Securing the Development & Supply Chain of Open Source Software (OSS)

Stéfane Fermigier

Cours à l'EPITA - 22 nov. 2022

Outline

- Open Source Software (OSS) is everywhere!
  - All software under attack & users are not updating to fixed versions of OSS
- How can OSS developers develop & distribute secure OSS (today)?
- How can potential OSS users select secure OSS (often by looking for that)?
- How OSS is developed & distributed - a supply chain (SC) model
  - … including how it's attacked & some countermeasures
- What's coming in the future? (SBOMs, etc.)

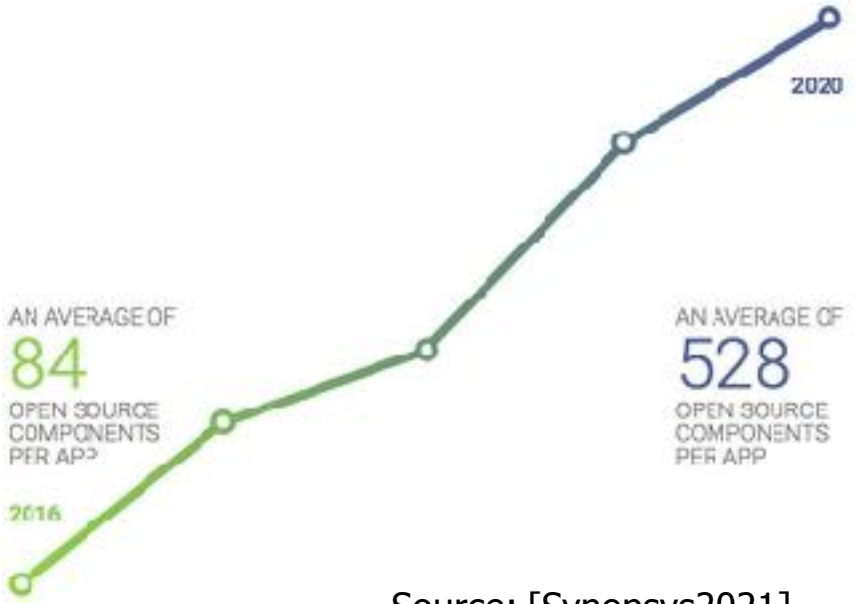
Open Source Source is critically important today

98%

Percent of general codebases and Android apps that contained OSS [Synopsys2021]

70%

Percent of codebase that was OSS on average [Synopsys2020]



Source: [Synopsys2021]

Source:
[Synopsys2021] "2021 Open Source Security and Risk Analysis Report" by Synopsys https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html
[Synopsys2020] "2020 Open Source Security and Risk Analysis Report" by Synopsys https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf

All software under attack, via vulnerabilities & supply chain

GitHub hacked, millions of projects at risk of being modified or deleted

POLICY

Cleaning up SolarWinds hack may cost as much as $100 billion

Government agencies, private corporations will spend months and billions of dollars to root out the Russian malicious code

Source: https://www.rollcall.com/2021/01/11/cleaning-up-solarwinds-hack-may-cost-as-much-as-100-billion/

SolarWinds' Orion attacked via a *subverted build environment*

Computer Science > Cryptography and Security

[submitted on 29 May 2020]

Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks

Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier

A software supply chain attack is characterized by the injection of malicious code into a software package in order to compromise dependent systems further down the chain. Recent years saw a number of supply chain attacks that leverage the increasing use of open source during software development, which is facilitated by dependency managers that automatically resolve, download and install hundreds of open source packages throughout the software life cycle. This paper presents a dataset of 174 malicious software packages that were used in real-world attacks on open source software supply chains, and which were distributed via the popular package repositories npm, PyPI, and RubyGems. These packages, dating from November 2015 to November 2019, were manually collected and analyzed. The paper also presents two general attack trees to provide a structured overview about techniques to inject malicious code into the dependency tree of downstream users, and to execute such code at different times and under different conditions. This work is meant to facilitate the future development of preventive and detective safeguards by open source and research communities.

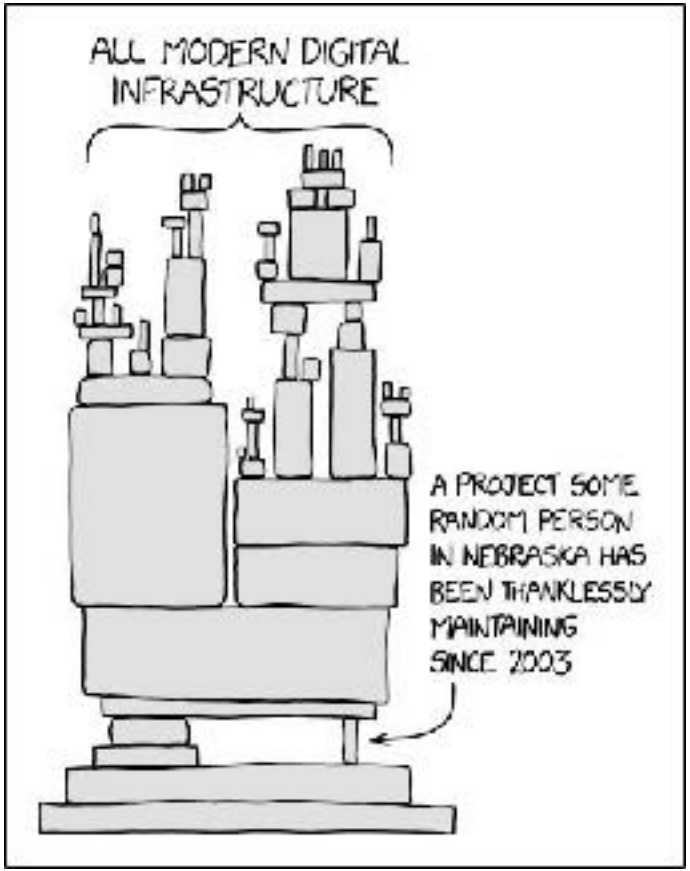
Source: https://arxiv.org/abs/2005.09535

Reviewed 174 OSS SC attacks 2015-2019,
61% malicious packages use typosquatting

Is OSS or proprietary software always more secure?

- Neither. The reality is that neither OSS nor proprietary always more secure
  - If you care, evaluate
- A design principle gives OSS a *potential* security advantage
  - Saltzer & Schroeder [1974/1975] defined secure design principles still valid today
  - Open design principle: “the protection mechanism must not depend on attacker ignorance”
  - OSS better fulfills this principle
  - “Many eyes” theory does work
    - Academics, science & engineering already based on peer review
    - Security experts widely perceive OSS advantage
- No software is perfect, so vulnerabilities may be found in well-run projects
  - Continuous careful review is *more* likely to detect vulnerabilities over time

Most projects & organizations are **not** able to **accurately summarize** the software that is running on their **systems**.



Common problem: Known-vulnerable reused software

- Problem: Failing to update reused known-vulnerable software (open & closed)
  - There's normally more OSS invisibly in use, so failure to update OSS is especially bad
- 84% of codebases had at least 1 OSS component with a known vulnerability, with an average of 158 vulnerabilities/codebase [Synopsys2021]
- Average known vulnerability was 2.2 years old [Synopsys2021]
- Android apps: 63% contain old OSS versions with known vulnerabilities; 44% of apps have high-risk vulnerabilities [Peril]
- 29% of codebases in 2019 AND 2020 included a vulnerable version of lodash fixed in July 2019 [Synopsys2021]
- 85% of codebases had OSS dependencies >4 years out of date [Synopsys2021]

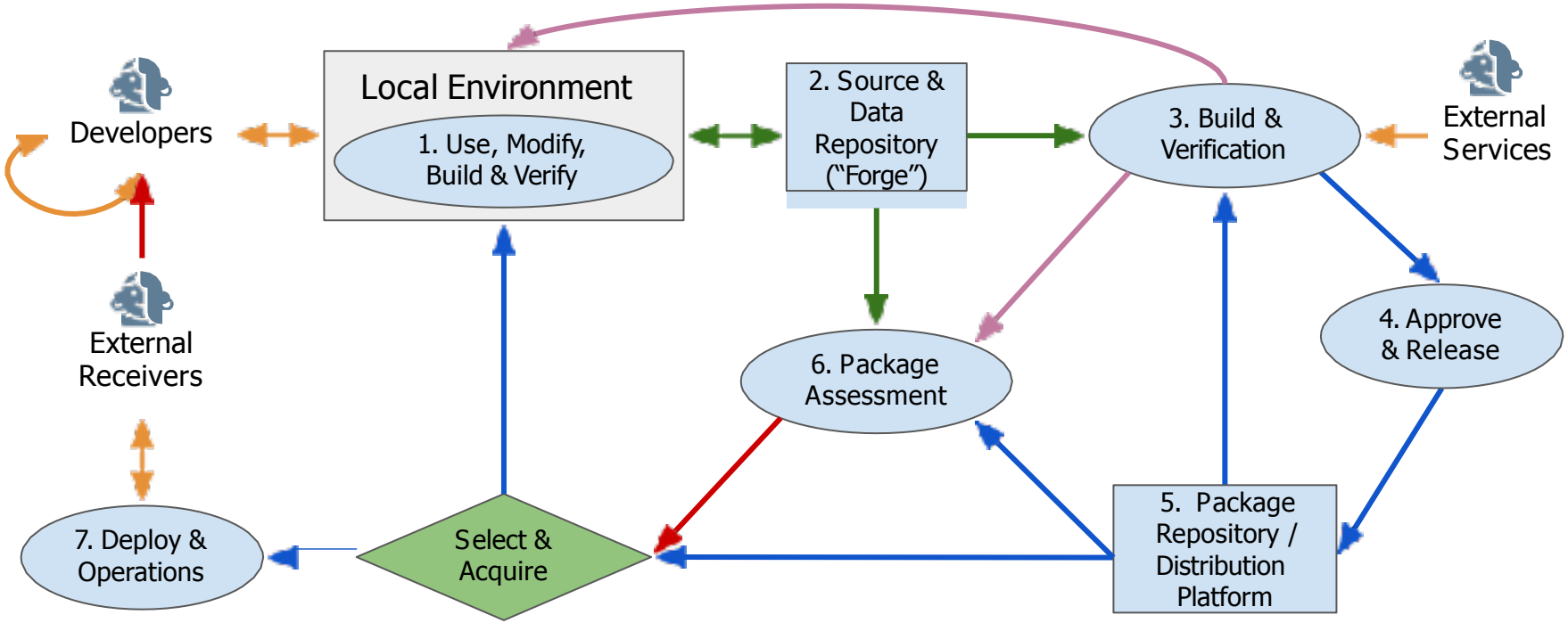
Many fail to update their reused software (~OSS) when their vulnerabilities are fixed

Source: [Synopsys2021] "2021 OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT" by Synopsys https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html ; [Peril] "Peril in a Pandemic: State of Mobile Application Security" by Synopsys;

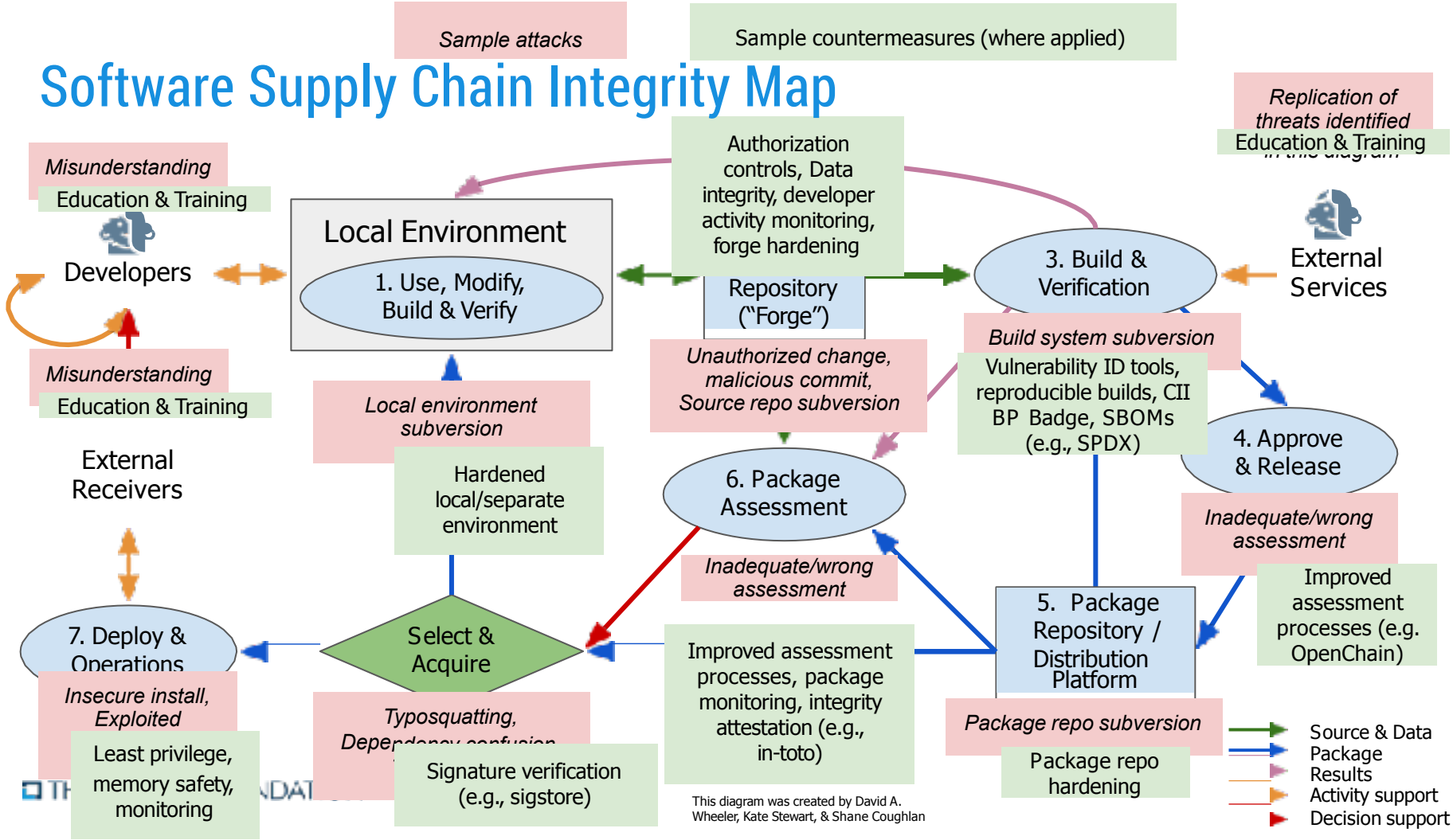
The OSS supply chain

abilian

Software Supply Chain Integrity Map



This diagram was created by David A. Wheeler, Kate Stewart, & Shane Coughlan


Sample attacks
Sample countermeasures (where applied)
Software Supply Chain Integrity Map
Replication of threats identified in this diagram
Education & Training
Misunderstanding
Education & Training
Developers
Local Environment
1. Use, Modify, Build & Verify
Authorization controls, Data integrity, developer activity monitoring, forge hardening
Repository ("Forge")
3. Build & Verification
External Services
Misunderstanding
Education & Training
Local environment subversion
Unauthorized change, malicious commit, Source repo subversion
Build system subversion
Vulnerability ID tools, reproducible builds, CII BP Badge, SBOMs (e.g., SPDX)
4. Approve & Release
External Receivers
Hardened local/separate environment
6. Package Assessment
Inadequate/wrong assessment
Inadequate/wrong assessment
Improved assessment processes (e.g. OpenChain)
7. Deploy & Operations
Select & Acquire
5. Package Repository / Distribution Platform
Insecure install, Exploited
Typosquatting, Dependency confusion
Improved assessment processes, package monitoring, integrity attestation (e.g., in-toto)
Package repo subversion
Least privilege, memory safety, monitoring
Signature verification (e.g., sigstore)
Package repo hardening
Source & Data
Package
Results
Activity support
Decision support
This diagram was created by David A. Wheeler, Kate Stewart, & Shane Coughlan

The cybersecurity executive order 14028

MAY 12, 2021

Executive Order on Improving the Nation's Cybersecurity

BRIEFING ROOM › PRESIDENTIAL ACTIONS

The cybersecurity executive order 14028

abilian

The NIST will be issuing guidance within the next year regarding the practices organizations will need to follow to ensure the security of the software supply chain.

Of particular importance to those using open source to develop applications is this part (emphasis ours):

> *(vi)* ***maintaining accurate and up-to-date data, provenance (i.e., origin) of software code or components****, and controls on internal and third-party software components, tools, and services present in software development processes, and performing audits and enforcement of these controls on a recurring basis;*
>
> *(vii) providing* ***a purchaser a Software Bill of Materials (SBOM)*** *for each product directly or by publishing it on a public website*

and...

> *(x) ensuring and* ***attesting, to the extent practicable, to the integrity and provenance of open source software used*** *within any portion of a product.*

Interlude: package managers

Package manager

From Wikipedia, the free encyclopedia

A **package manager** or **package-management system** is a collection of software tools that automates the process of installing, upgrading, configuring, and removing computer programs for a computer in a consistent manner.[1]

Synaptic, example of a full-featured package manager

A package manager deals with *packages*, distributions of software and data in archive files. Packages contain metadata, such as the software's name, description of its purpose, version number, vendor, checksum (preferably a cryptographic hash function), and a list of dependencies necessary for the software to run properly. Upon installation, metadata is stored in a local package database. Package managers typically maintain a database of software dependencies and version information to prevent software mismatches and missing prerequisites. They work closely with software repositories, binary repository managers, and app stores.

Package managers are designed to eliminate the need for manual installs and updates. This can be particularly useful for large enterprises whose operating systems typically consist of hundreds or even tens of thousands of distinct software packages.[2]

Comparison with installers [edit]

A package manager is often called an "install manager", which can lead to a confusion between package managers and installers. The differences include:

This box: view · talk · edit

Criterion	Package manager	Installer
Shipped with	Usually, the operating system	Each computer program
Location of installation information	One central installation database	It is entirely at the discretion of the installer. It could be a file within the app's folder, or among the operating system's files and folders. At best, they may register themselves with an uninstallers list without exposing installation information.
Scope of maintenance	Potentially all packages on the system	Only the product with which it was bundled
Developed by	One package manager vendor	Multiple installer vendors
Package format	A handful of well-known formats	There could be as many formats as the number of apps
Package format compatibility	Can be consumed as long as the package manager supports it. Either newer versions of the package manager keep supporting it or the user does not upgrade the package manager.	The installer is always compatible with its archive format, if it uses any. However, installers, like all computer programs, may be affected by software rot.

History [edit]

Early package managers, from around 1994, had no automatic dependency resolution[3] but could already drastically simplify the process of adding and removing software from a running system.[4]

By around 1995, beginning with CPAN, package managers began doing the work of downloading packages from a repository, automatically resolving its dependencies and installing them as needed, making it much easier to install, uninstall and update software from a system.[5]

Impact [edit]

Ian Murdock had commented that package management is "the single biggest advancement Linux has brought to the industry", that it blurs the boundaries between operating system and applications, and that it makes it "easier to push new innovations [...] into the marketplace and [...] evolve the OS".[21]

There is also a conference for package manager developers known as PackagingCon. It was established in 2021 with the aim to understand different approaches to package management.[22]

App store

From Wikipedia, the free encyclopedia

> *This article is about the concept. For the store for Apple mobile apps, see App Store (iOS/iPadOS). For the store for Apple Mac apps, see Mac App Store.*

An **app store** (or **app marketplace**) is a type of digital distribution platform for computer software called applications, often in a mobile context. Apps provide a specific set of functions which, by definition, do not include the running of the computer itself. Complex software designed for use on a personal computer, for example, may have a related app designed for use on a mobile device. Today apps are normally designed to run on a specific operating system—such as the contemporary iOS, macOS, Windows or Android—but in the past mobile carriers had their own portals for apps and related media content.[1]

Application-level package managers [edit]

- Bitnami: a library of installers for software packages for web applications;
- CocoaPods: a dependency manager for Swift and Objective-C Cocoa projects;
- Composer: a dependency Manager for PHP;
- Conda: a package manager for open data science platform of the Python and R;
- CPAN: a programming library and package manager for Perl;
- CRAN: a programming library and package manager for R;
- CTAN: a package manager for TeX;
- Docker: Docker, a system for managing containers, serves as a package manager for deploying containerized applications;
- Enthought Canopy: a package manager for Python scientific and analytic computing distribution and analysis environment;
- Gradle: a build system and package manager for Groovy and other JVM languages, and also C++;
- Ivy: a package manager for Java, integrated into the Ant build tool, also used by sbt;
- Leiningen: a project automation tool for Clojure;
- LuaRocks: a programming library and package manager for Lua;
- Maven: a package manager and build tool for Java;
- npm: a programming library and package manager for Node.js and JavaScript;
- NuGet: the package manager for the Microsoft development platform including .NET Framework and Xamarin;
- PAR::Repository and Perl package manager: binary package managers for Perl;
- PEAR: a programming library for PHP;
- pip: a package manager for Python and the PyPI programming library;
- RubyGems: a package manager and repository for Ruby;
- sbt: a build tool for Scala, uses Ivy for dependency management;

Comparison with build automation utility [edit]

Most software configuration management systems treat building software and deploying software as separate, independent steps. A build automation utility typically takes human-readable source code files already on a computer, and automates the process of converting them into a binary executable package on the same or remote computer. Later a package manager typically running on some other computer downloads those pre-built binary executable packages over the internet and installs them.

However, both kinds of tools have many commonalities:

- For example, the dependency graph topological sorting used in a package manager to handle dependencies between binary components is also used in a build manager to handle the dependency between source components.
- For example, many makefiles support not only building executables, but also installing them with `make install`.
- For example, every package manager for a source-based distribution – Portage, Sorcery, Homebrew, etc. – supports converting human-readable source code to binary executables and installing it.

A few tools, such as Maak and A-A-P, are designed to handle both building and deployment, and can be used as either a build automation utility or as a package manager or both.[17]

Le cas de Python

abilian

All solutions compared

	Installing python packages	Installing non-python packages	Managing python versions	Managing virtual environments	Environment reproducibility
pip	✅	✖*			
venv				✅	
pip-tools					✅
pyenv			✅		
conda	✅	✅	✅*	✅	
pipenv (+pyenv)	✅	✅	✅	✅	✅
poetry (+pyenv)	✅	✅	✅	✅	✅
Docker	*	*	*	*	✅

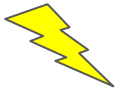
A thoroughly biased feature table

These tools have different scopes and purposes:

Name	Pip + venv	Pipenv	Poetry	pyenv	pythonfinder	Conda	this
Manages dependencies	✓	✓	✓			✓	✓
Resolves/locks deps		✓	✓			✓	✓
Manages Python installations				✓		✓	✓
Py-environment-agnostic				✓		✓	✓
Included with Python	✓						
Stores deps with project			✓		✓		✓
Requires changing session state	✓			✓			
Clean build/publish flow			✓				✓
Supports old Python versions	with `virtualenv`	✓	✓	✓	✓	✓	
Isolated envs for scripts							✓
Runs project fns from CLI		✓	✓				✓

Recommendations for developers

abilian



OSS Developer? Do this (now)

*If you're **evaluating** OSS, you should be looking for these!*

1. Learn how to develop & acquire secure software
   - *Secure SW Development Fundamentals* free course*, https://openssf.org/edx-courses/
   - Make software "secure by default" (easy to use securely, hardening)
2. Help projects earn CII Best Practices badge* https://bestpractices.coreinfrastructure.org/
3. Use many tools to find vulnerabilities via CI pipeline (build & verification environment)
   - Quality scanners (linters), security code scanners (static analysis), secret scanning, software composition analysis (SCA), web application scanners, fuzzers, …
   - One guide: https://github.com/ossf/wg-security-tooling/blob/main/guide.md
4. Monitor for known vulnerabilities in what you depend on
5. Enable rapid dependency update via using package managers & automated tests
   - Package managers may be system, language-level, and/or containers
   - Tests should include negative tests, be thorough enough to "ship if it passes"
6. Evaluate before selecting dependencies to use (typosquatting? malicious? secure?)
7. Make it *easy* for your users to update (e.g., stable APIs)
8. *Continuously improve* - attacks get better, so defenders also need to

Vulnerabilities are risks; manage the risks

*More information in next slides

Apply security design principles

- (Architectural) Design = how problem divided into components & interactions
- Design principles = rules-of-thumb to help you avoid serious design flaws
- There are many time-tested security design principles, e.g.:
  - Least privilege
  - Complete mediation (aka non-bypassability) - client-side JavaScript & mobile apps failures
  - Simplicity
  - Open design
  - Fail-safe defaults
  - Two-factor authentication
  - Minimize sharing
  - Easy to use
  - Apply acceptlists on untrusted inputs (define what's acceptable, reject the rest)

Know most common kinds of vulnerabilities & how to avoid

- Over 90% of vulnerabilities fit into a small set of categories
  - Knowing & preventing them reduces vulnerabilities by at least 1 order of magnitude
- Some widely-used carefully-crafted lists - know & use them
  - OWASP top 10 web application security vulnerabilities (for web apps)
  - CWE top 25 list (for anyone) + others "on the cusp"
- Examples of vulnerability categories
  - Injection vulnerabilities (e.g., SQL injection - counter with prepared statements)
  - Cross-site scripting (XSS)
  - Buffer overflows (and memory safety failures more generally. 70% Chrome & MS vulns)

Add vulnerability detection tools to CI pipeline

- Key: detect problems early
- Many different kinds of tools - include multiple kinds
- Practically all tools have false positives & false negatives
  - You still need to *think*
  - Try to have many tools (eventually)
- Greenfield (new start) vs. Brownfield (existing)
  - Greenfield: Add tools right now & make them sensitive (learn of & avoid problems)
  - Brownfield: Add tools slowly & greatly limit what they report, otherwise you'll be overwhelmed. Then increase sensitivity.

CII Best Practices Badge

- Identifies best practices for OSS projects
  - Goal: Increase likelihood of better quality & security. E.g.:
    - "The project sites… MUST support HTTPS using TLS."
    - "The project MUST use at least one automated test suite..."
    - "At least one static code analysis tool MUST be applied…"
    - "The project MUST publish the process for reporting vulnerabilities on the project site."
  - Based on practices of well-run OSS projects
- If OSS project meets best practice criteria, it earns a badge
  - Enables projects & potential users know current status & where it can improve
  - Combination of self-certification, automated checks, spot checks, public accountability
  - Three badge levels: passing, silver, gold
- Participation widespread & continuing to grow
  - >3,700 participating projects, > 500 passing+ projects in 2021-04
  - Current statistics: https://bestpractices.coreinfrastructure.org/en/project_stats
- A project within the OpenSSF Best Practices Working Group (WG)
- For more, see: https://bestpractices.coreinfrastructure.org

Recommendations for users of OSS

abilian

OSS User? Things to consider for evaluation (now)

1. Is there evidence that its developers *work to make it secure*?
   - See the previous "for developers" list!
   - E.g., CII Best Practices badge, security tools to detect vulnerabilities, documentation on why it's secure, evidence of security audits, SAFECode *Principles for Software Assurance Assessment*
2. Is it *easy to use securely*? (e.g., defaults)
3. Is it *maintained*?
   - Recent commits, issues handled, releases, multiple developers (ideally from >1 organization)
   - Multi-organization governance model / governing board
4. Does it have *significant use*? (beware of fads, but no users==no reviewers)
5. What is the software's *license*? (beware of no-license==not OSS)
   - Software Composition Analysis (SCA) tools, OpenChain*
6. If it is important, what is *your own evaluation* of the software? (OSS makes this possible!)*
   - Citizenship is not trustworthiness. Review the code/project you're using instead*
7. Did you acquire (download) it securely?*

Source: "Secure Software Development Fundamentals" course, https://openssf.org/edx-courses/ & https://github.com/ossf/secure-sw-dev-fundamentals

What is your own evaluation of the software?

- If the software is important to you, not examining it (OSS or not) is a risk
- Don't be afraid; even a brief review of its code can give insight
  - Again, is there evidence that the developers were trying to develop secure software?
    - Rigorous validation of untrusted input, use of prepared statements, etc.
  - Evidence of insecure or woefully incomplete software (e.g., forest of TODO statements)?
    - Nothing made by humans is perfect, but is it adequate for purpose?
  - What are the "top" problems reported by tools that look for vulnerabilities?
    - All such tools have false positives, so you must actually look
    - Consider working with project to fix actual problems
  - Is there evidence it's malicious?
    - Most malicious packages perform malicious actions during install, check those
    - Most aim at data exfiltration (check for extraction)
    - About half use obfuscation (look for obfuscation & encoded values that get executed)*
- What's the likelihood that packages were generated from putative source code?
  - E.g., is the package (including container images) from originating project and/or trusted org?
- Many organizations can do an in-depth analysis for a fee

*See "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks"

Malicious code & OSS

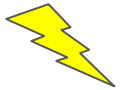
- OSS repositories demo great resilience vs. attacks
  - Linux kernel (2003); hid via "=" instead of "=="
    - if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
    - retval = -EINVAL;
    - Attack failed (CM, developer review, conventions)
  - SourceForge/Apache (2001), Debian (2003); Haskell (2015)
  - Countered & restored via external copy comparisons
- Linux kernel devs rejected all intentionally-vulnerable code from U of MN (2021)
- Malicious code can be made to look unintentional
  - Techniques to counter unintentional still vulnerabilities apply
- Attacker could try to bypass tools... but for OSS won't know what tools will be used!
- Borland InterBase/Firebird Back Door
  - user: politically, password: correct
  - Hidden for 7 years in proprietary product
  - Found after release as OSS in 5 months
  - Unclear if malicious, but has its form, & shows *improved* detection when switched to OSS

Can people insert malicious code in widely-used OSS?

- *Anyone* can insert malicious code into *any* software, proprietary or OSS
  - Just use a hex editor. Legal niceties are not protection
- Trick is to get result into user supply chain
- In OSS, requires:
  - subverting/misleading the trusted developers or trusted repository/distribution
  - *and* no one noticing the public malsource later
- Distributed source aids detection
- Large community-based OSS projects tend to have many reviewers from many countries
  - Makes undetected subversion more difficult

Things to consider when downloading (now)

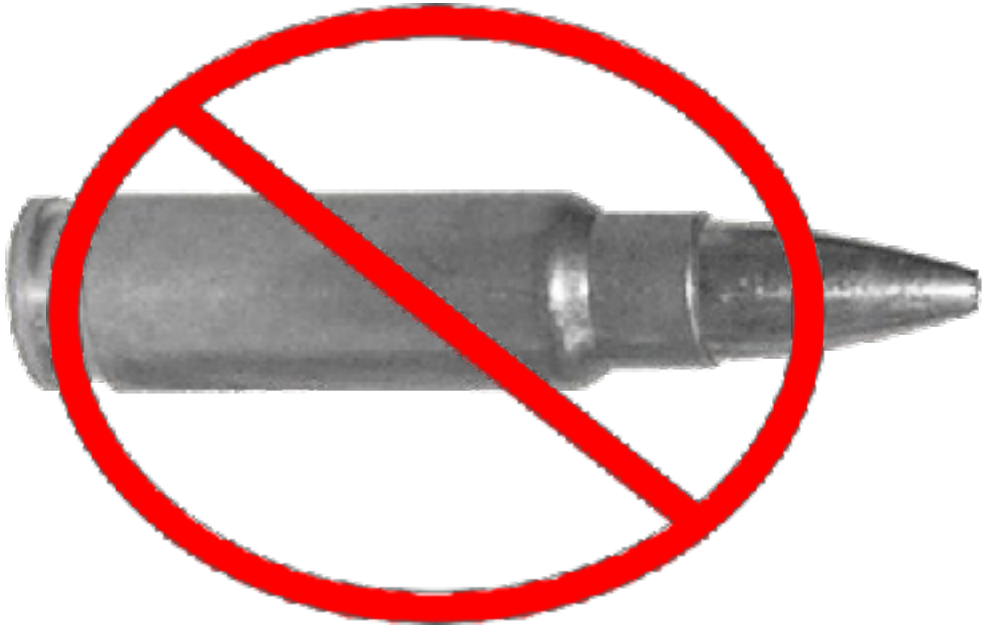
1. Make sure you have exactly the correct name before adding (counter typosquatting, most common OSS SC attack)
   a. Check for - vs _, 1 vs l, 0 vs O, Unicode
   b. Check popularity (download counts, followers, search engine) & date created
2. Download/install trustworthy way
   a. Use main site or "normal" redistribution site (e.g. package manager repository)
   b. Use https not http
   c. Download & delay (in case attack is revealed soon) - try to avoid running immediately
   d. Try to avoid using pipe-to-shell (e.g., curl … | sh) - cannot review, detectable by attackers
   e. If important & practical, try to verify that package is digitally signed by its expected creators



Things to consider when operating any software (now)

1. Protect, detect, respond
    - Protection is good, but you must *also* detect & respond to attacks in operations
    - Here "protect" includes "identify", and "respond" includes "recover"
2. Constantly monitor for vulnerabilities in all dependencies (OSS or not)
    - SCA tools enable this
    - GitHub, GitLab, Linux Foundation's LFX Tools, many others
3. If vulnerability found in dependency, examine quickly
    - If you *know* it can't be exploited in your environment, fine!
    - Otherwise, rapidly update, test (automated tests), ship to production
    - You must be *faster* than the attackers

There is no silver bullet!



Source: A silver bullet, Sir Magnus Fluffbrains, 2019, https://commons.wikimedia.org/wiki/File:Silver_bullet.png

Sample risks & countermeasures

- The following slides walk through the model
  - Show SOME risks & countermeasures
  - In some cases, developers would need to DO something, but evaluators can see what's done
  - For now, don't need to focus on the specific details

Key point: Attacks *can* be countered!

Developers & External Receivers

- Risk: Misunderstandings
  - Countermeasure: Education & training
  - Countermeasure: Processes to ensure best practices are consistently followed
  - Countermeasure: Tools, multi-party review to detect problems that slip in

1. Use, Modify, Build & Verify (in Local Environment)

- Risk: Local environment subversion
  - Countermeasure: Hardened local/separate environment
  - Countermeasure: Regular audits of the environment integrity

2. Source & Data Repository (“Forge”)

- Risk: Unauthorized change, malicious commit
  - Countermeasure: Authorization controls (e.g., 2FA)
  - Countermeasure: Data integrity analysis
  - Countermeasure: Developer activity monitoring
- Risk: Source repo subversion
  - Countermeasure: Forge hardening
  - Countermeasure: Forge audits

3. Build & Verification

- Risk: Build system subversion
  - Countermeasures: Build system hardening
  - Countermeasures: Software Bill of Materials (e..g., SPDX)
  - Countermeasures: Best practices (CII BP)
  - Countermeasures: Verified reproducible builds
  - Countermeasures: Vulnerability identification automation

4. Approve & Release

- Risk: Inadequate/wrong assessment
  - Countermeasure: Improved assessment processes (OpenChain ISO 5230)
  - Countermeasure: Automatically generation analysis SBOMs to automate policy checklists

5. Package Repository / Distribution Platform

- Risk: Package repo subversion
  - Countermeasure: Package repo / distribution platform hardening
  - Countermeasure: Code signing
  - Countermeasure: Verified reproducible builds

6. Package Assessment

- Risk: Inadequate/wrong assessment
  - Countermeasures: Software Bill of Materials (e.g., SPDX)
  - Countermeasures: Best practices (CII BP)
  - Countermeasures: Package monitoring
  - Countermeasures: Verified reproducible builds
  - Countermeasures: Integrity attestation (in-toto)

Select & Acquire

- Risk: Typosquatting (wrong package name)
  - Countermeasures: Software Bill of Materials
  - Countermeasures: Vulnerability identification automation
- Risk: Dependency confusion (wrong repository)
  - Countermeasures: Software Bill of Materials
- Risk: Selecting malicious packages
  - Countermeasures: Package monitoring
  - Countermeasures: Signature verification (sigstore)
- Risk: Selecting highly vulnerable packages
  - Countermeasures: Vulnerability identification automation

7. Deploy & Operations

- Risk: Insecure install
- Risk: Exploited vulnerabilities

Countermeasures:

- Least privilege, memory safety, monitoring & response

Additional topics

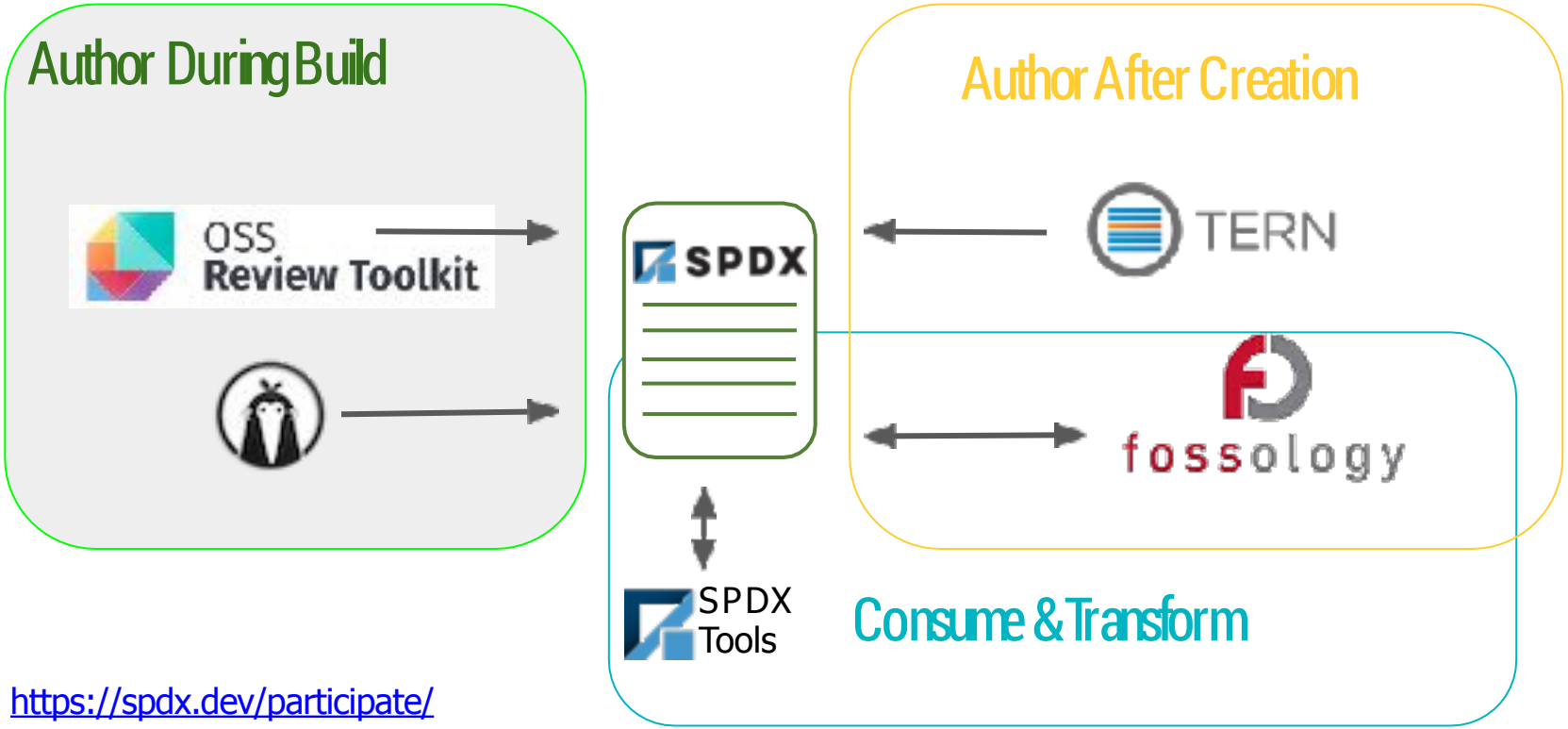
abilian

Verified reproducible builds

- SolarWinds' Orion (proprietary software) suffered a *build system* attack
  - The signed package received by customers was *not* generated from their source code
  - Most countermeasures fail against build system attacks, reproducible builds counter
- "Reproducible builds are a set of software development practices that create an independently-verifiable path from source to binary code."
  - "By promising identical results are always generated from a given source, this allows multiple third parties to come to a consensus on a "correct" result, highlighting any deviations as suspect and worthy of scrutiny."
  - Goal: have independently-verified reproducible builds; attackers must break them all
- Significant progress (e.g., Debian bullseye/i386 94.0% reproducible 2021-04)
- Typically requires build changes (forced dates, forced sorts, etc.)
  - Possible for closed & open source software
  - Closed source has added doable challenges (keeping source secret, independently buildable)

Sources: https://reproducible-builds.org/, https://www.linuxfoundation.org/en/blog/preventing-supply-chain-attacks-like-solarwinds/

SPDX is being used as a common exchange standard for SBOM information



https://spdx.dev/participate/

Increased use of memory-safe / safe languages

- Memory-safe/safe languages eliminate many security issues
  - Prevent many unintentional & mock-unintentional vulnerabilities
  - Most programming languages are memory-safe; use them when you reasonably can
  - C/C++/assembly are performant but not memory-safe/safe
  - Problem: Most languages cannot provide performance sometimes needed, C/C++ can
- Ongoing efforts to enable memory-safe languages in more situations
  - Esp. Rust; "safe" portion provides memory-safety & safe concurrency, yet performant
- Rustification: Transitioning (parts of) software to Rust, e.g.:
  - curl: optional http backend in Rust
  - Mozilla Firefox: increasing proportion in Rust
  - Linux kernel: ongoing effort to add support for drivers in Rust
- Many challenges (maturity issues), but promising
  - Rust currently has only 1 implementation, lacks support for some architectures (gcc ongoing)
  - Some capabilities need "Rust nightly builds" (but those extensions are becoming stable)

Developer Citizenship: The wrong worry

- Commercial software (open & closed) is developed in a *global* economy
- It is *possible* to estimate probable OSS developer location & citizenship
  - Timezones & typical commit times. Also, presented name & email address are often available
  - Generally can't know these for closed source; sometimes only company location "known"
- But focusing on OSS location/citizenship is generally a *waste of time*
  - "Legal location of company" is *irrelevant*, that's just a flag of convenience ($90 for Delaware LLC)
  - "Location of developer" is NOT the same as citizenship
  - "Citizen of [certain countries]" is NOT trustworthiness (Aldrich Ames, Robert Hanssen)
  - Powerful nation-states can bribe citizen or forge citizenship credentials if they care
  - Someone clearly developing code in an adversary state is probably not attacking
- Malicious state actor could attempt insertion, but OSS is less risky
  - With OSS you can review the code you'll use; closed source often requires blind faith
- With OSS, focus on evaluating *code*, not citizenship
  - This is how larger OSS projects counter potentially malicious developers
  - Esp. since top risks are unintentional vulnerabilities, old known-vulnerable code, typosquatting

(Largely based on a presentation by David Wheeler)