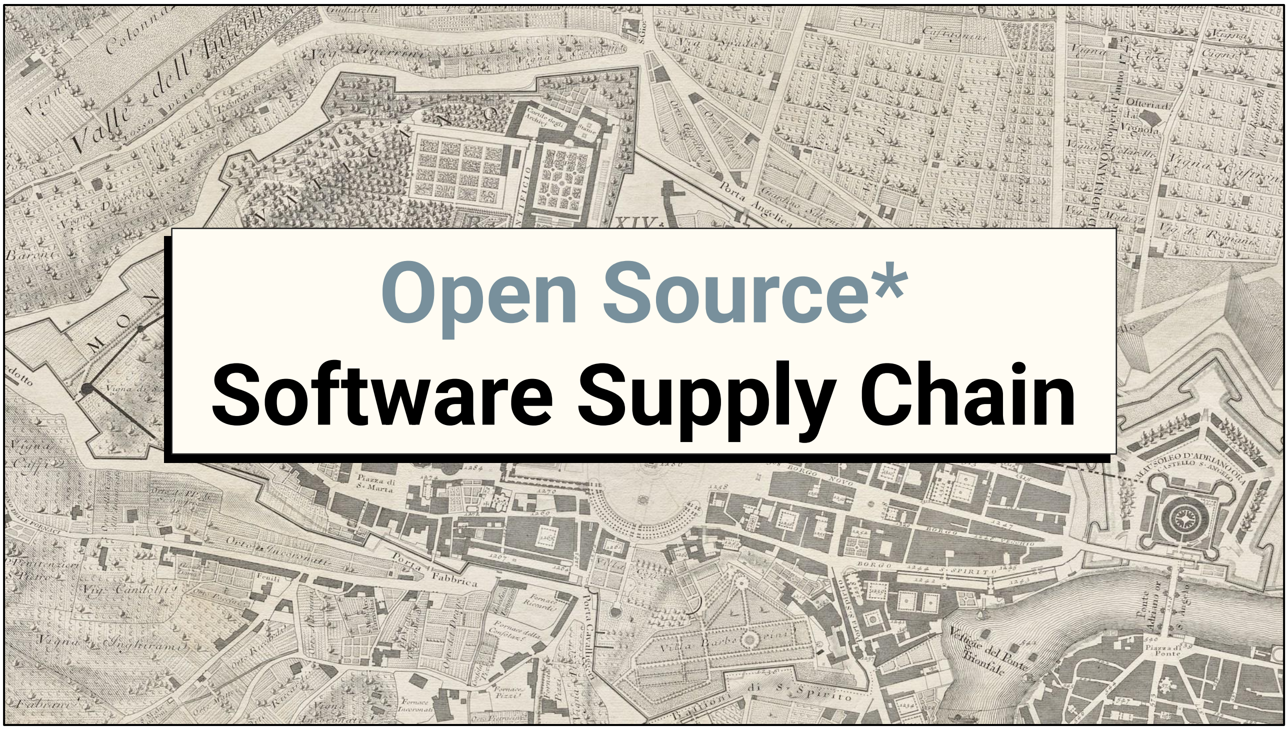


Software Supply Chain *Security*

The engineering of defenses against software supply chain attacks and vulnerabilities.

Russ Cox. 2025.
Fifty Years of Open Source Software Supply Chain Security

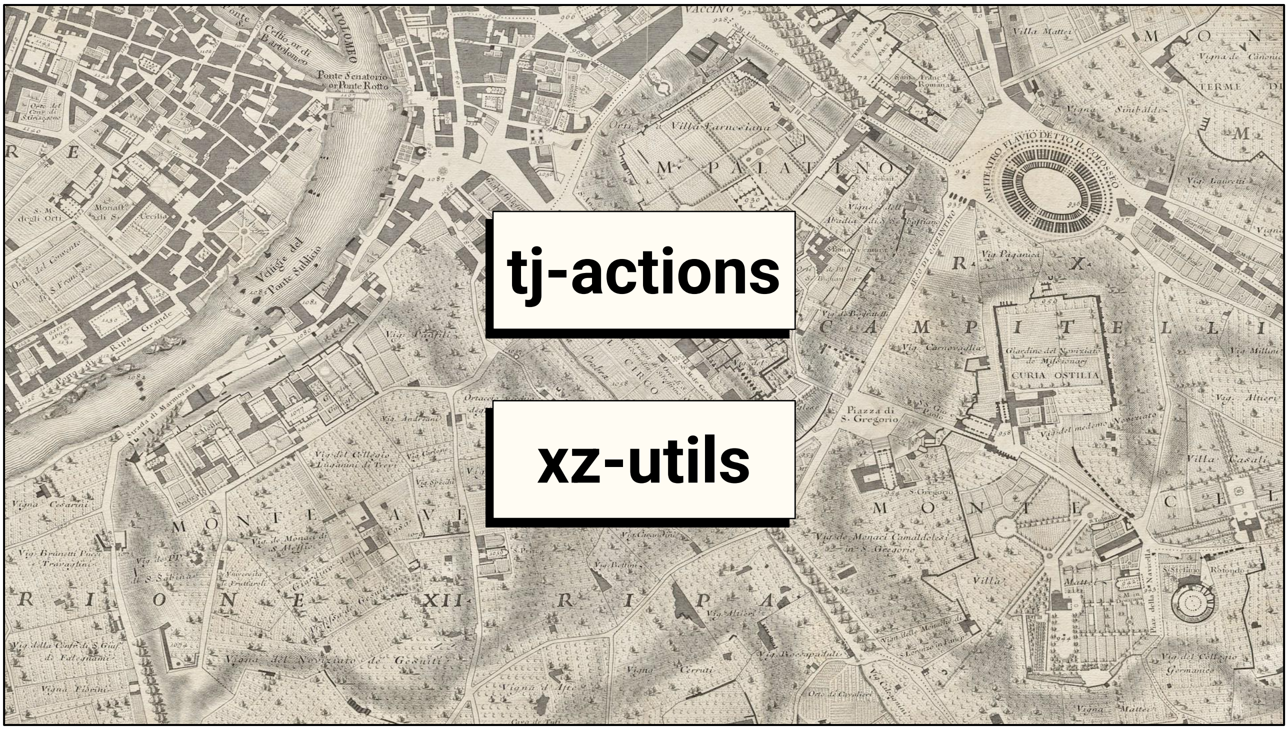
The Software Supply Chain Security definition falls out from the previous two as “the engineering of defenses against software supply chain attacks and vulnerabilities.”



Open Source*

Software Supply Chain

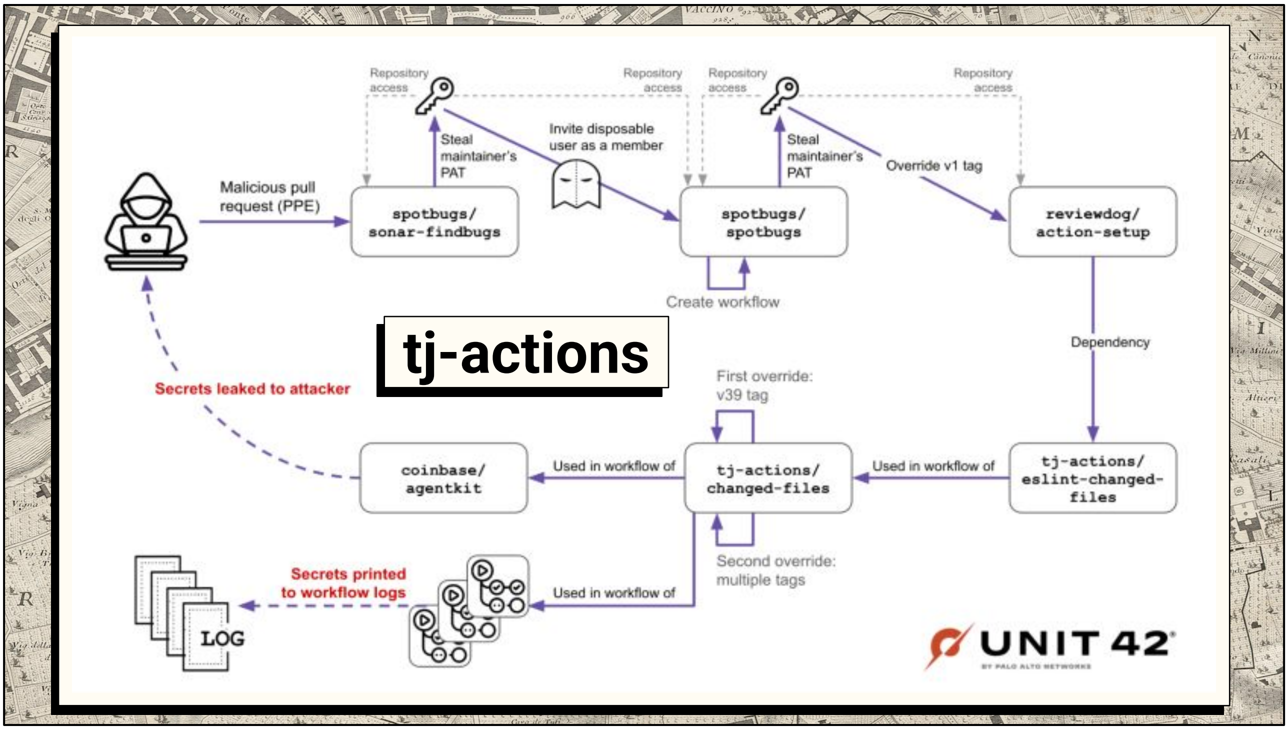
A small caveat on these definitions that they were defined as “Open Source Software Supply Chain”, but i feel they work well without the Open Source tag as well.



tj-actions

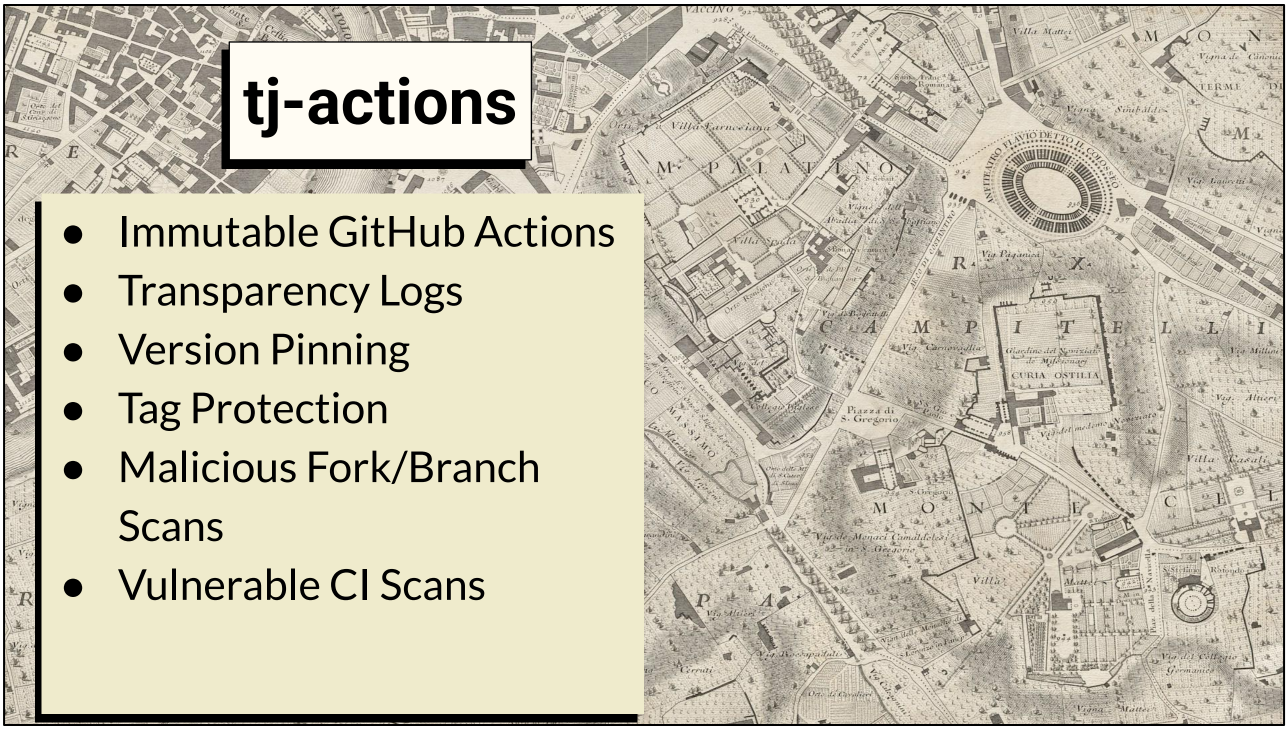
xz-utils

Let’s get a bit more concrete. And look at two recent attacks. Tj-actions was just a few months ago, and xz-utils was last year.



tj-actions

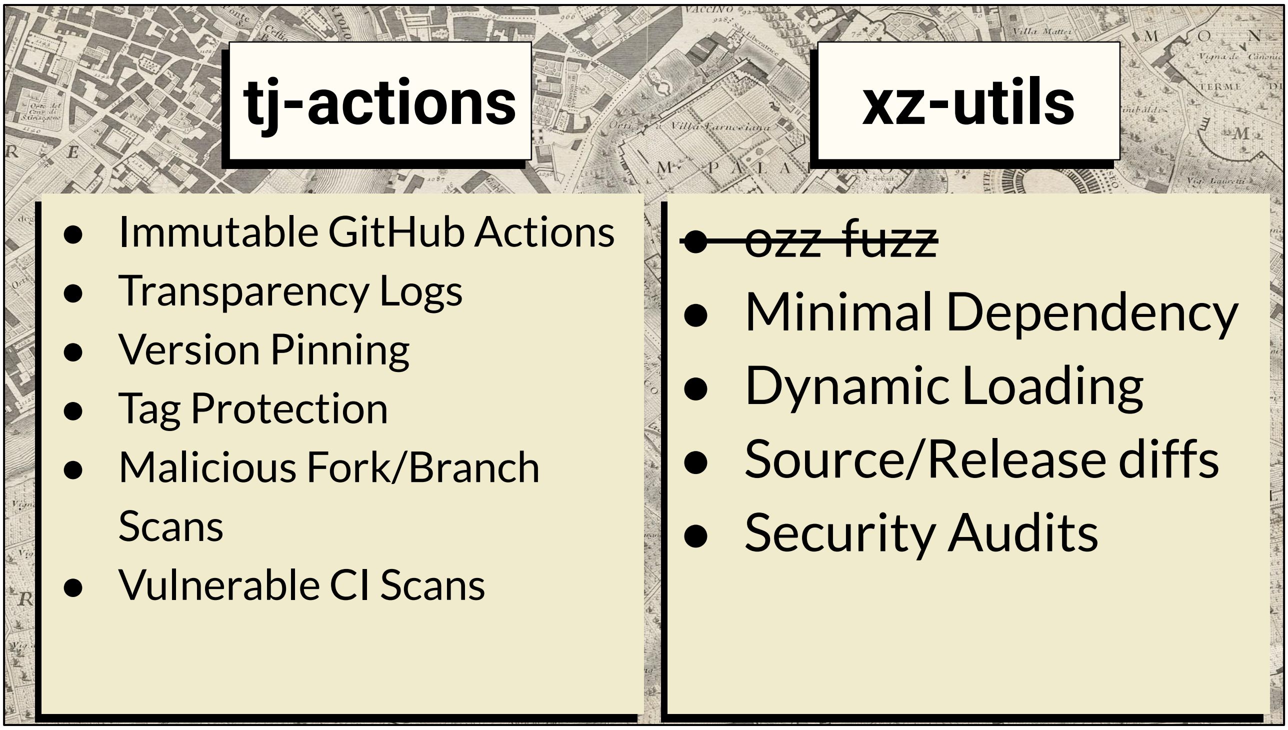
An important thing about tj-actions was how it took months of effort and secrets leaked across 6 different repositories for the final attack. This isn't a point and shoot attack, but a dedicated persistent attacker. There's more details in the leaflet that you can find at Rootconf.



tj-actions

- Immutable GitHub Actions
- Transparency Logs
- Version Pinning
- Tag Protection
- Malicious Fork/Branch Scans
- Vulnerable CI Scans

Instead of focusing on the attack, let us look at how we could have prevented or detected these attacks earlier. In the case of tj-actions - which used Github Actions to pivot and escalate - simple versioning pinning or immutable action artifacts would have prevented the attack. Tag Protection would have prevented the tag from being overwritten. Since the attack relied on malicious forks and branches being created as a recon and attack mechanism - any scanners looking out for temporary branches would have caught it. Existing CI system scanners such as “zizmor” would have caught the vulnerable pull_request_target vulnerability as well.



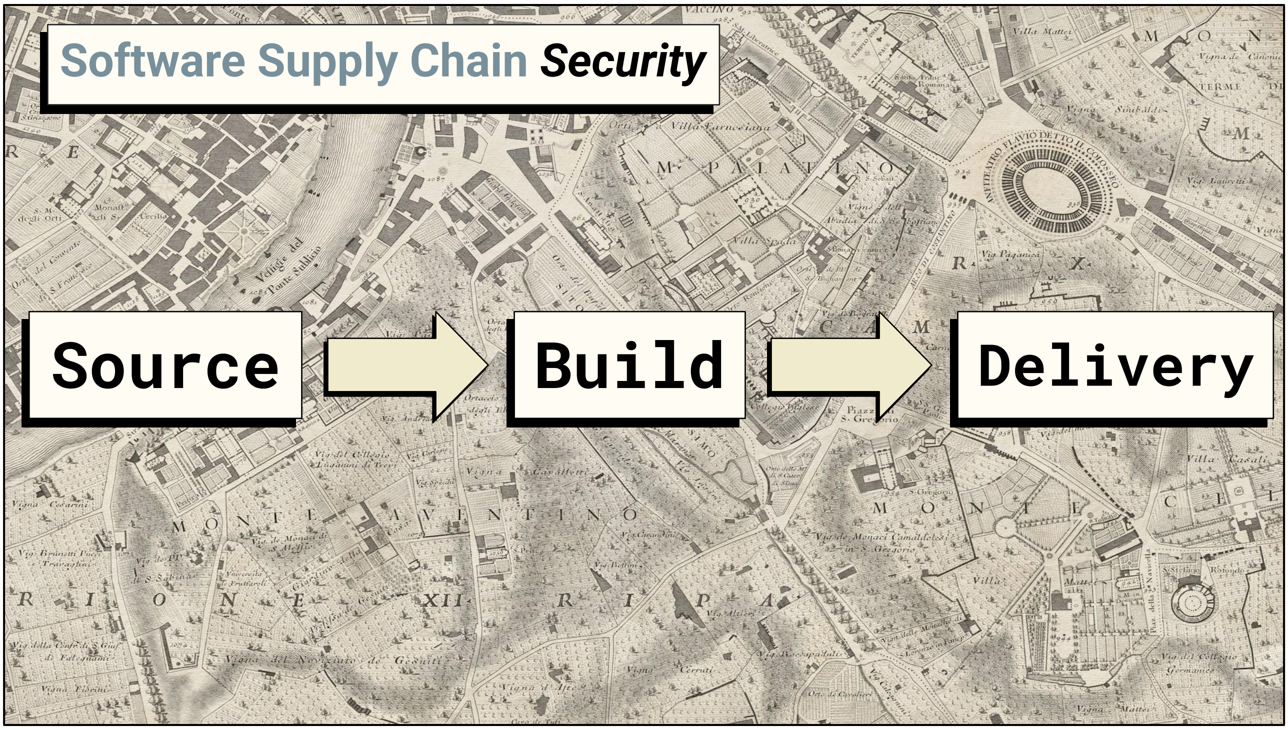
tj-actions

- Immutable GitHub Actions
- Transparency Logs
- Version Pinning
- Tag Protection
- Malicious Fork/Branch Scans
- Vulnerable CI Scans

xz-utils

- ~~ozz-fuzz~~
- Minimal Dependency
- Dynamic Loading
- Source/Release diffs
- Security Audits

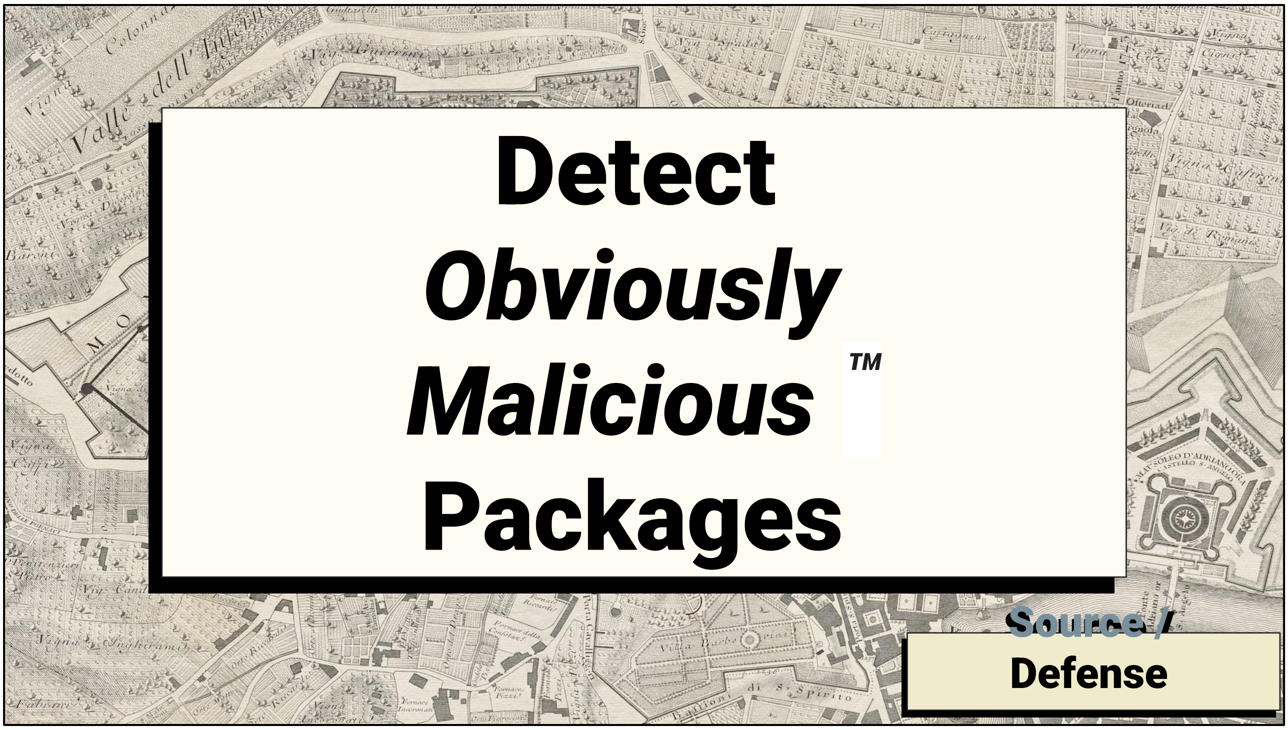
On xz-utils, since it was a malicious insider. ozz-fuzz where valgrind caught some bugs in the backdoor would not have helped, mainly because fuzzing tooling is not looking for backdoors. Switching to minimal dependencies would have helped, as Gentoo which didn’t link ssh to libsystemd wasn’t impacted. Dynamic Loading of dependencies at runtime (instead of Dynamic Linking) via dlopen style calls would have helped. Systemd has now switched to that already. There aren’t any systems doing source v/s release diffs, but if there were - they might have caught the autoconf backdoor as an obvious large change in the xz releases. And finally, a security audit might have picked up the malicious changes, and maybe the delivery mechanism.



Software Supply Chain *Security*

Source → Build → Delivery

This is the common framing in use for Software Supply Chain Security - breaking it into your source which is what goes into your software. Then the build stage where your code and the dependencies get built, and finally the delivery stage where your software reaches your end-users. A compromise in any of these three results in a software supply chain attack. Lets look at all three of these from a bird-eye lens - rapid fire



Detect *Obviously* *Malicious* ™ Packages

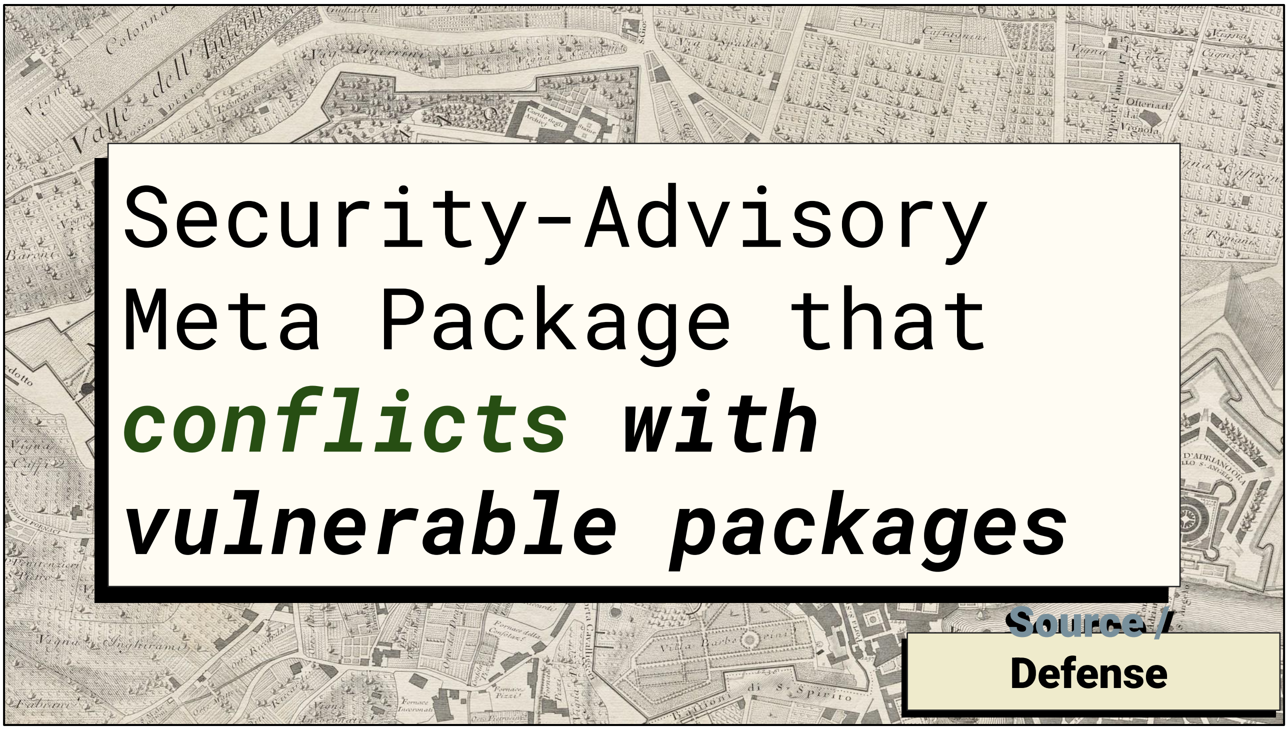
Source / Defense

The first is detecting obviously malicious packages. This isn’t hard. Packages with pre-install scripts that use base64 to encode a malware drop stand out very easily. Existing tooling can detect such malicious packages today. The harder problem is:

Prevent *Obviously Malicious* Package Installation

Source / Defense

Preventing such packages from being installed in the first place is much harder. However, there is tooling which can scan packages at install time and prevent them from being installed. Look at https://github.com/safedep/pmg or https://github.com/DataDog/supply-chain-firewall for example.



Security-Advisory Meta Package that ***conflicts with vulnerable packages***

Source / Defense

Another interesting approach is to create a meta package that conflicts with all known vulnerable packages. The PHP community has been doing it at https://packagist.org/packages/roave/security-advisories, and this one package prevents installation of known vulnerable packages.

Typosquatting
Slopsquatting

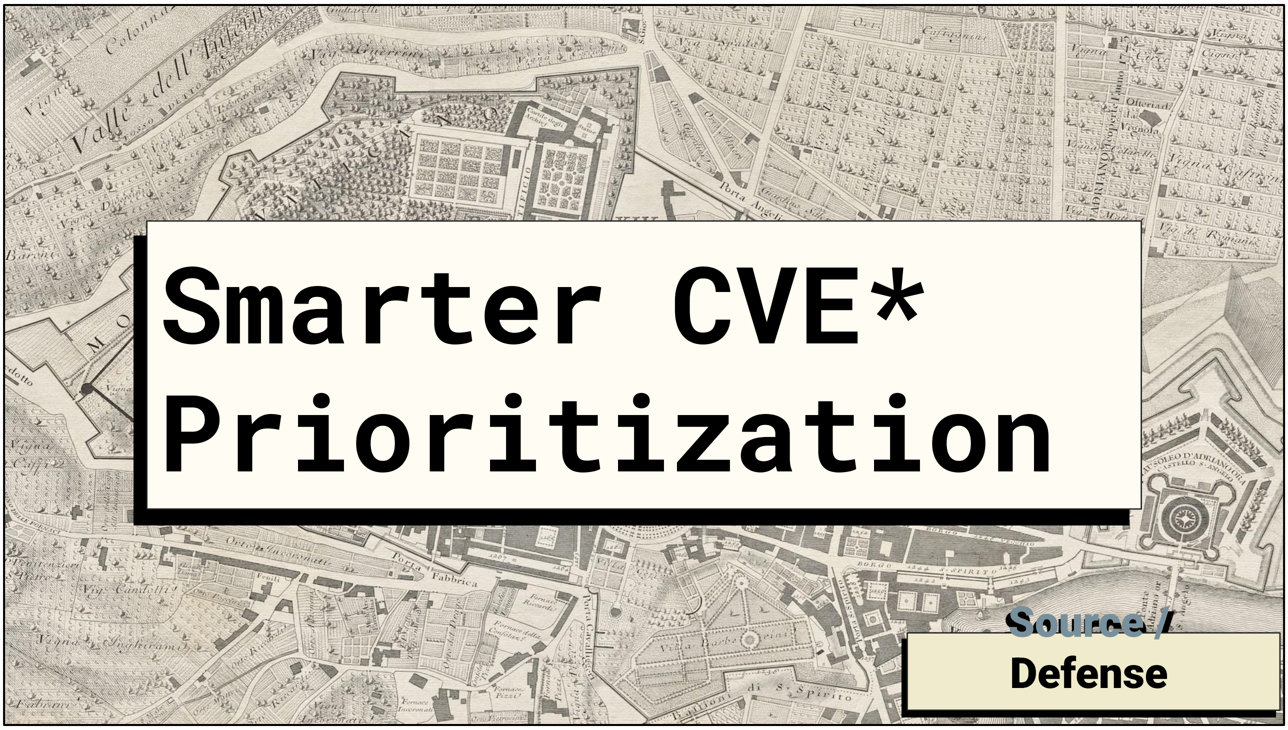
Source / **Attack**

Two new attacks are typosquatting and slopsquatting. Typosquatting is quite old, where malicious packages with a similar sounding name to a popular package are registered. Slopsquatting similarly is registering package names that are commonly hallucinated by LLMs, and using them for serving malicious code.

Improve Security Scan cadence ⏰

Source / Defense

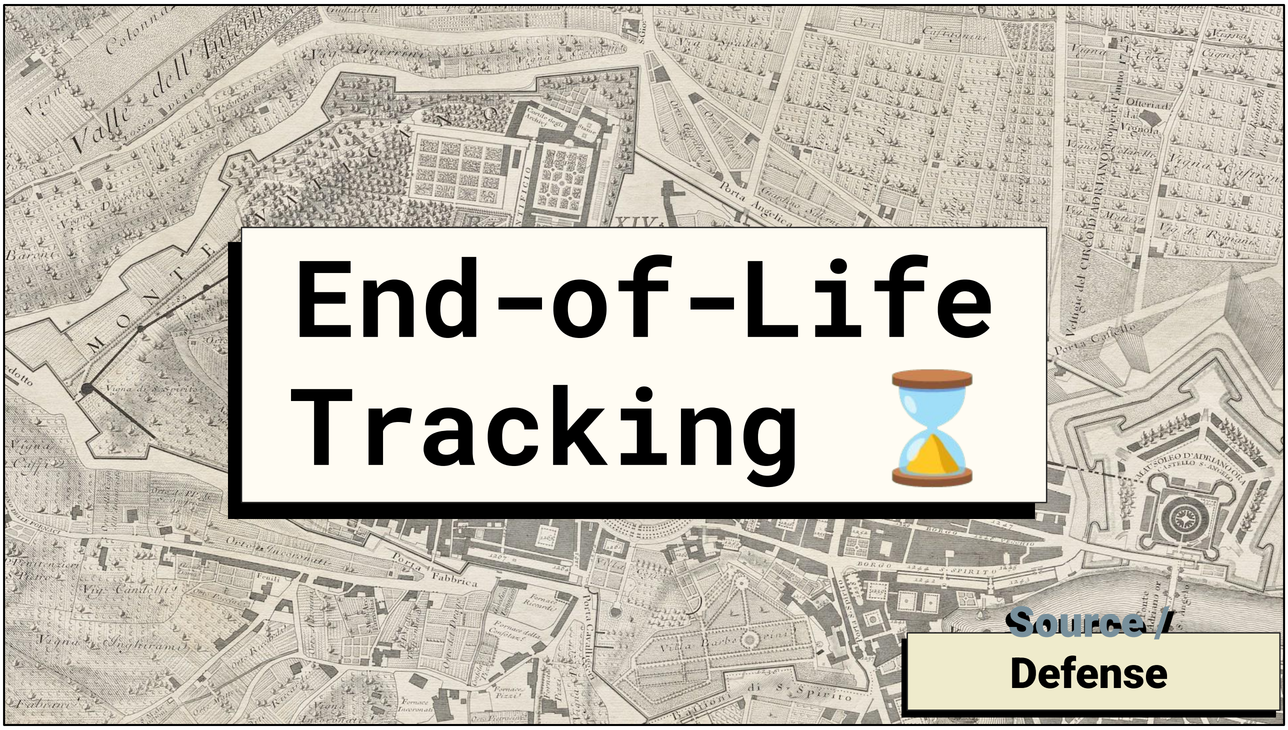
Increasing the security cadence of your scans, whatever shape these scans take, is an easy way to make your posture aligned with reality. Whatever it might be, get it lowered. See if you can scan every commit and every branch, instead of just scanning the master branch every day.



Smarter CVE* Prioritization

Source / Defense

Prioritize your CVEs (and other vulnerability data). Look at KEVs from CISA on what vulnerabilities are being exploited. Look at not just the CVSS score, but EPSS as well. Check what your distro is saying about the vulnerability.



End-of-Life Tracking ⏳

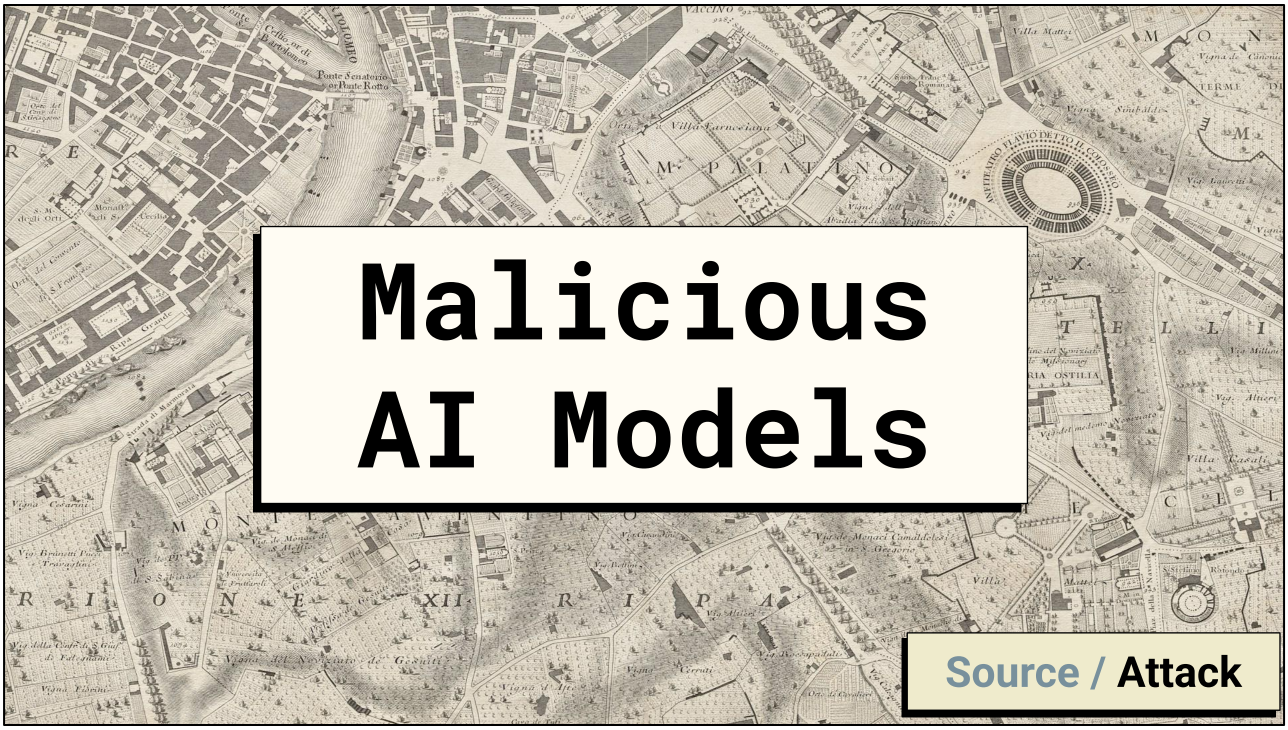
Source / Defense

Track your EOL dates. This is what I do at endoflife.date, so this is close to my heart, but EOL tracking is essential for your critical dependencies. Because when a CVE drops, you might not have a clear pathway to a safe upgrade because you were on an EOL version, but there was no alert till it was very late.

Trusted OSS Supplier

Source / Defense

Look at trusted/curated OSS Supplier Programs. Google, Chainguard run one each, but there’s other companies as well. Every distro in a way runs their own trusted oss program rebuilding packages from source code.



Malicious AI Models

Source / **Attack**

Another interesting attack is coming via malicious AI model pickling attacks, where a model can include deserialization attacks for malware.

OpenSSF Scorecard

Source / Defense

Look at OpenSSF scorecard to score your dependencies on how they are faring on security. Maybe there’s an opportunity to switch.

Audit your SBOMs

Source / **Defense**

Every security engineer will tell you that SBOMs are leaky - they don't reflect the reality of your production systems. Most SBOM tooling is a by-product of our existing package-management ecosystems, and thus misses on so much more (compiler toolchains, ad-hoc dependencies, downloaded binaries, software installed with a make install). Something that's easy to do is to take your SBOM and audit it - check it for missing software, across different layers of your stack.

Score your Dependencies

Source / Defense

Score your dependencies, not just with the OpenSSF scorecard. But there’s so much more you can use to score them - look at whether they are included in linux distros or assured source programs, check the number of dependents of a package, calculate your risk according to what it does, and where it runs. Look at whether the repository of the dependency still has any activity, whether it has been archived or yanked.

Security Commons Funding

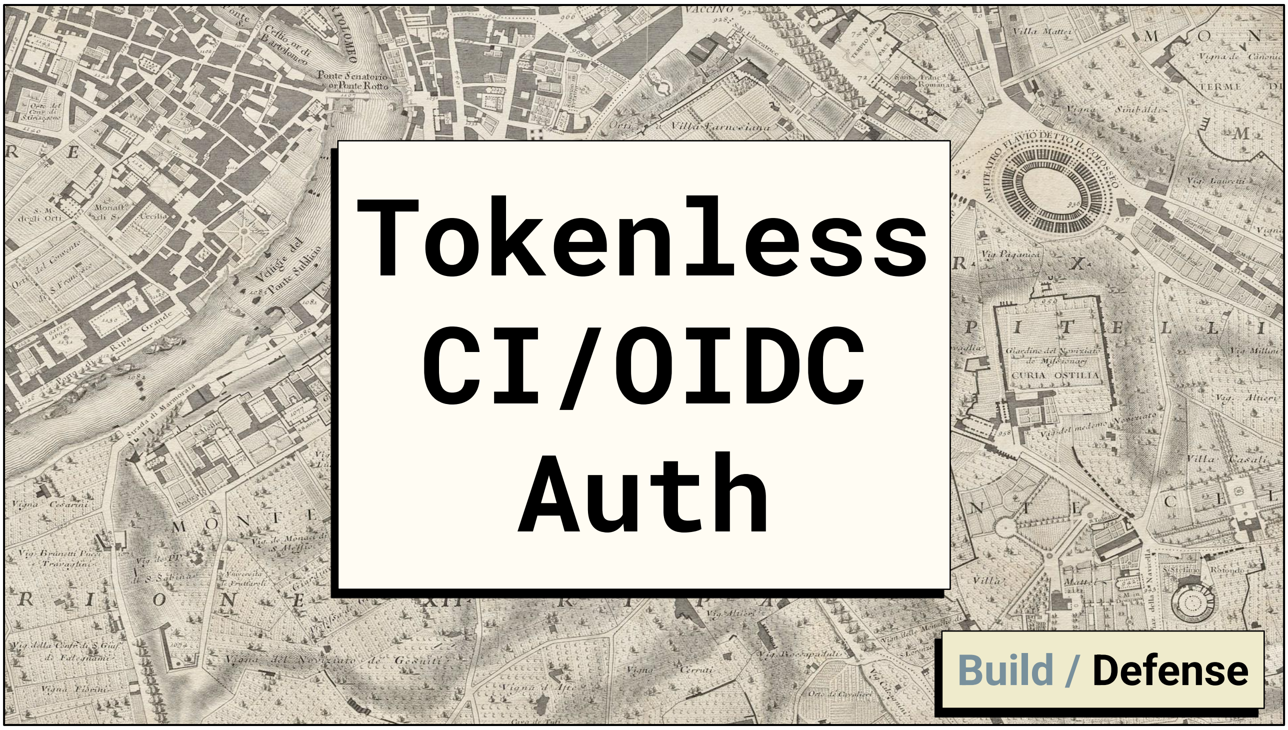
Source / Vulnerability

A quick note about the MITRE and NVD funding crises that happened last and this year. The leaflet in your hands talks a bit more about the crisis. The funding for both the NVD and the CVE programs is a single source US-agency funding which has caused a risk in the security ecosytem because these databases are critical to everything else that we do as an industry. These are also vulnerabilities in the system.

Commit /Release Signing

Build / Defense

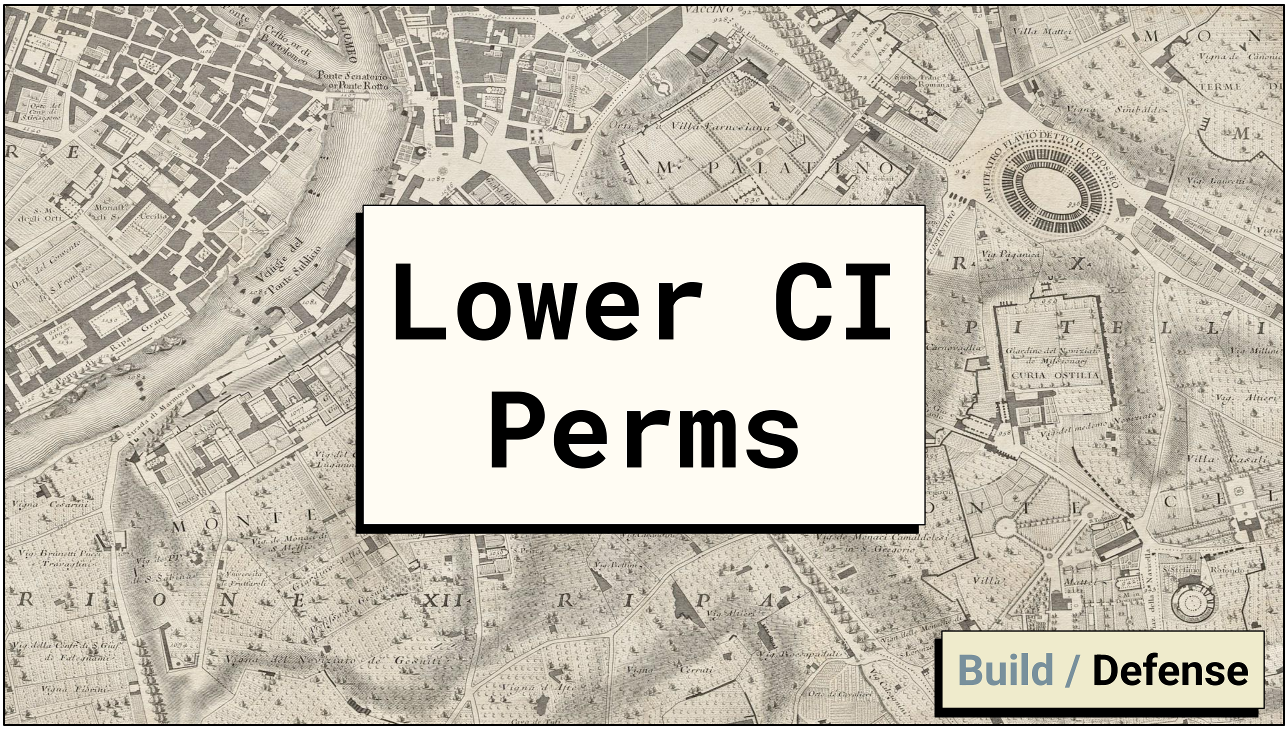
Moving to defense, a quick way to increase your confidence in the source code is to enforce comit/release signing for CI systems to be triggered, relying on a hardware signature or a JIT-OIDC-attestation. This avoids token-theft attacks from being easily leveraged to push code to the build systems.



Tokenless CI/OIDC Auth

Build / Defense

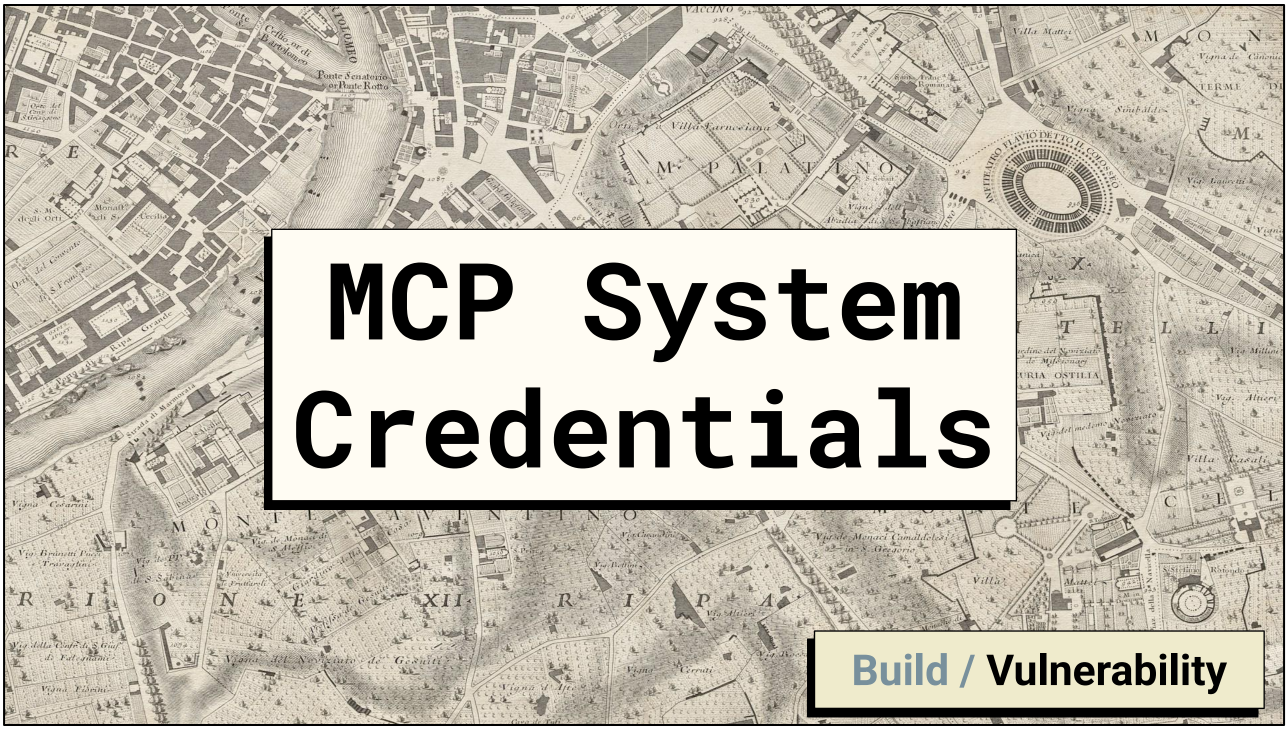
My favorite security system rollout in the last few years have been OIDC Attestation rollouts in CI systems that can be chained with external systems to deploy service-authentication mechanisms without any long-lived keys. See https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/configuring-openid-connect-in-cloud-providers for how GitHub Actions supports this for example.



Lower CI Perms

Build / Defense

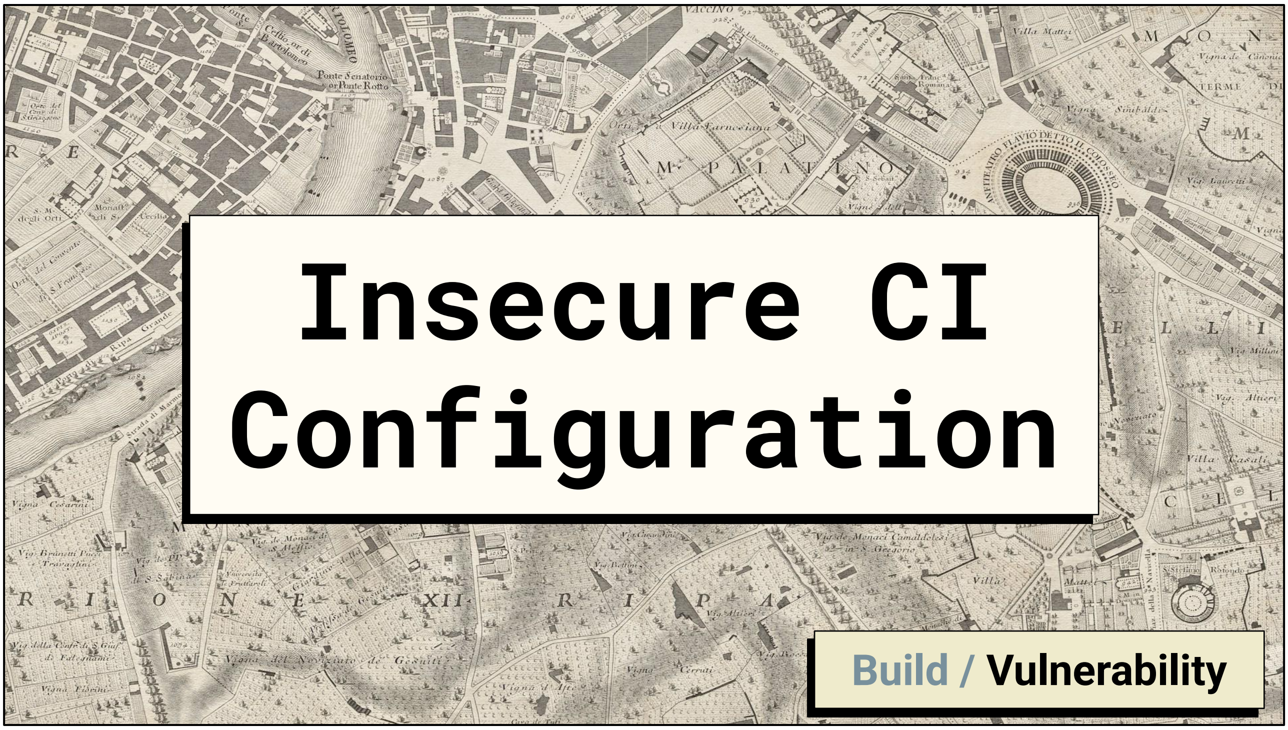
Age old security paradigm still applies: Run with least permissions in your CI systems. Split your risky pipelines by execution environments (jobs) to avoid environment leak attacks, such as the one used in the tj-actions attack.



MCP System Credentials

Build / **Vulnerability**

Another vulnerability in AI systems comes from how MCP (Model Context Protocol) clients save their credentials. With more MCP servers coming up every day, there are more and more tokens being stored alongside these MCP systems, which have a new risk of their own.



Insecure CI Configuration

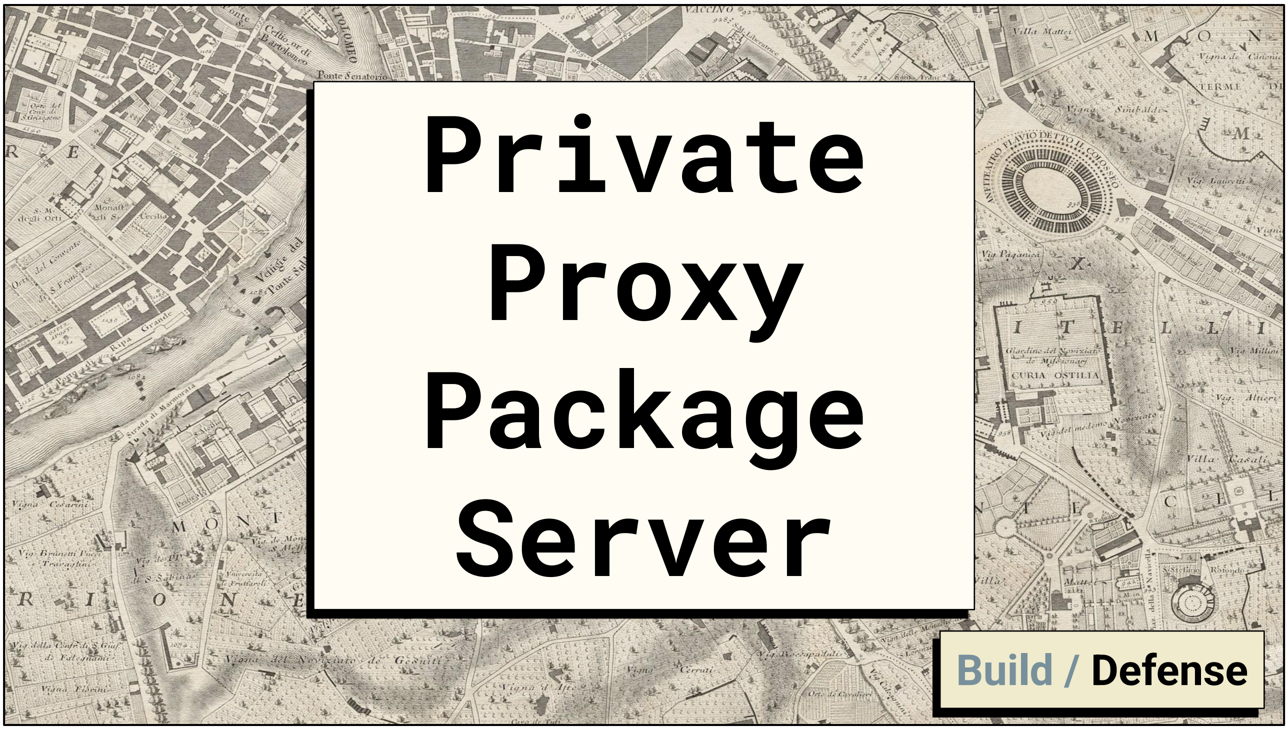
Build / **Vulnerability**

CI pipelines can be scanned by modern tooling like zizmor. Almost every platform also provides their Well Architected or Best Practice Guidelines for security. Benchmark yourselves against that, see if your tooling supports scanning against that. Avoid using pull_request_target - it is just too hard to use safely.

Lockfiles

Build / Defense

Lockfiles are great, and the guarantees that they provide are essential in modern SRE practices. But there’s still scenarios where lockfiles are not used often, such as distribution packages inside a container, or github actions. If you’ve the early wins, then focus on the harder ones.



Private Proxy Package Server

Build / Defense

Setup a private proxy package server. It gets you free caching that protects against upstream outages, but also lets you do far more interesting things - block installation of all known vulnerable packages. Alert the security team when an archived package is installed.

Publish SBOMs

Delivery / **Defense**

Publish your SBOMs, even if this is internally. SBOMs are amazing, and having your entire SBOMs in one place where your entire engineering team can look at it is a great leverage point for an org if used well. You can not only check what you're using, but there's a lot of data inside the SBOMs that can be leveraged for making important decisions.

Release Attestations

Delivery / Defense

Do release attestations via Sigstore. Gives you a guarantee of what you’re releasing has a clear and verifiable provenance

Trusted Publishing

Delivery / **Defense**

Trusted Publishing is using the OIDC Attestations in CI systems to publish your release packages. Doesn’t apply everywhere, but if you’re publishing packages to a place like NPM or PyPi, you can use Trusted Publishing. This also applies to publishing to something like a S3 bucket, it just takes a bit more work.

Release Diff Alerts

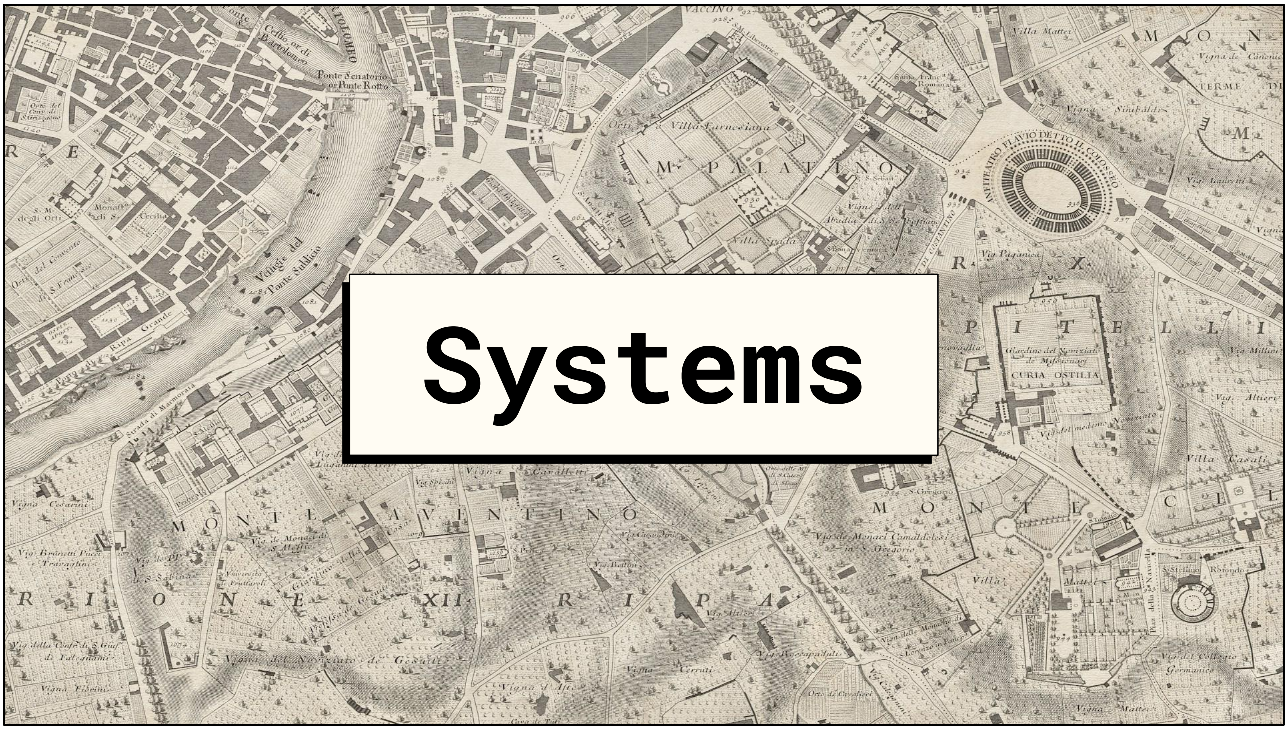
Delivery / **Defense**

Consume your release artifcats as an end-user would, and diff+validate them against your final builds. If you had a provenance attestation, this is easier but if not - do diffs. Many supply chain attacks in the past would have been detected much earlier with a process like this in place

Token Theft

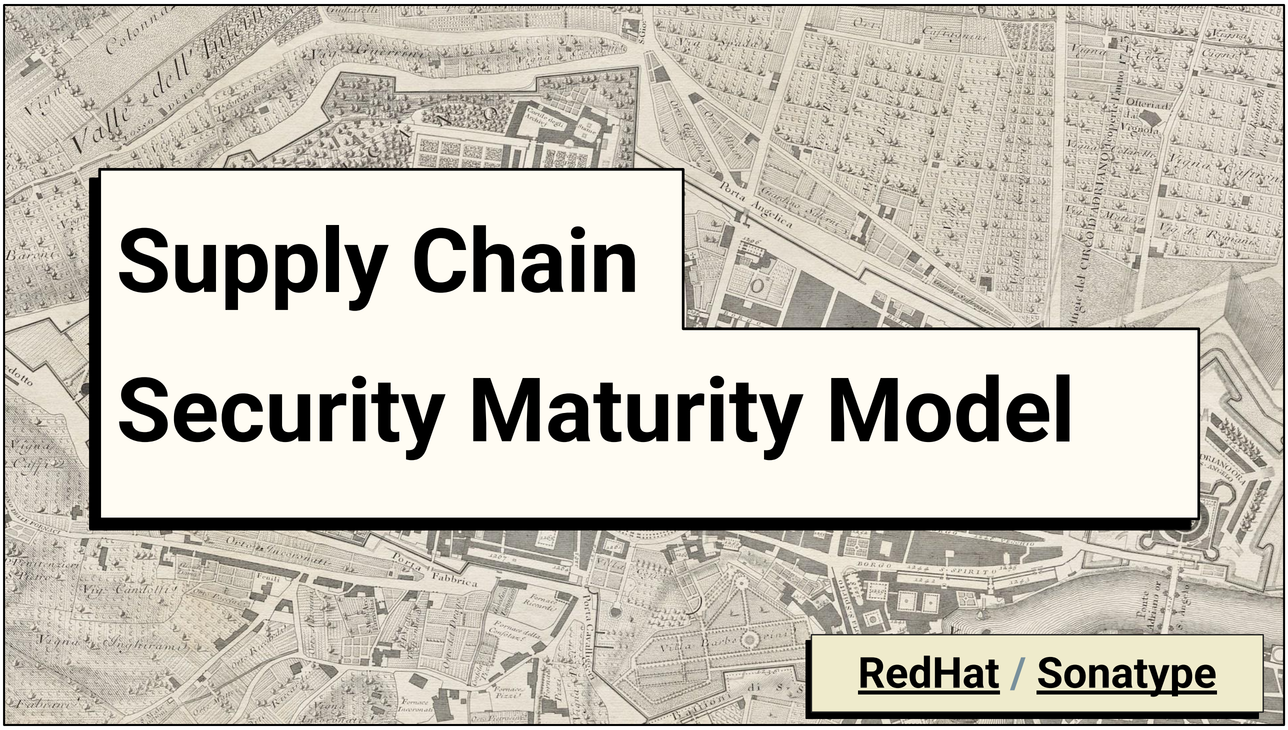
Delivery / **Attack**

Token theft attacks are fairly common escalation mechanisms. Setup canary tokens, and additional checks such as IP address allowlists against your tokens to both reduce the risk, and trigger alerts.



Systems

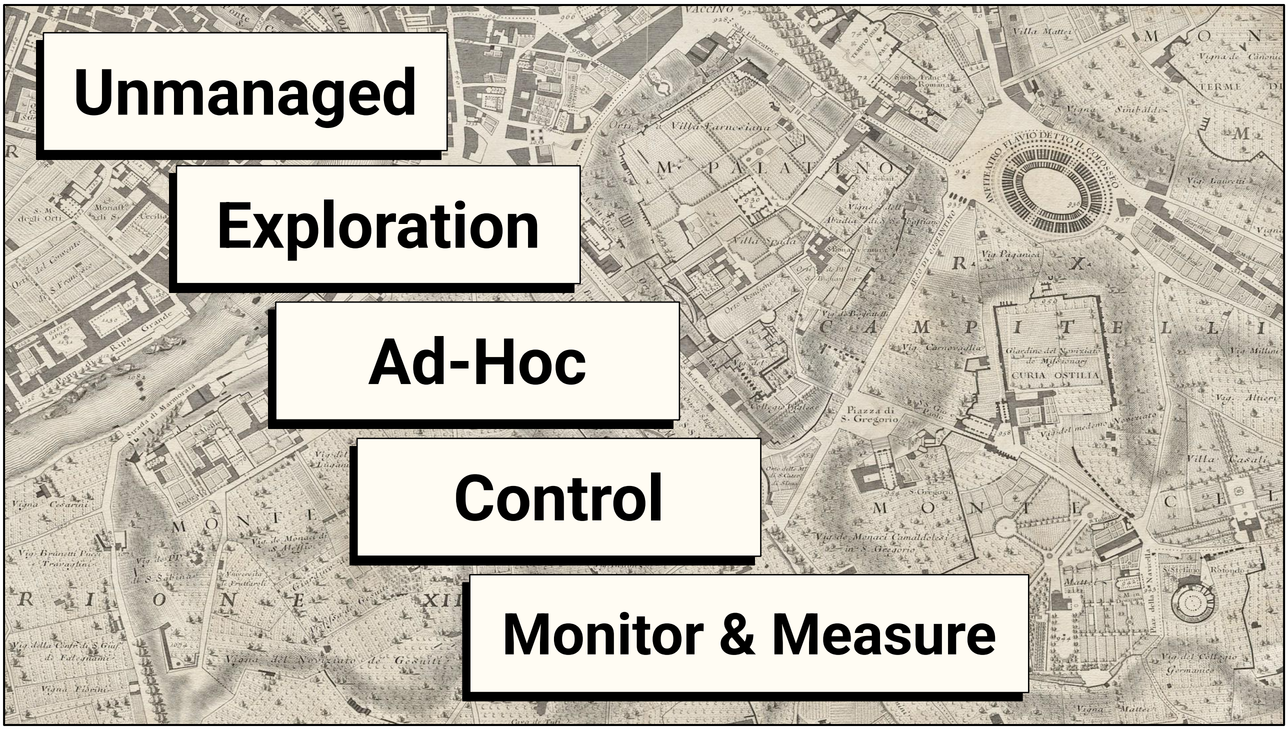
Phew! That was alot of ideas. But as a security team how can you deploy these systems in a reasonable manner?



Supply Chain Security Maturity Model

RedHat / Sonatype

I’ll start with a Software Supply Chain Security Maturity Model. There’s 2 companies that have published their take on it -Redhat and Sonatype.



Unmanaged

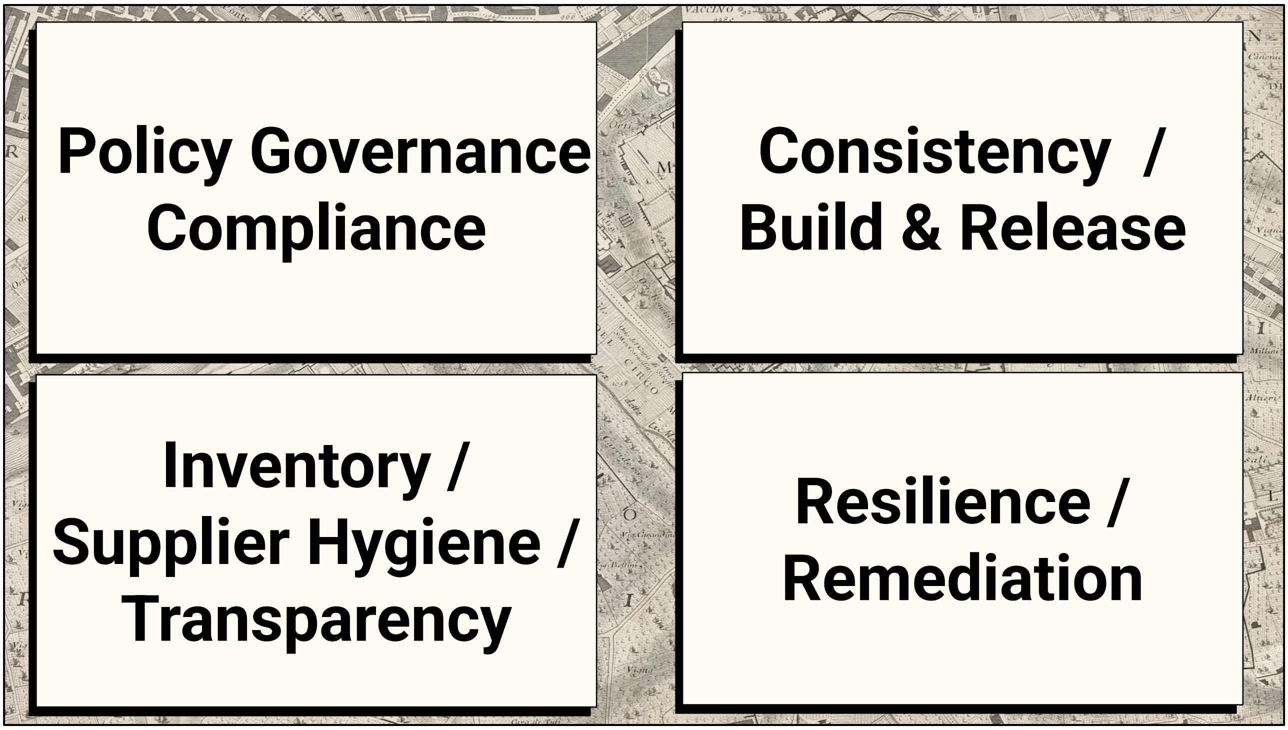
Exploration

Ad-Hoc

Control

Monitor & Measure

A maturity model benchmarks your organization on a given theme or competency across 4-5 different levels. The higher the level, the more mature your org in that field as the rolouts are more mature, controlled, and data reliant. At the earliest stages, it could be an unmanaged exploration, such as an org where any developer can install any dependency across any system.



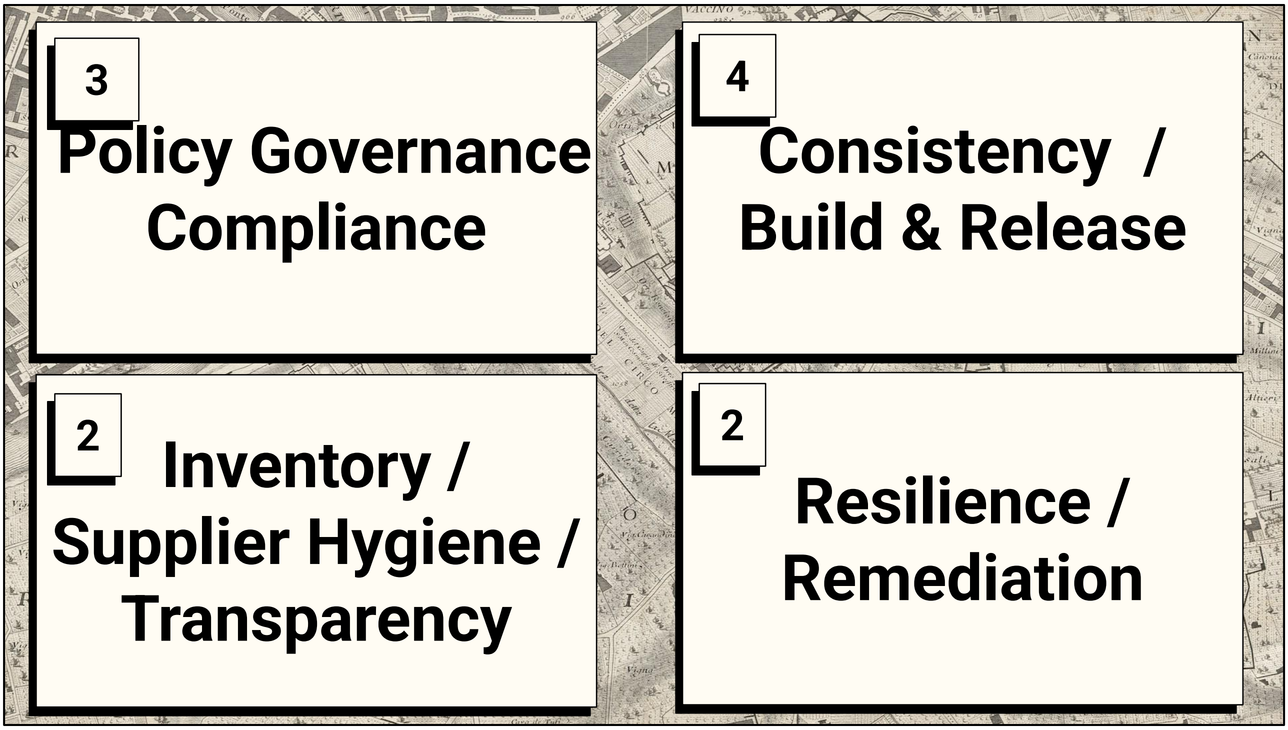
Policy Governance Compliance

Consistency / Build & Release

Inventory / Supplier Hygiene / Transparency

Resilience / Remediation

The maturity models split this progression over 4 broadly similar themes: Policy / Builds / Inventory / Remediation. The first tracks your compliance requirements and the maturity of your internal policies for things like OSS Consumption, license checks etc. The second tracks where you stand on the security and consistency of your builds - are they reproducible, are they pinned, are they homogenous? Finally, the last two track you on your supplier hygiene and remediation efforts for when something happens, such as a CVE showing up in scans.



3

Policy Governance Compliance

4

Consistency / Build & Release

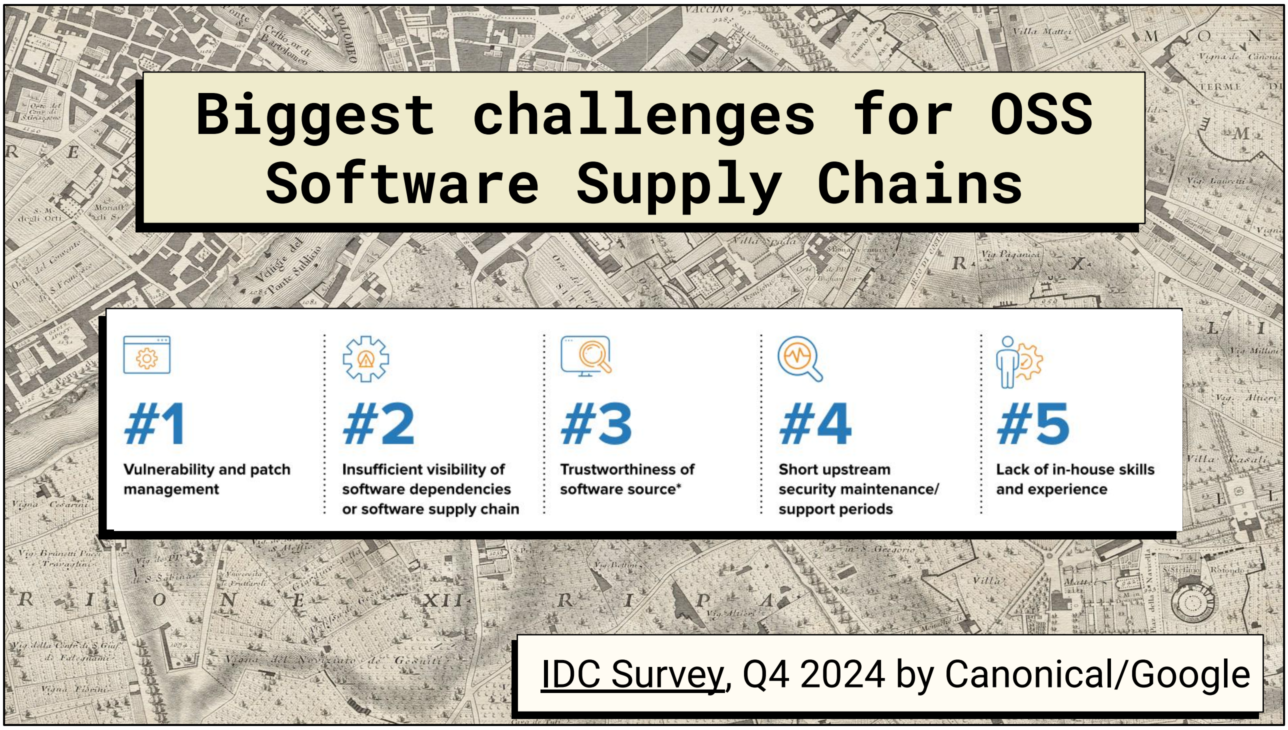
2

Inventory / Supplier Hygiene / Transparency

2

Resilience / Remediation

You will typically set your own targets for what changes will get you from Ad-Hoc to Controlled to Monitored maturity. Think of items like: Coverage on your OSS dependency tree, or SBOM coverage, or coverage of builds with provenance. For remediation, look at timelines and track how quick your response times are to a critical or high CVE. There’s a lot more in the guides that I’m referencing - do take a look.



Biggest challenges for OSS Software Supply Chains

#1
Vulnerability and patch management

#2
Insufficient visibility of software dependencies or software supply chain

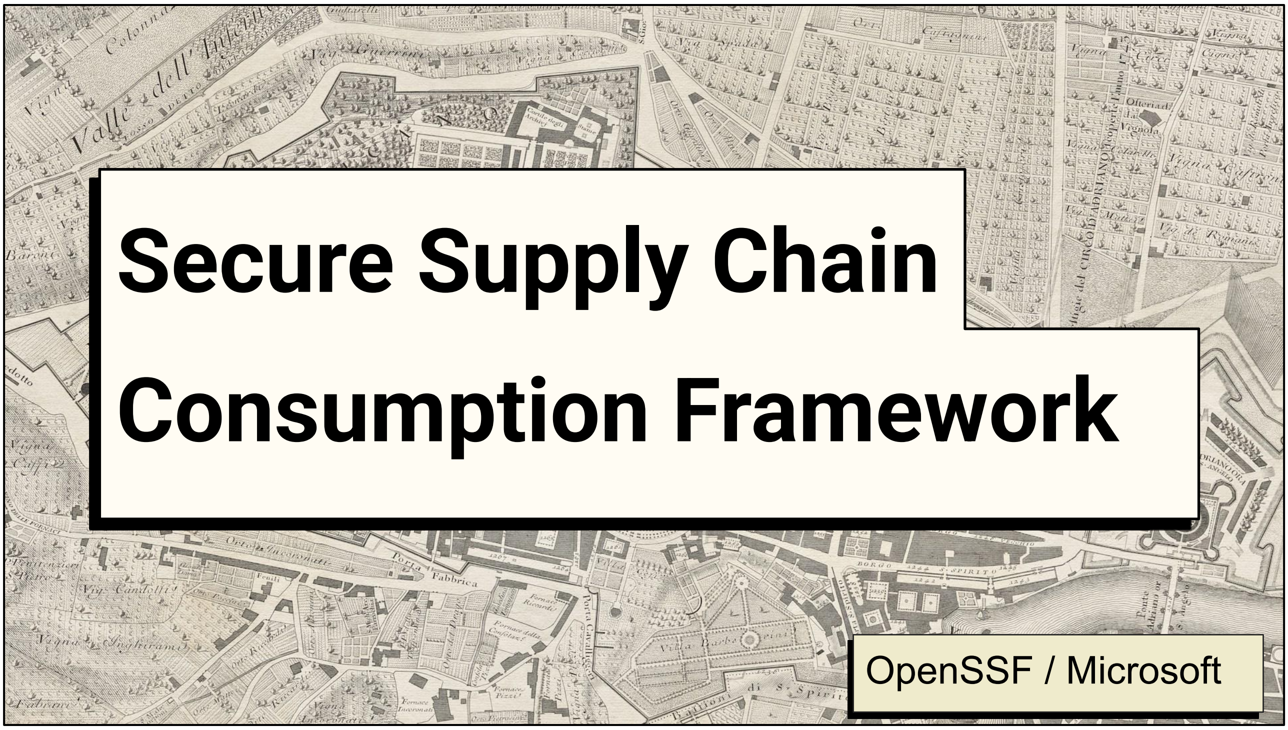
#3
Trustworthiness of software source*

#4
Short upstream security maintenance/ support periods

#5
Lack of in-house skills and experience

IDC Survey, Q4 2024 by Canonical/Google

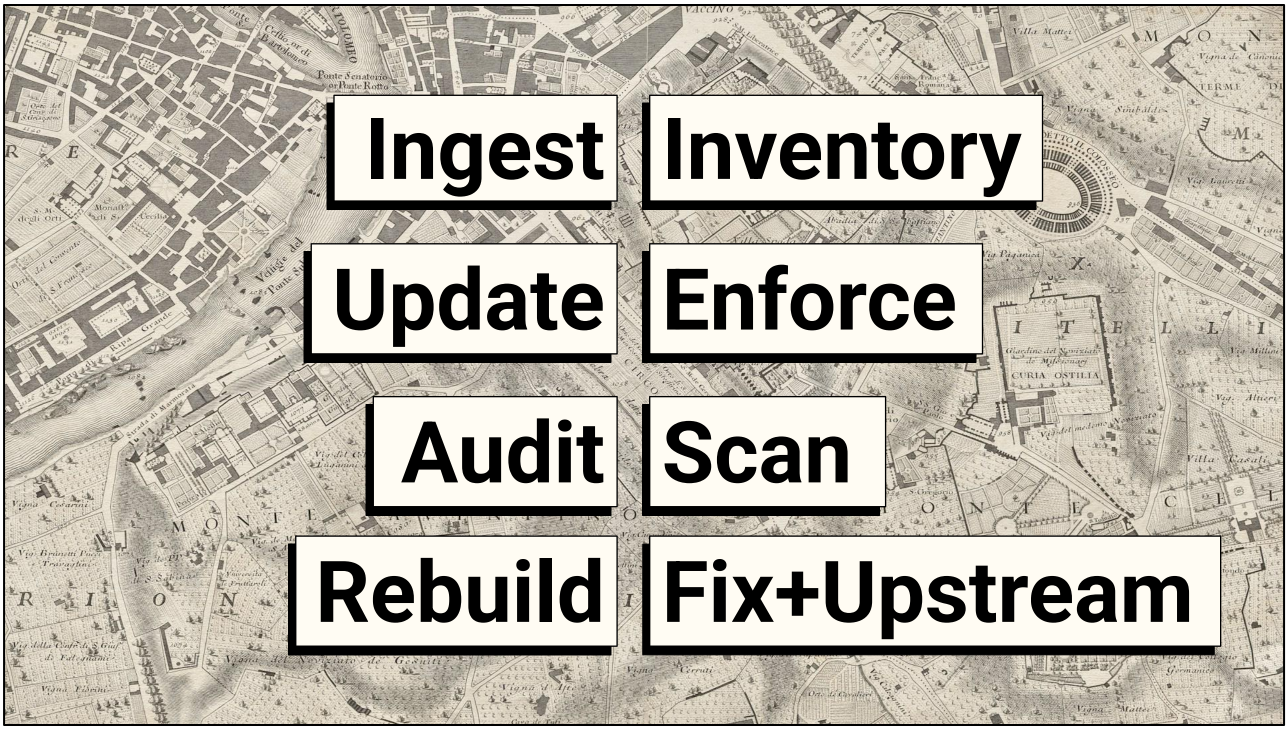
The Supply Chain Security world is quite broad, but as it turns out the majority of challenges that companies are facing come from Software ingestion itself. As per a IDC Survey commissioned by Canonical/Google, the number 1 challenge faced by enterprises is Vulnerability and patch management, followed by insufficient visibility into the software dependencies.



Secure Supply Chain Consumption Framework

OpenSSF / Microsoft

If you’d like to focus on the consumption side, look at the Secure Supply Chain Consumption Framework. Created by Microsoft, and then donated to the OpenSSF Foundation, it provides guidelines and levels on how you can securely consume third-party software.



Ingest

Inventory

Update

Enforce

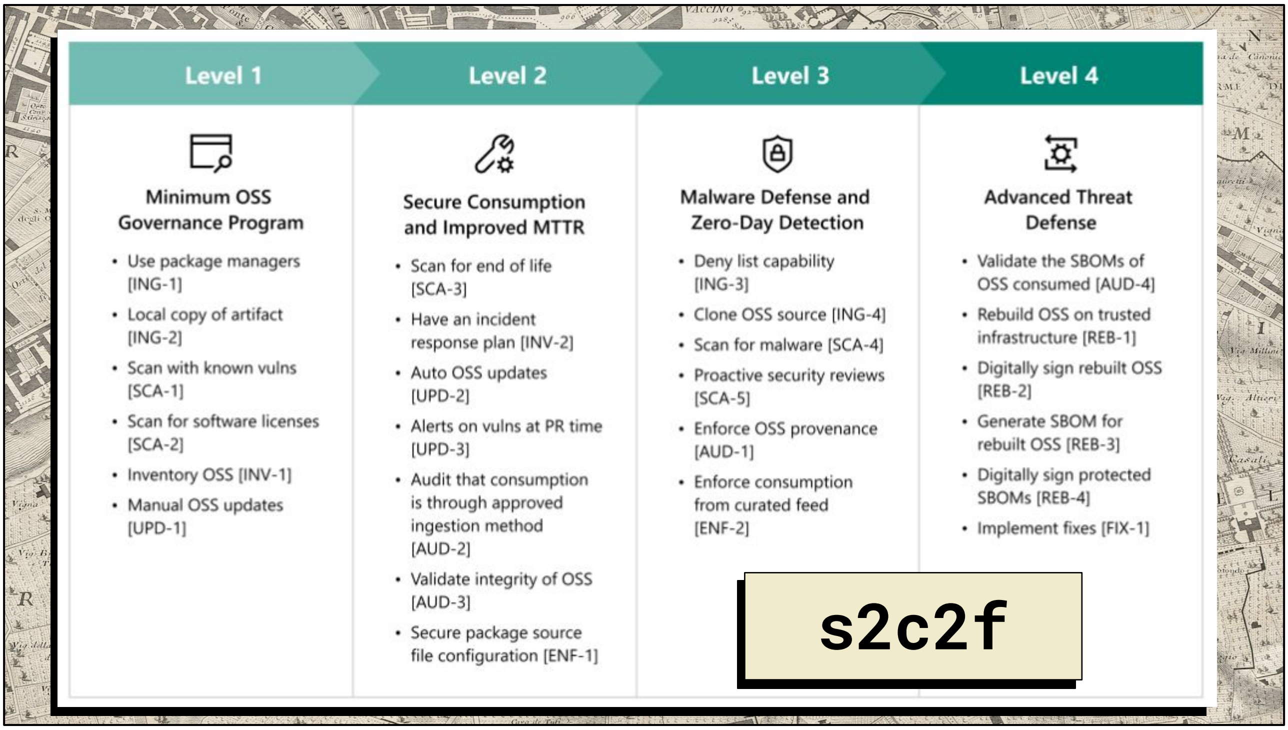
Audit

Scan

Rebuild

Fix+Upstream

It broadly covers tasks across 8 tracks: For every third-party component, how do you ingest it, update it, and track it in your inventory. Further, ti asks you to enforce your policies, scan and audit for the same. At the highest levels, it requires you to rebuild the OSS dependencies, and be prepared to patch them in-house if needed.



Level 1

Minimum OSS Governance Program

- Use package managers [ING-1]
- Local copy of artifact [ING-2]
- Scan with known vulns [SCA-1]
- Scan for software licenses [SCA-2]
- Inventory OSS [INV-1]
- Manual OSS updates [UPD-1]

Level 2

Secure Consumption and Improved MTTR

- Scan for end of life [SCA-3]
- Have an incident response plan [INV-2]
- Auto OSS updates [UPD-2]
- Alerts on vulns at PR time [UPD-3]
- Audit that consumption is through approved ingestion method [AUD-2]
- Validate integrity of OSS [AUD-3]
- Secure package source file configuration [ENF-1]

Level 3

Malware Defense and Zero-Day Detection

- Deny list capability [ING-3]
- Clone OSS source [ING-4]
- Scan for malware [SCA-4]
- Proactive security reviews [SCA-5]
- Enforce OSS provenance [AUD-1]
- Enforce consumption from curated feed [ENF-2]

Level 4

Advanced Threat Defense

- Validate the SBOMs of OSS consumed [AUD-4]
- Rebuild OSS on trusted infrastructure [REB-1]
- Digitally sign rebuilt OSS [REB-2]
- Generate SBOM for rebuilt OSS [REB-3]
- Digitally sign protected SBOMs [REB-4]
- Implement fixes [FIX-1]

s2c2f

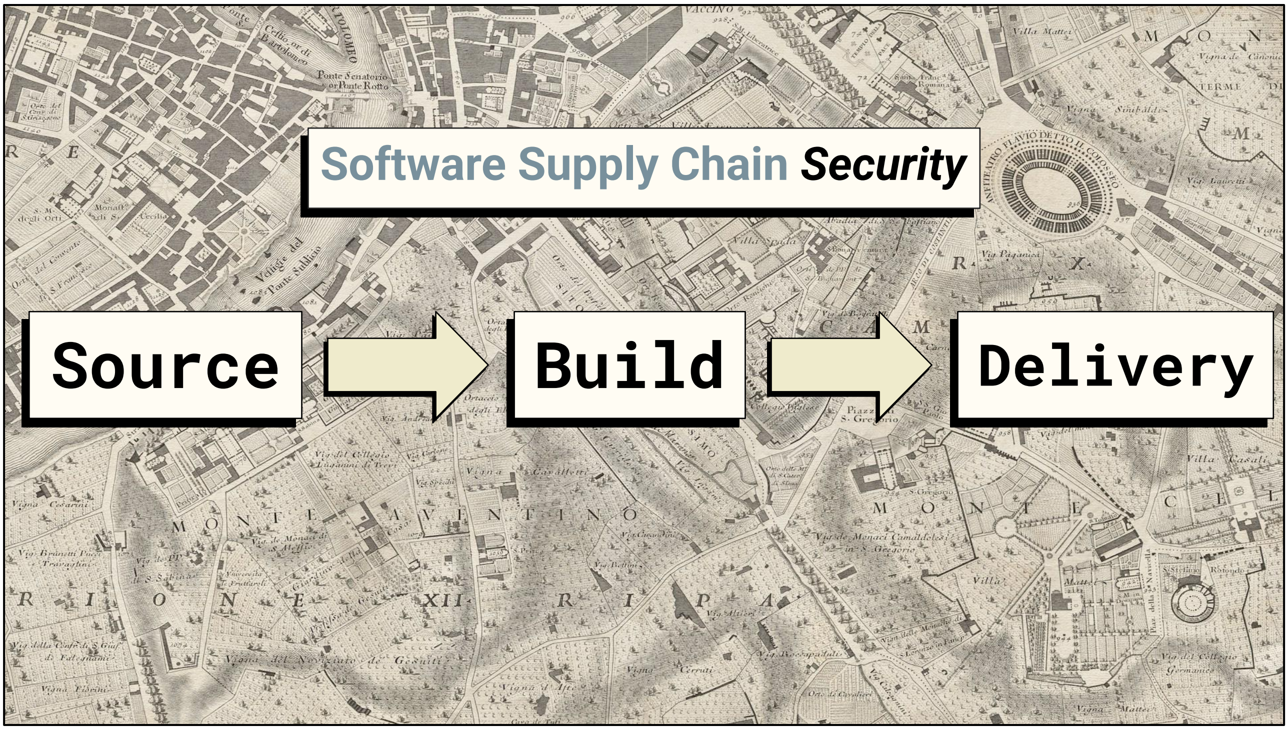
This is how the 4 Levels look like. At the basic level, it focuses on Updates, Scanning, and Ingestion guidelines. As you move up, it asks you to do harder things, like scanning for malware, or enforcing OSS provenance, and consuming OSS components from a curated feed - something we covered earlier.

SLSA

And then we get to SLSA (slsa.dev), which is a framework that primarily focuses on build provenance. Provenance is the verifiable information about software artifacts describing where, when, and how something was produced. It includes information about your build, the components, and exactly where and how it was run - ideally in an isolated environment. In case of a breach, the provenance can help you trace back where something happened, and do things like - knowing the extent of the compromise. All of this happens with cryptographic attestations (via sigstore or the like) so your consumers not only get the same guarantees, but they can also verify the same.

Open Software Supply Chain Attack Reference (OSC&R)

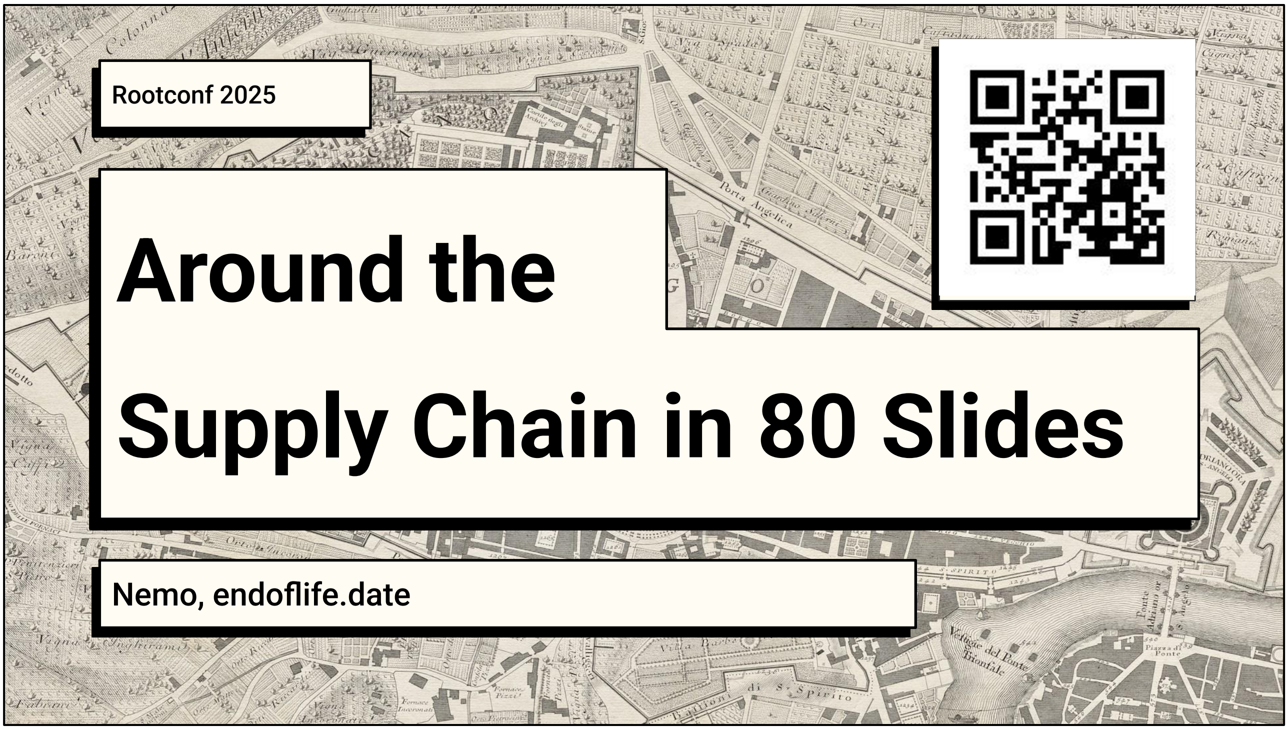
Something else that you can look at is the Open Software Supply Chain Attack Reference (OSC&R, pronounced OSCAR). It is an analogous framework to the MITRE ATT&CK framework, but focused on supply chain security attacks. It includes various techniques on recon, attacks, impersonation used in these attacks, as well as enumerated scenarios for lateral movements. It has several tracks, including Container Security, OSS Consumption, Cloud Security, Secret Consumption and a few more.



Software Supply Chain *Security*

Source → Build → Delivery

That’s all. I hope this was a nice intro to what Supply Chain Security looks like in 2025.



Rootconf 2025

Around the Supply Chain in 80 Slides

Nemo, endoflife.date

I’ll be around this track the entire day, come talk to me about EOL, SBOMs, or interesting supply chain security attacks.