SOFTWARE SUPPLY CHAIN BEST PRACTICES V2

Authors

Original White Paper authors
Marina Moore
Michael Lieberman
John Kjell
James Carnegie
Ben Cotton
Aditya Sirish A Yelgundhalli

Reviewers

Aonan Guan
Justin Cappos

Published

xxxxxxxxx

CLOUD NATIVE COMPUTING FOUNDATION

SCOPE

This new publication modernizes the original Supply Chain Best Practices whitepaper by highlighting changes and improvements to supply chain security tools and practices since the time of its writing. It also adds discussion of personas, and how different groups of people should approach this whitepaper to get relevant guidance. As with the original version, this paper provides the community with a series of recommended practices, tooling options, and design considerations that can reduce the likelihood and overall impact of a supply chain attack. It provides a holistic, end-to-end guide for organizations and teams to build a resilient and verifiable supply chain.

Some specific updates to the original whitepaper include:

- The addition of personas and guidance around how different audiences can approach this paper.
- Audit data handling best practices, which are clear, simple audits for third parties to run. These ensure third parties are securing data, limiting access, and monitoring for data security events as required.
- A discussion of Vulnerability Exploitability eXchange (VEX).
- A look at how to provide cryptographic verification to ensure correct actions were performed (e.g., ensuring tests were actually run).
- Updates to examples of attacks and tools to reflect recent changes.

INTRODUCTION

Software supply chain attacks cost over $45 billion in 2023, with projections exceeding more than $80 billion by 2026.[1] According to the Verizon DBIR 2024,[2] supply chain attacks grew 68% year-over-year. Aggregated risk from software supply chain compromises continues to grow as the relative ease of exploitation and exponential network effects of compromises have been demonstrated, encouraging further attacks. This growth is evidenced by what multiple software supply chain vendors are witnessing in the field, with some claiming growth rates as high as 200% recently.

Software supply chain attacks occur when the materials or processes of producing software are themselves compromised, resulting in vulnerabilities targeting downstream consumers of the produced software. Although the number of known, successful exploits remains comparatively small, the impact of these attacks has been extensive, as evidenced by incidents such as Log4j, NotPetya, and SolarWinds. The CNCF Technical Advisory Group (TAG) for Security maintains a detailed catalog of known supply chain attacks to raise awareness of the increasing frequency of such occurrences coupled with lower barriers to success.

This paper presents a holistic view of the software supply chain, providing information for both newcomers and experienced professionals in the field.

Non-goals

This paper does not detail tool-specific configurations, but it refers to existing documentation where it is reasonable and appropriate to ensure the reader has the most recent information. This paper does not provide instructions on assessing existing supply chains for security risk. This paper is not an end-all solution for resolving or preventing supply chain attacks. The components of this paper should be evaluated by the reader for usability and applicability in relation to their organization's risk tolerance and environmental requirements.

Projects and Products

Where possible the authors strive to provide the readers with CNCF open source projects as examples. However, it is recognized that some of these projects may not yet be productized or available "off-the-shelf." As a result, this paper may also refer to other projects, including both open source and commercial products. All such references, including references to existing documentation, are given as examples to the reader in an effort to provide real-world relatability. Readers should understand that these references are not an endorsement of any project or product by the CNCF or the paper's authors and reviewers.

TABLE OF CONTENTS

PART 1: HOW TO READ THIS PAPER **AUDIENCE/PERSONA**

At a high level, software supply chain security has two main perspectives: software producers and software consumers. Software producers are looking to secure the software they produce and distribute. Software consumers are looking to ensure that they only ingest and operate software that is safe and trustworthy. In addition, there are distributors focused on securely storing trustworthy software and distributing it to consumers.

Software producers are focused on developing software for any number of reasons, but want to deliver software with features that satisfy the needs of their consumers. Most producers care about security, but depending on their specific role within a project, team, or organization, it might not be a major part of their work. For example, open source maintainers, software developers working at end-user companies, or developers working at software vendors fall under this category.

Software distributors are focused on storing software artifacts along with metadata from producers and distributing them to consumers. Distributors often care about ensuring that they only store and distribute trustworthy software that meets a security standard. Open source distributors like PyPI and Maven Central are examples. Vendors can also distribute their software to customers or through internal company software repositories.

Software consumers are focused on ingesting and running software. This software can come from another project in the same organization, an external vendor, or it could be software from the open source community. Software consumers want to ensure the software they ingest and run is safe to use and is from trustworthy sources. They also want to be able to ensure this safety in a cost-effective manner. Most consumers don't want to run complex and expensive security scans against every piece of software they ingest, except in very high-risk scenarios. They want to be able to ingest metadata coming from trusted sources and verify as needed.

There are people, teams, and organizations that might do all three, but it is easy for us to break up the practices into the three categories of *software producer*, *software storage* and *distributor*, and *software consumer*. Even though someone may primarily fill the role of software producer, they will inevitably also act as a software consumer when pulling in dependencies when developing and building their software.

Below is a list of other personas that fall into one or more of the categories but it is worthwhile to call out some of their specific use cases. For each persona there is discussion as to where to start with software supply chain security. More details about the suggested best practices can be found in part 3 of this document.

Developer

Developers are not only responsible for adding new features and fixing bugs, but also managing application dependencies, version control system configuration (VCS), and often also Continuous Integration and Delivery (CI/CD) configurations. Moreover, much of their work happens locally[3], so managing the security of workstations has never been more important.

Fortunately, more and more developer work happens in source control. This is great news for security, consistency, recoverability, auditability, and collaboration, but it also means at the forefront of supply

chain security. The changes developers make (or fail to make) have a direct impact on an organization's supply chain security, and in many cases, also the supply chain security of downstream consumers, their consumers, and so on.

Where do I start?

One of the most impactful actions developers can take to improve and maintain the security of their supply chain is to ensure that all dependencies are kept up to date. This means accepting those automated PRs, running npm audit (or equivalent) regularly, updating container base images and keeping workstations up to date with the latest patches. Even this isn't an easy task, especially when it comes to keeping dependencies up to date on projects that are not currently being actively developed.

Then what?

The supply chain is only as secure as its weakest link, so even if all dependencies are kept up to date, a simple misconfiguration, leaked access token or unauthorized change to an upstream project can cause the whole chain to fall apart. As with all security, the best strategy for the supply chain is defense in depth (often referred to as "layers on the onion"). For developers, this means also paying attention to the other areas:

Choosing New Libraries and Base Images

- **Minimize Dependencies:** Where possible, minimize the number of dependencies so that each can be more easily audited. Use dependencies from the same organization or vendor to minimize sources of trust.
- **Publisher Trust:** Investigate whether the publishers of the libraries or base images are reputable,trustworthy, and whether their dependencies are also trustworthy. Minimize the number of publishers to simplify the work of investigating publishers.
- **Regular Updates and Patches:** Check if the publisher regularly updates their products and quickly releases patches for known vulnerabilities, including those from their dependencies.
- **Best Practices in Supply Chain Security:** Assess whether the publisher follows security best practices, including distributing signed attestations and securely managing dependencies.
- **Secure Supply Chain Consumption Framework (S2C2F):** Follow best practices for ingesting open source software, including how to handle upstream projects that might not adhere to supply chain security best practices.
- **Dependency Confusion of Package Managers:** Be vigilant about package names to avoid importing malicious packages that mimic legitimate ones.
- **Typo-squatting:** Double-check spellings in package names and URLs to avoid downloading counterfeit libraries that could contain malicious code.
- **Recursive dependencies:** Ensure that all recursive dependencies are audited by either you or the vendor. If possible, minimize recursive dependencies and perform more scrutiny on each one.

Protecting Source Code (see Producer)

- **Signing Commits/Pushes:** Use cryptographic signing of commits and pushes to verify author identity

and prevent unauthorized changes.

- **Git Security:** Implement strong access controls, use two-factor authentication for repository access, and regularly review access logs.
- **Workstation Security:** Ensure your workstation is protected with the latest antivirus software, firewalls, and is regularly updated to prevent vulnerabilities.
- **Secure Coding Practices:** Adopt and promote secure coding practices within your team to minimize vulnerabilities in the codebase.

Protecting the Build

- **Audit:** Regularly review and audit your build processes and automation tools to ensure they are free from vulnerabilities.
- **Scanning:** Integrate vulnerability scanning in your CI/CD pipeline to detect and address security issues before build and deployment.
- **Testing:** Ensure that tests are automatically run before deployment.
- **Shared Workflows:** Ensure use of shared CI/CD workflows and that they are secure and only include necessary permissions and access.
- **Isolation:** Utilize environment isolation techniques such as containerization to limit the impact of potential compromises, and to isolate one build from another.
- **Minimal build environment:** Keep the environments running the build minimal in order to minimize attack surface.
- **Supply Chain Levels for Software Artifacts (SLSA):** Aim to achieve higher levels of SLSA to ensure your build processes are comprehensively secure.
- **Reproducible builds:** Ensure that binaries are consistently generated from their source code to make tampering evident.

Supporting Downstream

- **VEX (Vulnerability Exploitability eXchange):** Provide clear and actionable information about the exploitability of vulnerabilities in components you produce.
- **Scoring System:** Provide a data-driven estimation, such as Exploit Prediction Scoring System (EPSS), for the likelihood a vulnerability in one or more components will be exploited in the wild.
- **Signed Attestations:** Use signed attestations to provide verifiable evidence of the integrity and origin of your software artifacts.
    - **Software Bill of Materials (SBOM):** Generate and distribute SBOMs to provide transparency about the versions and components in your software.
    - **Provenance:** Ensure that all software artifacts include information about their origin and the process used to build them.

Producer's Security Team

In the quest to secure the software supply chain, the synergy between developers, Chief Information Security Officers (CISOs), and operations teams emerges as a cornerstone. This collaboration is vital to preemptively identify vulnerabilities and swiftly respond to incidents. Through this collaboration, organizations can develop a culture of security awareness that can transform developers from mere code creators to responsible actors in the software supply chain (SSC). To do so, open communication must be established between the security team and developers, encouraging a proactive sharing of information on emerging threats. To aid in these endeavors, a selection of tools and processes must be established. In particular, the producer should:

- Have secure coding best practices in place to ensure integrity and consistency across the code base.
- Monitor dependencies used by their software, and respond to any incidents or vulnerabilities in these dependencies (see SBOM, VEX and metrics).
- Provide a template or stylesheet compatible with developers' preferred IDEs, containing the best coding guidelines.
- Use a vulnerability risk assessment solution to scan the code, packages, and third party dependencies. This assessment should cross-reference the CVSS database with databases such as Nessus to identify vulnerabilities and to discover the latest packages or implement fixes.

Compliance-focused, big-picture (CISO)

The Chief Information Security Officer (CISO) plays a pivotal role in managing organizational risk. Their duties include achieving and upholding compliance with regulatory standards, as well as providing evidence to auditors and the board. They are tasked with establishing and implementing policies and procedures derived from industry best practices, which must be adhered to by both their team and the management personnel across the organization. Although typically not engaged in daily cybersecurity or vulnerability issues, they must maintain a comprehensive understanding of the company's threat landscape. This includes determining the organization's risk tolerance, necessitating an assessment of current threats, estimated costs of mitigation, and the Mean Time to Remediate (MTTR). The ultimate objective is to safeguard the organization against large-scale data breaches and incidents that could severely impact the business and damage its reputation. Ignoring these risks is not an option, and failure to comply or inadequately handling threats may result in legal action.

Ops and Platform Teams Building Supply Chain Infrastructure

Ops/platform teams are responsible for selecting and configuring the tools that manage the supply chain. To tackle this critical step, a focus on simplicity and observability is essential.

Achieving Observability

Having a complete view and understanding the current state of the supply chain is crucial. Employing observability tools that aggregate data across various points of the supply chain enables real-time visibility into potential vulnerabilities. This awareness enables teams to proactively address security concerns, adapt to threats while continuously refining and improving security strategies.

Policy enforcement

The creation of explicit security policies sets the stage for secure supply chain practices; however, the real challenge lies in consistently enforcing these policies. Leveraging policy-as-code solutions allows for the automation of these enforcements, ensuring that every step and element of the supply chain adheres to established security standards. To strengthen policy enforcement, continuous validation ensures that security policies are not only adhered to at the outset, but also continuously improved as the supply chain evolves.

Security Team of Consumer (Verifier)

End-users of software must establish trust in the software they consume. In order to safely consume software, verifying the security, integrity, and authenticity of the product is essential to safeguard against supply chain vulnerabilities.

The first step in secure consumption for a verifier is ensuring that they are requesting the correct packages. Software repositories for open source software, such as npm and PyPI, allow any developer to upload packages. While this decreases the barrier to entry for developers, it means that not all software on these repositories will meet an organization's security standards, and attackers may upload malicious packages with techniques like typosquatting or dependency confusion. Ideally, the consumer will audit all consumed software, creating an allowlist of packages they consume. Even if an organization does not have the resources to audit every package, they can ensure that the same packages are requested on every download by creating a lock file with dependency digests or similar, reducing the opportunities for typos. Once packages have been selected, the next step is to verify their authenticity.

Verifying Software Integrity and Authenticity

Consumers should verify the source and integrity of software prior to utilization. Signature and checksum validation were once standard methods of doing so, but are no longer enough to ensure provenance and integrity. Attestations about software artifacts provide additional information about what the signer is attesting to and how the software was built. As such, consumers should verify these against their own supply chain policy.

Specifically, consumers should consider:

- Requesting and verifying attestations from the producers of software they consume.
- Evaluating provenance of that software.
- Understanding their dependencies through the use of SBOMs.

In some cases, upstream producers may not follow best practices, and may not provide attestations. When this happens, consumers can:

- Mitigate this risk by either auditing software that is consumed or building the software in-house.
- Engage with upstream producers to help improve their supply chain practices
- Perform Software Composition Analysis (SCA) on consumed packages.

End User Concerns and Use Cases

End users come in various forms: from enterprises that use open source software without releasing their software externally to non-technical users downloading open source software for personal use. This document focuses on the safe consumption of software without consideration of its intended use. For use cases where software is being used to develop more software look, refer to the Developer persona.

Where do I start?

For the end user consumer it might seem like security is not something you should have to deal with. You may think that it's the problem of those providing the software. This is mostly true when you buy software from vendors. There are often contracts or agreements in place that if a vulnerability comes up in vendor-provided software it's the responsibility of the vendor to report to end users and fix that vulnerability. This isn't true for open source software. A piece of open source software might be backed by a company or large non-profit organization like the CNCF, or it might be a project maintained by just one individual. In the latter case it's impossible to rely on an individual with which there's no agreement on your security.

As an end user, whether or not you're using software as part of a business or just for personal use, you need to start by figuring out how comfortable you are with different levels of supply chain risk. This is often called a risk appetite. If you are using open source software in an isolated environment - not touching any sensitive data - you might be a bit more open to using software that isn't following most security best practices. If you are using open source software to do something that deals with sensitive information you would want the software to be demonstrably more secure.

Once you have an understanding of what are the important things to focus time and money on in protecting you need to threat model.

Then what?

As an end user, once you have an understanding of what you're looking to protect and how much time and effort you're willing to spend on it, you need to get started on actually protecting yourself from potential security issues. Most end users are not security experts and most businesses' core competencies are not related to cybersecurity. The most important thing as an end user you can do is be vigilant. Download software from trustworthy places, update your software regularly, and follow other best practices.

This won't protect you from a lot of sophisticated software supply chain attacks, but it helps prevent the most common ones. Having more end user companies fund and contribute to the projects they use also helps here.

Protecting Ingestion (see consumer)

- **Publisher Trust:** Use software from trusted sources.
- **Verify Security:** Ingest security metadata like SBOMs, in-toto attestations like SLSA, etc. and verify that it fits your security needs.
- **Secure Supply Chain Consumption Framework (S2C2F):** Follow best practices around consuming open source software. The more technical your team is the more you can do on this front.

Protecting Runtime

- **Store Software Security Metadata:** Store SBOMs, in-toto attestations, etc. from your SDLC processes and your dependencies.

- **Have Supply Chain Observability:** Analyze supply chain metadata like SBOMs and in-toto attestations and correlate it with runtime behavior to better respond to supply chain incidents like vulnerabilities or attacks.
- **Have software update processes:** Understand what to do when there's a new version of software released or vulnerability discovered.

Protecting the Ecosystem

- **Fund Open Source:** Well-funded open source has a strong correlation with better security posture. Help fund security initiatives and critical open source projects.
- **Contribute to Open Source:** Contributing back to the open source projects you use is also a good way to help with supply chain security. This includes more than just code contributions, when you discover issues, report them.

PART 2: OVERVIEW
SECURING THE SOFTWARE SUPPLY CHAIN

A supply chain is "a process of getting a product to the customer." This sentence hides a lot of complexity. As we learned during the COVID-19 pandemic, the supply chains for manufactured goods are complex and can be disrupted by events anywhere along the way. A shortage of computer chips for cars can drive up the cost of new vehicles — and then used vehicles. Social distancing policies at loading docks results in dozens of ships waiting offshore to unload. An error in ship navigation could result in a blocked canal. The supply chain includes everything that goes into making a product through every upstream step.

Supply chains don't just exist for manufactured goods. Software has a supply chain, too. This includes the work that goes into writing, testing, and distributing the software. It also includes the same work for the applications and upstream libraries used to write, test, and distribute the software. The dependencies can go many layers deep, with developers not knowing what the software they build atop of contains. Software delivered to a consumer may contain malicious code or an active exploit prior to or upon installation and use; these vulnerabilities may be inserted or exploited at any stage in the development process, from early development to deployment.

When it comes to open source software, the traditional supply chain concept starts to look a little different from commercial software. Supply chains for commercial software are built on a vendor-buyer relationship defined by contracts with requirements for quality, functionality, support, and delivery times. Open source software doesn't require that two-way relationship. In fact, developers often don't know who is using the project until someone comes with a bug report or feature request. While commercial software vendors can charge extra if their customers need more stringent security requirements, open source projects have to find their own resources to meet those needs.

It's no surprise, then, that some open source maintainers push back on the idea of being part of a large company's supply chain, particularly when the open source project is a hobby or passion project. But

supply chain security is important for everyone, whether working on a single-developer project or a giant project like Kubernetes. No matter what software you're writing, you're using software to do it. Improvements to the security of the software ecosystem are beneficial for everyone. It's not just about producing more secure software, it's also about knowing what's in your own supply chain and who produced that software, so that you can respond quickly to vulnerabilities in upstream code.

In the rest of this document, we will provide an overview of software supply chain concepts before delving into best practices for each of its stages. All of these best practices are designed so that readers can start small, and build up until they have secured their entire software supply chain. Not all supply chains look alike, so much of this guidance leaves specific policy choices to the implementer to decide based on their threat model and risk assessment. A software supply chain is made up of many people and machines, and only by working together to secure each link in the chain can we make it more secure.

CONCEPTS

Before providing detailed security best practices for each of the steps in the software supply chain, some tools and concepts that are relevant throughout the software supply chain are discussed. These concepts come up in many of the stages of the supply chain, so they are described here for brevity.

Metadata

There are several types of metadata that can be associated with artifacts in order to improve software supply chain security. Discussed here are SBOMs, VEX, and attestations. All of the metadata produced during the software supply chain needs to be stored and distributed. Storage mechanisms include OCI and GitHub. These storage mechanisms allow software consumers to find and verify metadata associated with the artifacts they consume.

SBOM

A Software Bill of Materials (SBOM) is a detailed inventory of all components, libraries, and dependencies used in a software application and the environment in which it was built. SBOMs can help software consumers determine their transitive dependencies, and to determine whether they are impacted by a vulnerability. However, just having an SBOM has little value without tooling to interrogate, aggregate, and otherwise reason about it.

With the right tooling, SBOMs can help with:

- **Vulnerability Management:** Provides detailed visibility into software components, helping to identify and quickly remediate vulnerabilities
- **License Management:** Helps with the identification and management of software licenses in line with organizational policy to avoid legal issues.
- **Incident Response:** Facilitates quick identification and remediation of issues by providing a clear inventory of components.
- **Transparency:** Enhances understanding of the software supply chain, making it easier to track and

manage dependencies.

- **Compliance:** Helps meet regulatory requirements by documenting all software that comprises the application or system
- **Keeping the Ecosystem Agnostic:** Standard formats for SBOMs (such as CycloneDX and SPDX) allow for consistent communication across different tools and platforms.

SBOMs are often generated by scanning package manager indexes post-build (for example Syft and Trivy), but this approach can be error prone, not least because components are often built from source, so the identity of the package is harder to discover. The software build process is the only time when there's an intersection of the source code, dependencies, and produced software. This makes build tools the ideal producer of accurate and attested SBOMs. SBOMs are generated by the software producer or vendor, and represent the state of the released software as understood by the producer. As such, SBOMs may not contain information about all dependencies, and do not include vulnerability information about the package itself.

VEX

The generation of Vulnerability Exploitability eXchange (VEX) documents for software artifacts is a proactive measure to communicate the exploitability of vulnerabilities within specific environments. During this process, it is possible to include a parameter describing the state of given vulnerabilities present in the report (AFFECTED, NOT AFFECTED, UNDER INVESTIGATION, FIXED). For example, there might be false positives or trails of unused libraries. Those need to be listed inside the report, and specifying the status could help evaluating the actual security impact. VEX documents are typically in JSON or XML format. A parser or dedicated VEX processing tool can be used to ingest and analyze the data. The information within a VEX document can be used to assess the risk a vulnerability poses to the product and identify possible mitigation strategies. For example, an SBOM generated with CycloneDX can include a section dedicated to vulnerabilities within the listed components. This section can reference a separate VEX document or embed vulnerability details directly in the SBOM. Python tools like "vexy" can analyze a CycloneDX SBOM and generate a corresponding VEX document. This VEX file would then detail the exploitability of the identified vulnerabilities within the context of the specific product that uses those components. VEX is not yet widely adopted, however it is a useful tool to consider when producing SBOMs and consuming artifacts.

Attestations

Attestations are signed records of actions that occurred in the software supply chain. They enable policy evaluation by providing verifiable information about what occurred in the software supply chain. With these claims, a verifier can ensure that the steps described in policy were performed by the actors described in policy. in-toto, an incubating project of the CNCF, is a popular specification for software supply chain attestations. Attestations have a lifecycle that includes:

- **Creation:** Attestations are created during the execution of the software supply chain as signed statements by the actors performing steps. For example, the build system could create an attestation that includes the inputs and outputs of the build, signed by the workload identity of the build machine. There are a variety of tools for creating in-toto attestations, including Witness, Tekton and Syft.

- **Distribution:** Attestations must be distributed to verifiers and be associated with the particular artifact or metadata. Tools like TUF and Archivista can be used for such distribution (see Deployment and Distribution for details about secure distribution).
- **Verification:** Attestations must be verified against the supply chain policy. Verifiers can check for inconsistencies between the attestations of what occurred in the software supply chain, and what the policy requires from it. Such verification ensures that the supply chain policy is adhered to.
- **Storage:** Attestations should be stored after verification for future use. If a vulnerability is discovered later, attestations can be used as an audit trail to determine where in the supply chain the vulnerability was introduced. Storage mechanisms include Archivista, OCI, and GitHub.

Policy

Software supply chain policy describes requirements for the software supply chain, defined in software, including authorized actors and expected actions. This policy can then be evaluated against attestations as part of the verification process. Policy will evolve with an organization's supply chain security posture, first capturing the current process, then implementing stricter security requirements as the organization's threat model and risk assessment evolve. More detail about software supply chain policy, including its evaluation and enforcement, is described in a blog post by TAG Security.

Testing

Properly testing and verifying software is a crucial aspect of ensuring not only its functionality, but also its security. Testing should be considered and attested to at each step of the delivery lifecycle. Test results should be captured as signed, verifiable attestations and which are stored in a metadata store so that it can be evaluated against supply chain policy. Software supply chain policy should list any expected tests to ensure they are run before deployment.

Zero Trust

Software supply chain security heavily relies on an understanding of which actors are performing which actions in the software delivery lifecycle. Metadata like SBOMs and attestations help shed light on what is happening in the software supply chain, but the data are only as trustworthy as the identities that created them. Zero trust can be used for continuous verification of identities performing actions to determine whether these identities were authorized to perform those actions. For example, a system could verify which people wrote code, which systems built the software, and which systems ran deployments. The verified identities may be a public identity like the Linux Foundation or a pseudonymous identity. A pseudonymous identity can be associated with a cryptographic key or other consistent identifier that indicates the same actor is performing an action over time. Such consistency can be used to establish trust even without verifying a human identity.

Root of Trust

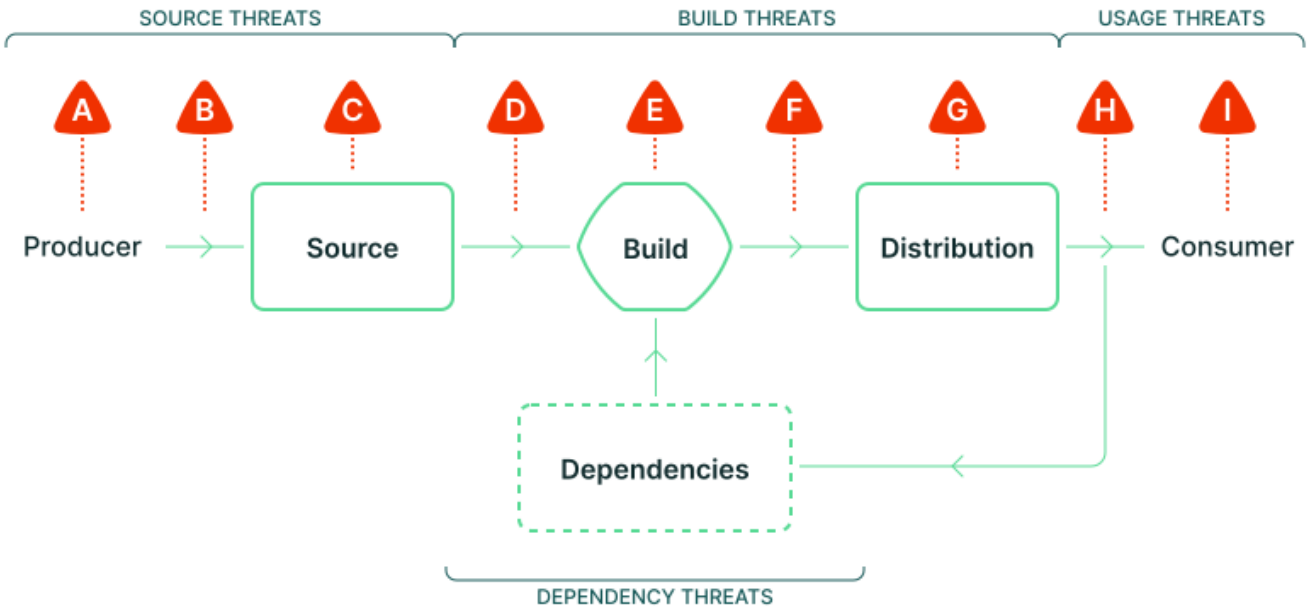
A root of trust is an entity that provides the initial trust anchor for a system. This trust anchor can then have a chain of trust that can be used to determine other trusted parties, such as developer signing keys. The root of trust should be hardened (for example, using offline keys, and requiring a threshold of keys), as it is used to establish trust in other keys or devices. For example, the Sigstore project uses a TUF root

of trust, which requires a threshold of offline keys to make changes to the root of trust, then this root chains trust to keys used by the Sigstore service. Clients embed the TUF root, and can use it to securely determine the currently trusted keys for Sigstore. In this way, clients always can receive updates to keys used by Sigstore, and in fact always have the most recent keys. The level of hardening of a root of trust is dependent on the organization's threat model and risk assessment. For example, a hobby project may use online keys to improve automation of the root of trust, while a certificate authority may require a threshold of offline keys for improved compromise resilience. Each organization should perform a risk assessment to determine the scope and severity of a root of trust breach, and harden it accordingly.

PART 3: BEST PRACTICES REFERENCE **SOFTWARE SUPPLY CHAIN OVERVIEW**

The software supply chain consists of all of the steps taken to move code from source to production. While every supply chain is unique, there are some general steps that are used by most software supply chains. Each of these is described in more detail below, along with specific security risks and recommendations for each stage of the software supply chain. The stages are:

- Source code
- Materials
- Build Pipelines
- Artifacts
- Deployments and Distribution



A Producer (entity)
B Authoring & reviewing
C Source code management
D External build parameters
E Build process
F Artifact publication
G Distribution channel
H Package selection
I Usage

To illustrate how these stages fit together, consider the above diagram from the SLSA project. This illustration of the supply chain threats highlights all the areas where something can go wrong. This can be either a malicious attack, or it could be benign but still causing vulnerabilities.

- **Producer** - Developer can be malicious or untrained
- **Authoring & Reviewing** - Mistakes can be made, reviewers can be malicious or miss something
- **Source Code Management** - Incorrect security settings, and the source code systems can get compromised
- **External Build Parameters -** Malicious or insecure inputs to the build, like a bad compilation flag
- **Build Process -** Build can be compromised or the software can be built insecurely
- **Artifact Publication -** A wrong or malicious artifact can be published
- **Distribution Channel -** Artifact host or tooling can be compromised
- **Package Selection -** Downloading the wrong package due to typo or other reason

This paper condenses the threats described in this diagram into 5 sections. The steps from this diagram map to the stages in this paper as follows:

- **Source code:** Threats from A, B, and C in the diagram (the source threats) map to the source code section below.
- **Materials:** The dependencies map to the materials section.
- **Build Pipelines:** Steps D and E are described in the build pipelines section.
- **Artifacts:** Step F maps to the artifact section.
- **Deployments and Distribution:** Steps G, H, and I are described in the deployments and distribution section.

This illustration of the software supply chain simplifies some aspects. Each software supply chain has dependencies on other software, which has its own software supply chain. In this way the software supply chain is recursive, with each piece of software building on the software supply chain of its dependencies. While software consumers can and should validate the software supply chain of their immediate dependencies, this will not encompass the entire supply chain unless they also validate the dependencies of those dependencies, and so on.

SOURCE CODE

The foundational construct of any software supply chain is the source code. The initial step in securing a supply chain is establishing and ensuring the integrity of the source code. “Integrity” in this context means that the source code and tags found in the repository have not changed or been altered from when the code was created by the developer. This includes potential malicious changes introduced by a local compromise on the developers machine. To maintain integrity, organizations should take steps to verify the provenance of the code added to their first party products and libraries.

It is fundamental to the supply chain that the development activity employs software development best practices. Agile methodologies of “continuous improvement” have been embraced in the industry and enabled through CI/CD pipelines and automated testing. These pipelines must be properly configured to access source code on-demand in order to build, deploy, and release artifacts at the cadence the organization needs.

Identity and Access management (IAM), including role creation and local developer accounts, is the biggest attack vector, regardless of platform or vendor[4]. It is critical to carefully manage IAM policies, including requiring multi-factor authentication (MFA), to ensure they are not overly permissive, but still provide both developers and agents secure access to source code. Pipeline agents and human developers must have their access and privileges calibrated to their roles within the organization and be given secure means to authenticate to those roles.

While the recommendations here assume that source code is tracked in a Git repository, they apply to any modern version control system (VCS). The assumption is that there is a centrally managed canonical copy of the repository (for example, on github.com) that all developers contribute their changes to and pull new changes from. If the VCS is not Git, it’s also necessary to confirm what integrity guarantees the VCS provides for its tracking of files and changes, such as the hashing algorithm.

Verification

Declare protected repository namespaces

The repository responsible for storing source code and tracking changes must also include a policy that indicates which parts of the repository are protected and who can make changes to them. For example, repositories typically have a primary branch such as main. Changes to this branch and a handful of others should be approved by trusted maintainers. Another example of protected namespaces are in enterprise contexts that use a monorepo to manage multiple projects. As different teams may be responsible for different projects, this should be clearly defined in the repository’s policy.

Source control platforms such as GitHub and GitLab include mechanisms for specifying protected branches as well as protected files/folders. Open source tools like Peribolos (part of Prow) can be used to declare access control rules for GitHub. This tool is used by Kubernetes. Another option that also enables *verification* of policy enforcement is gittuf, an OpenSSF sandbox project.

Authenticate and monitor repository activity

The primary copy of the repository which serves as a synchronization point for all developers should limit who can push to it. This baseline access control protection is offered by all popular source control

platforms. However, in addition, each push should also be verified to ensure the developer is authorized to push to that repository namespace. This goes hand in hand with the use of protected namespaces mentioned above: while that sets expectations on the policy applicable to a protected namespace, here that policy is applied to verify the validity of a push to the repository.

There are several mechanisms to authenticate the developer pushing to the repository. Popular source control platforms like GitHub already do so, and enforce policies such as branch protection and code review requirements. Other mechanisms that are independently verifiable include Git's support for signed pushes as well as the "reference state log" maintained by gittuf for a Git repository.

Finally, all repository activity should be recorded in an audit log, and this audit log should be monitored. As before, source control platforms include audit logs on a repository and organization basis. These logs should be inspected to detect unauthorized changes to a repository or the policies themselves due to the compromise of a privileged developer account.

Enforce independent 2-party review

The author(s) of a request may not also be the approver of the request. At least two individuals, one of which should have write access to the branch and are independent of the request, should review and approve. Some organizations may find an engineering manager or a lead engineer, who is an informed security practitioner, to be sufficient for this role. Reviewers should have a similar level of knowledge as the author(s), implying equal or greater expertise to provide informed review.

Require verification attestations / confirmation

Verification of source code policies should be captured as attestations. This allows other systems such as a CI pipeline to verify that source code policies were enforced and met for a specific revision prior to performing a build. Ongoing work such as the SLSA source track is defining guidelines about how to generate attestations at different levels.

Automation

Prevent committing secrets to source code repository

Secrets such as credential files, SSH keys, access tokens, and API keys should not be committed to the source code repository unless encrypted prior to placement. Tooling exists to detect secret key leaks, such as git-secrets, detect-secrets, or trufflehog, which can be integrated using either a client-side hook (pre-commit), server-side hook (pre-receive or update), or as a step in the CI process.

Some DevOps platforms and integrations provide a default list of files which cannot be pushed to the repository. These lists can be extended to include more domain specific files which may contain sensitive materials.

Automate software security scanning and testing

Software best practices include the use of unit, functional, and end-to-end testing to identify bugs and errors in the code before release. In addition, security specific scans should be performed, including Static Application Security Tests (SAST) and Dynamic Application Security Tests (DAST). Static scan tooling should be integrated as early in the development process as possible, including integration into the IDE. For more details on application security best practices refer to the OWASP standards and tools.

During the build process the metadata from security tooling such as SAST tooling should be recorded and linked to a hash of the build artifact to provide provenance. Both the coverage and results of these tests should be published as part of the repository information to help downstream consumers of software better assess the stability, reliability, and/or suitability of a product or library, possibly as an in-toto attestation.

Controlled Environments

Establish and adhere to contribution policies

Modern VCS platforms allow repository administrators to define configuration options or rules to enforce security, hygiene and operational policies. These rules can help enforce policies during different stages of software development allowing standardization across different team members. For example, automated deletion of branches, policies for code reviews, role-based contribution and automated checks can be performed. It is recommended that repository administrators define a baseline of these rules. General contribution policies should also define what is and is not considered an acceptable contribution so that potential contributors are made aware in advance.

Define roles aligned to functional responsibilities

Define roles corresponding to the different actors interacting with source code repositories, and assign access controls based on their responsibilities while enforcing the principle of least privilege. Examples of roles include Developer, Maintainer, Owner, Reviewer, Approver, and Guest. Grant each role fine-grained permissions for repository access control.

Secure Authentication

Enforce MFA for accessing source code repositories

Multi-factor authentication (MFA) should be required at the VCS level for all software projects. MFA provides an additional layer of security, normally by requiring a soft or physical token in addition to traditional credentials, thus requiring two compromises in order for an attack to be successful.

Use SSH keys to provide developers access to upstream source code repositories

Developers contributing source code require a non-interactive way to access source code hosted repositories from their development tools. Instead of using passwords which are prone to common hacking techniques like brute force, password guessing and password spraying, SSH keys or SSH certificates should be used. Agents in CI/CD pipelines should also be configured to access repositories using SSH Keys. All modern platforms like GitHub, GitLab, and BitBucket provide guidance on configuring access using SSH keys, including rotation and revocation of these keys.

There are environments where SSH may not be a viable authentication mechanism due to network policies. One possible approach to authentication is frequently rotating access tokens, scoped to only the necessary functions. It is important to note that the problem with tokens is their distribution, exposing credentials to Adversary-in-the-Middle and other classes of attack. Tokens, in this context, should only be used if they are short lived and issued with an out-of-band identity management system such as SPIRE.

Use short-lived/ephemeral credentials for machine/service access

Modern VCS platforms allow the use of randomly generated short-lived tokens for the access management of machines and services such as CI/CD pipeline agents. Short-life credential issuance encourages the use of fine grained permissions and automation in provisioning access tokens. Short-lived signing keys, such as those used by Sigstore, do not require key rotation as new keys are used for each signing event. These permissions further restrict the type of operation that can be performed at the repository level.

Short-lived access tokens should be considered instead of password-based credentials so that tokens are continuously authenticated. These tokens shouldn't be confused with the long-lived and non-refreshable personal access tokens (PAT) such as those available on GitHub, as these PATs will have similar security properties as a shared secret or password type of credential.

MATERIALS

The overall quality and consistency of any "manufacturing" process largely depends on the quality and consistency of the goods used as inputs. Producers must take care to verify the quality of these materials. For software supply chains, the quality of the dependencies, direct or transitive[5] should be verified. Ensuring dependencies are retrieved from trusted sources and have not been tampered with is a key part of the software supply chain. Depending upon the required risk profile, it may be appropriate to identify trusted repositories and rely directly upon them, in addition to disabling access to all other repositories.

Second and Third-Party Risk Management

When making determinations on accepting, limiting, or denying the use of second- and third-party software, organizations should use risk management processes to understand the risk presented by these materials. Not all software an organization may intend to use has the same level of support behind it, which may reflect significant differences in the overall level of quality, security, or responsiveness to issues across dependencies. Particularly from the perspective of security, organizations should consider whether proposed dependencies have mechanisms for managing externally reported vulnerabilities.

Most foundation- or corporate-supported open source projects make it standard practice to report Common Vulnerabilities and Exposures (CVEs) and patch affected code. However, a CVE is a lagging signal. It requires a significant effort to establish and publish a CVE. A series of continual "operational health" metrics should be used to evaluate the true state of security for a project. The Open Source Security Foundation (OpenSSF) has produced the OpenSSF Scorecard project aimed at providing a rubric to evaluate the security posture and operational health of open source projects. In addition, projects should be evaluated to determine whether they adhere to the best practices documented in this paper.

The number of CVEs by itself is not a particularly meaningful metric as not all vulnerabilities have the same impact. Common Vulnerability Scoring System (CVSS) and Package Vulnerability Scores (PVS) are useful frameworks for assessing and managing vulnerabilities within the software supply chain. Vulnerability Exploitability eXchange (VEX) documents are an additional source of information. VEX documents give attestations from software providers about whether or not a package is affected by

a particular vulnerability. For example, a CVE in a software's source code may not be present in the shipped binaries because the vulnerability is conditioned on a compile-time option that wasn't used.

These specifications offer a standardized approach to evaluating software packages and their dependencies, facilitating clear and consistent decision making during evaluation. The integration of the frameworks into security checks and gates throughout the software supply chain helps reduce risk and improve compliance with regulatory requirements.

Additionally, organizations should always monitor changes to second and third party software and services. Software and service vendors should list any changes to their Service Level Agreement (SLA) for the service or changes to the software (for example, maintaining a changelog). It is incumbent on the consumer to review and understand these changes and to analyze how they will impact their supply chain.

Verification

Verify third party artifacts and open source libraries

Third party artifacts need to be verified using a range of approaches. All third party artifacts, open source libraries, and any other dependencies should be verified as part of continuous integration.Verification involves verifying their checksums against a known good source and checking any cryptographic signatures. Any software ingested should be scanned using Software Composition Analysis (SCA) tools to detect whether any vulnerable software is present in the final product. For even higher confidence, perform penetration and fuzz testing to ensure any third party artifacts are resistant to common attacks and no basic security flaws are present.

Require SBOMs and VEX statements from third-party suppliers

Where possible, software vendors should be required to provide SBOMs containing the explicit details of the software and versions used within the supplied product and VEX statements that indicate whether specific vulnerabilities affect the software. This provides a clear and direct link to the dependencies, removing doubt on the version of dependencies that are used and thus removing the requirement of using heuristics via SCA tools. Ideally, signed metadata from the build process should accompany the SBOM.

Track dependencies between open source components

A register of a project's open source components, dependencies, and vulnerabilities should always be maintained to help identify any deployed artifacts with new vulnerabilities. One of the most popular open source inventory implementations is OWASP Dependency-Track. Generating, receiving, or maintaining a supply chain inventory will help identify the software vendors, suppliers, and sources used in an organization, as well as the associated software and versions. Cataloging this inventory into a human and machine readable format allows organizations to correlate vulnerabilities as they are published against in-use or prospective software and to establish any impact.

Build libraries from source code

Software dependencies are often distributed in binary form such as tarballs, or stored in a package management repository. However, there is often no explicit connection between the binary in a

repository and the source code itself. This connection can be made with attestations (such as those used by npm) or through content-addressable build systems (such as Nix). When this connection is not made it is entirely possible that the compiled version uploaded to a package repository contains additional or different code when compared to the source code repository. This often happens, for example, when developers add debugging capabilities to their code. However, this lack of a one-to-one correlation between source code and binaries also opens up the possibility of malicious additions to the package manager's version of a dependency. Consequently, when possible, it is safer and more reliable to build the binaries yourself directly from the source code. This provides a clear link between the library source code and the compiled binary. However, because this process is time consuming and resource intensive, it may only be feasible within highly regulated environments.

Define trusted package managers, repositories, and libraries

If building from source is not a viable option for every component, components from a trusted supplier which regularly maintains and updates those binary components in a backwards compatible way should be used. They should be vetted by using metrics and other testing methods. The supplier should publicly document their processes, and have a security incident response process with well defined SLAs.

While there are public repositories and package managers that sign packages and provide the means to verify those signatures, highly regulated environments should minimize ingestion from public repositories, and try to rely on verification by build reproducibility. Less highly regulated environments should only ingest public packages if they can assess the risk level with techniques such as CVE scanning. If organizations choose to host their own package managers and artifact repositories, they should restrict build machines to pull from only those sources. These hosted package managers must be properly configured to ensure that they do not have malicious packages and are using best practices for software distribution. When organizations choose to instead utilize external repositories, they should ensure that packages are signed by the correct authors and limit the pages ingested from untrusted repositories (such as through the use of an allowlist or other validation).

Generate an immutable SBOM of the code

While rebuilding a software artifact, it is best practice to also generate its SBOM. An SBOM provides a machine readable inventory of the contents of the artifact. Consumers of the software will then be able to analyze the SBOM, correlate it with vulnerability data to directly identify vulnerabilities. There are currently two well known SBOM specifications: SPDX and CycloneDX.

Ideally, the build process should generate the SBOM. For third party software, an SBOM can be generated using software composition analysis. SBOM data generated by using the build-time data will be more accurate because the build process has visibility into the dependencies used to generate the software. An SBOM generated through scanning isn't likely to capture issues such as the toolchain vulnerabilities or manually installed software.

Automation

Scan software for vulnerabilities

Before software dependencies are brought into a system and periodically thereafter, they should be analyzed to ensure the vulnerabilities they include pose an acceptable risk to the organization. If the dependency poses an unacceptable risk, perform triage using a vulnerability management process to

decide whether other mitigations may be applied to reduce risk, or whether the software or update should be blocked.

Note that instances may exist where the specific usage of the software does not expose or use the specific code containing the vulnerability. For these occurrences, analysis mechanisms such VEX or CVSS may be provided to mitigate the vulnerability.

Managing software licenses

While not directly a security concern, licensing obligations are an important consideration when adding new dependencies. The Linux Foundation maintains the Open Compliance Program which provides several tools to ensure released software meets legal and regulatory requirements. Licensing metadata should be recorded during the build process and distributed within the artifact's SBOM.

Run software composition analysis on ingested software

At a minimum, a Software Composition Analysis (SCA) tool should be run against any dependencies used within the software being built. The SCA tool will use heuristics to identify the direct and transitive dependencies, and can also verify the contents of the SBOM. Additionally, utilizing the SARIF (Static Analysis Results Interchange Format) can enhance the process by supporting both SBOM integration and other static analysis capabilities. This data can then be correlated against data from a number of feeds containing vulnerability metadata to highlight any vulnerabilities in the dependent packages. There are several nuances with this approach that may lead to false positives. Errors may occur when libraries are mapped against vulnerability feeds due to the use of different or inaccurate library identifiers (e.g. namespace or version strings). Validation of the security of the open source components should occur before the build process and can complement a range of checks appropriate to organizations risk tolerance.

BUILD PIPELINES

The build pipeline "assembles" the software artifacts that will be made available to downstream consumers.

Securing the build pipelines is crucial to securing the software supply chain:

- **Build steps:** The function or task to be performed at any point in the "assembly line." A build step should have a single responsibility which may be, for example, to retrieve sources, compile an artifact, or run tests.
- **Build workers:** The machinery or infrastructure carrying out the task. Historically, a single server might have completed all the steps, but in a cloud native environment the build worker is typically a container which has a 1:1 correlation with a particular step.
- **Build tools:** Any software dependencies required to generate and ensure the integrity of the final artifact(s).
- **Pipeline orchestrator:** The overall build pipeline managing the CI/CD workflow. Deploys build steps and workers to complete the stages of that pipeline.

Additionally, build metadata produced by these components should be signed and recorded externally in order to facilitate out-of-band verification.

The pipeline is created by joining together a series of hardened build steps. Each build step should be verified as trusted: either by implementing each step through a hardened container image stored within a secured repository and deploying to a hardened orchestration platform such as Kubernetes, or by having attestations for the ingested materials and operations that stem from a trusted root. As an example, the US Air Force's Platform One is an implementation of this concept, which leverages hardened containers from the Platform One centralized artifact repository called "Iron Bank." This repository contains signed container images hardened according to the DoD Container Hardening Guide.

By building the pipeline from hardened components, the likelihood of successful compromise of any build step is reduced. The pipeline orchestrator controls exactly what each stage is able to perform, implementing the required build step without additional software and reducing the attack surface of each component in the pipeline. The entire pipeline should be designed such that the compromise of an individual build step (or even multiple build steps) is contained and does not lead to compromise of the entire pipeline. Out-of-band verification provides defense in depth and mitigates against anything short of loss of the infrastructure's administrator credentials.

Guiding principles for securing the build pipeline include:

- Each component in the pipeline, from infrastructure to code, should have a single responsibility. This division of responsibility should support least privilege authorisation.
- Steps should have clearly defined build stage inputs and outputs (artifacts) to allow greater control over data flow.
- Clearly defined output parameters enable signing of data to provide non-repudiation of artifacts and metadata.
- All aspects of the pipeline infrastructure and configuration should be immutable.
- Pipeline steps themselves should be subject to automated testing to validate the efficacy of the security controls within the pipeline.
- Pipeline steps should produce signed attestations for out-of-band verification of the build process.

For more detail about build pipelines, see the TAG Security Secure Software Factory Whitepaper.

Verification

Enforcing policy

Project and organizational release policy should be maintained as software supply chain policy. Metadata produced during the build process should be analyzed to ensure that the products, materials, and processes used during the build pipeline fall within requirements defined in the policy. Verification of the build policy should be performed in a cryptographically provable way. The in-toto project can be used to secure a chain of pipeline stages end-to-end with cryptographic guarantees. It enables supply chain owners to define a "layout" of the software supply chain, and includes a workflow to produce metadata files, or "links," by hashing and signing the inputs and output of the steps in the pipeline. The presence and output of each build step should be attested to during the build. Finally, in-toto includes a

verification workflow that analyzes the links to ensure that they meet the constraints set in the layout. The SLSA framework uses in-toto attestations, combined with defined maturity levels that specify the build policy. Build metadata should be evaluated against the policy by using tools such as Open Policy Agent (OPA). Open Policy Agent provides a language to enforce policy on JSON data derived from any source, such as Terraform or SPDX.

Validate environments and dependencies before usage

The build environment's sources and dependencies must come from a secure, trusted source of truth. Checksums and any signatures should be validated both in the downloading or ingestion process, and again by the build worker. This should include validating package manager signatures, checking out specific Git commit hashes, and verifying digests of input sources and binaries. After completing this validation, the downloading process could sign all binaries or libraries to generate an attestation or include them in an internal repository that allows future steps to ingest them directly.

Validate runtime security of build workers

Out-of-band verification of runtime environment security, as defined by execution of policies using tools such as seccomp, AppArmor, and SELinux, provides defense in depth against attacks on build infrastructure. Policy rule sets should be created and applied to build infrastructure. High privilege kernel capabilities such as debugger, device, and network attachments should be restricted and monitored. Findings should be forwarded to organizational Security Information and Event Management (SIEM) systems for remediation.

Validate build artifacts through verifiably reproducible builds

A deterministic build produces equivalent outputs when given the same inputs and enables the detection of unintended changes (whether malicious, such as malware and backdoors, or accidental). Verifiably reproducible builds improve on this by enabling cryptographic attestations that the given inputs produce the same output. A verifiably reproducible build is a build process where, given a source code commit hash and a set of build instructions, an end user should be able to reproduce an identical artifact. We can build on these cryptographic assertions to provide additional security properties, such as artifact flow integrity. Assurance of the build is obtained when multiple builders, on separate infrastructure, achieve consensus of the build artifact signature.

The ability to build software in a verifiably reproducible manner is of growing importance as the software industry sees more attacks on build infrastructure. Building software in a reproducible way is not a trivial task however, and care needs to be taken to capture the required build environment and remove non-determinism from all aspects of the process. In what follows, several recommendations specific to the production of reproducible builds are given.

Recommendations For Reproducible Builds

Lock and Verify External Requirements From the Build Process

External resources introduce a source of risk into systems. Verification by reproducibility gives security administrators more confidence that external software has not been compromised. Reaching out to external sources at build time can be a challenge when trying to make builds reproducible. External resources can change or disappear unexpectedly. Third party packages that are part of the build process

can be vendored (included in the revision control system alongside the code that depends on them). When this is not feasible, hashes should be recorded of any remote data for verification during future builds. Pinning specific versions of external requirements increases the consistency and ability to verify build assets throughout a project's lifecycle. However, if pinned versions are not regularly updated, vital bug fixes and security patches may be missed. Therefore, it is important to use tools like dependabot or renovatebot to track and update dependency versions regularly. If an external dependency cannot be verified in some way, replacement or complete removal of the dependency should be considered.

Find and Eliminate Sources Of Non-Determinism

Timestamps, locale differences, and embedded version information are just a few of the things that may affect determinism in the build process. A compiler may produce different output if source files are passed in differing orders. Some compilers may embed build information such as build time. The things that may affect determinism in a particular build depends on which tools and compilers are being used. Reproducible Builds documents and offers solutions for many of these. Diffoscope can be used to identify the sources of build non-determinism.

Record The Build Environment

In order to reproduce a build environment, the versions and hashes of all tools and required configurations should be recorded and distributed with the software's source. Compilers, system libraries, build paths, and operating systems are some of the items that are required to be recorded to properly reproduce a build environment. Debian has file format that they use for this purpose. Storing snapshots of historical build environments allows some forensic analysis and may be more achievable for some teams.

Automate Creation of Build Environments

Other developers or verifiers who wish to build your software need to be able to recreate the build environment with minimum effort. A project may use scripts to quickly retrieve and set up the required tools in a virtual environment. For example, tools like Vagrant can be used to declaratively create a virtual machine, or Dockerfiles and container images.

Distribute Builds Across Different Infrastructure

To detect potential attacks on the build infrastructure an architecture can be deployed to distribute the builds to multiple hosts. Each host independently and deterministically builds the same component. A hash of each resulting artifact can then be verified to ensure that the results match, and any divergence can be examined. To successfully attack such an architecture, the attacker must compromise multiple disparate systems. Rebuilderd is an example of a tool that can be used to set up independent rebuilder environments on different infrastructure. It currently supports rebuilding Arch Linux packages, with support for Debian packages planned for the future.

Automation

Build and related continuous integration/continuous delivery steps should all be automated through a pipeline defined as code

Steps such as build, linting, scanning, and validation should be clearly defined in code that describes a pipeline. This means that all steps from code checkout, to checksum and signature validation, to building and compilation, to publishing, and eventually to deployment should be automated. By putting all steps

in a pipeline as code, the likelihood of human error is reduced, and the same steps are taken on each build. The only manual steps during this process should be any code reviews or sign offs.

Standardize pipelines across projects

CI/CD processes should be standardized across the enterprise for ease of maintenance and to ensure secure configurations. This process should be enforced by templating CI/CD pipelines or using shared workflows and verifying that pipelines and shared workflows meet organizational standards. Open source tools like Jenkins, Tekton, Argo, and GitLab provide the ability to either template pipelines or include shared workflows in pipelines for enforcing organizational release processes. Organizations should take care that use of these templates and shared workflows can not be bypassed by developers.

Verification of pipelines should occur before release or distribution. This can ensure that pipelines are following best practices. in-toto provides a layout specification enabling out-of-band verification of CI/CD pipelines.

Provision a secured orchestration platform to host the build pipeline

During the initialization phase of the platform, trust should be bootstrapped in the system. Methods to bootstrap trust vary by installation type. The best practices described in the cloud native security whitepaper (including protection from unauthorized access, immutability, availability, auditing and accountability) should be followed during the platform hardening process. The CNCF provides the Cloud Native Trail Map as a starting point for architecting a secure application platform. Additionally storage and network access need to be provisioned. The provisioning process should be described as IaC and happen through an automated and audited process. Cluster administrative credentials should be secured with hardware tokens. The use of GitOps as a deployment methodology ensures all changes to the system are tracked and can be integrated with organizational identity management systems.

Build Workers Should be Single Use

Build Workers, the workloads that perform the Build Steps, should only be used once before being thrown away. This lowers the blast radius of a compromised build worker, and limits the attack surface by keeping the lifespan of a build worker to a single build operation. Long-lived build workers are prone to configuration drift and can lead to increased risk of attack. Shorter build worker uptime makes it easier to reason about the operations performed during its lifecycle. For use cases where future forensics might need to be performed, it is prudent to either snapshot the build worker or to keep it running and quarantine it after its build.

Controlled Environments

Ensure Build Pipeline has minimal network connectivity

The build pipeline should have no network connectivity other than to connect to the trusted sources of source code, the dependency repository and code signing infrastructure.The build workers will require a secure shared storage capability to pass data between each worker. This prevents the workers from repeatedly retrieving remote data for the build and thus improves performance. This shared storage must be encrypted and secured to prevent tampering.

Segregate the Duties of Each Build Worker

When planning what each build worker will be responsible for, consider segregation of duties within the domain of a particular build. It is generally better to have specific build workers handle specific parts of a build as opposed to having a single worker handle all steps, e.g. lint, compile, submit for remote scanning, push artifact etc. Splitting the build between workers reduces the attack surface for any compromised worker.

Pass in Build Worker Environment and Commands

A build worker should have a hermetic (e.g. isolated and sealed) environment given to it from the broader pipeline, and no ability to define its own environment. This is because a compromised worker can potentially create an environment of the attacker's choosing, including hostile tooling and persistent threats to pivot or retain access. In addition to the environment, the worker's commands or actions should be passed in explicitly at provisioning time. There should be minimal decision logic in the worker itself. One method of doing this might be to use Kubernetes pod or job objects to deploy the workers for each step in the pipeline. The object definitions can be used to specify the environmental variables, commands, and volumes relevant to that particular step, while the base image itself remains minimal. For environments which do not use Kubernetes as an orchestrator, other methods exist for externally declaring the environment and commands that a container should execute.

Write Output to a Separate Secured Storage Repository

The output artifacts of builds require similar security considerations as the inputs and the environment definition. The output artifacts should be written to separate storage from the inputs. A process separate from the worker should then upload the artifacts to the appropriate repository for downstream consumption.

Secure Authentication and Access

Only allow pipeline modifications through "pipeline as code"

The pipeline configuration should be deployed through the pipeline as code and should be immutable. It should not be possible for an administrator to modify an instantiated pipeline to ensure an attacker posing as an administrator cannot interfere with it directly. This model requires appropriate authentication and authorisation to be in place for the configuration of the pipeline.

Define user roles

Organizations must define user roles in a build pipeline which should be used to define permission boundaries. In high security environments, no single user should be able to attest to the validity of a software release. For example, policy administrators are responsible for ensuring organizational policy is encoded into software pipelines and attestation gates. Administrators are responsible for the maintenance and innovation of a build pipeline. Inevitably there will be shared responsibilities across these roles. Teams should carefully define the boundaries and delineate expectations around those shared responsibilities, especially when it comes to security concerns. For example: how much responsibility do application developers have for security within the pipeline? What are they not responsible for?

Follow best practices for establishing a root of trust from an offline source

Root of trust for a build pipeline should follow standard methods for their establishment from an offline source.

Use Short-Lived Workload Certificates

Workloads should be issued short lived credentials with automated rotation. The CNCF maintained SPIFFE/SPIRE project provides a robust identity and certificate orchestration system. The publication "Solving The Bottom Turtle" is an excellent resource to consider when designing an organizational workload identity system. Certificate rotation policy is a decision based on the trade-off of system availability and the security impact of a lost key. In situations where the use of long-lived certificates is required, a robust certificate revocation system should be implemented.

Deploy monitoring tools to detect malicious behavior

Additional security techniques should be integrated into the build pipeline itself to monitor for suspicious or unexpected activity, such as attempts to connect to unexpected endpoints during the build process. It is especially valuable to provide enhanced monitoring of stages within the pipeline that affect the resulting artifacts (as opposed to stages performing testing on them) as any deviations from their normal process may be a sign of compromise.

Sign Every Step in the Build Process

Individual steps in the build process should be attested to for process integrity. Build step inputs, outputs, and process traces should be collected and evaluated as part of the software release and distribution process. The final artifact bundle should include these collective signatures and itself be signed to give integrity to the completed artifact and all its associated metadata. These signatures may use keyless signing methods, like Sigstore, to tie signatures to a developer identity rather than long-lived keys. The CNCF sponsors in-toto, The Update Framework (TUF), and the SPIFFE/SPIRE projects, each of which support the framework for such an attestation/control system.

ARTIFACTS

An artifact is the output of the build pipeline once the software has been built and packaged for distribution. Software artifacts, along with corresponding build metadata, should be hashed and signed by authorized entities. The signing of software artifacts should follow a process ensuring the integrity and provenance of the artifact at build time, helping to establish trust. Cryptographic keys used for each artifact should be part of a chain of trust stemming from a secure root of trust. The signing of an attestation for an artifact is a method of indicating that an artifact has been vetted and approved to be used in a given environment. Trust is established through cryptographically generated signatures at build time based on a secure hash of the artifact and, in more complex scenarios, whether the artifact has been signed by another process in the supply chain. For example, consider a scenario where the artifact is

signed at build time and the resulting signature is verified before the artifact is scanned for compliance and security.

The signing of artifacts and creation of signed attestations should be performed at each stage of an artifact's life cycle, along with the verification of signatures and attestations from prior stages, to ensure end-to-end trust. For added protection, encryption can be used to protect the confidentiality of the artifact, and to ensure only authorized parties are able to use the artifact.

Verification

Validate the Signatures Generated at Each Step

The integrity and provenance of images, deployment configuration, and application packages included in artifacts should all be validated using the signatures generated by each step in its build process to ensure compliance with the organization's requirements. This includes signatures on attestations. The contents of attestations should additionally be verified against the software supply chain policy. Additionally, software metadata, such as SBOMs, should have verified signatures to ensure objects in an artifact's manifest or dependency metadata store have not been tampered with between build and runtime.

Perform additional checks on the artifact

Perform additional testing or validation on the built artifact to ensure it meets security expectations. Once these checks are performed, add a signature to the artifact indicating that it is ready for release. This signing key for this validation service should be delegated to as the trusted key for this artifact.

Policy

In addition to validating the signatures on attestations, the verifier should also check that these attestations adhere to the software supply chain policy. For example, this policy could ensure that the output of the source code is the same as the input to the build system, or that the correct entity signed each attestation.

Automation

Use TUF to manage signing of artifacts

The Update Framework (TUF) is a graduated CNCF project that enables the secure distribution of artifacts by combining trust, compromise resilience, integrity, and freshness. Several TUF implementations exist that can be leveraged by new adopters, including RSTUF and tuf-on-ci. Signatures and metadata about artifacts are stored, often adjacent to an OCI registry. TUF makes use of a "root-of-trust" model to delegate trust from a single root to the individual teams or developers who sign artifacts. This should be used to delegate to the expected signing entity for each artifact, which may be a build server or developer. It uses additional metadata to allow clients to verify the freshness of content in a repository and protect against common attacks on update systems. Clients can make use of public keys to verify the contents of the repository.

Use a store to manage metadata from in-toto

Organizations that generate in-toto metadata need a way to track and store this metadata. This may be a database or a dedicated solution such as Archivista or Grafeas. Archivista is a CNCF project that provides a graph and storage mechanism for in-toto attestations. Grafeas is an open source artifact metadata API supporting in-toto link attestations as a type, and has a supporting Kubernetes admission controller called Kritis.

Distribute in-toto metadata with TUF

The in-toto metadata, along with the supply chain policy for an artifact needs to be securely distributed to users to prevent tampering and rollback attacks. Delegations in TUF can be used to associate each image with its policy and in-toto metadata, while preventing attacks on their distribution. This TUF metadata can be stored alongside the artifact or in a metadata store, such as those discussed in the Metadata section to allow for secure discovery and verification.

Controlled Environments

Limit which artifacts any given party is authorized to certify

It is important that a software delivery and update system must not grant trust universally or indefinitely. The system must make it clear which artifacts or metadata a given party is trusted to certify using selective trust delegations, such as those in TUF. Trust must expire at predefined intervals, unless renewed. Finally, the idea of trust must be compartmentalized — a party must only be trusted to perform the tasks assigned to it.

Build in a system for rotating and revoking private keys

It is insufficient to cryptographically sign components being distributed and assume they're protected. Instead, the system must be prepared for *when, not if*, its private keys are compromised. The ability to rotate and revoke private keys must be built into the distribution mechanism. This distribution mechanism must allow users to ensure that they are using a currently trusted set of keys, and not keys that have previously been revoked. This requires a notion of timeliness, similar to the one used by TUF. Additionally, multiple keys must be used, especially for different tasks or roles, and a threshold of keys must be required for important roles. Finally, minimal trust must be placed in high-risk keys like those that are stored online or used in automated roles.

Use a container registry that supports OCI image-spec images

An internal image registry should be deployed and configured to support internal artifact distribution with the security properties described in this section. This might be accomplished by distributing metadata using the recently standardized OCI artifact manifest.

DEPLOYMENTS AND DISTRIBUTION

Software delivery systems have been historically prone to several types of attacks. The Update Framework (TUF) has been designed to be resistant to these attacks. Therefore, any system designed to distribute software artifacts and their corresponding metadata must have several properties that enable them to counter the attacks defined in the TUF spec: Trust, Compromise Resilience, Integrity, and Freshness. In addition the system must contain preventive and detective capabilities to monitor its security posture and report if attempts to compromise are discovered.

Verification

Ensure clients can perform Verification of Artifacts and associated metadata

Clients receiving software artifacts and policy from the distribution mechanism must be able to verify the integrity of the downloaded files using the policy. It's also vital that the view a client has of the repository is consistent and up to date so the client sees the latest version of all the files it has access to. This is especially necessary for highly volatile repositories.

Clients must verify the metadata associated with the artifacts. In addition to authenticating the metadata, such as by verifying its signature(s), the contents of the metadata must be verified as well. For example, if SLSA provenance is captured for a build process as an in-toto attestation, the signature on the attestation as well as the provenance metadata must be verified against the policy. Adopters can use different tooling depending on the type of metadata that must be verified, with tools like Witness shipping with policy features to verify different types of in-toto attestations.

Ensure clients can verify the "freshness" of files

Since software updates are used to deliver bug fixes and security patches, it is important for clients to have access to the latest available versions. Clients must be in a position to recognize when they are being provided files that are out of date. Clients must also recognize if they are unable to obtain updates that exist on the repository.

Air-gapped deployment

In air-gapped environments, verifiers will not have live access to artifacts and attestations. While these artifacts and attestations can be copied into the air-gapped deployment, there may be a lag leading to expired signatures or a failure of freshness checks. To address this, air-gapped deployments may use a time in the past (such as when artifacts were copied) when making expiration and freshness checks. Further, all verification should be done before artifacts are copied into the air-gapped environment, with an attestation that such verification occurred. With this attestation, the verifier can ensure that the artifacts were valid at the time of ingestion.

Admission controller/deployment gate

All software supply chain metadata should be verified at the deployment gate, for example by a

Kubernetes admission controller. By performing verification at this stage, any mistakes or vulnerabilities in the software supply chain can be caught and addressed early in the process. Verification at the deployment gate can supplement, but not replace verification at the end of the pipeline. Further metadata may be generated or tampered with between the deployment gate and the end user, so verification must also be done by the end user. The deployment gate verification allows for early detection.

Automation

Use The Update Framework

TUF has been used to bootstrap trust in delivering software supply chain metadata, specifically those pertaining to in-toto. The Datadog model combines TUF and in-toto to provide end-to-end guarantees to the consumer. This model has been documented in a blog post and two in-toto Enhancements (ITEs), ITE-2 and ITE-3.

Continuous vulnerability scanning

The state of an application's dependencies is not static because vulnerabilities could be discovered after deployment. SBOMs and VEX documents can help track which new vulnerabilities might affect the software.

An Example of a Secure Supply Chain

This example assumes there is a policy that defines who should do what inside the supply chain for an application.

1. A developer commits code on their workstation.
2. Tools generate an attestation for that commit based on the developer's identity.
3. Developer pushes the commit to a remote code repo.
4. A build is triggered based on the push.
5. The build runs pre-build linting, scanning, and other security steps.
6. Each step has a signed attestation generated with an identity tied to the instance of that step running.
7. The build runs any compilation, packaging, and SBOM generation steps.
8. Each step has a signed attestation generated with an identity tied to the instance of that step running.
9. The build runs any post-build scanning and security steps.
10. Each step has a signed attestation generated with an identity tied to the instance of that step running.
11. The build pushes the packaged artifact to a package repository.
12. The package repository verifies that the artifact has all required attestations to be published
13. A consumer which can be a human or an automated system like a deployment system ingests the attestations for the package and confirms with the package system that is what they're downloading.

14. Once the package is downloaded the consumer verifies that what they received is what is in the attestations.
15. The consumer performs any other security related scans for their risk appetite for that artifact.

OUT OF SCOPE

The software supply chain security space is vast, so not all topics could be included in this document. The following is a non-comprehensive list of topics that did not fit in this paper, but could be explored in future work.

- Supply chain threat modeling
- Hardware supply chain security
- Details about determining supply chain policy (see Policy-as-Code in the software supply chain)
- Application security
- Certificate/root of trust management
- Monitoring
- Governance (see Automated governance WG)
- More discussion of zero trust
- Incident response and mitigation
- Compliance

ENDNOTES

1 Juniper Research Study Reveals Staggering Cost of Vulnerable Software Supply Chains, May 2023

2 Verizon Data Breach Information Report, May 2024 9th Annual State of the Software Supply Chain Report, October 2023

3 IDE's in the cloud are gaining popularity, so this might change in the future.

4 https://www.cisa.gov/news-events/news/4-things-you-can-do-keep-yourself-cyber-safe

5 Direct dependencies are explicitly defined by a developer, but transitive dependencies are defined upstream and often not known to the developer. For a deeper explanation see Direct Dependencies vs. Transitive Dependencies.

THANK YOU!

CLOUD NATIVE
COMPUTING FOUNDATION