



Microsoft® Open Source Software (OSS) Secure Supply Chain (SSC) Framework

Simplified Requirements

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal reference purposes.

© 2022 Microsoft Corporation. All rights reserved.

Licensed under [Community Specification License 1.0](#)

Table of Contents

Document Change Record	3
Introduction	4
About the Microsoft OSS SSC Framework	4
What is the OSS SSC Framework?	5
Common OSS Supply Chain Threats.....	6
OSS SSC Framework Practices.....	8
Target Audience	8
OSS SSC Framework Practices.....	8
The OSS SSC Framework Implementation Guide.....	12
Target Audience	12
OSS SSC Framework Levels of Maturity	12
How to Assess Where Your Organization is in the Maturity Model?	14
OSS SSC Framework Requirements	16
OSS SSC Framework Tooling Availability.....	18
Implementing the OSS SSC Framework by Level	19
Conclusion.....	26
Appendix: Mapping OSS SSC Framework Requirements to Other Specifications	27
Appendix: References	29

Document Change Record

Date	Author	Version	Change Reference
8/1/2022	Adrian Diglio (Microsoft)	1.0	Initial release

Introduction

The purpose of this paper is to illustrate the core concepts of the Open Source Software (OSS) Secure Supply Chain (SSC) Framework to outline and define how to securely consume OSS dependencies, such as NuGet and NPM, into the developer's workflow. For the purposes of this Framework, we define OSS as any source code, language package, module, component, container, library, or binary that you can consume into your software project as a dependency that does not have a paid-support contract. This guide provides a dedicated framework to enhance any organization's OSS governance program to address supply chain threats specific to OSS consumption.

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 paper is split into two parts: a solution-agnostic 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 paper presents:

- An overview of the Microsoft OSS SSC Framework Practices.
- Common supply chain threats with examples and how OSS SSC Framework can help.
- An overview of the OSS SSC Framework Implementation Guide and Maturity Model.
- A process for assessing your organization's maturity.
- Detailed walkthrough of the OSS SSC Framework implementation requirements and tools.
- A mapping of the OSS SSC Framework requirements to other specifications.

The guidance provided in this paper is targeted toward 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.

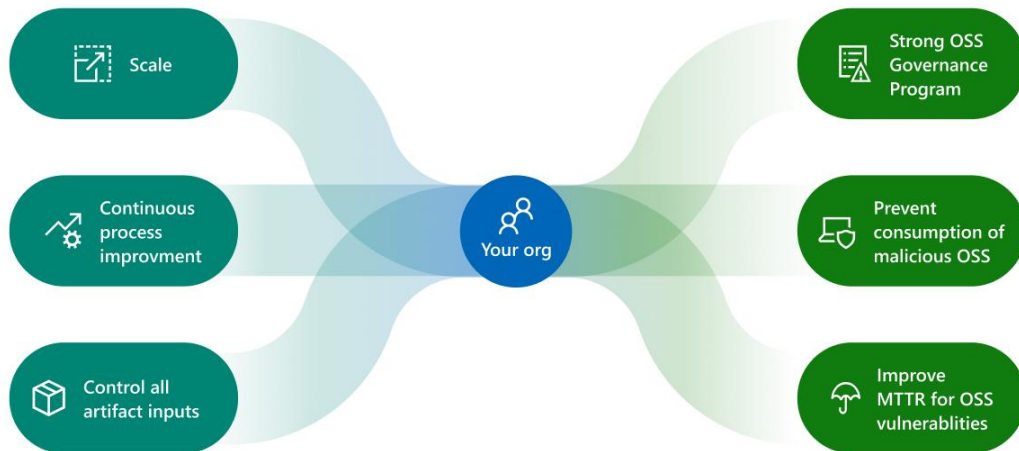
About the Microsoft OSS SSC Framework

The OSS SSC Framework is a security assurance and risk reduction process that is focused on securing how developers consume open source software. As a Microsoft-wide initiative since 2019, the OSS SSC Framework 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 across Microsoft. Using a threat-based risk-reduction approach, the goals of the OSS SSC Framework 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

The Microsoft OSS SSC Framework (described later in this document) is modeled after three core concepts—*control all artifact inputs, continuous process improvement, and scale*.

3 core concepts



3 goals of the framework

- *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 OSS SSC Framework 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 OSS SSC Framework places heavy emphasis on understanding the new threats to the OSS supply chain and *requires* regular evaluation of OSS SSC Framework controls and introduction of changes in response to new technology advancements or new threats.
- *Scale:* The Microsoft OSS SSC Framework 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, Microsoft's OSS SSC Framework tools were developed to secure how they consume OSS today at scale without requiring a central internal registry or central governance body.

What is the OSS SSC Framework?

The OSS SSC Framework is a combination of requirements and tools for any organization to adopt. The Framework includes a capability maturity roadmap to help establish a secure OSS ingestion process to

protect developers from OSS supply chain threats and to establish a strong governance program to manage your organization's use of OSS.

Common OSS Supply Chain Threats

The Microsoft OSS SSC Framework 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 with links to real examples. It also identifies which OSS SSC Framework requirements mitigate the threat. To see the full list of requirements and their benefits, please see the *OSS SSC Framework Requirements* later in this document.

For other sources of OSS threats, please see the following links:

- [Threats, Risks, and Mitigations in the Open Source Ecosystem](#)
- [Taxonomy of Attacks on Open-Source Software Supply Chains](#)
- [Software Supply Chain Threats](#)

OSS Supply Chain Threat	Real Example	Mitigation via OSS SSC Framework Requirement
Accidental vulnerabilities in OSS code or Containers that we inherit	SaltStack	UPD-2 UPD-3
Intentional vulnerabilities/backdoors added to an OSS code base	phpMyAdmin	SCA-5
A malicious actor compromises a known good OSS component and adds malicious code into the repo	ESLint incident	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	Typosquatting	AUD-1 ENF-2 SCA-4
A malicious actor compromises the compiler used by the OSS during build, adding backdoors	CCleaner	REB-1
Dependency confusion, package substitution attacks	Dependency Confusion	ENF-1 ENF-2

An OSS component adds new dependencies that are malicious	Event-Stream incident	SCA-4 ENF-2
The integrity of an OSS package is tampered after build, but before consumption	How to tamper with Electron apps	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	left-pad	ING-2 ING-4
OSS components reach end-of-support/end-of-life and therefore don't patch vulnerabilities	log4net and CVE-2018-1285	SCA-3
Vulnerability not fixed by upstream maintainer in desired timeframe	Prototype Pollution in Lodash	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	Ua-parser-js	AUD-1 ENF-2 SCA-4

OSS SSC Framework Practices

Target Audience

This section is a solution-agnostic description of what should be implemented to secure your organization's OSS supply chain. The guidance is useful to compliance/risk managers, security managers, engineering managers, and Chief Information Security Officers (CISOs).

OSS SSC Framework Practices

Practice 1: Ingest It

I can ship any existing asset if external OSS sources are compromised or unavailable.

Sample threat scenarios addressed by this job:

- The Docker Hub repository becomes compromised
- A team might be targeted by a dependency confusion attack
- Azure itself is unavailable and we need access to OSS assets to restore it
- A package becomes permanently unavailable (i.e. left-pad is removed)

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 ([CVE-2021-24105](#)).

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)

Practice 2: Scan It

I know if any OSS artifact in my pipeline has vulnerabilities or malware.

Sample threat scenarios addressed by this job:

- A team tries to use an OSS package with a known vulnerability
- A team is already using an OSS package believed to be secure, but a new vulnerability in that package is later publicly disclosed
- A team tries to use an OSS package that is known to steal bitcoins (i.e. the *event-stream* scenario)
- A team tries to use an OSS package with a backdoor

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).

Practice 3: Inventory It

I know where OSS artifacts are deployed in production.

Sample threat scenarios addressed by this job:

- A critical vulnerability is discovered in log4j, and the incident response team wants to know all the production services using log4j so they can appropriately staff and coordinate a response effort

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.

Practice 4: Update It

I can deploy updated external artifacts soon after an update becomes publicly available.

Sample threat scenarios addressed by this job:

- A team is currently using three different vulnerable NuGet packages and upgrading each package will be a substantial amount of work for the team. The team chooses to start by upgrading the most widely deployed package.

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.

Given the [SaltStack incident](#), where a vulnerability was exploited within 3 days after announcement, 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 such as Dependabot to auto-generate Pull Requests (PRs) to update vulnerable OSS become critical capabilities for securing your supply chain.

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.

Sample threat scenarios addressed by this job:

- A well-meaning but misguided developer bypasses the official engineering pipeline to update an OSS package directly in a release; however, this new version contains a known vulnerability

- An attacker with network access intentionally bypasses the official engineering pipeline to deploy malware to a service

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's coming through the standardized consumption tools (such as a package repository solution) established by your organization.

Practice 6: Enforce It

I can rely on secure and trusted OSS consumption within my organization.

Sample threat scenarios addressed by this job:

- A developer bypasses the official engineering pipeline to consume an OSS package with a known vulnerability

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. An example of enforcement is to reroute DNS traffic or configure builds to break if they try to consume OSS from untrusted sources.

Practice 7: Rebuild It

I can rebuild from source code every OSS artifact I'm deploying.

Sample threat scenarios addressed by this job:

- A team uses a malicious OSS package with a hidden backdoor (which could happen via traditional exploitation, political influence, blackmail or even threats of violence); as a result, the package's binaries do not match its source code.
- An attacker gains access to build infrastructure and modifies generated binaries during the build process; as a result, illicit changes can be injected in a manner that is essentially invisible to its original authors and users alike.

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 (e.g. [Reproducible Builds](#)).

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.

Sample threat scenarios addressed by this job:

- A team has taken a dependency on a package; the package is later discovered to have a critical vulnerability and the maintainer needs help/more time to fix the issue

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. Microsoft suggests a number of different ways to contribute:

- Financial support and participating in foundations or even individual projects: [GitHub Sponsors](#), [OpenCollective](#), etc.
- Bounty programs (such as [SOS Rewards](#)) and sharing best practices and tools with projects around security
- Being present and participating in key open source projects to share fixes or expertise
- See Microsoft’s approach toward contributing to open source for more ideas [Microsoft’s Open Source Program](#) | [Microsoft Open Source](#)

The OSS SSC Framework Implementation Guide

Target Audience





This section details a maturity model, which splits the practices in the previous section into 4 levels to achieve. There is also a list of tools your organization can implement to meet each security level in the framework. The guidance is useful to software developers, Continuous Integration and Continuous Development (CI/CD) administrators, and security practitioners.

OSS SSC Framework Levels of Maturity

When the OSS SSC Framework 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.

Level 1	Level 2	Level 3	Level 4
 <p>Minimum OSS Governance Program</p> <ul style="list-style-type: none"> • Use package managers • Local copy of artifact • Scan with known vulns • Scan for software licenses • Inventory OSS • Manual OSS updates 	 <p>Secure Consumption and Improved MTTR</p> <ul style="list-style-type: none"> • Deny list capability • Scan for end life • Have an incident response plan • Auto OSS updates • Alert on vulns at PR time • Audit that consumption is through the approved ingestion method • Validate integrity of OSS • Secure package source file configuration 	 <p>Malware Defense and Zero-Day Detection</p> <ul style="list-style-type: none"> • Clone OSS source • Scan for malware • Proactive security reviews • Enforce OSS provenance • Enforce consumption from curated feed 	 <p>Advanced Threat Defense</p> <ul style="list-style-type: none"> • Validate the SBOMs of OSS consumed • Rebuild OSS on trusted infrastructure • Digitally sign rebuilt OSS • Generate SBOM for rebuilt OSS • Digitally sign protected SBOMs • Implement fixes

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 – 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.

How to Assess Where Your Organization is in the Maturity Model?

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 OSS SSC Framework 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? (e.g. [packages.lock.json](#) for NuGet, [package-lock.json](#) for NPM, etc.)
 - 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 OSS SSC Framework 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 OSS SSC Framework categorizes its requirements into maturity levels to better help you prioritize investments in improvements. Additionally, the OSS SSC Framework 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.

OSS SSC Framework Requirements

Below is a table of the requirements mapped to the 8 different practices. Two of the requirements have prerequisites identified that are outside the scope of this document to list as requirements.

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	L2	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, such as colors v 1.4.1 , 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) <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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.

OSS SSC Framework Tooling Availability

Comprehensive Tooling available in v1.0 of the OSS SSC Framework:

The guidance and tooling in this document are a combination of paid and free tools from both Microsoft and across the industry.

Tooling available in future iterations of the OSS SSC Framework:

In the future, Microsoft plans on releasing more tools to help organizations secure their software supply chain end-to-end.

Implementing the OSS SSC Framework by Level

Below is a table of the OSS SSC Framework requirements with example tools from across the industry or detailed instructions to implement them, sorted by maturity level. Many of the tools referenced below are freely available and are listed as such. Some tools that are individually listed are available through a bundled offering, such as [GitHub Advanced Security](#) (GHAS). We aren't specifically endorsing any tool or service, as they each have different strengths or weaknesses. We recommend performing a thorough evaluation before deciding on a specific solution, including tools not referenced in this document.

This table maps each Framework requirement to corresponding level and Framework practice. To see the full list of requirements and their benefits, please see the *OSS SSC Framework Requirements* earlier in this document.

Practice name	L1	L2	L3	L4
Ingest it – save a local copy of artifacts and source code	<p>[ING-1] Use package managers trusted by your organization</p> <p>[ING-2] Saving a local copy of the OSS artifact can be done by adopting an integrated package caching solution into your CI/CD infrastructure. All developers across your organization should standardize their consumption methods (using governed workflows) so that security policy can be enforced.</p> <p>Free Tools: VCPKG for C/C++ OSS, Pulp</p> <p>Paid Tools: Artifacts, GitHub Packages, Azure Container Registry, PackageCloud</p>	<p>[ING-3] Having a Deny List capability to block ingestion of vulnerable and malicious OSS components is a required defensive tool in incident response situations. Having an incident response team that can rapidly respond and update the deny list is also critical.</p> <p>Paid Tool: Nexus Firewall</p>	<p>[ING-4] Saving a local copy of the OSS source code</p> <p>Free Tool: Duplicating a repo</p>	
Scan It - for vulnerabilities and malware	<p>[SCA-1] It is required to scan for known vulnerabilities of your dependencies. Choosing a tool that gets vulnerabilities from more</p>	<p>[SCA-3] Scanning OSS to determine if it is end of life is crucial to ensure that you are not taking dependencies on OSS that is no longer updated.</p>	<p>[SCA-4] Given the rise in malicious OSS packages over the years, it is critical that OSS be scanned for malware prior to consumption.</p>	

	<p>places than just CVEs is important to ensure that you are being informed from across multiple vulnerability sources.</p> <p>Free Tool: GitHub Dependency Graph</p> <p>Paid Tool: Snyk Open Source, Mend SCA</p> <p>[SCA-2] In addition to scanning for vulnerabilities, OSS should be scanned for software licenses.</p> <p>Free Tool: ScanCode</p>	<p>Free Tool: OpenSSF Scorecard</p>	<p>Free Tool: Mend Supply Chain Defender, OpenSSF Package Analysis</p> <p>Paid Tool: Nexus Firewall, Checkmarx SCA</p> <p>[SCA-5] Without doing proactive security analysis to look for zero-day vulnerabilities, there would be entire threat categories that would go unmitigated, such as back-doors.</p> <p>Free Tools: OSSGadget, DevSkim, Attack Surface Analyzer, Application Inspector, CodeQL, OneFuzz, RESTler</p> <p>Paid Tool: Semgrep</p>	
Inventory It - OSS usage and deployment	<p>[INV-1] Establishing an inventory of all developer OSS dependencies is critical when responding to an incident as an ingested malicious component would need to be deleted from the developer's desktop, the package caching solution,</p>	<p>[INV-2] Have an incident response plan that leverages your inventory and your deny list.</p> <p>Free Tool: Incident Response Reference Guide</p>		

	<p>and the software/service that in production that consumed the package. Knowing which projects are using which OSS components and their versions across your enterprise is vital toward supporting rapid Incident Response.</p> <p>Free Tool: Component Detection, SBOM Generator for 1st party code, Syft, Tern, SCA tooling</p> <p>Paid Tool: Dependency Graph w/ Insights via GHAS</p>			
Update It	<p>[UPD-1] Update vulnerable OSS manually.</p>	<p>[UPD-2] Automating patching OSS dependencies to address known vulnerabilities in a timely manner.</p> <p>Free Tool: Dependabot, Renovate</p> <p>[UPD-3] Display OSS vulnerabilities as comments in Pull Requests.</p>		

		Paid Tool: Dependency Review via GHAS		
Audit It - provenance and consumption workflows		<p>[AUD-2] Audit that developers are consuming OSS through the approved ingestion method. You can search for binaries that are checked into the repo.</p> <p>Free Guide: Searching Code</p> <p>[AUD-3] Validate integrity of the OSS that you consume into your build.</p> <p>Free Tool: NuGet CLI verify command</p>	<p>[AUD-1] Verify the provenance of all OSS components to ensure they come through the official supply chain.</p> <p>Paid Tool: Nexus Firewall</p>	<p>[AUD-4] Validate the SBOMs of OSS that you consume into your build.</p> <p>Free Tool: Community Attestation Service</p>
Enforce It - OSS consumption meets security policy		<p>[ENF-1] Securing the configuration of how build pipelines consume OSS components.</p> <p>Free Tools: NuGet Package Source Mapping, Version pinning and Lock Files</p>	<p>[ENF-2] Enforcing teams to only consume packages from a curated feed is the goal of this OSS SSC Framework.</p> <p>Paid Tool: Nexus Firewall</p>	
Rebuild It - from source				<p>[REB-1] Rebuilding from source in a trusted build environment removes the risk of consuming a package that may have</p>

				<p>been victim to a CCleaner/SolarWinds style build-time attack.</p> <p>Free Tools: Oryx, DotNet.ReproducibleBuilds, Reproducible-Builds.org, OSS Reproducible, rebuilderd</p> <p>[REB-2] Digitally sign the OSS you rebuild.</p> <p>Tool: Notary, SigStore</p> <p>[REB-3] If you are rebuilding the OSS yourself, you can automate Software Bill of Material (SBOM) generation at build time. This helps capture the supply chain information for each package to enable you to better maintain auditability and blast radius assessments.</p> <p>Free Tool: SBOM Generator on rebuilt 3rd party code</p> <p>[REB-4] Digitally sign the SBOMs you produce.</p>
--	--	--	--	---

				Free Tool: Notary
Fix It + Upstream				<p>[FIX-1] In extreme cases, when a newly discovered vulnerability is so severe and you cannot wait for an upstream maintainer to implement a fix, you should 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.</p> <p>Free Tool: Follow confidential disclosure guidelines</p>

Conclusion

The goal of this paper is to provide a *simple* framework for the pragmatic inclusion of secure OSS consumption practices in the software development process. It outlines a series of discrete, non-proprietary security development activities that when joined with effective process automation and maturation levels represent the steps necessary for an organization to objectively claim compliance with the Microsoft OSS SSC Framework as defined by the requirements identified in Level 3 of the OSS SSC Framework Maturity Model.

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.

Appendix: Mapping OSS SSC Framework Requirements to Other Specifications

There are many other security frameworks, guides, and controls. This section maps the OSS SSC Framework requirements to other relevant specifications including NIST SP 800-161, NIST SP 800-218, CIS Software Supply Chain Security Guide, OWASP Software Component Verification Standard, SLSA, and the CNCF Software Supply Chain Best Practices.

Requirement ID	Requirement Title	References
ING-1	Use package managers trusted by your organization	CIS SSC SG: 3.1.5 OWASP SCVS: 1.2 CNCF SSC: Define and prioritize trusted package managers and repositories
ING-2	Use an OSS binary repository manager solution	OWASP SCVS: 4.1 CNCF SSC: Define and prioritize trusted package managers and repositories
ING-3	Have a Deny List capability to block known malicious OSS from being consumed	
ING-4	Mirror a copy of all OSS source code to an internal location	CNCF SSC: Build libraries based upon source code
SCA-1	Scan OSS for known vulnerabilities	SP800218: RV.1.1 SP800161: SA-10, SR-3, SR-4 CIS SSC SG: 1.5.5, 3.2.2 OWASP SCVS: 5.4 CNCF SSC: Verify third party artefacts and open source libraries, Scan software for vulnerabilities, Run software composition analysis on ingested software
SCA-2	Scan OSS for licenses	CIS SSC SG: 1.5.6, 3.2.3 OWASP SCVS: 5.12 CNCF SSC: Scan software for license implications
SCA-3	Scan OSS to determine if its end-of-life	SP800218: PW.4.1 SP800161: SA-4, SA-5, SA-8(3), SA-10(6), SR-3, SR-4 OWASP SCVS: 5.8
SCA-4	Scan OSS for malware	
SCA-5	Perform proactive security review of OSS	SP800218: PW.4.4 SP800161: SA-4, SA-8, SA-9, SA-9(3), SR-3, SR-4, SR-4(3), SR-4(4) OWASP SCVS: 5.2, 5.3,
INV-1	Maintain an automated inventory of all OSS used in development	OWASP SCVS: 1.1, 1.3, 1.8, 5.11 CNCF SSC: Track dependencies between open source components

INV-2	Have an OSS Incident Response Plan	SP800218: RV.2.2 SP800161: SA-5, SA-8, SA-10, SA-11, SA-15(7)
UPD-1	Update vulnerable OSS manually	
UPD-2	Enable automated OSS updates	
UPD-3	Display OSS vulnerabilities as comments in Pull Requests (PRs)	
AUD-1	Verify the provenance of your OSS	CIS SSC SG: 3.2.4 OWASP SCVS: 1.10, 6.1 SLSA: Provenance – Dependencies complete
AUD-2	Audit that developers are consuming OSS through the approved ingestion method	CIS SSC SG: 4.3.3
AUD-3	Validate integrity of the OSS that you consume into your build	CIS SSC SG: 2.4.3 OWASP SCVS: 4.12 CNCF SSC: Verify third party artefacts and open source libraries
AUD-4	Validate SBOMs of OSS that you consume into your build	CNCF SSC: Require SBOM from third party supplier
ENF-1	Securely configure your package source files (i.e. nuget.config, .npmrc, pip.conf, pom.xml, etc.)	SP800218: PO.5.2 CIS SSC SG: 2.4.2, 3.1.7, 4.3.4, 4.4.2
ENF-2	Enforce usage of a curated OSS feed that enhances the trust of your OSS	SP800218: PO.5.2 CIS SSC SG: 2.4.3, 3.1.1, 3.1.3
REB-1	Rebuild the OSS in a trusted build environment, or validate that it is reproducibly built	CIS SSC SG: 2.4.4 SLSA: Build - Reproducible
REB-2	Digitally sign the OSS you rebuild	SP800218: PS.2.1

REB-3	Generate SBOMs for OSS that you rebuild	SP800218: PS.3.2 SP800161: SA-8, SR-3, SR-4 CIS SSC SG: 2.4.5 OWASP SCVS: 1.4, 1.7 CNCF SSC: Generate an immutable SBOM of the code
REB-4	Digitally sign the SBOMs you produce	CIS SSC SG : 2.4.6
FIX-1	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	

Appendix: References

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

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