Secure Supply Chain Consumption Framework (S2C2F)

Version 1.1

Pre-draft

CONTENTS

Secure Supply Chain Consumption Framework (S2C2F)

Forward

Some of the elements of this document may be the subject of patent rights. No party shall be held responsible for identifying any or all such patent rights.
Any trade name used in this document is information given for the convenience of users and does not constitute endorsement.

This document was prepared by OpenSSF's Secure Supply Chain Consumption Framework (S2C2F) Special Interest Group governed by the Supply Chain Integrity Working Group.

Introduction

The purpose of this document is to illustrate the core concepts of the Secure Supply Chain Consumption Framework (S2C2F) to outline and define how to securely consume open source software (OSS) dependencies, such as NuGet and NPM, into the developer's workflow. This framework is applicable to OSS dependencies consumed into the developer's workflow, such as any source code, language package, module, component, container, library, or binary. This guide provides a dedicated framework to enhance any organization's OSS governance program to address supply chain threats specific to OSS consumption.
The Secure Supply Chain Consumption Framework (S2C2F) is a security assurance and risk reduction process that is focused on securing how developers consume open source software. The S2C2F provides security guidance and tools throughout the developer inner-loop and outer-loop processes that have played a critical role in defending and preventing supply chain attacks through consumption of open source software. Using a threat-based risk-reduction approach, the goals of the S2C2F are to:

1. Provide a strong OSS governance program
2. Improve the Mean Time To Remediate (MTTR) for resolving known vulnerabilities in OSS
3. Prevent the consumption of compromised and malicious OSS packages

OSS has become a critical aspect of any software supply chain. Across the software industry, developers are using and relying upon OSS components to expedite developer productivity and innovation. However, attackers are trying to abuse these package manager ecosystems to either distribute their own malicious components, or to compromise existing OSS components.
This document is split into two parts: a solution-agonistic set of practices and a maturity model-based implementation guide. The practices section should be utilized by individuals like Chief Information Security Officers (CISOs) and security, engineering, compliance/risk managers while the implementation guide should be utilized by software developers and other security practitioners.
This document presents:

- An overview of the Secure Supply Chain Consumption Framework (S2C2F) Practices.
- Common supply chain threats and how the S2C2F can help.
- An overview of the S2C2F Implementation Guide and Maturity Model.
- A process for assessing your organization's maturity.
- Detailed walkthrough of the S2C2F implementation requirements and tools.
- A mapping of the S2C2F requirements to other specifications.

1 Scope

The document outlines how to securely consume OSS dependencies, such as NuGet and NPM, into the developer's workflow.

It is applicable to organizations that do software development, that take a dependency on open source software, and that seek to improve the security of their software supply chain.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
artifact
Work products that are produced and used during a project to capture and convey information (e.g., models, source code).
[SOURCE: NIST SP 800-160v1r1 from ISO 19014:2020]

3.2
artifact store
A repository to store the artifacts used in a build. This can be used to ingest artifacts from the store instead of directly from the external source.

3.3
OSS
Open Source Software; software that ensures that the end users have freedom in using, studying, sharing and modifying that software.
Note 1 to entry: Additional definition about OSS can be found at Reference 1.
[SOURCE: The Free Software Definition]

3.4
packaged artifacts

A logical way to assemble artifacts together. Also called packages, these are sets of program files, data files, etc. developed by programmers to perform specific tasks.

3.**5**
source code artifact
Artifacts remain in their project source code form and have not been converted into a different format.

3.6
upstream source
The source repository and project where contributions happen and releases are made. The contributions flow from upstream to downstream.
[SOURCE: What is an open source upstream? (redhat.com)]

3.7
vulnerability
A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).
[SOURCE: NVD - Vulnerabilities (nist.gov)]

4 Specifications

4.1 Secure Supply Chain Consumption Framework Practices

4.1.1 Practice 1: Ingest It

I can ship any existing asset if external OSS sources are compromised or unavailable.
The first step towards securing a software supply chain is ensuring you control all the artifact inputs. To satisfy this practice, there are two ingestion mechanisms: one for packaged artifacts and one for source code artifacts.

For **packaged artifacts**, we require ingestion into an artifact stores – Linux package repositories, artifact stores, OCI registries – to fully support *upstream sources*, which transparently proxy from the artifact store to an external source and save a copy of everything used from that source. When using a mix of internal and external packaged artifacts, it is important to secure your package source file configuration to protect yourself from dependency confusion attacks.

For **source code artifacts**, we require mirroring external source code repositories to an internal location. Mirroring the source in addition to caching packages locally is also useful for many reasons:

- Business Continuity and Disaster Recovery (BCDR) purposes, so that your organization can take ownership of code if a critical dependency is removed from the upstream

- Enables proactive security scans to look for backdoors and zero-day vulnerabilities
    - Enables your organization to contribute fixes back upstream
- Enables your organization to perform fixes if needed (in extreme circumstances)

4.1.2 Practice 2: Scan It

I know if any OSS artifact in my pipeline has vulnerabilities or malware.
Once we control all artifact inputs, we must scan all inputs to trust them. This trust is built using scanners that look for vulnerabilities, malware, malicious or anomalous behavior, extraneous code, and other known or previously undiscovered issues (i.e. zero-day vulnerabilities).

4.1.3 Practice 3: Inventory It

I know where OSS artifacts are deployed in production.
Once we have ingested and scanned the artifacts entering the software supply chain, we must ensure that we have an inventory where each artifact is used, by knowing in which services it is deployed and in which products it was released. This is required for incident response scenarios so that teams affected by a compromised package can be contacted so the appropriate actions can be taken to remove the affected package.

4.1.4 Practice 4: Update It

I can deploy updated external artifacts soon after an update becomes publicly available.
Once we have ingested, scanned, and inventoried where each artifact is used, we can enable developers to *fix* issues with artifacts that have already been used by knowing the supply chain processes that released the product/service that needs the fix.

Every organization should aspire to patch vulnerable OSS packages in under 72 hours so that you patch faster than the adversary can operate. Using tools to auto-generate Pull Requests (PRs) to update vulnerable OSS become critical capabilities for securing your supply chain.

4.1.5 Practice 5: Audit It

I can prove that every OSS artifact in production has a full chain-of-custody from the original artifact source and is consumed through the official supply chain.

Now that we have ingested, scanned, inventoried, and provided the ability to update any artifact that has come through the software supply chain properly, you must have the ability within your organization to audit OSS consumption to see if it is coming through the standardized consumption tools (such as a package repository solution) established by your organization.

4.1.6 Practice 6: Enforce It

I can rely on secure and trusted OSS consumption within my organization.
All OSS artifacts must be consumed from trusted sources and through the official OSS consumption channels. The next step is to enable enforcement of the supply chain so that all artifacts that in any way impact a production service/release must come through the full supply chain.

4.1.7 Practice 7: Rebuild It

I can rebuild from source code every OSS artifact I'm deploying.
Until now, we have assumed that we took our inputs at the beginning of the supply chain *as-is*: as the package, container, or other delivery vehicle provided by the author. For key artifacts that are business-critical and for all artifacts that are inputs to High Value Assets, this assumption may not be sufficient. Hence, the next step to secure the supply chain is creating a chain of custody *from the original source code* for every artifact used to create a production service/release.

The baseline REBUILD IT requirement is to enable developers who have a critical dependence on certain OSS components to ingest source code (including discovering the source code, which is not always linked to the built artifact), rebuild it (possibly developing build scripts along the way, if they're not part of the source code), make any post-build modifications (e.g. signing), cache the rebuilt artifact, and advertise the internally-rebuilt version's existence to other teams in the organization. One other potential method is the use of multiple third parties to build and come to consensus on a 'correct' artifact.

4.1.8 Practice 8: Fix It + Upstream

I can privately patch, build, and deploy any external artifact within 3 days of harm notification and confidentially contribute the fix to the upstream maintainer.

When To Use This: *This is intended to be used only in extreme scenarios and for temporary risk mitigation. It should only be used when the upstream maintainer is unable to provide a public fix within an acceptable time for your Organization's risk tolerance. The first action any organization should take is to confidentially report the vulnerability to the upstream maintainer AND help suggest a fix.*
Once we can rebuild any artifact used in the software supply chain, the final step is to be able to privately fix it while confidentially disclosing the vulnerability to the upstream maintainer. Assuming that the team that ingested the source and rebuilt the artifact has allowed PRs to their forked copy of the source and set up CI builds appropriately, then anyone needing to private fix a component can use the normal PR workflow. The only additional work needed is the ability to distribute the private fix as widely within the organization as is needed.

Related to the note below, the implemented fix should be confidentially contributed to the upstream maintainer to give back to the community.

Important Note: The Fix It + Upstream practice should not be perceived as being at odds with supporting communities and projects. If an organization chooses to take a dependency on open source, they should also find ways to give back to the community.

4.2 Secure Supply Chain Framework Levels of Maturity

When the S2C2F was first developed, the strategy to secure our OSS supply chain was comprised of 8 practices. Since all 8 practices cannot be reasonably implemented at the same time, the following maturity model organizes the requirements from each of the 8 practices into 4 different levels. It allows an organization to make incremental progress from their existing set of security capabilities toward a more secure defensive posture.

Additionally, the maturity model considers different threats and themes at each Maturity Level.

Depending on the projects and their criteria, you may have a mix of framework levels implemented across projects. Additionally, Level 4 of the Maturity Model has a high estimated cost to implement compared to the risk/reward, and therefore should be considered as an aspirational north star vision for your organization. While it is difficult to implement Level 4 at scale across your organization, it is feasible to implement Level 4 on your most critical dependencies for your most critical projects.

Level 1 – Using a package caching solution, performing an OSS inventory, plus scanning and updating OSS represents the most common set of OSS security capabilities across the software industry today.

Level 2 – This maturity level focuses on shifting security further left by improving ingestion configuration security, decreasing MTTR to patch OSS vulnerabilities, and responding to incidents. The SaltStack vulnerability in 2020 showed us that adversaries were able to start exploiting CVE-2020-11651 within 3 days of it being announced. Even though a patch was available, organizations were not able to patch their systems fast enough. Thus, a key component of this level leverages automation to help developers keep their OSS hygiene healthy and updated. The ideal goal is for organizations to be able to patch faster than attackers can operate.

Level 3 – Proactively performing security analysis on your organization's most used OSS components and reducing risk to consume malicious packages are the themes of this maturity level. Scanning for malware in OSS before the package is downloaded is key toward preventing compromise. Then, to perform proactive security reviews of OSS requires that an organization can clone the source code to an internal location. Proactive security reviews help you look for the not-yet-discovered vulnerabilities, as well as identifying other threat categories such as detecting backdoors.

Level 4 – This level is considered aspirational in most cases. Rebuilding OSS on trusted build infrastructure is a defensive step to ensure that the OSS was not compromised at build time. Build time attacks are performed by the most sophisticated adversaries and do not occur very frequently. Thus, this level of maturity is what's required to defend against the most sophisticated adversaries. Additionally, rebuilding OSS has many subtle technical challenges such as what to name the package to prevent collisions with upstream? How to make sure all developers use the internal package instead of the external? Rebuilding also enables you to implement fixes (if needed) and deploy them at scale across your organization.

Annex A (informative) – Framework core concepts

The S2C2F is modeled after three core concepts— *control all artifact inputs, continuous process improvement, and scale.*

- Control All Artifact Inputs: There are a myriad of ways that developers consume OSS today: git clone, wget, copy & pasted source, checking-in the binary into the repo, direct from public package managers, repackaging the OSS into a .zip, curl, apt-get, git submodule, and more. Securing the OSS supply chain in any organization is going to be near impossible if developer teams don't follow a uniform process for consuming OSS. Enforcing an effective secure OSS supply chain strategy necessitates standardizing your OSS consumption process across the various developer teams throughout your organization, so all developers consume OSS using governed workflows.

- Continuous Process Improvement: To help guide organizations through continuous process improvement, we have organized the S2C2F into a maturity model. This helps organizations prioritize which requirements they should implement first. Since security risk is dynamic and new threats can emerge at any time, the S2C2F places heavy emphasis on understanding the new threats to the OSS supply chain and _requires_ regular evaluation of S2C2F controls and introduction of changes in response to new technology advancements or new threats.
- Scale: The S2C2F tools were designed with scale in mind. Some organizations may attempt to secure their OSS ingestion process through a central internal registry that all developers within the organization are supposed to pull from. However, what if one developer chooses to pull straight from pypi.org or npmjs.com? Is there anything preventing them from doing so? A central internal registry also has the problem of requiring a team to manage the process and workflow, which is extra overhead. As such, the S2C2F tools were developed to secure how they consume OSS today at scale without requiring a central internal registry or central governance body.

Annex B (normative) - Requirements and respective practices, and implementation details

Table B.1 breaks down the requirements of the framework and maps them to the 8 practices. It also includes the benefits of each requirement.

Practice	Requirement ID	Maturity Level	Requirement Title	Benefit
Ingest it	ING-1	L1	Use package managers trusted by your organization	Your organization benefits from the inherent security provided by the package manager
	ING-2	L1	Use an OSS binary repository manager solution	Caches a local copy of the OSS artifact and protects against left-pad incidents, enabling developers to continue to build even if upstream resources are

				unavailable
	ING-3	L3	Have a Deny List capability to block known malicious OSS from being consumed	Prevents ingestion of known malware by blocking ingestion as soon as a critically vulnerable OSS component is identified or if an OSS component is deemed malicious
	ING-4	L3	Mirror a copy of all OSS source code to an internal location	Business Continuity and Disaster Recovery (BCDR) scenarios. Also enables proactive security scanning, fix it scenarios, and ability to rebuild OSS in a trusted build environment.
Scan It	SCA-1	L1	Scan OSS for known vulnerabilities (i.e. CVEs, GitHub Advisories, etc.)	Able to update OSS to reduce risks
	SCA-2	L1	Scan OSS for licenses	Ensure your organization remains in compliance with the software license
	SCA-3	L2	Scan OSS to determine if its end-of-life	For security purposes, no organization should take a dependency on software that is no longer receiving updates
	SCA-4	L3	Scan OSS for malware	Able to prevent ingestion of malware into your CI/CD environment
	SCA-5	L3	Perform proactive security review of OSS	Identify zero-day vulnerabilities and confidentially contribute fixes back to the upstream maintainer

Inventory It	INV-1	L1	Maintain an automated inventory of all OSS used in development	Able to respond to incidents by knowing who is using what OSS where. This can also be accomplished by generating SBOMs for your software.
	INV-2	L2	Have an OSS Incident Response Plan	This is a defined, repeatable process that enables your organization to quickly respond to reported OSS incidents
Update It	UPD-1	L1	Update vulnerable OSS manually	Ability to resolve vulnerabilities
	UPD-2	L2	Enable automated OSS updates	Improve MTTR to patch faster than adversaries can operate
	UPD-3	L2	Display OSS vulnerabilities as comments in Pull Requests (PRs) • **Prerequisite**: Two-person PR reviews are enforced.	PR reviewer doesn't want to approve knowing that there are unaddressed vulnerabilities.
Audit It	AUD-1	L3	Verify the provenance of your OSS	Able to track that a given OSS package traces back to a repo
	AUD-2	L2	Audit that developers are consuming OSS through the approved ingestion method	Detect when developers consume OSS that isn't detected by your inventory or scan tools
	AUD-3	L2	Validate integrity of the OSS that you consume into your build	Validate digital signature or hash match for each component
	AUD-4	L4	Validate SBOMs of OSS that you consume into your build	Validate SBOM for provenance data, dependencies, and its digital signature for SBOM integrity

Enforce It	ENF-1	L2	Securely configure your package source files (i.e. nuget.config, .npmrc, pip.conf, pom.xml, etc.)	By using NuGet package source mapping, or a single upstream feed, or using version pinning and lock files, you can protect yourself from race conditions and Dependency Confusion attacks
	ENF-2	L3	Enforce usage of a curated OSS feed that enhances the trust of your OSS	Curated OSS feeds can be systems that scan OSS for malware, validate claims-metadata about the component, or systems that enforce an allow/deny list. Developers should not be allowed to consume OSS outside of the curated OSS feed
Rebuild It	REB-1	L4	Rebuild the OSS in a trusted build environment, or validate that it is reproducibly built • **Prerequisite**: Sufficient build integrity measures are in place to establish a trusted build environment.	Mitigates against build-time attacks such as those seen on CCleaner and SolarWinds. Open Source developers could introduce scripts or code that aren't present in the repository into the build process or be building in a compromised environment.
	REB-2	L4	Digitally sign the OSS you rebuild	Protect the integrity of the OSS you use.
	REB-3	L4	Generate SBOMs for OSS that you rebuild	Captures the supply chain information for each package to enable you to better maintain your dependencies, auditability, and blast radius assessments
	REB-4	L4	Digitally sign the SBOMs you produce	Ensures that consumers of your SBOMs can trust that the contents have not been tampered with

Fix It + Upstream	FIX-1	L4	Implement a change in the code to address a zero-day vulnerability, rebuild, deploy to your organization, and confidentially contribute the fix to the upstream maintainer	To be used only in extreme circumstances when the risk is too great and to be used temporarily until the upstream maintainer issues a fix.

Annex C (informative) - Maturity model assessment

Any maturity assessment should be done at the Organization level so that it assesses multiple different OSS consumption processes from across different development teams. Some teams may have more mature processes than others, even within a single organization, so it's best to perform a company-wide assessment to determine OSS consumption practices across a diverse set of software development teams. The steps to perform a Maturity Assessment are below:

1. **Prepare for Assessment**. The first step is to understand the concepts behind the S2C2F so you feel comfortable engaging with developers and engineers to inquire about their existing tools, capabilities, and workflows. Next, identify a good sample size of diverse development teams from across the company to interview.
2. **Perform the Assessment**. This is where you assess the organization's degree of maturity in software developer OSS management, security, and consumption processes. Here are a set of example questions that you can ask:
    a. What type of OSS do you consume in your project? (e.g. native C/C++, NuGet, PyPI, npm, etc.)
    b. How are you consuming your OSS into your project? (e.g. Using a Package Cache solution such as Azure Artifacts, commands such as curl or git clone, checking in the OSS into the repo, etc.)
    c. Where do you consume your OSS from? (e.g. NuGet.org, npmjs.com, pypi.org, etc.)
    d. Do you use a mix of internal-only packages and external packages? (*This can make you susceptible to Dependency Confusion attacks*)
    e. Does your package source file (e.g. nuget.config, pom.xml, pip.conf, etc.) contain multiple feeds in its configuration? (*This can make you susceptible to Dependency Confusion attacks*)
    f. Do you do anything custom with how you consume OSS? (e.g. consuming private forks of projects, putting Golang components into a NuGet, etc.)
    g. Does your project use package lock files?
    h. How does your team inventory the use of OSS within your project? What tools are used?
    i. How is your team made aware when a vulnerability exists in an OSS component? What tool is used?
    j. At what point in the Software Development Lifecycle (SDLC) are OSS vulnerabilities surfaced? (e.g. after release? During build? As comments in PRs?)

k. How fast is OSS updated to address known vulnerabilities? (e.g. what is the Mean Time To Remediate)
l. Is updating OSS a manual or automated process? (e.g. using Dependabot)
m. Do you perform integration tests of how your software interfaces with the dependencies you have to validate that there are no breaking changes?
n. Do you scan OSS for malware prior to use?
o. Is your team able to block ingestion of a known-bad/malicious package?
p. Does your team clone open source code internally?
q. Does your team perform any sort of security reviews or scans of OSS before using?
r. Does your team contribute bug fixes back to the upstream OSS maintainer?
s. Do you rebuild any of the open source internally?
t. Do you have an incident response plan or playbook for reacting to an incident of consuming a malicious OSS component?

3. **Plan for Improvements.** Based on the interviews and answers you received from across your organization, you should be able to determine where you fall within the S2C2F Maturity Levels. It's possible that some teams may be ahead of others, so your focus should be on elevating all development teams to a specific Maturity Level. It's suggested that you accomplish this by driving standardization in both process and tooling across your software development teams for consuming OSS.
The S2C2F categorizes its requirements into maturity levels to better help you prioritize investments in improvements. Additionally, the S2C2F recommends tooling with specific capabilities that mitigates against the known supply chain threats, but you probably should make business decisions about which set of tools are right for your business and your security goals.

Annex D (informative) - Common OSS Supply Chain Threats

The S2C2F was designed based on known threats (i.e. tactics and techniques) used by adversaries to compromise OSS packages. The table below is a comprehensive compilation of OSS supply chain threats. It also identifies which S2C2F requirements mitigate the threat.

OSS Supply Chain Threat	Mitigation via S2C2F Requirement
Accidental vulnerabilities in OSS code or Containers that we inherit	UPD-2 UPD-3
Intentional vulnerabilities/backdoors added to an OSS code base	SCA-5
A malicious actor compromises a known good OSS component and adds malicious code into the repo	ING-3 ENF-2 SCA-4

A malicious actor creates a malicious package that is similar in name to a popular OSS component to trick developers into downloading it	AUD-1 ENF-2 SCA-4
A malicious actor compromises the compiler used by the OSS during build, adding backdoors	REB-1
Dependency confusion, package substitution attacks	ENF-1 ENF-2
An OSS component adds new dependencies that are malicious	SCA-4 ENF-2
The integrity of an OSS package is tampered after build, but before consumption	AUD-3 AUD-4
Upstream source can be removed or taken down which can then break builds that depend on that OSS component or container	ING-2 ING-4
OSS components reach end-of-support/end-of-life and therefore don't patch vulnerabilities	SCA-3
Vulnerability not fixed by upstream maintainer in desired timeframe	FIX-1
Bad actor compromises a package manager account (e.g. npm) with no change to the corresponding open source repo and uploads a new malicious version of a package	AUD-1 ENF-2 SCA-4

Bibliography

Here is a list of hyperlinks for documents mentioned within this paper:

1. The Free Software Definition
2. The Open Source Definition

3. Supply Chain Risk Management Practices for Federal Information Systems and Organizations (nist.gov)
4. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities (nist.gov)
5. CIS WorkBench / Benchmarks (cisecurity.org)
6. OWASP Software Component Verification Standard | OWASP Foundation
7. SLSA • Supply-chain Levels for Software Artifacts
8. tag-security/CNCF_SSCP_v1.pdf at main · cncf/tag-security (github.com)