CLOUD NATIVE COMPUTING FOUNDATION

The Secure Software Factory

A reference architecture to securing the software supply chain

by the Security Technical Advisory Group

Contents

Introduction

A software supply chain is the series of steps performed when writing, testing, packaging, and distributing application software to end consumers. Given the increased prominence of software supply chain exploits and attacks, the Cloud Native Computing Foundation (CNCF) Technical Advisory Group for Security published a white-paper titled "Software Supply Chain Best Practices",[1] which captures over 50 recommended practices to securing the software supply chain. That document is considered a prerequisite for the content described in this reference architecture.

This publication is a follow-up to that paper, targeted at system architects, developers, operators, and engineers in the areas of software development, security and compliance. This reference architecture adopts the "Software Factory" model[2] for designing a secure software supply chain.

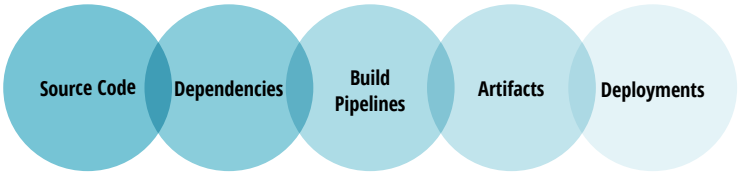
This reference architecture has been produced after a thorough evaluation of avail-able tooling as of early 2022. The components selected are open source, cloud native, and prioritise security.

Problem Scope: Software Supply Chain Security

The practices that the "Software Supply Chain Best Practices" whitepaper captures are predicated on four overarching principles:

- Defence in depth (Layered end-to-end security controls)
- Signing and Verification
- Artifact Metadata Analytics
- Automation

Those four principles are in turn applied and organised around five functional areas deemed as the entities in a software factory:



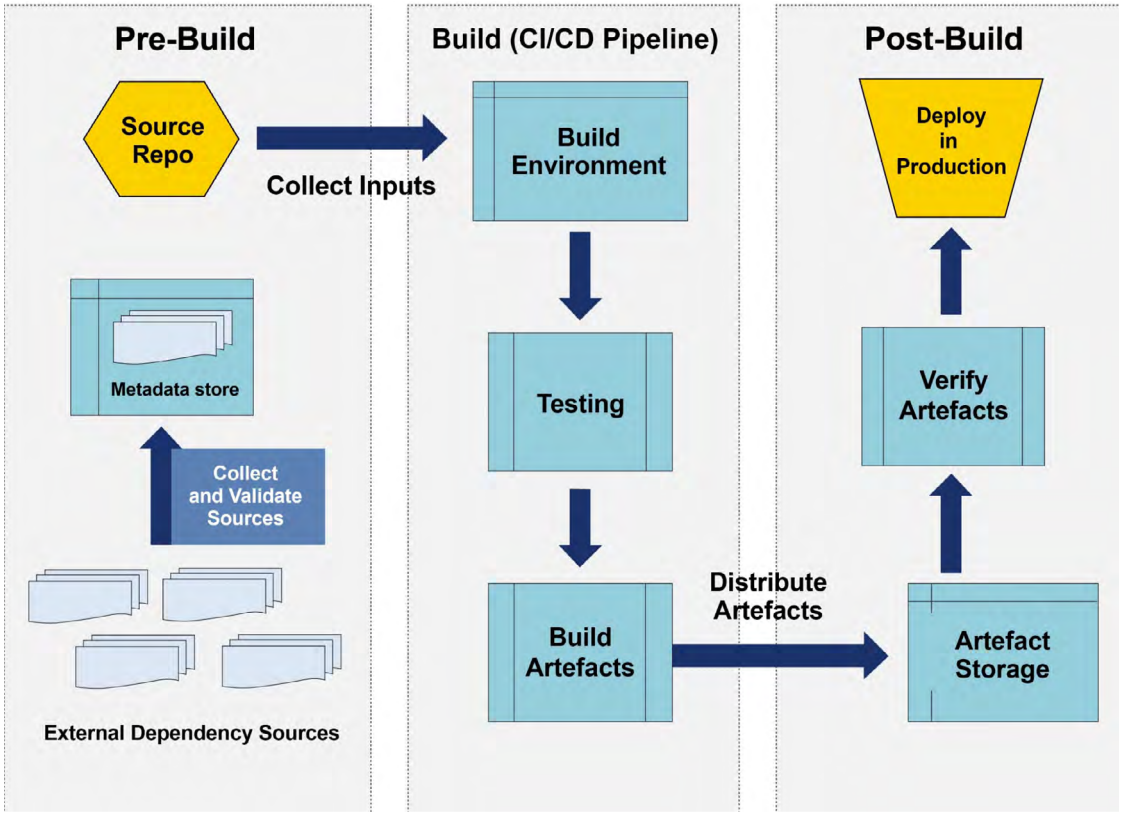
When thinking about how to secure those entities, there are two broad ways of organising security controls:

1. Around three critical concerns:
   - **1.1. Provenance verification**: assurance that existing assumptions of where and how an artifact originates from are true and that the artifact or its accompanying metadata have not been tampered with during the build or delivery processes.
   - **1.2. Trustworthiness**: assurance that a given artifact and its contents can be trusted to do what it is purported to do (ie, is suitable for a purpose). This involves judgement on whether the code is safe to execute and making an informed decision about accepting the risk that executing the code presents.
   - **1.3. Dependencies**: recursive checking of an artifact's dependency tree for trustworthiness and provenance of the artifacts it uses.

1. By stages of activity (see diagram):
   - **1.1. Pre-Build**: principally concerned with development and handling of the source code and with the collection and storage of dependencies.
   - **1.2. Build**: the process of building, testing, and packaging an artifact according to its build specifications.
   - **1.3. Post-Build**: principally concerned with the storage, delivery, deployment, continuous verification.

1 Markdown version of the Software Supply Chain Best Practices paper can be found at https://github.com/cncf/tag-security/blob/main/supply-chain-security/supply-chain-security-paper/sscsp.md and is referenced throughout this document for convenience of navigation

2 https://en.wikipedia.org/wiki/Software_factory



In the matrix below, we attempt to overlay these entities, concerns, and activity stages with one another:

Stages	Pre-Build	Build	Post-Build
Entities:	• **Source Code:** Development and Handling • **Materials:** Selection, Collection, and Storage	• **Source Code and Dependencies:** As Inputs • **Build Pipelines:** Components performing the build • **Artifacts:** As outputs	• **Artifacts:** Storage and Verification • **Deployments:** Verification of artifacts
Concerns:	• **Provenance:** Developer Contributions, Dependency Definitions • **Trustworthiness:** Developer Contributions • **Dependencies:** Dependency provenance and trustworthiness	• **Provenance:** Integrity of the build, collection of metadata and attestations, signing of artifacts	• **Provenance:** Verification of Attested Metadata • **Trustworthiness:** Consumer judgement of artifact's worth • **Dependencies:** Recursive analysis of both Provenance and Trustworthiness by consumers

This reference architecture focuses specifically on the critical concern of provenance and primarily on the activity stage of the "build." There are numerous other publications and guides which address issues around trustworthiness, including practices like SAST/DAST scanning, code signing, etc, including the CNCF Software Supply Chain Best Practices Paper. We direct readers to these documents for more information on those facets of supply chain security.

Our decision to emphasize provenance and the build pipeline in this paper is based on the foundational role provenance verification plays in other supply chain security concerns. Provenance provides an attestation that an output was derived from the claimed inputs. If you are relying on the results of SAST/DAST scans of a software artifact to inform your decision on its trustworthiness, you need to know that those claims are accurate. By validating that provenance came from a trusted party's identity, you get a level of assurance that those claims are accurate. An identity is provided by a system or user signing the provenance and a trusted identity is one certified by the end user for a specific purpose. All of these claims are foundational to being able to make informed decisions about an artifact's trustworthiness: you need to know the provenance attestation has been signed, and that signature is by a trusted identity, which has been certified for a specific task.

How to Read This Document

This paper offers a high-level treatment of a secure software factory. This is designed to explain the necessary interfaces and control structures for each component of a software factory to generate verifiable provenance. The theoretical treatment in this architecture should provide guidance on what features and/or configurations are required, which will allow readers to pick the tools they prefer to create their secure software factory.

This document also provides multiple suggestions around how the components in the Secure Software Factory should be configured and managed for secure operation. Some practices must be followed in order to satisfy the definition of the SSF, however many of the practices and how they are followed is dependent on the risk appetite of a project or organization.

The Secure Software Factory

"Architects look at thousands of buildings during their training, and study critiques of those buildings written by masters. In contrast, most software developers only ever get to know a handful of large programs well—usually programs they wrote themselves—and never study the great programs of history. As a result, they repeat one another's mistakes rather than building on one another's successes."

— THE ARCHITECTURE OF OPEN SOURCE APPLICATIONS

The subsequent sections detail how a Secure Software Factory ought to be structured and how its different parts interact.

Key Diagrams

Secure Software Factory Landscape

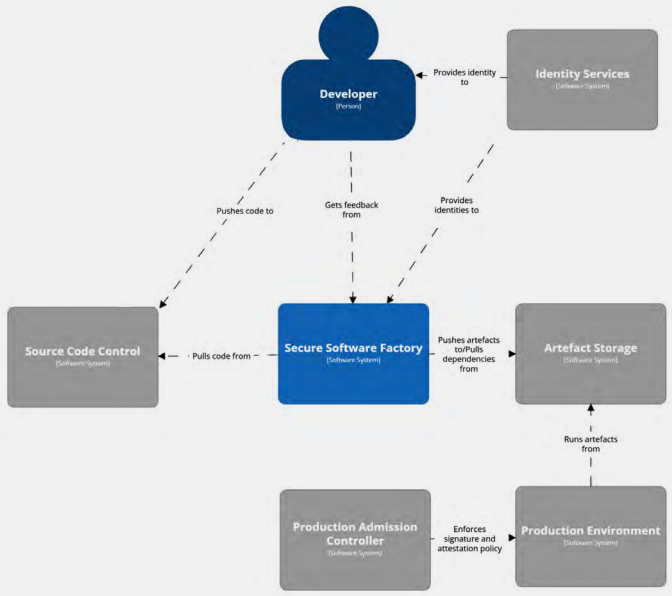
The Secure Software Factory (SSF) fits in an organization's software and IT environment. Within that environment, the SSF has both upstream and downstream dependencies. Upstream, the SSF depends on Identity and Access Management for both human users and other software services. During a pipeline run, the SSF relies on Source Code Control for fetching the code to be built and on Artifact Storage for dependencies required for the build. It also relies on Identity and Access Management for providing identities to the components making up the SSF. Downstream, the SSF is depended on for providing attestations and signatures regarding artifacts which can be used by production systems to determine artifact provenance and make policy decisions about artifact deployment.

Secure Software Factory Components/Elements

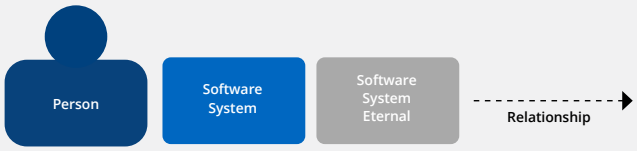
FIGURE 2 shows how the various services running inside of the Secure Software Factory interact with each other, and a portion of the external services they depend on. The diagram is simplified, and doesn't show every interaction between each tool. For example, in a real environment, Runtime Visibility monitors more than just the Build Environment. The remainder of this document illustrates how the services interact and function in further detail.

FIGURE 1

Secure Software Factory Landscape



Legend



How to read relationships (arrows):

Arrows show the initiator of actions in the system. For instance, an arrow pointing to a system A ---> B means that A is initiating an interaction with B, therefore A has the responsibility to start the action. This arrow notation should not be read as a data low, though the arrow can highlight the context for the interaction.

For example, the following example:

"[The Application] — retrieves-data-from -----> [The DB]"

Is read:

"The application retrieves data from the DB."

FIGURE 2

Secure Software Factory Components/ Elements

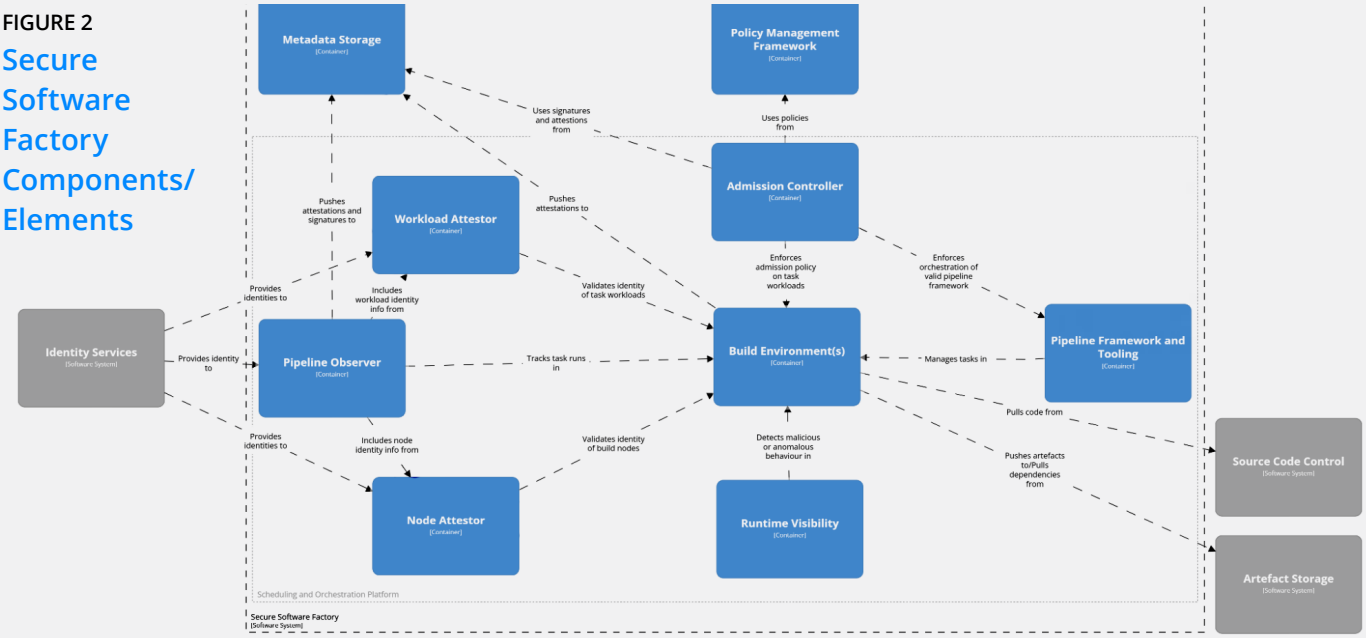
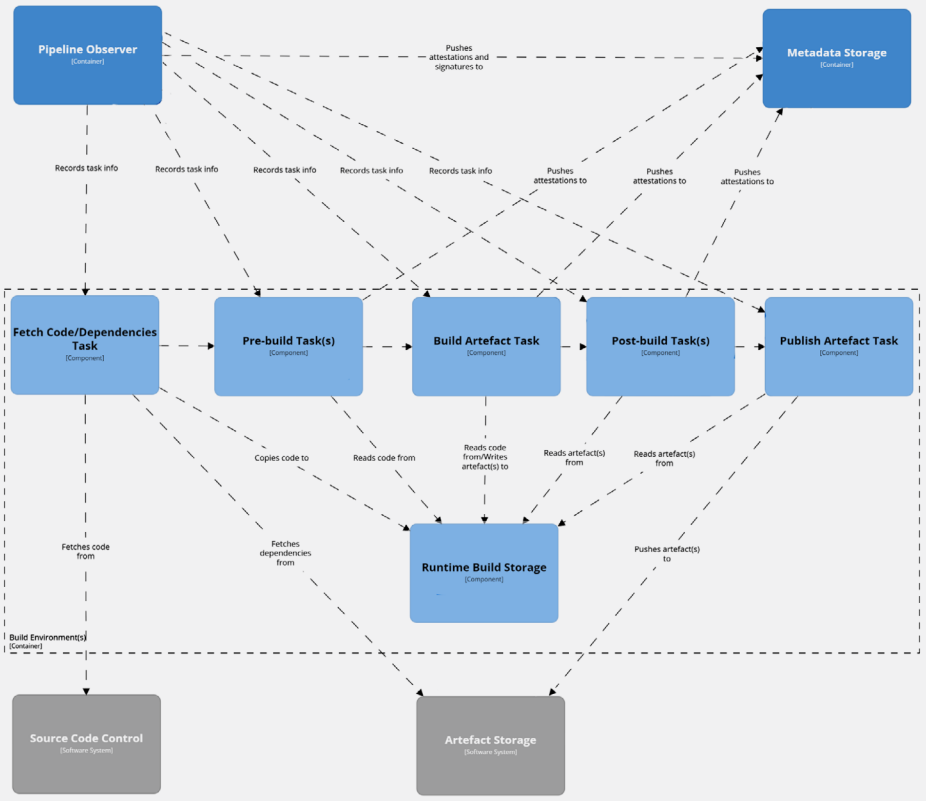


FIGURE 3

Pipeline Run Example



Pipeline Run Example

FIGURE 3 is intended to show an example Pipeline Run inside the SSF. Each step includes a validation call to an admission controller. We have left this out of the above diagram for better visual clarity. Some tasks might interact with other external services outside the scope of the SSF. The exact number of tasks depends on the requirements of your project.

There are a few important takeaways from **FIGURE 3**.

- The Pipeline Observer records what Tasks occur in what order.
- The Tasks interact with a form of Runtime Build Storage during normal operation.
- The storage in some cases might be shared between tasks, while in other cases it might not. For additional guidance on shared storage configuration, please refer to the CNCF Supply Chain Security Best Practices whitepaper. Additional storage guidance is provided with the CNCF Storage Whitepaper.

Not every task will provide attestation or additional metadata, but those that do support this capability should be signed and securely stored in a source of truth. This is due to factors such as the ability to reliably instrument the metadata (e.g. can't observe self or no hardware/software interfaces to do so), or in cases where metadata produced is not actionable.

Components of the SSF

The SSF that manufactures secure software can be broken down into several categories of components, like that of a regular factory. These are the core components, the management components and the distribution components. The core components are responsible for the central task of the Secure Software Factory: taking the inputs of the factory and processing them to create the output artifacts. The management components ensure that the factory runs in accordance with policy. It ensures that the processes of the factory are validated in the right way, and provides evidence and documentation of the outputs of the factory. The distribution components are in charge of moving the products of the factory to where they can be made available for usage, as well as to provide guidance and tools to ensure that outputs of the factory are consumed safely.

The "Core" Components

The core components can further be classified into 3 stages: the Scheduling and Orchestration Platform, which runs all the other components, the Pipeline Framework, which details the basic layout of the build pipeline, and the Build Environments, which perform the actions defined in the pipeline.

Scheduling and Orchestration Platform

A Secure Software Factory seeks to run its components in the most secure way possible. All other components of the SSF leverage this platform to schedule their jobs to perform their respective actions.

For additional guidance on scheduling and orchestration, please refer to the CNCF Cloud Native Security Whitepaper.

Pipeline Framework and Tooling

Pipelines are a core part of the SSF as they encode the concrete workflow for building the software artifacts. This typically follows a Continuous Integration (CI) workflow, i.e. repeatable and declaratively defined sets of tasks intended to download, build, and test code. In a cloud native context, the pipeline tooling uses a scheduling and orchestration platform to run each task in a container.

Given that the pipeline is running on the scheduling and orchestration platform, it must be treated as any other workload the platform manages, including being subject to the same security requirements and measures. At a minimum, all container images used in the pipeline must be subject to signature verification and scanned for any known vulnerabilities.

Build Environments

The build environment is the actual container(s) or worker(s) where the source code is turned into a machine-usable software product, which we refer to as an artifact. Existing CI frameworks typically follow ephemeral execution patterns, wherein a new instance is created only for the lifetime of each execution job. This pattern should be further extended to create a new instance of the scheduling platform to host every new build pipeline. The build environment must generate evidence and an automated attestation about the input parameters, actions, and tools used during the build, such that they can be independently validated to assure build security.

The "Management" Components

A SSF will use a Policy Management Framework to enforce various controls and gates. This should include policies around identities of users who may invoke the pipeline, worker nodes where the pipeline should be executed, and container images that can be used in the pipeline. These policies are dependent on the risk appetite of the organization or project. It will then utilize a series of monitoring components to verify conformity with these policies: Node Attestors, Workload Attestors, and Pipeline Observers.

Policy Management Framework

A SSF needs policies that define the actors for each step in the build process. For example, a policy might define the actor (human or otherwise) authorized to sign metadata for a particular task. These policies are important at the time of verification within, for example, an admission controller, where they are used to validate that the right actors performed the respective tasks.

Policies should follow cloud native and supply chain security best practices. For more information on policy management and good policy concepts, refer to the CNCF SIG-Security K8s Policy Management Whitepaper.

Attestors and Observers

There are three basic components of the SSF which monitor or attest to policy adherence:

- Node Attestors verify the identity (and authenticity) of the node, authenticating a node's request for its identity document
- Workload Attestors verify the identity (and authenticity) of the workload process on a node, authenticating a workload's request for its workload identity document
- Pipeline Observers, which capture the verifiable metadata from pipeline processes.

Node attestors and workload attestors work in conjunction to ensure the node selected for running the work is authorized to host that workload and that it is not compromised. Pipeline observers then build upon this evidence by generating additional metadata about individual tasks executed in the pipeline to provide comprehensive assurance across the build process. This synthesis allows later steps to validate that previous steps were completed as expected and provides a level of guarantee around the provenance claims and legitimacy of the final artifacts from the SSF.

All metadata from Node Attestors, Workload Attestors, and the Pipeline Observer should be signed and included as part of the metadata documents output from the SSF.

The "Distribution" Components

Upon completion of a pipeline run, the SSF outputs several artifacts. Artifacts must be available to downstream consumers and securely stored. Signatures for artifacts should also be stored such that they can easily be found and verified. These signatures can be stored, for convenient discoverability and distribution, alongside the artifact or in a separate location.

Artifact Repository

The Artifact Repository stores artifacts the SSF outputs. This repository should be accessible from both the build and deploy environments. The stored artifacts may include container images, Helm charts, SBoMs, and their corresponding signatures. In some cases, the artifact repository can also serve as the storage location for metadata, such as SBoMs, attestations, and signatures. In other cases, users may prefer to store these items separately or in multiple locations.

Admission Controller

An Admission Controller in the SSF limits what workloads can be run in the SSF's Scheduling and Orchestration Platform. "Admission control", in a general sense, is the act of enforcing policies around the consumption of components in a system.

In the SSF, there are multiple levels at which admission control must occur:

- **Enforcing policies on the sources and packages pulled into a build,** including "intermediate artifacts" passed between steps in the build pipeline. For example, evaluating whether these objects have been properly signed or came from a known and trusted party.
- **Enforcing policies around the components of the factory itself.** The scheduling and orchestration platform should perform admission checks to ensure all such components are trusted and verifiable.

- **Enforcing policies on the build steps.** This typically includes verifying pipeline definitions and all the referenced images to be used during execution.

In order of execution, admission control proceeds as follows:

1. When admitting the build request, the Admissions Controller validates that steps satisfy defined policies.
2. When steps that fetch dependencies are executed, the Admission Controller must enforce policies on the dependencies that are sourced into the environment (e.g. source, binary dependencies, base images).
3. When steps execute user-provided code, the Admission Controller uses a network jail to enforce an "admit nothing" policy because we do not trust that code to self-regulate.
4. When steps that publish artifacts are executed, they must produce attestations to satisfy the Admission Controllers that may be encountered downstream.

Outside of simple build execution, relevant areas to admission control include:

- The components that are "admitted" to the node host environments
- Policy enforcement on the build control plane (incl. admission control), which recurses (who watches the watchers?).

In addition to the above inputs, it is assumed that the following checks are being handled when deploying to production.

- Security controls for admission controller itself (identity of the controller and validation)
- Metadata inputs for different policies
- Diff signatures or policies validation (interface with CA's for validating certs), Notary services
- Enforcement points
- Interfaces with Signing services/notary service/signature validation services
- Mutating the definition of workloads to include additional metadata
- Outputs or error messages after enforcement/blocking admission
- Signing check as a label that could be used by a workload attestor to grant access to signing keys.

Note: Artifact signatures should be verified against the associated public keys before deployment. Any generated provenance information should also be verified.

The Variables—Inputs and Outputs to and from the SSF

Inputs

Source Code

Source code encompasses the human readable representation of applications being built by the Secure Software Factory, associated dependencies being built from source or that are interpreted instead of compiled, code for the build pipelines (Pipeline-as-Code) and infrastructure (Infrastructure-as-Code). Source code is the primary input for the SSF. The users and operators of the SSF must decide what programming languages they support, where to host source code, and what tools to integrate for testing and scanning. The SSF assumes that source code uses version control systems like Git, which have a preserved history, and that the repository has an appropriate regime for review and testing in place that is appropriate for the needs and use cases of the repository. For securing the source code see recommendations that can be found in the "Source Code" section of CNCF Supply Chain Security Best Practices whitepaper.

Software Dependencies

Almost all software depends on other software which needs to be collected before building the target software. These dependencies should be validated against a security policy. It is recommended to pin to validated attestations or signatures of dependencies if available in order to validate that it was built by a trusted identity. In addition, it is recommended to pin to the checksum of upstream dependencies in order to ensure you are pulling the exact version you expect.

Software dependencies carry with them serious risks that are too often overlooked. The shift to easy, fine-grained software reuse has happened so quickly that we do not yet understand the best practices for choosing and using dependencies effectively, or even for deciding when they are appropriate and when not.

For both security and availability, it's recommended to maintain a local mirror of any external dependencies. This mirror may be limited to only dependencies that have passed a security scan or trusted and curated source of truth. The mirror also prevents downtime if the upstream repository becomes unavailable and provides a single network endpoint to secure for dependency ingestion.

More recommendations and specifics on securing dependencies can be found in the "Materials" section of the CNCF Supply Chain Security Best Practices whitepaper.

User Credentials

User credentials are identifiers for both human users and services (e.g. automation agents) and can authenticate these actors at multiple points in the SSF and its supporting services. Credentials should meet baseline security requirements as defined in the CNCF Supply Chain Security Best Practices whitepaper.

Cryptographic Material

Cryptographic material input into the SSF fall into two categories:

1. Materials used for identification of a particular entity.
2. Materials used for attestation/verification of a particular activity.

The first category includes certificates, tokens, and keys used for authenticating nodes, scheduling and orchestration platforms, workloads, services, and users. It might also include certificates corresponding with recognized Certificate Authorities and trust bundles for validating and cross-authenticating all of these materials.

The second category includes material such as signing keys deployed by users or services to attest to the work they have performed. Unlike traditional signing architectures, the modern software factory doesn't directly use a single signing key. Multiple signing keys have trust delegated to a specific domain, processes/users/services to limit the impact of a compromised key.

All cryptographic materials must conform and comply with standards for their type and purpose and are generated in a cryptographically secure manner. In addition, they should also expire based on the lifetime of their purpose and security and access control policy. We assume that they are securely distributed to the necessary entities and are properly configured for use by those entities. The specific mechanisms for producing, signing, and distributing these certificates will be left to the user to implement and are beyond the scope of this paper.

Pipeline Definitions

CI/CD pipelines define the steps in the application build process. The specific implementation of a pipeline will vary from organization to organization. However, all pipeline definitions should follow security best-practices that include:

1. **Persistence & Source Control:** Pipeline definitions should be defined as "code" (Pipeline-as-Code) in a declarative fashion, and as such, must meet all the security expectations for source code defined above. Additionally, pipeline definitions must be managed through a source control process (ie, git) that limits changes to only authorized users following standard protocols (ie, submitting changes via a pull request) and code reviews, partnering with security engineering, along with the particular tools being used. Once your pipeline assembly is complete, make sure to persist all relevant artifacts.

2. **Sign Pipeline Definitions:** Sign your pipeline definitions to ensure non-repudiation. During signing, sign pipeline specifications including all the images used for execution.

3. **Pipeline Audit:** Perform regular audits of your pipeline definitions to ensure the integrity of the pipeline is maintained.

4. **Static Scan:** Pipelines typically need access to various user credentials that are provided to the pipeline at runtime (e.g. git-token, OCI-registry-token, etc.). Make sure these credentials are not hard-coded in the definitions. In general, limit the use of hard-coded configurations in the definitions.

Outputs

Artifacts

A software artifact is the principal output of the Secure Software Factory. Artifacts may include binaries, software packages, container images, signatures, and attestations. They are what will be consumed by downstream users. Artifacts should be accompanied by the appropriate metadata to demonstrate their provenance (described below), stored in a secure artifact repository, and distributed through secured and well understood mechanisms. The exact nature of the artifact itself and the implementation of these requirements will vary depending on factors like language, package type, and target platform(s). Therefore, these implementation details are beyond the scope of the Secure Software Factory.

Public Signing Keys

In order to verify the signatures included in a software factory's metadata, downstream consumers will need the public keys associated with those signatures.[3] The root certificates may be included as an output from the SSF, but they should be distributed separately from the artifact and the metadata itself to allow additional verification of the certificate authenticity. Certificate chains linking the signing key to a root certificate should be included as an output from the SSF, and they should be distributed with the artifact being signed, allowing verifiers to validate a signature is trusted by an approved root certificate. As these keys should be identical to the cryptographic material used as an input to the pipeline, the security considerations already discussed for cryptographic material as inputs apply.

Metadata Documents

Throughout execution of the pipeline, a number of metadata documents are generated. Examples include test reports, vulnerability reports, and Software Bills of Material (SBOMs). These documents are a snapshot of the build that produced them. For example, a vulnerability report reflects CVEs known at the time of the build, but might become outdated as new vulnerabilities are discovered and shared. Similarly, an SBOM reflects what is in a particular build. It will always be valid for that build, but future builds with slightly different dependencies/version constraints must generate a new/updated SBOM. The following practices are recommended for managing metadata documents:

1. **Timestamp inclusion:** Always explicitly include a timestamp associated with the document.[4]
2. **Persistence:** Make sure when stored that documents are immutable, version controlled and signed.
3. **Metadata Links:** Link all metadata documents to the final deliverable artifact. For example, for a microservice application build pipeline, link the test, vulnerability, and SBOM record to the particular container image they are generated from.

Secure Software Factory Functionality

This section goes through the primary actions that the SSF performs in normal operation. It describes how a project runs through the SSF and how the SSF helps secure the supply chain by establishing and tracing provenance through the build pipeline.

All Stages: Attesting Identity of Nodes, Pipeline Orchestration, Tasks and Workloads and Establishing Provenance

Actors:

- Scheduling and Orchestration Platform
- Pipeline
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage

It is important to call out this sub-action as it happens in most other actions of the SSF. This is the key piece of the SSF in establishing and tracing provenance from source code to artifact of a given project. This provenance can then

3 By using identity federation, it is possible for verification to be achieved without actual proof of possession of the keys. In cases where this is the method of choice, public signing keys will not need to be provided.

4 Note that for Reproducible Builds, the timestamp may be extra metadata included alongside the document so that the content can be checked for reproducibility.

be used in conjunction with other tooling and auditing to better make claims on the veracity of software.

In general the following is how the action works though there might be a few caveats specified in the other actions:

Initial Setup:

1. Spin up a node
2. Node Attestor establishes identity of node.

Action Steps:

1. Pipeline or Pipeline task is triggered/orchestrated
2. Workload Attestor establishes identity of Pipeline or task
3. Pipeline Observer captures metadata for Pipeline or task.
   a. This includes inputs, timestamps, outputs, as well as other metadata
4. Pipeline Observer signs metadata with key or cert based on identity provided by Workload Attestor

All Stages: Admissions Control for the SSF itself

Actors:

- Scheduling and Orchestration Platform
- Pipeline
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage
- Admission Controller
- Artifact Storage

As noted in the discussion of the Admissions Controller above, both build workers (the containers performing pipeline steps) and intermediate artifacts (the outputs of previous steps passed along to the next steps in a build) should be verified before they are admitted into the SSF. This should be part of every stage in the pipeline.

Stage 1: Secure the data flow in the pipeline

As tasks execute inside a pipeline, they typically produce some new artifacts like an image, binary or evidence report. These artifacts are then consumed by subsequent tasks to perform their respective functions. Such sharing of artifacts between tasks normally achieved through shared storage resources. It is important to regulate access to these shared resources across tasks.

To achieve this objective, avoid using a single storage workspace across all tasks in the pipeline. Create multiple storage workspaces that are exclusively shared between the tasks that need to communicate some data/results. For instance, for a simple pipeline shown below, avoid using a single shared storage for all tasks and use exclusive storage sharing. And when possible set access-policies (RW, RO) while mounting these storage in the tasks.

Stage 2: Configuration of Pipeline

Actors:

- Developer
- Tech Lead
- Security Engineer
- Scheduling and Orchestration Platform
- Pipeline Platform

The primary component configured as part of normal operation of the SSF is the Pipeline. Both creation of a new Pipeline as well as modification of an existing Pipeline have similar modes of operation and so this section represents both.

The secure software factory expects that you store pipeline configuration as code and that the code is stored in a secure source code repository with adequate controls.See both “Source Code” and “Pipeline Definitions” in the inputs section above for more information about the SSF’s expectations regarding both of these types of inputs. The goal of these controls is to make sure that the pipeline definition

itself has trustworthy provenance. In a cloud native context, these components are often deployed as containers and treated as artifacts in their own right. Ensuring we have adequate provenance for those components increases our assurance about the provenance of the artifacts they build.

When configuring and designing the pipeline, there consider that:

- Individual tasks and steps should have limited in scope and are well defined. sing templates and linting rules during the development of the pipeline itself aids this.
- Configuring the pipeline to respond automatically to well-defined triggers in the Software Development Life Cycle.

Stage 3: Trigger Pipeline

Actors:

- Developer
- Scheduling and Orchestration Platform
- Pipeline Platform
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage

The first step in the SSF is that something triggers a pipeline. This can be a manual, event-driven, or timed trigger. Common triggers are web hooks and manual triggering through an API call or dashboard. Regardless of what trigger you use, parameters to the trigger that can affect the build, like compile flags, should be restricted in order to minimize the attack surface.

The SSF secures this by capturing and validating the inputs and other metadata like timestamps through the Pipeline Observer. This is then signed by a key or certificate provided by the Workload Attestor that is associated with the identity of the workload. The Workload Attestor then has its identity attested to by the Node Attestor. This signed metadata is then pushed to Metadata Storage where it becomes a supply chain link that other parts of the SSF can link to and can later be used to validate and audit veracity of the artifact(s) built in the SSF.

Stage 4: Ingest Source for Project

Actors:

- Scheduling and Orchestration Platform
- Pipeline Platform
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage
- External Source Code Control

The first step in the SSF is that something triggers a pipeline. This can be a manual, event-driven, or timed trigger. Common triggers are web hooks and manual triggering through an API call or dashboard. Regardless of what trigger you use, parameters to the trigger that can affect the build, like compile flags, should be restricted in order to minimize the attack surface.

Stage 5: Ingest Dependencies for Project

Actors:

- Scheduling and Orchestration Platform
- Pipeline Platform
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage
- Internal/External dependency repos

After ingesting source code, the next step is to download the dependencies for the artifact you are building. This is a separate step from the ingestion of the source for a couple of reasons. In line with the build best practices in this document(reference here) and the CNCF Supply Chain Security Best Practices whitepaper, the pipeline steps should be kept as minimal and atomic as possible. In the case of this step, it allows you to download the source and sign it as a single atomic action. Then you can validate after downloading dependencies that the source code wasn't changed by a compromised dependency install. Some package managers can run arbitrary execution actions on the system without adequate controls.

Once dependencies are installed on shared storage they are hashed and that metadata is signed and pushed to Metadata Storage.

Stage 6: Run Build for Project

Actors:

- Scheduling and Orchestration Platform
- Pipeline Platform
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage

This is arguably the most critical step of the Pipeline. This step is the one that performs common "build" actions to generate an artifact such as compilation, building an image, etc. The build is a common attack vector in supply chain attacks, therefore it is crucial to keep this step atomic, minimal, and more importantly, hermetic.[5] When available you should strive for reproducible builds.[6]

The build process performs code compilation or transformation (e.g. source code to byte code for compiled languages). Leverage pipeline observers to record the command, options and parameters used during compilation.

Given the need for the build to be hermetic, the task running the build should have no network or most other external capabilities and have build parameters pushed at the task level. The only external access the task should have is to shared storage containing the source and dependencies required. The build must write the artifact to new shared storage explicitly for the artifact. More information on this can be found in the "Build Worker Environment and Commands" section of the Supply Chain Best Practices paper.

After the operation of the build the metadata associated with the build, e.g. input parameters, hash of produced artifact, etc. are signed and pushed to Metadata Storage.

Stage 7: Publish Artifact

Actors:

- Scheduling and Orchestration Platform
- Pipeline Platform
- Pipeline Observer
- Node Attestor
- Workload Attestor
- Metadata Storage
- Artifact Storage

In the final build stage, compiled artifacts are packaged into appropriate distribution format (container image, rpm, tar.gz, etc.). Artifacts may be published to an artifact store, external from the SSF. Artifacts must be hashed and signed along with any applicable metadata that can be pulled from the artifact. That signed metadata is then stored in Metadata Storage.

5 A hermetic build is a self-contained build. This means that all the inputs must be defined in the build. This is often done by pinning all dependencies based on cryptographic hashes. This also means the build should have no access to any resources not defined in the build, most commonly the network.

6 A reproducible build is a build where given identical inputs the build generates identical bit for bit outputs.

Appendix A: Inputs and Outputs Summary

Inputs

Inputs of the SSF	Assumptions/Recommendations About Those Inputs	What We're Not Specifying in this Version
Source Code	• Version controlled with stored history • Commits are signed • History cannot be overwritten (no force merges) • Has an appropriate testing and code review regime in place	• Where code is hosted • Specific test types or tooling to use
Dependencies	• Defined with version and immutable reference (e.g. hash) constraints • (ideally) something approximating an SBOM and/or source of provenance • Have appropriate update and review procedures in place	• Format of SBOM/Provenance for dependencies • Types of testing to perform on dependencies • Source repositories allowed for dependencies
User Credentials	• Users use MFA • Users use SSH or PATs for repository access • Users have signing certificates	• User roles/permissions • Key/Certificate Rotation Policy • How users are authenticated
Machine/Workload Credentials	• Automatically rotated short-lived credentials to identity application services	
Signing keys	• Meet or exceed current NIST guidelines for the type of key/certificate with regards to length, randomness, etc.	• How keys/certificates are generated and by whom? • How keys/certificates are distributed and by whom?
Pipeline Definitions	• Maintained as Infrastructure-as-Code/Pipeline-as-Code meeting all the above specs for Source Code, Dependencies, User Credentials, etc. • Builds task definitions	
Build Images	• Either bootstrapped or created by the SSF • Signatures are verified by the SSF Admission Controller	

Outputs

Ouputs of the SSF	Assumptions/Reccomendations About those Inputs	What We're Not Specifying in this Version
Artifacts (Requires addition)	• Includes signed and validated metadata in an appropriate storage mechanism	• What storage mechanism to use (unless we find there are really good reasons to recommend one)
Public Signing Keys	• Meet or exceed current NIST guidelines for the type of key/certificate with regards to length, randomness, etc.	• How keys/certificates are generated and by whom? • How keys/certificates are distributed and by whom?
Metadata Documents (Requires addition)	• Must be in machine readable format • Must be signed except in cases where signing is not supported by the tooling • Should include reference to other artifacts allowing for chaining of metadata documents	• What formats to use

Appendix B: Best practices x Reference Architecture

Stage	Practice	Categories
Securing the Source Code	Verification: Commits and tags are signed	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Enforce full attestation and verification for protected branches	**Assurance**: High **Risk**: High
	Automation: Secrets are not committed to the source code repository unless encrypted	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: The individuals or teams with write access to a repository are defined	**Assurance**: High **Risk**: High
	Automation:Automate software security scanning and testing	**Assurance**: Moderate to high **Risk**: Moderate to high
	Controlled Environments: Establish and adhere to contribution policies	**Assurance**: Moderate to high **Risk**: Moderate to high
	Controlled Environments: Define roles aligned to functional responsibilities	**Assurance**: Moderate to high **Risk**: Moderate to high
	Controlled Environments: Enforce an independent four-eyes principle	**Assurance**: Moderate to high **Risk**: Moderate to high
	Controlled Environments: Use branch protection rules	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication: Enforce MFA for accessing source code repositories	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication: Use SSH keys to provide developers access to source code repositories	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication: Have a Key Rotation Policy	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication: Use short-lived/ephemeral credentials for machine/service access	**Assurance**: Moderate to high **Risk**: Moderate to high
Securing the Materials	Verification: Verify third party artifacts and open source libraries	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Require SBOM from third party suppliers	**Assurance**: Moderate to high **Risk**: High
	Verification: Track dependencies between open source components	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Build libraries based upon source code	**Assurance**: High **Risk**: High
	Verification: Define and prioritize trusted package managers and repositories	**Assurance**: High **Risk**: High

Stage	Practice	Categories
	Verification: Generate an immutable SBOM of the code	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Scan software for vulnerabilities	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Scan software for license implications	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Run software composition analysis on ingested software	**Assurance**: Moderate to high **Risk**: Moderate to high
Securing the Build Pipelines	Verification: Cryptographically guarantee policy adherence	**Assurance**: High **Risk**: High
	Verification: Validate environments and dependencies before usage	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Validate runtime security of build workers	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Validate Build artifacts through verifiably reproducible builds	**Assurance**: High **Risk**: High
	Reproducible Builds: Lock and Verify External Requirements From The Build Process	**Assurance**: Moderate to high **Risk**: Moderate to high
	Reproducible Builds: Find and Eliminate Sources Of Non-Determinism	**Assurance**: Moderate to high **Risk**: Moderate to high
	Reproducible Builds: Record The Build Environment	**Assurance**: High **Risk**: High
	Reproducible Builds: Automate Creation Of The Build Environment	**Assurance**: High **Risk**: High
	Reproducible Builds: Distribute Builds Across Different Infrastructure	**Assurance**: High **Risk**: High
	Automation: Build and related continuous integration/continuous delivery steps should all be automated through a pipeline defined as code	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Standardize pipelines across projects	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Provision a secured orchestration platform to host software factory	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Build Workers Should be Single Use	**Assurance**: High **Risk**: Moderate
	Controlled Environments: Ensure Software Factory has minimal network connectivity	**Assurance**: High **Risk**: High
	Controlled Environments: Segregate the Duties of Each Build Worker	**Assurance**: High **Risk**: High
	Controlled Environments: Pass in Build Worker Environment and Commands	**Assurance**: High **Risk**: High

Stage	Practice	Categories
	Controlled Environments: Write Output to a Separate Secured Storage Repo	**Assurance**: High **Risk**: High
	Secure Authentication/Access: Only allow pipeline modifications through "pipeline as code"	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication/Access: Define user roles	**Assurance**: Moderate to high **Risk**: Moderate to high
	Secure Authentication/Access: Follow established practices for establishing a root of trust from an offline source	**Assurance**: High **Risk**: High
	Secure Authentication/Access: Use short-lived Workload Certificates	**Assurance**: High **Risk**: High
Securing the Artifacts	Verification: Every step in the build process should be signed/attested for process integrity	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Every step in the build process should verify the previously generated signatures	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Use TUF/Notary to manage signing of artifacts	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Use a store to manage attestations	**Assurance**: Moderate to high **Risk**: Moderate to high
	Controlled Environments: Limit which artifacts any given party is authorized to certify	**Assurance**: High **Risk**: High
	Controlled Environments: Rotation and revokation of private keys should be supported	**Assurance**: High **Risk**: High
	Controlled Environments: Use a container registry that supports OCI image-spec images	**Assurance**: High **Risk**: High
	Encryption: Encrypt artifacts before distribution & ensure only authorized platforms have decryption capabilities	**Assurance**: High **Risk**: High
Securing Deployments	Verification: Ensure clients can perform Verification of Artifacts and associated metadata	**Assurance**: Moderate to high **Risk**: Moderate to high
	Verification: Ensure clients can verify the "freshness" of files	**Assurance**: Moderate to high **Risk**: Moderate to high
	Automation: Use a framework for managing software updates	**Assurance**: High **Risk**: High

Appendix C: Glossary

SSF — Secure Software Factory
SAST — Static Application Security Testing
DAST — Dynamic Application Security Testing
NIST — National Institute of Standards and Technology
SBOM — Software Bill of Materials
MFA — Multi-factor Authentication
SSH — Secure Shell
PAT — Personal Access Token
API — Application Programmable Interface
.rpm — Redhat Package format
.deb — Debian package format
.tar.gz — Compression and packaging format (tar — abbreviation of tape archive)

Contributors

Aditya Sirish A Yelgundhalli (NYU)

Alexander Floyd Marshall (Raft)

Andres Vega (VMware)

Andrew Block (Red Hat)

Aradhna Chetal (TIAA)

Axel Simon (Red Hat)

Brandon Lum (Google)

Brandon Mitchell (IBM)

Cole Kennedy (TestifySec)

Dan Papandrea (Sysdig)

Glaicimar Aguiar (Hewlett Packard Enterprise)

Jason Hall (Red Hat)

John Kjell (VMware)

Marina Moore (NYU)

Matt Moore (Chainguard)

Michael Lieberman (Citi)

Parth Patel (IBM)

Priya Wadhwa (Chainguard)

Shripad Nadgowda (IBM T.J. Watson Research Center)

Acknowledgements

The Cloud Native Computing Foundation supported the creation of this reference architecture. As with the "Best Practices for Supply Chain Security", the authors followed a "collaborative knowledge production" methodology. This effort took place over the span of five months of weekly online meetings. The majority of authors are members of the CNCF Technical Advisory Group for Security, which you can join. Go to the TAG repository site.

This was a remarkable collaboration between large technology companies and startups.

The coordination and facilitation was provided by Andres Vega (VMware), Brandon Lum (Google), Dan "Pop" Papandrea (Sysdig) and Michael Liebermann (Citi).

We'd also like to thank a number of contributors from whom we had excellent input and feedback and as leading practitioners in the field did much of the work that we write about in this document:

Aeva Black

Allan Friedman

Dan Lorenc

David Wheeler

Ed Warnicke

Emily Fox

Frederick Kautz

Jacques Chester

Jonathan Meadows

Justin Cormack

Remy Greinhofer

Tiffany Jordan

References

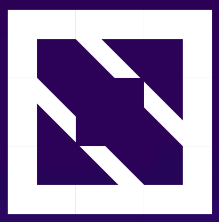
Software Factory: https://en.wikipedia.org/wiki/Software_factory

CNCF TAG-Security: https://github.com/cncf/tag-security

CNCF Supply Chain Security Paper: https://github.com/cncf/tag-security/blob/main/supply-chain-security/supply-chain-security-paper/CNCF_SSCP_v1.pdf

CNCF Cloud Native Security Whitepaper: https://github.com/cncf/tag-security/blob/main/security-whitepaper/CNCF_cloud-native-security-whitepaper-Nov2020.pdf

Kubernetes Policy Management Whitepaper: https://github.com/kubernetes/sig-security/blob/main/sig-security-docs/papers/policy/CNCF_Kubernetes_Policy_Management_WhitePaper_v1.pdf



CLOUD NATIVE
COMPUTING FOUNDATION