# Journey to the Center of Software Supply Chain Attacks

Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, Olivier Barais

# Journey to the Center of Software Supply Chain Attacks

Piergiorgio Ladisa, *SAP Security Research, Université de Rennes 1*
Serena Elisa Ponta, *SAP Security Research*
Antonino Sabetta, *SAP Security Research*
Matias Martinez, *Universitat Politècnica de Catalunya-BarcelonaTech*
Olivier Barais, *Université de Rennes 1, Inria, IRISA*



*Abstract—This work discusses open-source software supply chain attacks and proposes a general taxonomy describing how attackers conduct them. We then provide a list of safeguards to mitigate such attacks. We present our tool "Risk Explorer for Software Supply Chains" to explore such information and we discuss its industrial use-cases.*

Open-Source Software (OSS) is ubiquitous in modern applications. It may constitute more than 90% of the code of a commercial application and it is widely used across the technology stack and the development and operation lifecycle. Due to the complexity of the modern software supply chain, attackers have multiple opportunities to inject malicious code into open-source components and infect downstream users.

In recent years, we have observed an exceptional increase in the number and type of attacks on OSS [12]. For example, a recent case affected a nightly build of PyTorch, a popular framework to build machine-learning models: the attackers sneaked malicious code through a dependency by abusing the dependency resolution mechanism of `pip` [13]. Other software supply chain attacks (e.g., infection of SolarWind's Orion platform [10]) impacted suppliers of government agencies and critical infrastructures. For this reason, several national security agencies reported software supply chain attacks as one of the primary threats [2], [3] and different efforts (both public and private) arose to increase the security of software supply chains (e.g., SLSA[1]).

There is no doubt about the fact that the software supply chain is more and more often under attack, and that this problem is receiving considerable attention both by the industry and the academic community. However, we have also observed that the existing literature is somewhat fragmented, also due to the lack of a general, technology-independent description of how attackers inject malicious code into OSS projects.

In [5] we presented a taxonomy of attack vectors OSS supply chains, which are independent of specific programming languages and technologies. Moreover, we introduced a set of general safeguards addressing the identified attack vectors. To ease the visualization and exploration of the taxonomy, as well as to enable its extension by the open-source and security communities, we have developed a visualisation tool named *Risk Explorer for Software Supply Chains* [6], which is accessible online and is released as open source.

In this work we will guide you through a journey to the center of OSS supply chain attacks.

We prepare the descent by describing the attack surface of OSS development model, associated risks, and the attacker model. Then, we present the taxonomy covering **117 unique attack vectors** related to OSS supply chains and **33 safeguards** geared towards the proposed taxonomy. This taxonomy, which is an updated version of the one presented in [5], was built by examining **370 resources** encompassing real-world attacks and scientific and grey literature. We dig deeper by analysing the vectors of real world attacks covered by those resources and we contrast the prevalence of each such vector with the attention that it received from the research community.

The journey continues with the review of the *Risk Explorer for Software Supply Chains* [6] and four industrial use cases.

We then position our taxonomy with respect to the existing frameworks for software supply chain security.

The journey's end provides insights on the content of malicious packages and on the current challenges that the software industry faces to secure the OSS supply chain.


Digital Object Identifier 10.1109/XXX.0000.0000000
[1] https://slsa.dev/

## Preparing the descent

To understand the existing attack vectors, first we describe the attack surface of the OSS supply chain, i.e., what are the *systems* and *stakeholders* involved in the creation of OSS artifacts. Then we provide a general overview of the risks of OSS supply chains. Finally, we present what are the characteristics of the attacker in our context.

### Attack Surface: OSS Development Model

The OSS supply chain denotes all the systems and stakeholders involved in the development, build, and distribution of OSS artifacts to downstream users. Figure 1 describes at high-level the common OSS development model.

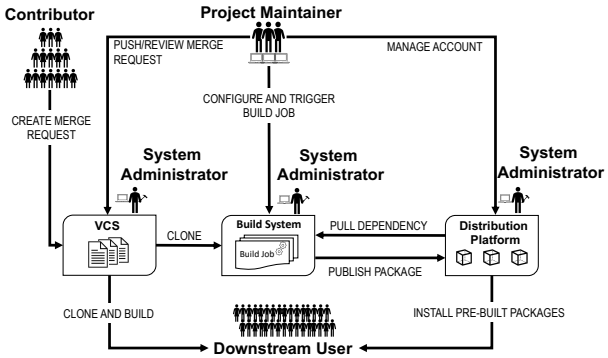



FIGURE 1: Stakeholders, systems and dataflows related to the development, build and distribution of OSS artifacts [5].

The systems considered comprise Version Control System (VCS), build systems, and distribution platforms (e.g., package repositories).

**Version Control Systems** host the source code of the OSS project, as well as metadata, configuration files, and other resources. They track and manage all the changes of the codebase throughout the development process. Plain VCSs like Git do not require user authentication, but complementary tools (e.g., GitHub) offer additional functionalities (e.g., issue trackers) or security controls (e.g., authentication, review workflows).

**Build Systems** consume the project's code to produce a binary artifact, e.g., an executable or compressed archive, which can be distributed to downstream users. The build commonly involves so-called dependency or package managers (e.g., pip for Python) which determine and download all dependencies necessary for the build to succeed. Continuous Integration (CI)/Continuous Delivery (CD) pipelines are often used to automate the build, test, and deployment of project artifacts.

**Distribution Platforms** distribute pre-built OSS artifacts to downstream users, e.g., upon the execution of package managers or through manual download. They include not only the well-known public package repositories like PyPI or Maven Central but also internal and external mirrors, Content Delivery Network (CDN) or proxies.

**Workstations of OSS Maintainers and Administrators.** OSS project maintainers and administrators of the abovementioned systems have privileged access to sensitive resources, e.g., the codebase, a build system's web interface, or a package repository's database. Therefore, their workstations are in the scope of the attack scenario.

Concerning the stakeholders involved in the OSS supply chain, they include contributors, project maintainers, system and service administrators, and downstream users.

**Contributors** contribute to an OSS project with limited access to project resources. Common way of contributions to the VCS involve the submission of merge requests.

**Project Maintainers** have privileged access to project resources. For example, they are in charge of reviewing and integrating contributors' merge requests, configure build systems and trigger build jobs, or deploy ready-made artifacts on package repositories.

**System and Service Administrators** have the responsibility to configure, maintain, and operate any of the above-mentioned systems. These stakeholders can include, e.g., employees of 3rd-party VCS hosting providers, members of OSS foundations that operate private build systems for their projects, or employees of companies running package repositories (like npm or private mirrors).

**Downstream Users** consume OSS project artifacts. They can opt to consume the source code from VCS (and then build it themselves) or, as is more common, download pre-built versions from distribution platforms. In the context of downstream development projects, the download is typically automated by package managers like pip or npm, which help in automatically identifying and obtaining the direct and transitive dependencies for a certain project.

Both systems and stakeholders have to be considered as roles, multiple of which can be exercised by a single host, individual, or third party service. For example, maintainers of an OSS project typically consume artifacts of other projects

## Risks of Open-Source Software Supply Chains

The above-described systems are inherently distributed, and the stakeholders are partly unknown or anonymous. The real identities of project collaborators, both contributors and maintainers, are not necessarily known. Accounts, including anonymous ones, gain trust through continued contributions of quality (meritocracy).

Each open-source component used comes with its own systems and stakeholders, thereby multiplying an attack surface having both technical and social facets. As in other adversarial contexts, attackers require finding single weaknesses, while defenders need to cover the whole attack surface, which in this case spans the whole supply chain.

Even heavily used open-source projects receive only little funding and contributions, making it difficult for maintainers to securely run projects or promptly react to security incidents. Moreover, the larger the user base (direct and indirect) the more attractive an open-source project becomes for attackers.

Downstream consumers have no control over and limited visibility into given projects' security practices. The sheer number of dependencies makes rigorous reviews impractical for a given consumer, forcing them to trust the community for a timely detection of vulnerabilities and attacks.

## Attacker Model

To identify attack vectors related to OSS supply chains, we make the following assumptions on the attacker.

Primary goal of the attacker is to **insert** malicious code in open-source artifacts such that it is executed in the context of downstream projects, e.g., during its development or runtime. Targeted assets can belong both to developers of downstream software projects, or their end-users, depending on the attacker's specific intention. In fact, the focus of the taxonomy is not on *what* malicious code does, but *how* attackers place it in upstream projects.

> *The focus of the taxonomy is not what malicious code does, but how attackers place it in upstream projects.*

Initially, attackers only have access to publicly available information and publicly accessible resources, which they can collect and analyze following the Open Source Intelligence (OSINT) approach. Of course, due to the nature of open-source projects, many project details are freely accessible, e.g., project dependencies, build information, or commit and merge request histories. Attackers can interact with any of the stakeholders and resources depicted in Figure 1, e.g., to communicate with maintainers using merge requests or issue trackers or to create fake accounts and projects.

Attackers can target any kind of project (e.g., libraries, word processors). Downstream consumers can be affected directly or indirectly and in addition the scope of the attacker can be either to reach a large pool of consumers or to target a specific group. This is possible by conditioning the execution of malicious code depending, e.g., on the lifecycle phase (install, test, etc.), application state, operating system, or properties of the downstream component it has been integrated into [9].

## Our start: Taxonomy of Attacks

This section present the taxonomy consisting of 117 unique attack vectors, collected through the review of scientific and grey literature (details in [5]). Such taxonomy takes the form of an *attack tree* to systematically represent the attacker goals and techniques. In an attack tree, the root node represents the attacker's top-level goal, which is iteratively refined by its children into subgoals.

To create the taxonomy depicted in Figure 2 we conducted a Systematic Literature Review (SLR) of both scientific and grey literature. We have done our best to collect as many resources as possible, although we refrain from claiming to have achieved completeness. By conducting a user survey, the taxonomy has been validated and assessed by 17 experts in software supply chain security and 134 software developers.

*Conducting an Open-Source Supply Chain Attack*

This is the attackers' top-level goal and happens by injecting malicious code into an OSS project such that it is downloaded by downstream consumers, and executed upon installation or at runtime.

The entire taxonomy unfolding below this high-level goal is depicted in Figure 2 and summarized in the following. The 1st-level child nodes of the tree reflect different degrees of interference with existing packages.

*Develop and Advertise Distinct Malicious Package from Scratch*

This node covers the creation of a brand new OSS project, with the intention to use it for spreading malicious code from the beginning or at a later point in time. Besides creating the project, the attacker is required

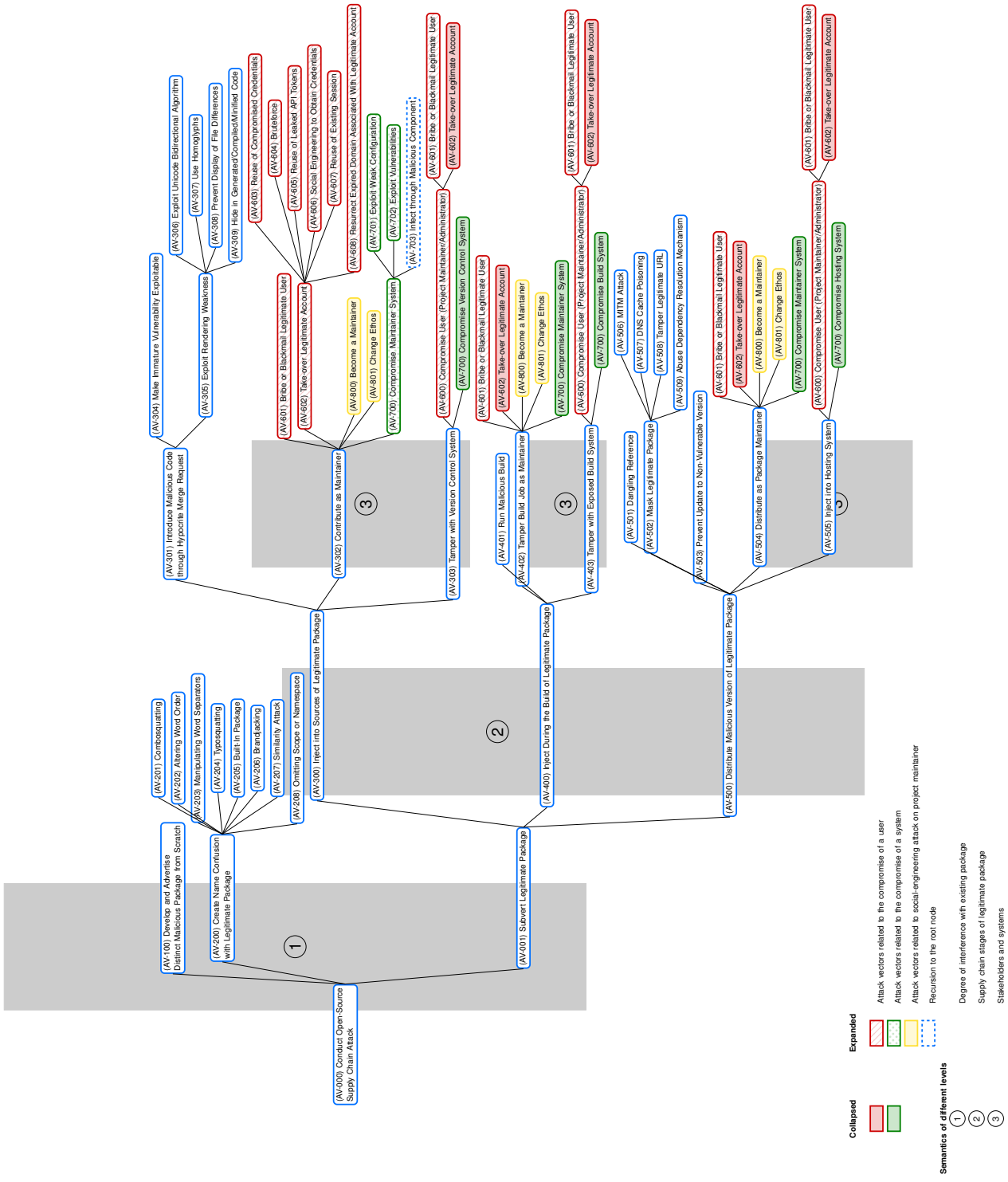



FIGURE 2: Refined version of the taxonomy for OSS supply chain attacks [5].

to advertise the project to attract victims. Real-world examples affect PyPI, npm, Docker Hub or NuGet.

*Create Name Confusion with Legitimate Package*
This node covers attacks that consist of creating project or artifact names that resemble legitimate ones, suggest trustworthy authors, or play with common naming patterns. Once a suitable name is found, the malicious artifact is deployed, e.g., in a source or package repository, in the hope of being consumed by downstream users. As the deployment does not interfere with the resources of the project that inspired the name (e.g., legitimate code repository, maintainer accounts) the attack is relatively cheap.

Child nodes of this attack vector relate to sub-techniques applying different modifications to the legitimate project name: *Combosquatting* adds pre or post-fixes, e.g., to indicate project maturity (`dev` or `rc`) or platform compatibility (`i386`). *Altering Word Order* re-arranges the word order (`test-vision-client` vs. `client-vision-test`). *Manipulating Word Separators* alters or adds word separators like hyphens (`setup-tools` vs. `setuptools`). *Typosquatting* exploits typographical errors (`dajngo` vs. `django`). *Built-In Package* replicates well-known names from other contexts, e.g., built-in packages or modules of a programming language (`subprocess` for Python). *Brandjacking* includes the name of popular brands/organizations (e.g., `twilio, aws`) to suggest that such package comes from a trustworthy author (`twilio-npm`). *Similarity Attack* creates a misleading name in a way different from the previous categories (`request` vs. `requests`).

*Subvert Legitimate Package*
This node covers all attacks aiming to corrupt an existing, legitimate project, which requires compromising one or more of its numerous resources depicted in Figure 1. As a result, this subtree is much larger compared to the previous ones, especially because subtrees related to user and system compromises occur multiple times in the different supply chain stages.

The remainder of this section is dedicated to sub-techniques of this first-level node.

*Inject into Sources of Legitimate Package* relates to the injection of malicious code into a project's codebase. From the attacker's point of view, this has the advantage to affect all downstream users, no matter whether they consume sources or pre-built binary artifacts (as part of the codebase, the malicious code will be included during project builds and binary artifact distribution). This vector has several sub-techniques. Taking the role of contributors, attackers can use *hypocrite merge requests* to turn immature vulnerabilities into exploitable ones, or exploit IDE rendering weaknesses to hide malicious code, e.g., through the use of Unicode homoglyphs and control characters , or the hiding and suppression of code differences.To *contribute as maintainer* requires to obtain the privileges necessary for altering the legitimate project's codebase, which can be achieved in different ways: using Social Engineering (SE) techniques on legitimate project maintainers, *changing the ethos* (e.g., as in the case of protestwares), *taking over legitimate accounts* (e.g., reusing compromised credentials, or *compromising the maintainer system* (e.g., exploiting vulnerabilities). The latter can also be achieved through a malicious (OSS) component, e.g., IDE plugin, which is reflected through a recursive reference to the root node. The legitimate project's codebase can also be altered by *tampering with its VCS*, thus, bypassing a project's established contribution workflows. For instance, by compromising system user accounts, or by exploiting configuration/software vulnerabilities, an attacker could access the codebase in insecure ways.

*Inject During the Build of Legitimate Package*
Greatly facilitated by language-specific package managers like Maven or Gradle for Java, it became common to download pre-built components from package repositories rather than OSS project's source code from its VCS. Therefore, the injection of malicious code can happen during the build of such components before their publication.Though the spread is limited compared to injecting into sources, the advantage for the attacker is that the detection of malicious code inside pre-built packages is typically more difficult, especially for compiled programming languages. One sub-technique is *running a malicious build job* to tamper with system resources shared between build jobs of multiple projects(e.g., the infection of Java archives in NetBeans projects). An attacker can also *tamper the build job as maintainer*, e.g., by taking over legitimate maintainer accounts, becoming a maintainer, or compromising their systems (e.g., XCodeGhost malware). Similarly, the attacker could compromise build systems, especially online accessible ones, e.g., by compromising administrator accounts or exploiting vulnerabilities.

*Distribute Malicious Version of Legitimate Package*
Pre-built components are often hosted on well-known package repositories like PyPI or npm, but also on less popular repositories with a narrower scope. In addition, the components can be mirrored remotely or locally, made available through CDNs (e.g., in the case of JavaScript libraries), or cached in proxies. This attack vector and its sub-techniques cover all cases where attackers tamper with mechanisms and systems involved in the hosting, distribution, and download of pre-

built packages. *Dangling references* (re)use resource identifiers of orphaned projects, e.g., names or URLs. *Mask legitimate package* targets package name or URL resolution mechanisms and download connections. Their goal is the download of malicious packages by compromising resources external to the legitimate project. This includes Man-In-The-Middle (MITM) attacks, DNS cache poisoning, or tampering with legitimate URLs directly at the client. Particularly, package managers follow a (configurable) resolution strategy to decide which package version to download, from where, and the order of precedence when contacting multiple repositories. Attackers can *abuse such resolution mechanisms* and their configurations.Attackers can also *prevent updates to non-vulnerable versions* by manipulating package metadata, e.g., by indicating an unsatisfiable dependency for newer versions of a legitimate package. Finally, the involvement of systems and users in package distribution results in attack vectors similar to previous ones. Attackers can take the role of legitimate maintainers, thus, *distribute as maintainer*, e.g., by taking over package maintainer accounts (e.g., `eslint`), thesecond most common attack vector after typosquatting. They can also compromise maintainer systems, or directly *inject into the hosting system*, e.g., by compromising administrator accounts or exploiting vulnerabilities.

### Remark

While the taxonomy presented is largely agnostic, some attack vectors are specific to certain ecosystems. *Abuse Dependency Resolution Mechanism* attacks depend on the approach and strategy used by the respective package manager to resolve and download declared dependencies from internal and external repositories. For instance, Maven, npm, pip, NuGet or Composer were affected by the dependency confusion attack, while Go and Cargo were not [4]. Several attacks below *Exploit Rendering Weakness* depend on the interpretation and visualization of (Unicode) characters by user interfaces and compiler/interpreters [1]. Also name confusion attacks need to consider ecosystem specificities, especially *Built-In Packages*.

**Pills of History of Supply Chain Attacks**

Since its first appearance in 1984 with the publication of Ken Thompson's *Reflections on Trusting Trusts*, the discussion around software supply chain security has kept growing, with a larger increase in recent years. In terms of attacks, to the best of our knowledge, the first known OSS supply chain attacks date back to 2010 (cf. Figure 3b). The first consists of the implantation of a backdoor in the source code of ProFTPD1.3.3c [8] while the second is the well-known case of Operation Aurora [7]. Until 2016, attacks mainly consist in the injection of malicious code into source code or during builds, or the deployment of malicious versions of legitimate softwares in hosting systems. It's in 2017 that we detect the first attack that exploited the attack vector *Create Name Confusion with Legitimate Package*, which probably is the most exploited attack vector today. In particular, this first attack involved uploading packages to PyPI with a name squatting that of popular projects such as urllib3 and acquisition (using different sub-techniques). Prior to this attack, there's just one non-peer-reviewed work discussing the attack vector *Create Name Confusion with Legitimate Package*, that is Nikolai Tschacher's Master thesis [14] published in 2016, one year before the first attack of this kind.

## A rapid recovery: Safeguards

This section presents an overview about safeguards against OSS supply chain attacks, which were identified through literature review and generalized to become agnostic of specific programming languages or ecosystems.

The complete list of the 33 safeguards can be found in Table 1, including a classification after control type and ordered according the Utility-to-Cost ratio as assessed by 17 domain experts [5]. All safeguards are mapped to the attack vector(s) (described above) they (partially or fully) mitigate, some to the top-level goal due to addressing all vectors (e.g., establishing a vetting process), others to more specific subgoals. Some safeguards can be implemented by one or more stakeholders, while others require the involvement of multiple ones to be effective (e.g., signature creation and verification).

Both implementation and use of those safeguards can incur non-negligible costs, also depending on the specific programming language and ecosystem. Thus, the selection, combination and implementation of safeguards require careful planning and design, to balance required security levels and costs.

In our work [5] we present the qualitative assessment of the utility and cost of each safeguard, conducted by surveying both domain experts and software developers.

| Safeguard | Utility-to-Cost | Control Type | | | | Stakeholders Involved | | | Attack-Vector Addressed |
|---|---|---|---|---|---|---|---|---|---|
| | | Directive | Preventive | Detective | Corrective | OSS Maintainer | 3P Service Prov. | OSS Consumer | |
| Protect production branch | 2.10 | | ✓ | ✓ | | • | | | AV-301, AV-302 |
| Remove un-used dependencies | 2.05 | | ✓ | | | | | • | AV-001 |
| Version pinning | 1.68 | | ✓ | | | | | • | AV-001 |
| Dependency resolution rules | 1.58 | | ✓ | | | | | • | AV-501, AV-508, AV-509 |
| User account management | 1.50 | | ✓ | | ✓ | • | • | | AV-302,AV-402,AV-504,AV-600 |
| Secure authentication (e.g., Multi-Factor Authentication (MFA), password recycle, session timeout, token protection) | 1.48 | | ✓ | | | • | • | | AV-*00 → AV-602 |
| Use of security, quality and health metrics | 1.35 | ✓ | ✓ | | | • | • | • | AV-000 |
| Typo guard/Typo detection | 1.34 | | ✓ | ✓ | | | • | • | AV-200 |
| Use minimal set of trusted build dependencies in the release job | 1.32 | | ✓ | | | • | | | AV-400 |
| Integrity check of dependencies through cryptographic hashes | 1.32 | | | ✓ | | | | • | AV-400, AV-500 |
| Maintain detailed Software Bill of Materials (SBOM) and perform Software Composition Analysis (SCA) | 1.24 | | ✓ | ✓ | | • | • | • | AV-000 |
| Ephemeral build environment | 1.24 | | ✓ | | | • | | | AV-400 |
| Prevent script execution | 1.23 | | ✓ | | | | | • | AV-000 |
| Pull/Merge request review | 1.21 | | ✓ | | | • | | | AV-301, AV-302 |
| Restrict access to system resources of code executed during each build steps | 1.21 | | ✓ | | | • | | | AV-400 |
| Code signing | 1.19 | | | ✓ | | • | • | • | AV-200, AV-500 |
| Integrate Open-Source vulnerability scanner into CI/CD pipeline | 1.15 | | | ✓ | | | | • | AV-000 |
| Use of dedicated build service | 1.09 | | ✓ | | | • | | | AV-400 → AV-700 |
| Preventive squatting the released packages | 1.07 | | ✓ | | | • | • | | AV-200 |
| Audit | 1.05 | ✓ | | ✓ | | • | • | | AV-000 |
| Security assessment | 1.05 | | | ✓ | | • | • | | AV-000 |
| Vulnerability assessment | 1.05 | | | ✓ | | • | • | | AV-000 |
| Penetration testing | 1.05 | | | ✓ | | • | • | | AV-000 |
| Reproducible builds | 1.02 | | | ✓ | | • | • | • | AV-400, AV-500 |
| Isolation of build steps | 1.00 | | ✓ | | | • | | | AV-400 |
| Scoped packages | 1.00 | | ✓ | | | • | • | | AV-509 |
| Establish internal repository mirrors and reference one private feed, not multiple | 0.97 | | ✓ | | | | | • | AV-501,AV-502, AV-504, AV-505 |
| Application Security Testing | 0.95 | | | ✓ | | | • | • | AV-000 |
| Establish vetting process for Open-Source components hosted in internal/public repositories | 0.95 | | ✓ | | | | • | • | AV-000 |
| Code isolation and sandboxing | 0.93 | | | | ✓ | | | • | AV-000 |
| Runtime Application Self-Protection (RASP) | 0.88 | | | ✓ | ✓ | | | • | AV-000 |
| Manual source code review | 0.85 | | | ✓ | | | • | • | AV-300 |
| Build dependencies from sources | 0.73 | | ✓ | | | | • | • | AV-400, AV-500 |

TABLE 1: Safeguards against OSS supply chain attacks shown in the order of the mean of their Utility-to-Cost ratio assessed by 17 experts [5]. Each safeguard is also characterised by control type, stakeholder(s) involved in their implementation, and a mapping to mitigated attack vectors (cf. Figure 2 to resolve their identifiers) [5].

*Common Safeguards* comprises 4 countermeasures that require all stakeholders to become active, i.e., project maintainers, open-source consumers, and administrators (service providers). For example, a detailed SBOM has to be produced and maintained by the project maintainer, ideally using automated SCA tools. Following, the SBOM must be securely hosted and distributed by package repositories, and carefully checked by downstream users in regards to their security, quality, and license requirements.

*Safeguards for Project Maintainers and Administrators* comprises 8 safeguards. *Secure authentication*, for instance, suggests service providers to offer MFA or enforce strong password policies, while project maintainers should follow authentication best-practices, e.g., use MFA where available, avoid password reuse, or protect sensitive tokens.

*Safeguards for Project Maintainers* includes 7 countermeasures. Generally, OSS projects use hosted, publicly accessible VCSs. Maintainers should then, e.g., conduct careful *merge request reviews* or enable *branch protection rules* for sensitive project branches to avoid malicious code contributions. As project builds may still happen on maintainers' workstations, they

are advised to use *dedicated build services*, especially *ephemeral environments*. Additionally, they may *isolate build steps* such that they cannot tamper with the output of other build steps.

*Safeguards for Administrators and Consumers* comprises 5 countermeasures. For example, both package repository administrators and consumers can opt for *building packages directly from source code*, rather than accepting pre-built artifacts. If implemented by package repositories, this would reduce the risk of subverted project builds. If implemented by consumers, this would eliminate all risks related to the compromise of 3rd-party build services and package repositories, as they are taken out of the picture.

*Safeguards for Consumers* includes 9 countermeasures that may be employed by the downstream users. The consumers of OSS packages may reduce the impact of malicious code execution when consuming by *isolating the code and/or sandboxing* it. Another example is the *establishment of internal repository mirrors* of vetted components.

### Remark

Some of the presented safeguards are specific to selected package managers, namely *Scoped packages* (Node.js) and *Prevent script execution* (Python and Node.js). All others are relevant no matter the ecosystem, however, control implementations and technology choices differ, e.g., in case of *Application Security Testing*.

## Deeper and deeper: Popularity of Attack Vectors.

At the time of writing (February 2023) we collected a total of 369 references, of which 81 discuss real-world attacks and 126 are peer-reviewed papers. Figure 3 shows the number of publications by year for all the references (Fig. 3a), for the references related to real-world attacks (Fig. 3b), and for the peer-reviewed references (Fig. 3c).

By grouping the peer-reviewed and real-world attacks references per attack vector we observe the following.

Among the peer-reviewed references we have, in order of popularity:

1) 35 resources discussing the general problem of *Conducting OSS Supply Chain Attack (AV-000)*;
2) 33 resources discussing security aspects and issues about *distribution platforms (AV-500)*;
3) 24 resources discussing security aspects and issues of *build systems (AV-400)*;
4) 19 resources discussing security aspects and issues of *VCS (AV-300)*;
5) 8 resources discussing problems on packages *creating name confusion with legitimate packages (AV-200)*;
6) 3 resources discussing problems on *malicious packages developed and advertised from scratch (AV-100)*.

For what concerns real-world attacks we have, in order of popularity:

1) 26 attacks that exploited the *creation of name confusion with legitimate packages (AV-200)*;
2) 25 attacks that exploited attack vectors in the context of *distribution platforms (AV-500)*;
3) 20 attacks that exploited attack vectors in the context of *VCS (AV-300)*;
4) 14 attacks that exploited attack vectors in the context of *build systems (AV-400)*;
5) 7 attacks that consisted of the *development and advertisement of a malicious package from scratch (AV-100)*.

It is straight-forward to observe that the most exploited vector among real-world attacks (i.e., *create name confusion with legitimate packages*) is the least discussed in academic papers. For other attack vectors there is almost a match between attack vectors addressed in peer-reviewed papers and popularity of attacks. The prevention of package name squatting is complex and there exist legitimate uses for which organizations do upload packages with similar names, e.g., using the same prefix for all packages they develop to make them easily recognizable. Academic work has begun to propose techniques for the detection of malicious code in package repositories. Though they do not target solely name squatting, one of the safeguards most discussed among peer-reviewed papers, *Application Security Testing (AST)* (both static and dynamic), is protecting against such attacks as well as it has the advantage of being general (regardless of whether the malicious package name is squatted or not). However, vetting entire package repositories is computationally burdensome and the false-positive rate must be low so that analyst manual review is practically feasible.

Another aspect to highlight is the fact that academic works discuss the general problem of OSS supply chain attacks and, after AST, the most discussed countermeasures are respectively SCA/SBOM and vulnerability assessment.

*The most exploited vector among real-world attacks (i.e., create name*

*confusion with legitimate packages) is the least discussed in academic papers.*

## A tool to the rescue: SAP's Risk Explorer for Software Supply Chain

Given the size of the taxonomy and the amount of information associated with it, we develop a companion tool to ease the access and consumption of the taxonomy. The tool is open-source[2] to foster the creation of a community that can benefit of and contribute to the taxonomy. When pull requests are merged in the main branch, a new version of the tool is automatically deployed.

Users can collapse and expand the different nodes of the attack tree to explore the attack surface of open-source-based software development. The description of the respective attack vector, references, as well as associated safeguards are shown below the tree (cf. Figure 4). This exploratory mode of visualization offers to the user the benefit of managing visual complexity and accessing information in more consumable portions on-demand.

A share button generates a deep link to individual attack vectors, which can be referenced from 3rd party websites, e.g., in training material or in security advisories that need to reference the attack vector(s) used in a given attack and have a clearer explaination of it. A search field in the top-right corner allows searching for attack vectors by name. Upon selection, the respective path from the root node to the selected attack vector is expanded and highlighted in red. The second search field right below is for safeguards. Upon selection, all the nodes mitigated by the respective safeguard are colored in green (cf. Figure 4). All attack vectors, safeguards, and bibliographical references can also be displayed in tabular form. The tabular display of attack vectors also allows showing information about associated safeguards in a modal window. References can be sorted after title, publication year, and affected ecosystem.

### Industrial Use Cases

The Risk Explorer tool can support the following industrial activities.

**Training and Awareness.** The Risk Explorer tool enables an interactive visualization of the taxonomy, the description of attack vectors and safeguards, and the dataset of resources reviewed during the systematization of knowledge. Thus, it can be used to better understand security risks in software supply chains and raise awareness among developers and security practitioners.

For example, let us assume that we heard about the attack to the *event-stream* package. To figure out which attack vector was used, we can search for such package in the Risk Explorer through the tabular view of references (cf., Fig. 5). In the column *Related Attack vectors* we find which attack vector is assigned to. We can then click on the deep link to locate such attack within the taxonomy. At this point, the risk explorer provides the description of the specific attack vector used to infect the *event-stream* package, other attacks that used the same strategy, and mitigating safeguards that would protect against this attack. Looking at the path from the specific attack vector up to the root node, we can learn about the chain of goals that led to the supply chain attack. By stopping at a certain depth of the taxonomy we can learn about what other possibilities attackers could have used and more generic safeguards that would protect against a broader spectrum of attacks.

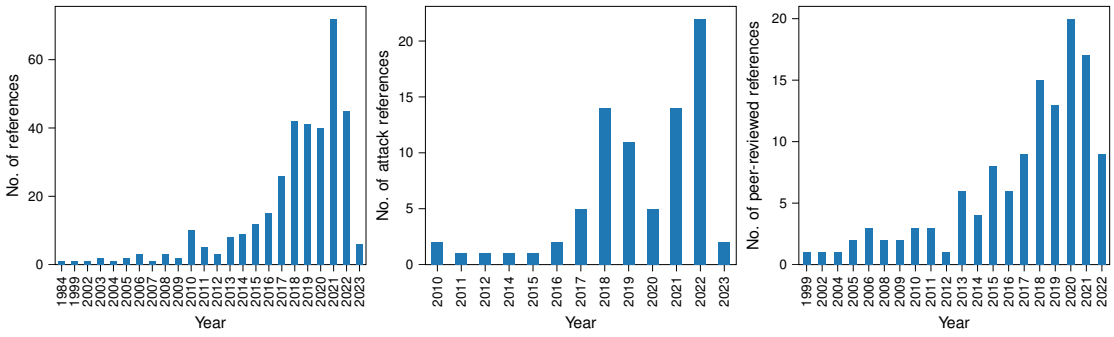
A security expert at SAP used the tool when providing guidance on supply chain security across the entire organization. She reported that, through the suggested safeguards and linked references, the tool *"helped learning more about preventive and detective measures"*. Content, but especially the combination of features offered by the tool, e.g. *"the visualization and mapping, [and the] ability to see cross-references"*, was reported as a strength. The expert concluded that she *"will continue to reference to this research while improving security practices for development"*.

*"[The Risk Explorer] helped learning more about preventive and detective measures."*

**Threat Modeling.** During this activity, the goal is to outline the threats to which a system is (potentially) subjected and then identify possible countermeasures that can be put in place to protect against those threats.

By presenting an extensive list of attack vectors, the tool can support threat modeling activities in the context of OSS supply chain attacks. In particular, once the architectural diagram of the infrastructure is outlined, one can identify to which threats (i.e., attack vectors) the actors and systems may be subject to. Moreover,

[2] https://github.com/SAP/risk-explorer-for-software-supply-chains




(a) (b) (c)

FIGURE 3: Number of collected references per years of publication. (a) References of all types; (b) References discussing real-world attacks; (c) Peer-reviewed references.

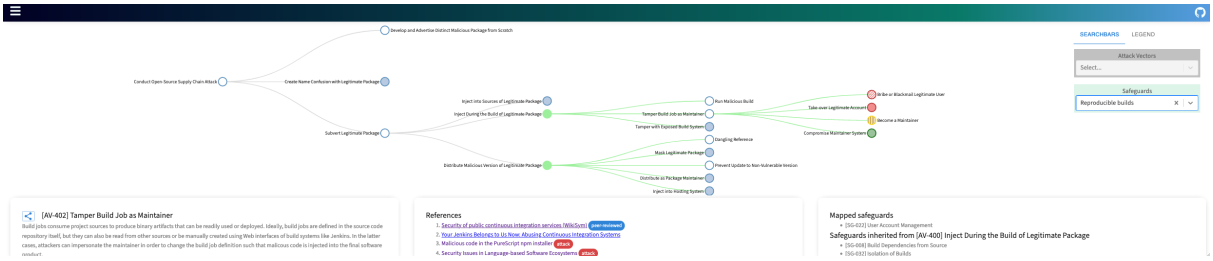


FIGURE 4: Screenshot of the attack tree visualization (in green attack vectors covered by safeguard *Reproducible builds*) [6].

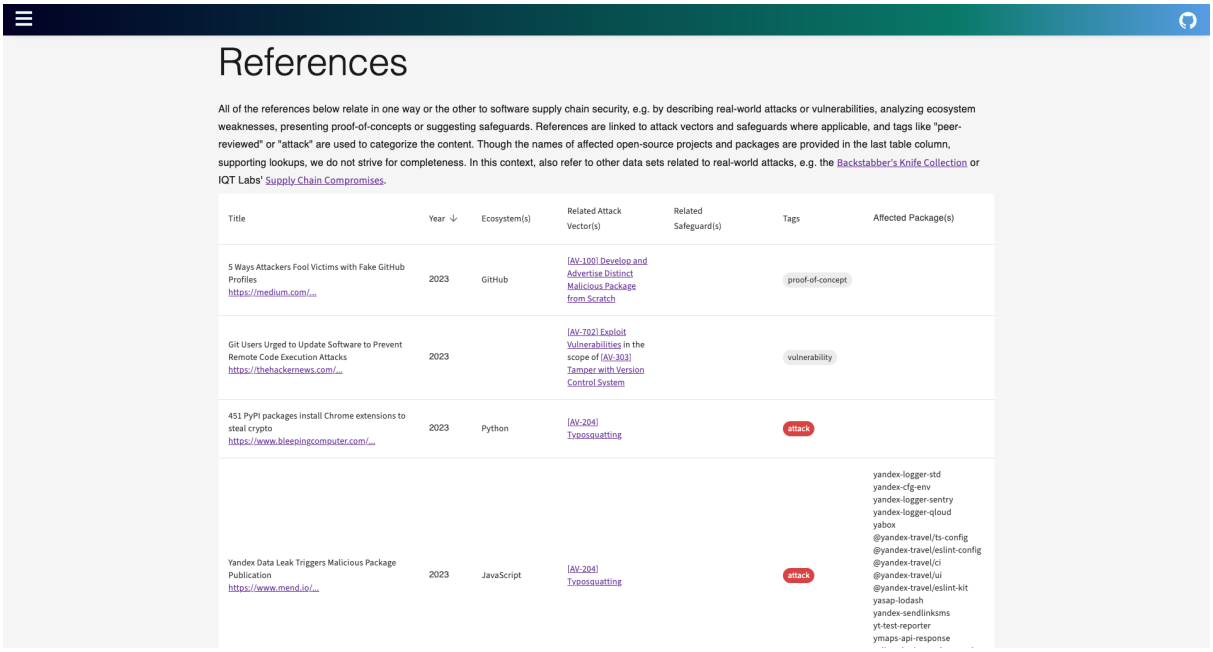



FIGURE 5: Page containing the tabular view of all the references categorized. They can be sorted by title, year of publication, related attack vectors, related safeguards, and by type.

the cross-reference to the related countermeasures helps in understanding how to protect the infrastructure from those threats.

**Scope red-team activities.** Red-teaming activities consist of simulating real-world attack scenarios to evaluate the security capability of a system.

Similarly to the ATT&CK Navigator[3], the tool can also be used for red/blue team activities planning and security assessments. The enumeration of attack vectors (child nodes) according to different goals (parent nodes) provides a check-list of possible strategies to be adopted (as an attacker) during the security assessment of a project.

**Gap analysis of safeguards.** The visual highlight of attack vectors covered by given safeguards through the safeguards searchbar (cf. Fig. 4) can help in the gap-analysis, thus to evaluate which attack vectors are being mitigated by the safeguards in place and which, if any, would need to be implemented.

### Adding a new attack reference: the case of PyTorch-nightly.

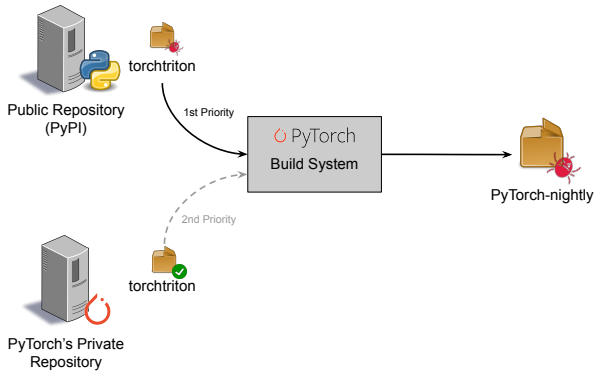



FIGURE 6: Highlight of the attack vector used in the PyTorch-nightly compromise within the taxonomy.

Let us assume that we learn about a new attack and we want to contribute by assigning it to the appropriate attack vector within the tree. We consider as an example the recent PyTorch-nightly's compromise (December 25th-30th, 2022).

The security advisory states that *"a malicious dependency package (torchtriton) [...] was uploaded to the Python Package Index (PyPI) code repository with the same package name as the one we ship on the PyTorch nightly package index. Since the PyPI index takes precedence, this malicious package was being installed instead of the version from our official repository"* [13]. The attack is depicted in Figure 6.

To assign the attack to a node of the taxonomy, we need to navigate it from the root node, establishing which path to follow when moving to deeper nodes. From the root node, the first question we ask ourselves is whether the attack consists of developing a malicious package from scratch, exploiting a name similar to that of a legitimate project, or whether a legitimate package has been subverted. Since `torchtriton` is a legitimate dependency used by PyTorch-nightly, we fall in the last case. From the node *Subvert Legitimate Package* (AV-001), we need to discriminate whether the attack occurred via the VCS, build system or the distribution platform. In the compromise of `torchtriton`, neither the source code nor the build infrastructure were compromised, the attack occurred in the PyPI package repository and thus we fall in the last node *Distribute Malicious Version of Legitimate Package* (AV-500). By reading the security advisory, the root cause of the attack was caused by the fact that `torchtriton` was only present in the PyTorch nightly package index (used to build the PyTorch-nightly project) and not in the public PyPI repository. Thus, the legitimate dependency `torchtriton` was *masked* by *abusing the dependency resolution mechanism* (AV-509). The PyTorch-nightly's compromise can thus be assigned to node (AV-509) (cf. Fig. 7).

Whenever an attack exploits new ways of injecting malicious code, it may happen that the existing vectors may not covering the newly identified behavior. In such cases an extension of the taxonomy itself has to be considered.

## We're not alone: Related frameworks

Open-source and other organizations have recognized the importance of securing open-source software and open-source based supply chains in the development of secure software and services. Several frameworks have been proposed to provide practices for a secure software lifecycle.

The Enduring Security Framework[4] (a public-private cross-sectional working group led by NSA, CISA, and ODNI) released actionable guidance to developers, suppliers, and customers to secure the entire software supply chain. "Securing the Software Supply Chain for Developers" targets the software development lifecycle and considers threats to the development of secure code, to the verification of third-party components, to the hardening of the build system, and to the delivering of code and provides recommended mitigations. "Securing the Software Supply Chain for Suppliers" focuses on how vendors should identify threats that could compromise their organi-

[3] https://mitre-attack.github.io/attack-navigator/

[4] https://www.nsa.gov/About/Cybersecurity-Collaboration-Center/Cybersecurity-Partnerships/ESF/

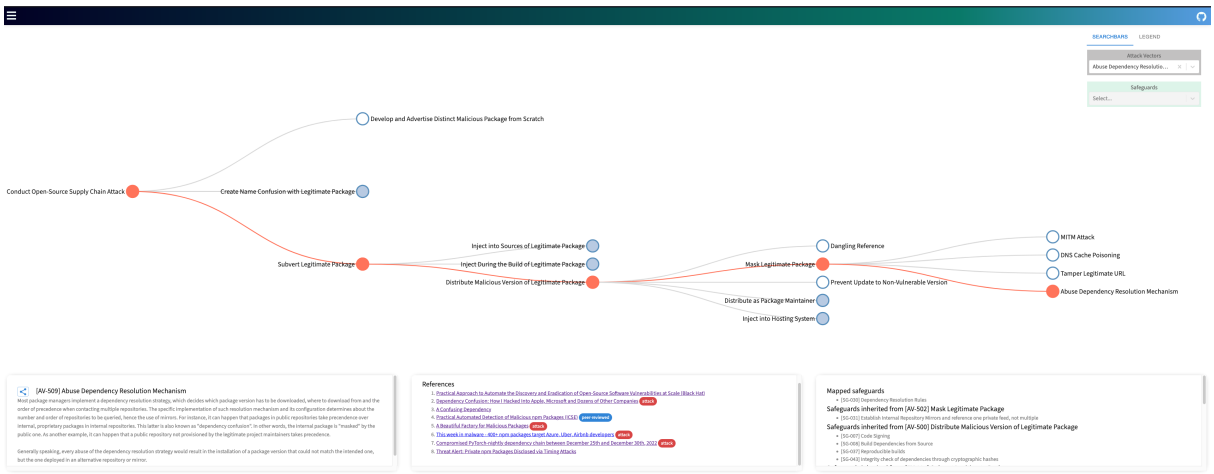



FIGURE 7: Highlight of the attack vector used in the PyTorch-nightly compromise within the taxonomy.

zation, software development, software product, and software delivery. “Securing the Software Supply Chain for Customers” focuses on the best practices for the acquisition, deploymnet, and operation of the software product.

The Microsoft OSS Secure Supply Chain Consumption Framework (S2C2F)[5] combines requirements and tools to reduce risks associated with the consumption of open-source software. It is based on three core concepts: control all consumed open-source software, use of maturity model to help in prioritizing the requirements to implement, and secure the software supply chain at scale. Compared to the framework “Securing the Software Supply Chain for Developers”, the Microsoft S2C2F only focuses on the secure consumption of open source software, but provides a more detailed guidance in this context. It defines 8 practices (e.g., scan and update third-party components) and a list of associated operational requirements (e.g., scan OSS for malware, perform security reviews of OSS). It also comes with a maturity model that organizes the requirements into 4 different levels and lists tools that can support fulfilling the requirements.

The OWASP Software Component Verification Standard (SCVS)[6] aims at establishing a framework for identifying activities, controls, and best practices, which can help in identifying and reducing risk in a software supply chain. Similarly to S2C2F, the OWASP SCVS provides 6 families of controls (e.g., inventory, pedigree and provenance) and 3 levels of verification requiring an increasing number of requirements for higher assurance. However, it applies to software components in general and thus the requirements are not specific for OSS consumed within the supply chain like in the case of the S2C2F framework.

The Supply chain Levels for Software Artifacts (SLSA)[7] is an OpenSSF project and consists of a checklist of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure in your projects, businesses or enterprises. It defines four levels of assurance, from simple provenance information via a documented, automated build process, to high confidence and trust via peer-review of source code changes with hermetic, reproducible builds. Compared to Microsoft S2C2F and OWASP SCVS, SLSA has a narrower scope as it focuses on integrity, thus ensuring that the consumed code has not been tampered.

The different frameworks overlap as they all aim to provide guidance and recommendations and some efforts to map them have been done, e.g., the OSS SSC framework includes a mapping of requirements to other relevant specifications including OWASP SCVS and SLSA. However, all frameworks lack a systematic representation of how attacks can be carried out in order to clearly define what are the threats covered by the provided recommentations.

The taxonomy presented in this work complements such efforts. It takes the perspective of the attacker and provides the most complete taxonomy of attacks whereas the frameworks above focus on

[5] https://www.microsoft.com/en-us/securityengineering/opensource/osssscframeworkguide

[6] https://owasp.org/www-project-software-component-verification-standard/

[7] https://slsa.dev/

safeguards by providing recommendations. Similar to what the MITRE ATT&CK framework does for attacks to infrastructures, it helps in describing how attackers can exploit the software supply chain to spread malwares. Having a complete taxonomy is key for establishing whether the identified recommendations cover the known attack vectors. By providing a base of knowledge related to attacks, it helps in designing and researching novel countermeasures and may also facilitate the comparison of the existing frameworks.

## From attack vectors to malicious code

As mentioned before, the scope of the taxonomy is to describe *how* attackers can inject malicious code in upstream projects, regardless the actual malicious content injected. To complement, looking at the malicious code provides valuable information about the behavior the attackers are trying to achieve and how different ecosystems are affected.

*Malware Behavior*

As first mentioned in [9], the main objectives observed in existing OSS supply chain attacks are:

- *Reverse shell*, which consists in spawning a shell process and redirects both its input and output through an open socket to the attacker machine.
- *Droppers* connect to an attacker-controlled host to download a second-stage payload that will be then executed. The remote payload can be read directly through the connection or be temporarily stored in a local file.
- *Data exfiltration* (most common behavior [9]) aims at reading sensitive information (e.g., environment variables, files) to then sends it to an attacker-controlled endpoint.
- *Denial of Service (DoS)* is typically achieved either through resource exhaustion (e.g., fork-bombs) or by deleting system files.
- *Financial gain* obtained by executing crypto miners in the target system.

During a recent analysis we performed on the NPM and PyPI ecosystem[8], we also observed the presence of *research proofs-of-concept* aiming at testing execution functionalities offered by the language and/or the specific package manager to show the potential risks when installing such packages. Another unwanted behavior we observed is the presence of *rickrolling attacks*. When we observed and reported such packages, the resepective security teams decided not to remove them as they do not consider rickrolling as malicious behavior.

*Malware Techniques*

We also observed cases where obfuscation techniques are used. Also note that code obfuscation may be more or less interesting for an attacker depending on the ecosystem. In case of interpreted languages, downloaded packages contain the malware's source code, which makes it more accessible to analysts compared to compiled languages. The presence of encoded or encrypted code in such packages prooved being a good indicator of compromise [11], as there are few legimitate use-cases for open-source packages (e.g., minification, mostly used for frontend JavaScript libraries). Still, the quantity of open-source packages and versions makes manual inspection very difficult, even if source code is accessible. [9] showed that malicious packages usually have no obfuscation or use simple techniques (e.g., base64), however a small fraction of the packages we studied used obfuscation with more complex mechanisms, such as encrypting the code (e.g., AES-256) or with custom encodings of strings and identifiers. When it comes to compiled code, well-known techniques like packing, dead-code insertion or subroutine reordering make reverse engineering and analysis more complex.

For an attack to succeed, malicious code needs to be executed. Attackers achieve this either at installation time, during test-cases, or at runtime (e.g., by embedding the payload in a specific function or initializer). The execution at installation time is possible in only certain ecosystems. For Python and Node.js, this is commonly achieved through installation hooks, which trigger the execution of code provided in the downloaded package (e.g., in `setup.py` for Python or `package.json` for JavaScript). A comparable feature is not present in most compiled languages, like Java or C/C++.

To increase the chances of succeeding, attackers may conduct malware campaigns, which consist in the distribution of a large number of packages that implement the same malicious behavior to increase the reach towards downstream users. These campaigns can affect only one ecosystem or be cross-language (i.e., same malicious behavior implemented in different languages and spread to related ecosystems).

*Malware Reporting*

How malicious packages are reported to the respective security teams varies from one ecosystem to another. For example, NPM offers the reporting functionality only through UI within the official website: the reporter

[8] Paper under submission to a conference that requires anonymity

has to search for the package in a searchbar, navigate to the project page, click on the button 'Report Malware', and finally fill in the report form. If multiple reports are submitted in a short time frame, CAPTCHA checks are enabled in the website. For PyPI, instead, the official procedure to report malicious packages consists in sending an e-mail to the security team[9] with the names of the packages and (preferably) the link to the lines of code containing the malware highlighted using *Inspector*[10]. Becaue of the way they are structured, these procedures make it complex to report multiple malwares at the same time, especially considering the popularity of malware campaigns.

> *"[current reporting mechanisms] make it complex to report multiple malwares at the same time, especially considering the popularity of malware campaigns."*

Once reported and confirmed, malicious packages are removed from package repositories and are no longer publicly accessible. Given the significance of the problem and the interest from both academia and industry in developing detection methods for software supply chain attacks, it would be beneficial to maintain a dataset of these attacks (both packages and associated metadata).

> *"Once reported and confirmed, malicious packages are removed from package repositories and are no longer publicly accessible. [...] it would be beneficial to maintain a dataset of these attacks (both packages and associated metadata)."*

## The journey ended? Benefits and open challenges.

Our work systematizes knowledge about OSS supply chain security by abstracting, contextualizing and classifying existing works. The proposed taxonomy can benefit future research by offering a central point of reference and a common terminology. The comprehensive list of attack vectors and safeguards can support assessing the security level of open-source projects, e.g., to conduct comparative empiric studies across projects and ecosystems and over time.

An open challenge in OSS supply chain attacks is the detection of malicious code. The availability of source code in ecosystems for interpreted languages suggests that malware analysis is more straightforward. Still, recent publications focus on those ecosystems, especially JavaScript and Python [5], partly due to their popularity, but also because existing malware analysis techniques cannot be easily applied. More subtle attacks, such as intentional insertion of vulnerabilities, complicate detection since they require analysis of the context of the change to distinguish it from an accidentally introduced vulnerability. Additionally, code generation and the difficulty in identifying VCS commits that correspond to pre-built components make malware analyis of source code difficult.

[9] https://pypi.org/security/
[10] https://inspector.pypi.io

## REFERENCES


1. Nicholas Boucher and Ross Anderson. Trojan source: Invisible vulnerabilities, 2021.
2. European Network and Information Security Agency. Enisa threat landscape 2021, 2021. [Online; accessed 20-October-2021].
3. Joseph R. Biden JR. Executive order on improving the nation's cybersecurity, 2021. [Online; accessed 20-October-2021].
4. Alexander Kjäll, Stian Kristoffersen, and Ståle Pettersen. How we protected ourselves from the dependency confusion attack. *URL: schibsted.com/blog/dependency-confusion-how-we-protected-ourselves/*, 2021.
5. P. Ladisa, H. Plate, M. Martinez, and O. Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 167–184, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.
6. Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Risk explorer for software supply chains: Understanding the attack surface of open-source based software development. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED'22, page 35–36, New York, NY, USA, 2022. Association for Computing Machinery.
7. Stuart McClure, Shanit Gupta, Carric Dooley, Vitaly Zaytsev, Xiao Bo Chen, Kris Kaspersky, M Spohn, and R Permeh. Protecting your critical assets-

lessons learned from operation aurora. *Tech. Rep.*, 2010.

8. Ryan Naraine. Open-source ProFTPD hacked, backdoor planted in source code. https://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code/. [Accessed 15-Mar-2022].
9. Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks, 2020.
10. Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the solarwinds incident. *IEEE Security Privacy*, 19(2):7–13, 2021.
11. Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. *arXiv preprint arXiv:2202.13953*, 2022.
12. Sonatype. 8th Annual State of the Software Supply Chain Report. https://www.sonatype.com/state-of-the-software-supply-chain/introduction, 2023. [Accessed 01-Feb-2023].
13. PyTorch Team. Compromised pytorch-nightly dependency chain between december 25th and december 30th, 2022. https://pytorch.org/blog/compromised-nightly-dependency/. [Accessed 02-Feb-2023].
14. Nikolai Philipp Tschacher. *Typosquatting in programming language package managers*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2016.