NIST Special Publication 800
NIST SP 800-204D

Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines

Ramaswamy Chandramouli
Frederick Kautz
Santiago Torres-Arias

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204D

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Special Publication 800
NIST SP 800-204D

Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

Frederick Kautz
TestifySec

Santiago Torres-Arias
Electrical and Computer Engineering Department
Purdue University

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204D

February 2024

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

Certain commercial equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

Authority
This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

NIST Technical Series Policies
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History
Approved by the NIST Editorial Review Board on 2024-01-31

How to Cite this NIST Technical Series Publication:
Chandramouli R, Kautz F, Torres-Arias S (2024) Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204D. https://doi.org/10.6028/NIST.SP.800-204D

Author ORCID iDs
Ramaswamy Chandramouli: 0000-0002-7387-5858

Contact Information

sp800-204d-comments@nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

Additional Information

Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/204/d/final, including related content, potential updates, and document history.

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

The predominant application architecture for cloud-native applications consists of multiple microservices, accompanied in some instances by a centralized application infrastructure, such as a service mesh, that provides all application services. This class of applications is generally developed using a flexible and agile software development paradigm called DevSecOps. A salient feature of this paradigm is the use of flow processes called continuous integration and continuous deployment (CI/CD) pipelines, which initially take the software through various stages (e.g., build, test, package, and deploy) in the form of source code through operations that constitute the software supply chain (SSC) in order to deliver a new version of software**.** This document outlines strategies for integrating SSC security measures into CI/CD pipelines.

Keywords

actor; artifact; attestation; CI/CD pipeline; package; provenance; repository; SBOM; SDLC; SLSA; software supply chain.

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Patent Disclosure Notice

NOTICE: ITL has requested that holders of patent claims whose use may be required for compliance with the guidance or requirements of this publication disclose such patent claims to ITL. However, holders of patents are not obligated to respond to ITL calls for patents and ITL has not undertaken a patent search in order to identify which, if any, patents may apply to this publication.

As of the date of publication and following call(s) for the identification of patent claims whose use may be required for compliance with the guidance or requirements of this publication, no such patent claims have been identified to ITL.

No representation is made or implied by ITL that licenses are not required to avoid patent infringement in the use of this publication.

Table of Contents

Acknowledgments

The authors would like to express their thanks to Isabel Van Wyk of NIST for her detailed editorial review, both for the public comment version as well as for the final publication.

Executive Summary

Cloud-native applications are made up of multiple loosely coupled components called microservices. This class of applications is generally developed through an agile software development life cycle (SDLC) paradigm called DevSecOps, which uses flow processes called Continuous Integration/Continuous Delivery (CI/CD) pipelines.

Analyses of recent software attacks and vulnerabilities have led both government and private-sector organizations involved in software development, deployment, and integration to focus on the activities involved in the entire SDLC. This collection of activities constitutes the software supply chain (SSC), and the integrity of the individual activities contributes to the overall security of an SSC. Threats can arise from attack vectors unleashed by malicious actors during SSC activities as well as defects introduced when due diligence practices are not followed by legitimate actors during the SDLC.

Executive Order (EO) 14028, NIST's Secure Software Development Framework (SSDF) [2], other government initiatives, and industry forums have discussed the security of SSC and provided a roadmap to enhance the security of all deployed software. This document uses this roadmap as the basis for developing actionable measures to integrate the various building blocks of SSC security assurance into CI/CD pipelines to enhance the preparedness of organizations to address SSC security in the development and deployment of cloud-native applications. To demonstrate that the SSC security integration strategies for CI/CD pipelines meet the objectives of SSDF, a mapping of these strategies to the high-level practices in the SSDF has also been provided.

Building a robust SSC security edifice requires various artifacts, such as a software bill of materials (SBOM) and frameworks for the attestation of software components. Since the specification of these artifacts, their mandatory constituents, and the requirements that processes using them must satisfy are continually evolving through projects in government organizations and various industry forums, they are beyond the scope of this document.

1. Introduction

Cloud-native applications typically consist of multiple loosely coupled services or microservices and are sometimes accompanied by an integrated application service infrastructure, such as a service mesh. The applications are developed through an agile software development life cycle (SDLC) paradigm called DevSecOps, which uses flow processes called Continuous Integration/ Continuous Delivery (CI/CD) pipelines. The security of applications during runtime is ensured through various security measures, such as assigning unique service identities for microservices and subjects that invoke those services and policy enforcement through proxies. However, sophisticated attacks on software have been carried out through the stealthy introduction of attack vectors during various activities in the SDLC, which collectively constitute the software supply chain (SSC). Thus, in the context of cloud-native applications, SSC security assurance measures must be integrated into CI/CD pipelines.

1.1. Purpose

This document outlines strategies for integrating SSC security assurance measures into CI/CD pipelines to protect the integrity of the underlying activities. The overall goal is to ensure that the CI/CD pipeline activities that take source code through the build, test, package, and deployment stages are not compromised.

1.2. Scope

SSC security assurance measures use various artifacts, such as a software bill of materials (SBOM) and frameworks for the attestation of software components. The specification of these artifacts, their mandatory constituents, and the requirements that processes using them must satisfy are continually evolving through projects in government organizations and various industry forums and are, therefore, beyond the scope of this document. Rather, this document focuses on actionable measures to integrate various building blocks for SSC security assurance into CI/CD pipelines to enhance the preparedness of organizations to address SSC security in the development and deployment of their cloud-native applications.

1.3. Target Audience

This document is intended for a broad group of practitioners in the software industry, including site reliability engineers, software engineers, project and product managers, and security architects and engineers.

1.4. Relationship to Other NIST Documents

This document is part of the NIST Special Publication (SP) 800-204 series of publications, which offer guidance on providing security assurance for cloud-native applications that are developed and deployed using the DevSecOps SDLC paradigm that uses CI/CD pipelines. SP 800-204C [1]

discusses DevSecOps, which is an agile software development paradigm for cloud-native applications that focuses on the various types of code involved in microservices-based applications that are supported by a service mesh infrastructure. SP 800-218 [2] provides a comprehensive list of high-level practices and tasks for ensuring SSC security under the Secure Software Development Framework (SSDF) based on the directives in Executive Order (EO) 14028 [3]. Other documents in the SP 800-204 series outline the mechanisms for enforcing various types of access controls for inter-service calls in the microservices environment during runtime.

This document presents strategies for integrating SSC security into CI/CD pipelines through the identification of workflow tasks that can meet the goals of the various high-level practices outlined in the SSDF. Not all practices and tasks outlined in the SSDF may be applicable to the environment under discussion in this document – i.e., cloud-native applications developed using the DevSecOPs SDLC paradigm with CI/CD pipelines, representing a specific application architecture and SDLC, respectively. The SSDF is agnostic to both application architecture and the SDLC paradigm. However, to demonstrate that the SSC security integration strategies for CI/CD pipelines meet the objectives of SSDF, Appendix A provides a mapping of these strategies to the high-level practices in the SSDF. However, tasks relating to secure software design and the enterprise-level vulnerability management strategies are beyond the scope of this document and these are indicated in Appendix B.

1.5. Document Structure

This document is organized as follows:

- Section 2 presents a series of definitions for modelling and understanding software supply chains and their compromises.
- Section 3 provides a broad understanding of common risk factors and potential mitigation measures with a particular focus on the software developer environment.
- Section 4 provides the background for CI/CD pipelines, the broad security goals of the processes involved, and the entities that need to be trusted.
- Section 5 outlines strategies for integrating SSC security assurance measures into CI/CD pipelines.
- Section 6 provides a summary and conclusions.
- Appendix A provides a mapping of the SSC security integration strategies for CI/CD pipelines to the SSDF's high-level practices.
- Appendix B provides a justification for the omission of certain measures related to SSDF practices in this document.

2. Software Supply Chain (SSC) — Definition and Model

2.1. Definition

Most activities in the SSC strongly affect the resulting software product. As such, the security of each individual activity is paramount for the security of the end result. This includes both the integrity of the activities themselves as well as the assurance that all activities were carried out and — conversely — that no unauthorized activities were injected into the chain.

While software composition (e.g., dependency management) is under the purview of software supply chain activities, other often overlooked activities are central to the software supply chain. This includes writing source code; building, packaging, and delivering an application; and repackaging and containerization.

An SSC attack can take on several forms, such as:

- Subverting, removing, or introducing a step within the SSC to maliciously modify or sabotage the resulting software product
- Stealing credentials from the build system to mint and sign unauthorized malicious software
- Causing naming collisions

SSC attacks can have a wide range of consequences that affect the correctness, integrity, or availability of a software product (e.g., making upstream dependencies unavailable). In practice, attackers often target the activities mentioned above to implant backdoors and subsequently compromise a target (i.e., end product) or exfiltrate sensitive information once the application is delivered.

SSC security should also account for discovering and tracking software security defects rather than simply mitigating attacks. To facilitate this, the software bill of materials (SBOM) must be shared with end users so that they can build inventories of software components. However, while SBOMs enable the identification of components and provenance, they do not provide enough information to address vulnerabilities nor content to address software defects. Hence, SBOMs alone cannot be used for vulnerability management. They simply provide the list of components to focus on when addressing vulnerabilities or defects in software.

2.2. Economics of Security

SSC attacks have two fundamental properties that make them appealing to attackers. First, they allow attackers to infiltrate highly-regulated environments through less secure but legitimate channels. Second, due to the highly-interconnected nature of supply chains, they allow for widespread damage in a short period of time.

Insufficient care in operating highly regulated environments throughout the SDLC often allows motivated attackers to identify weak spots in the chain. In the case of SOLORIGATE [4], for

example, attackers identified a single point of compromise that delivered software to multiple government agencies. Such attacks are also stealthy because they typically propagate through legitimate channels, such as software updates, which allows for widespread damage to users of the target software. These attacks are successful because of the significant amount of implicit trust present in these legitimate channels, and a first defensive measure calls for the removal of this implicit trust. Since attackers typically seek this avenue to obtain short-term benefits, widespread attacks of this nature often rely on the use of private crypto miners and cryptojackers. This is evidenced in the prevalence of these vectors existing in breadth-first approaches, such as typo and combosquatting attacks. Regardless of the motivations of the attackers, both vectors highlight the possibility of devastating impacts when attacks are successful.

2.3. Governance Model

Due to the distributed nature of an SSC, multiple practices, developer cultures, security and quality expectations, and legislative frameworks exist. As a consequence, there is no unified governance model, and these distinct models often overlap.

2.4. SSC Model

At a high level, an SSC is a collection of steps that create, transform, and assess the quality and policy conformance of software artifacts. These steps are often carried out by different actors who use and consume artifacts to produce new artifacts. For example, a build step uses a series of artifacts as tools (e.g., a compiler and a linker) and consumes artifacts (i.e., source code) to produce a new artifact (i.e., the compiled binary).

Without a loss of generality, this same definition can be applied to other actions, such as writing code, packaging an application inside of a container, and performing quality assurance. This definition also encompasses more activities than are colloquially considered. That is, it includes elements of secure software development, secure build systems, and dependency management. These elements collectively define the SSC model.

While this simplified model can accommodate multiple activities, mitigations and attacks may surface in different, nuanced ways for each activity.

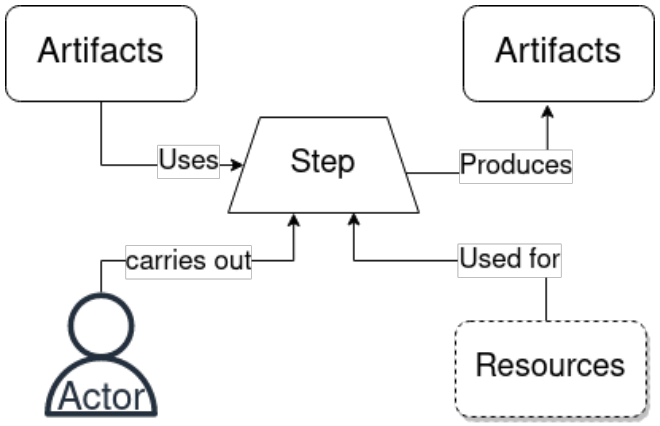

Fig. 1. Interaction between the different elements[1] of a software supply chain (SSC) step[2]

2.4.1. Software Supply Chain Defects

Much like software defects (i.e., bugs), defective artifacts can propagate throughout an SSC and affect its security posture. A noteworthy example of such a defect is Log4Shell [5], where a vulnerability in a highly-used software artifact allowed attackers to compromise a large number of targets with very little effort.

If software is used in a manner that it was not originally intended or configured for, it may result in an insecure state. However, while the line between a defect and an attack is often blurred in the context of SSC, the guiding principle is that of intent — that is, whether or not the upstream actor intended to exploit that defect. In the context of software engineering, not all defects are vulnerabilities, regardless of intent. Vulnerabilities may be present for other reasons, and that presence does not guarantee exploitation, which is what defines an attack. Malicious actors complete the defect-attack chain by intentionally introducing weaknesses that they can later exploit.

2.4.2. Software Supply Chain Attacks

In contrast to defects, an SSC attack is when a malicious party tampers with the steps, artifacts, or actors within the chain to compromise the consumers of a software artifact down the line.

Explicitly, an SSC attack is a three-stage process:

1. **Artifact, step, or actor compromise:** An attacker compromises an element of the SSC (see Fig. 1) to modify an artifact or the information of such.
2. **Propagation:** The attack propagates throughout the chain.

[1] An "actor" can also be a non-human, such as a build orchestrator.
[2] An SSC step stands for an SSC activity (e.g., build).

3. **Exploitation:** The attacker exploits the target to achieve their goals (e.g., exfiltration of data, cryptojacking).

3. SSC Security — Risk Factors and Mitigation Measures

This section considers the various risk factors that are applicable to the SDLC environment and the mitigation measures that can counter those risks.

3.1. Risk Factors, Targets, and Types of Exploits in an SSC

The risk factors in an SSC typically include:

- Vulnerabilities in the developer environment
- Threat actors
- Attack vectors
- Attack targets (i.e., assets)
- Types of exploits

3.1.1. Developer Environment

Developer workstations and their environments present a fundamental risk to the security of an SSC and should not be trusted as part of the build process since they are at risk of compromise. Mature SDLC processes accept code and assets into their software configuration management (SCM) mainline and versions branches only after code reviews and scanners are in place.

3.1.2. Threat Actors

Threat actors are generally:

- External attackers who seek privileged access to an SSC
- Disgruntled employees or contractors who perpetuate insider threats

External attackers may include foreign adversaries, criminal organizations, and cyber-activists who target an SSC for various reasons, such as espionage or sabotage. Internal attackers pose a significant risk, as they may have insider access to sensitive information — often using legitimate access rights — that allow them to launch attacks or steal confidential information. Additionally, both categories of threat actors may use a variety of techniques to compromise the SDLC environment and steal or manipulate software, such as phishing, malware, social engineering, and physical access. Therefore, companies should be aware of these risks and take appropriate measures (see Sec. 3.2) to secure their SSC.

Non-malicious threat actors may also impact the security of supply chains, such as a software engineer who inadequately manages secrets through a lack of tooling or purposeful subterfuge for ease of use. Organizations should be aware of these situations and take suitable measures to avoid such practices.

3.1.3. Attack Vectors

Attack vectors in an SSC include:

- Malware
- Code reuse or the ingest of libraries and dependencies
- Social engineering
- Network-based attacks
- Physical attacks

Attack vectors can originate from various sources, including malware attacks on developer workstations, social engineering attacks on developers, network-based attacks on the development environment, and physical attacks on the hardware or networks used by developers. These different attack vectors require distinct countermeasures, including endpoint protection software, network security controls, access control policies, and physical security measures. Companies should identify potential risks and vulnerabilities, assess their security posture, and implement appropriate defensive measures to mitigate threats to their SDLC environment.

In the case of ingested code, it is essential to verify the provenance information of the component being used to ensure that it is what it says it is and is coming from an expected source. Mitigations for this involve caching or curating packages and components for preferred use.

3.1.4. Attack Targets (Assets)

The assets targeted under an SSC may include:

- Source code
- Credentials
- Sensitive data
- Internal operations
- Build systems

A software developer's workstation typically contains various assets, including source code, credentials, and access to sensitive information, such as personally identifiable information (PII), protected health information (PHI), intellectual property (IP), cryptographic materials (e.g., software artifact signing keys), and proprietary information. Companies should identify critical assets and implement controls to protect them from unauthorized access, such as access controls, multi-factor authentication, encryption of data at rest and in transit, and data loss prevention (DLP) measures.

3.1.5. Types of Exploits

Exploits in the context of attack vectors and targeted assets in an SSC typically include:

- Injection of vulnerable or malicious dependencies into an SSC
- Stolen credentials that grant access to other systems
- Injection of malicious or vulnerable code into repositories
- Stealing secrets by submitting merge requests

Threat actors may seek to compromise various components of the SDLC process, including source code, testing environments, development tools, and build pipelines. They may introduce vulnerabilities, malware, or stolen credentials to gain access to other systems or compromise sensitive data. Such threats can result in financial losses, reputational damage, physical damage and legal consequences.

To inject malicious code into repositories, attackers may perform an operation called “forking,” which allows the attacker to copy some repository and freely make modifications outside of the original project. The attacker then initiates a pull request to merge the forked project with the original project. If the project maintainer accepts the request without properly and adequately reviewing the changes and determining them to be suitable, they will merge them into the original project, thus introducing malicious code into the repository.

When open-source code is used, an artifact or package is often pulled from a repository based on the reputation of the developer or the repository. However, there is no guarantee that pulled code is the same software that the developer authored and checked into their source-code repository. The following actions could have potentially occurred, resulting in a lack of assurance or an inability to trust the code:

- The source code could have been modified.
- Vulnerabilities could have been introduced due to an insecure build system.
- Checks, such as scanning and various types of tests (e.g., static, dynamic, or interactive), may have been bypassed in the CI/CD process.
- The repository owner may have improperly configured the repository, allowing malicious actors to submit pull requests with the intention of stealing secrets configured within a CI/CD pipeline.

3.2. Mitigation Measures

A secure SDLC environment can reduce the likelihood of security incidents and ensure the confidentiality, integrity, and availability of software assets and systems. It is crucial to assess security risks and implement appropriate defensive measures to protect software supply chains against compromise.

The following generic mitigation measures are applicable to the entire SDLC but are particularly relevant to an SSC:

- Patch management
- Dependency management
- Authentication and authorization
- Malware protection
- Secure SDLC
- Data protection
- Physical security
- Audit and monitoring
- Adherence to applicable security standards (e.g., regulatory requirements)

Organizations can implement various controls to mitigate risks to their SDLC environment, including regular patch management, robust authentication, granular authorization, malware protection, secure SDLC practices, data protection measures, physical security controls, and auditing and monitoring tools. They should regularly assess their security posture, identify potential weaknesses and vulnerabilities, and implement appropriate defensive measures to address them. Organizational network policies that account for and actively block maliciously known content-serving domains can reduce the use of software from non-curated or undesired locations. Another integral part of SSC security involves capturing the dependencies (e.g., package name, version) of the artifacts in a central repository. Organizations should also ensure that their SDLC environment remains compliant with various security and other relevant standards, such as the Open Worldwide Application Security Project (OWASP) Top Ten, SP 800-53, Health Insurance Portability and Accountability Act (HIPAA), and Payment Card Industry Data Security Standard (PCI DSS).

The choice of a mitigation approach will depend on the organization's customized threat model. However, all developer systems should meet a predefined minimum baseline for security to ensure that the operating system and applications are kept up to date with the latest security patches, individual and unshared user accounts are adequately protected, and proper access controls are enforced when interacting with SCM.

3.2.1. Baseline Security

Independent and open-source developers will need to follow best practices to protect their own systems. Government and enterprise environments should establish and adhere to a well-defined security policy that meets regulatory requirements and industry best practices. Since the development of such a policy is out of scope for this document, readers should refer to SP 800-53r5 (Revision 5) [6] for a more complete treatment of this topic.

The following are some baseline security measures that should be adopted when integrating open-source software (OSS) components into any enterprise project:

- The security team should establish a policy for trusted sources of OSS (e.g., allow lists) that includes reviewing minimum coding requirements, reputational standards, and distributing source code in a digitally signed package.
- The security team should approve the merging of unverified sources of OSS.
- Developers should download OSS as source code rather than pre-compiled libraries or binaries, when available.
- Developers should verify digital signatures, run vulnerability scans, check for recent updates on newly downloaded OSS's source-code packages, and generate an SBOM with dependency scanning on the first commit in order to identify the risks of any upstream or downstream dependencies within the OSS.
- Artifacts should be scanned in internal repositories for newly discovered or identified defects and the ability to stop their use in builds based on criticality.
- CI/CD processes should be audited regularly, and automation should be introduced wherever possible to improve the performance of activities and operations.
- There should be isolated CI/CD environment and elevated administrator credentials for the deployment of applications in clouds.
- There should be enhanced real-time monitoring and alerting mechanisms to detect suspicious activities in CI/CD servers, especially activities that might indicate the exfiltration of sensitive data or the tampering of builds.

3.2.2. Controls for Interacting With SCM Systems

Developers use their workstations to create, edit, and test source code. This process requires developers to pull source code from the SCM, modify the source code, and submit changes (i.e., patches) back to the SCM. The proposed changes should adhere to the SDLC processes defined by the organization. Pull access to the software depends on the policies of the software project in question (e.g., open-source projects typically allow anyone to pull, replicate, modify, and share the source code with minimal or copyleft restrictions). Proprietary software vendors often enforce strict rules that describe who is allowed to access the source code and under what conditions. In all cases, write access to the SCM should be considered a high risk and tightly controlled. A mature SDLC process allows developers to propose patches to the SCM, but another developer should perform a code review before the patch is merged. Code analysis tools should be implemented to catch common mistakes, but care should be taken to not inundate the developers with too many false positives to prevent alert fatigue.

4. CI/CD Pipelines — Background, Security Goals, and Entities to be Trusted

DevSecOps is an agile paradigm used for the development and deployment of cloud-native applications. This paradigm consists of a series of stages that takes code from variously sourced repositories (e.g., first-party or in-house, third parties or open-source/commercial) to perform tasks or activities, such as building, packaging, testing, and deploying.

In this document, the term "artifacts" denotes source code as well as the things generated from it, such as builds and packages. Each of the artifacts is associated with an owner. The logical containers that hold these artifacts are called repositories. The build process is based on application logic-driven dependencies and generates builds using many individual source-code artifacts that are stored in build repositories. The build artifacts are tested and used to generate packages whose artifacts are then stored in designated repositories and scanned before being deployed in testing or production environments. These stages and the various tasks performed at each stage are collectively called CI/CD pipelines. In other words, CI/CD pipelines use processes called workflows to transform source artifacts to deployable packages in production environments.

A common approach to SSC security in all of these workflows is to generate as much provenance data as possible. Provenance data are associated with the chronology of the origin, development, ownership, location, and changes to a system or system component, including the personnel and processes that enabled those changes or modifications. The generation of these data should be accompanied by corresponding mechanisms to validate, authenticate, and leverage them in policy decisions.

From the above description of CI/CD pipelines and associated activities, one can identify the set of security assurance measures that need to be added:

- Internal SSC security practices that are applied during the development and deployment of first-party software
- Security practices that are applied with respect to the procurement, integration, and deployment of third-party software (i.e., open-source and commercial software modules)

4.1. Broad Security Goals for CI/CD Pipelines

There are two security goals in the application of SSC security measures or practices in CI/CD pipelines:

1. Actively defend the CI/CD pipeline and build processes.
2. Ensure the integrity of upstream sources and artifacts (e.g., repositories).

The most common approach is to introduce security measures into the CI/CD platform, which allows developers to automate their build, test, and deployment pipelines. There are many open-source and commercial CI/CD platforms available on the market.

4.2. Entities That Need Trust in CI/CD Pipelines — Artifacts and Repositories

Zero trust architectures focus on protecting resources such as hardware systems (e.g., servers), services and the application itself. The entities that access these assets (e.g., users, services, and other servers) are not inherently trusted, and the primary goal of zero trust architecture is to establish this trust. In the context of CI/CD pipelines, the scope of trust is much larger and requires, at a minimum, the following steps:

- The entities involved in performing various SSC activities (e.g., building, packaging, deployment) should be authenticated through the verification of credentials. Based on this authentication, appropriate permissions or access rights are assigned to those entities based on enterprise business policies through a process called authorization.
- The integrity of artifacts and the repositories where they are stored should be ensured through the verification of the digital signatures associated with them. This integrity assurance results in trust.
- The establishment of trust above should be a recurring process throughout the CI/CD system since artifacts travel through various repositories to ultimately become the final product.
- The inputs and outputs of each build step should be verified to ensure that the correct steps have been executed by the expected component or entity.

Table 1 gives examples of entities (i.e., artifacts and repositories) that need to be trusted in typical CI/CD pipelines [7].

Table 1. Top-level entities in the trust chain of typical CI/CD pipelines[3]

Artifact[a]	Repository
First-party code — In house	SCM
Third-party code — Open source or commercial	Artifact managers for language, containers, etc.[b]
Builds	Build repository
Packages	Package repository

[a] Here, the artifacts include only those that go directly into the final software products. Other artifacts used for the security assurance of CI/CD processes (e.g., SBOMs, vulnerability reports, and model registries that use AI models) must also be trusted.
[b] The addition of attestations like VSA does not necessarily imply that the artifact manager is trusted.

[3] The trust chain includes sub-elements, components, workers, attestors, and other mechanisms that must establish and reestablish trust through interactions (i.e., handoffs of inputs and outputs).

5. Integrating SSC Security Into CI/CD Pipelines

In order to outline the strategies for integrating SSC security into CI/CD pipelines, it is necessary to understand the workflows in each of the two pipelines (i.e., CI pipelines and CD pipelines) and their overall security goals.

The prerequisites to activating CI/CD pipelines include the following:

- Harden the CI/CD execution environment (e.g., VM or pod) to reduce its attack surface.
- Define roles for the actors who operate the various CI/CD pipelines (e.g., application updaters, package managers, deployment specialists).
- Identify the granular authorizations to perform various tasks, such as generating and committing code to SCMs, generating builds and packages, and checking various artifacts (e.g., builds and packages) into and out of the repositories.
- Automate the entire CI/CD pipeline through the deployment of appropriate tools. The driver tools for CI and CD pipelines are at a higher level and invoke a sequence of function-specific tools, such as those for code checkouts from repositories, edits and compilation, code commits, and testing (e.g., static application security testing [SAST], dynamic application security testing [DAST], and software composition analysis [SCA] testers). In general, the driver tools or build control plane execute at a higher level of trust than the individual functional steps, such as build.
- Define CI/CD pipeline activities and associated security requirements for the development and deployment of application code; infrastructure as code, which contains details about the deployment platform; and policy as code and configuration code, which specify runtime settings (e.g., Yet Another Markup Language (YAML) files).

5.1. Securing Workflows in CI Pipelines

The workflows in the CI pipeline mainly consist of build operations, push/pull operations on repositories (both public and private), software updates, and code commits.

The overall security goals for the framework used for securely running CI pipelines include:

- The capability to support both cloud-native and other types of applications.
- Standard compliant evidence structures, such as metadata and digital signatures
- Support for multiple hardware and software platforms
- Support for infrastructures for generating the evidence (e.g., SBOM generators, Digital signature generators)

The following subsections consider the SSC security tasks for the various workflows in CI. Although providing support for artifact testing (that generates tamper-proof records of test runs and associated results) is an important security goal, it is beyond the scope of this document.

5.1.1. Secure Build

The following tasks are required to obtain SSC security assurance in the build process:

- Specify policies regarding the build, including (a) the use of a secure isolated platform for performing the build and hardening the build servers, (b) the tools that will be used to perform the build, and (c) the authentication/authorization required for the developers performing the build process.
- Enforce those build policies using techniques such as an agent and policy enforcement engine.
- Ensure the concurrent generation of evidence for build attestation to demonstrate compliance with secure build processes during the time of software delivery.

A common technique for facilitating the second task is to wrap commands from a CI tool with capabilities to gather evidence and ultimately create an evidence trail of the entire SDLC [8]. The first type of evidence is from the build system itself, which should be able to confirm that the tools or processes used are in an isolated environment. This provides internal operational assurance. The second type of evidence that should be gathered consists of the hash of the final build artifact, files, libraries, and other materials used in the artifacts and all events. This is then signed by a trusted component of the build framework that is not under the control of the developers using a digital certificate to create the attestation, which provides verifiable proof of the quality of the software to consumers and enables them to verify the quality of that artifact independently from the producer of the software, thus providing consumer assurance. In this context, the artifact is the build generated by a series of CI process steps.

In the context of "concurrent generation of evidence," the evidence generated should be enabled by a process with a higher level of trust or isolation than the build itself to protect against tampering. The generation of such evidence requires verification within the build as it occurs.

The attestation for a build consists of the following components [9]:

1. Environment attestation: Environment attestation involves an inventory of the system when the CI process happens and generally refers to the platform on which the build process is run. The components of the platform (e.g., compiler, interpreter) must be hardened, isolated, and secure.
2. Process attestation: Process attestation pertains to the computer programs that transformed the original source code or materials into an artifact (e.g., compilers, packaging tools) and/or the programs that performed testing on that software (i.e., code testing tool). It is sometimes difficult for tooling that simply observes CI processes to distinguish between data that should populate the process attestation and data that should populate the materials attestation. A file read by tooling that performs the source transformation may be used to influence the choices that the transformation tool makes, or it might be included in the output of the transformation itself. As a result, the population of the process attestation should be considered "best effort."

3. Materials attestation: Materials attestation pertains to any raw data and can include configuration, source code, and other data (e.g., dependencies).
4. Artifacts attestation: An artifact is the result or outcome of a CI process. For example, if the CI process step involves running a compiler (e.g., GNU Compiler Collection (GCC)) on a source code written in C, the artifact that will result is an executable binary of that source code. If the step involves running a SAST tool on the same source code, the artifact will be the "Scan Result." The step that generated it can be a final or intermediate step. An attestation pertaining to this newly generated product falls under the category of artifacts attestation.

The requirements associated with signed evidence (i.e., attestation) and its storage must include the following:

- The attestations must be cryptographically signed using a secure key.
- The storage location must be tamper-proof and protected using robust access control.

The attestations can then be used to evaluate policy compliance. A policy is a signed document that encodes the requirements for an artifact to be validated. The policy may include checks as to whether each of the functionaries involved in the CI process has used the right keys to generate the attestations, the required attestations are found, and the methodology to evaluate the attestation against its associated metadata has also been specified. The policy enables the verifiers to trace the compliance status of the artifact at any point during its life cycle.

The above capabilities collectively provide the following assurances:

- The software was built by authorized systems using authorized tools (e.g., infrastructure for each step) in the correct sequence of steps.
- There is no evidence of potential tampering or malicious activity.

5.1.2. Secure Pull-Push Operations on Repositories

The first SSC security task is to secure source-code development practices. In the context of CI/CD pipelines, code resides in repositories, is extracted by authorized developers using a PULL operation, is modified, and is then put back into the repositories using a PUSH operation. To authorize these PULL-PUSH operations, two forms of checks are required:

1. The type of authentication required for developers authorized to perform the PULL-PUSH operations. The request made by the developer must be consistent with their role (e.g., application updater, package manager). Developers with "merge approval" permissions cannot approve their own merges.
2. The integrity of the code in the repository can be trusted such that it can be used for further updates.

The various mechanisms for ensuring the trustworthiness of the code in the repository are:

- **PULL-PUSH_REQ-1:** The project maintainer should run automated checks on all artifacts covered in the change being pushed, such as unit tests, linters, integrity tests, security checks, and more.
- **PULL-PUSH-REQ-2:** CI pipelines should only be run using tools when confidence is established in the trustworthiness of the source-code origin of those tools.
- **PULL-PUSH-REQ-3:** The repository or source-code management system (e.g., GitHub, GitLab) should either a) run CI workflows in sandboxed environments without access to the network, any privileged access, or the ability to read secrets or b) have built-in protection that incorporates a delay in CI workflow runs until they are approved by a maintainer with write access. This built-in protection should go into effect when an outside contributor submits a pull request to a public repository. The setting for this protection should be at the strictest level, such as "Require approval for all outside collaborators" [10].
- **PULL-PUSH_REQ-4:** If there are no built-in protections available in the source-code management system, then external security tools with the following features are required:
  - Functionality to evaluate and enhance the security posture of the SCM systems with or without a policy (e.g., Open Policy Agent (OPA)) to assess the security settings of the SCM account and generate a status report with actionable recommendations.
  - Functionality to enhance the security of the source-code management system by detecting and remediating misconfigurations, security vulnerabilities, and compliance issues.

An example of such a tool is the popular open-source tool OpenSSF scorecard.[4]

5.1.3. Integrity of Evidence Generation During Software Updates

The software update process is typically carried out by a special class of software development tool called software update systems. Ensuring the security of these software update systems plays a critical role in the overall security of an SSC. Threats to software update systems primarily target the evidence generation process so as to erase the trail of updates and prevent the ability to determine whether the updates were legitimate or not.

There are several types of software update systems [11]:

- Package managers that are responsible for all of the software installed on a system
- Application updaters that are only responsible for individual installed applications

[4] See https://securityscorecards.dev/.

- Software library managers that install software that adds functionality, such as plugins or programming language libraries.

The primary task performed by a software update system is to identify the files that are needed for a given update ticket and download trusted files. At first glance, it may appear that the only checks needed to establish trust in downloaded files are the various integrity and authenticity checks performed by verifying the signatures on the metadata associated with individual files or the package. However, the very process of signature generation may be vulnerable to known attacks, so software update systems require many other security measures related to signature generation and verification.

The evolving framework for providing security for software update systems has incorporated many of these required security measures into its specification and prescribed some others for future specifications. A framework is a set of libraries, file formats, and utilities that can be used to secure new and existing software update systems. The framework should protect the signing operation by requiring the policy defined in Sec. 5.1.1 to be satisfied prior to performing the signing operation. The following are some of the consensus goals for the framework:

- The framework should provide protection against all known attacks on the tasks performed by the software update systems, such as metadata (hash) generation, the signing process, the management of signing keys, the integrity of the authority performing the signing, key validation, and signature verification.
- The framework should provide a means to minimize the impacts of key compromise by supporting roles with multiple keys and threshold or quorum trust (with the exception of minimally trusted roles designed to use a single key). The compromise of roles that use highly-vulnerable keys should have minimal impact. Therefore, online keys (i.e., keys used in an automated fashion) should not be used for any role that clients ultimately trust for files they may install [11]. When keys are online, exceptional care should be taken in caring for them, such as storing them in a Hardware Security Module (HSM) and only allowing their use if the artifacts being signed pass the policy defined in Sec. 5.1.1.
- The framework must be flexible enough to meet the needs of a wide variety of software update systems.
- The framework must be easy to integrate with software update systems.

5.1.4. Secure Code Commits

Appropriate forms of testing should be performed before code commits, and the following requirements must be met:

- SAST and DAST tools (covering all languages used in development) should be run in CI/CD pipelines with code coverage reports being provided to developers and security personnel.

- If open-source modules and libraries are used, dependencies must be enumerated, understood, and evaluated for policy (potentially using appropriate SCA tools). The security conditions that they should meet for their inclusion must also be tested. Dependency file detectors should detect all dependencies, including transitive dependencies with preferably no limit to the depth of nested or transitive dependencies that are to be analyzed [19].

One SSC security measure required during code commits is the prevention of secrets getting into the committed code. This is enabled by a scanning operation for secrets and results in a feature called push protection [12], [20]. This feature should satisfy the following requirements:

- **COMMIT-REQ-1:** (e.g., personal access token) Evaluate committed code for adherence to organizational policy, including the absence of secrets such as keys and Application Programming Interface (API) tokens. The detected secrets should be displayed prominently through media such as security dashboards, and appropriate alerts should be generated upon detection of policy violations with documented methods to remediate violations.
- **COMMIT-REQ-2:** Push protection features should be enabled for all repositories assigned to an administrator [13]. Such protection should include the verification of developer identity/authorization, the enforcement of developer signing of code commits, and file name verification [21].

5.2. Securing Workflows in CD Pipelines

Supply chain security measures also apply to controls during the CD process. The following are some due diligence measures that should be used during CD. These measures can be implemented by defining verification policies for allowing or disallowing an artifact for deployment.

- **DEPLOY-REQ-1:** For code that is already in the repository and ready to be deployed, a security scanning sub-feature should be invoked to detect the presence of secrets in the code, such as keys and access tokens. In many instances, the repository should be scanned for the presence of secrets, even before being populated with code, since their presence in a repository can mean that the credentials are already leaked, depending on the repository's visibility.
- **DEPLOY-REQ-2:** Before merging pull requests, it should be possible to view the details of any vulnerable versions through a form of dependency review [15], [19].
- **DEPLOY-REQ-3:** If a secure build environment and associated process have been established, it should be possible to specify that the artifact (i.e., container image) being deployed must have been generated by that build process in order to be cleared for deployment.

- **DEPLOY_REQ-4:** There should be evidence that the container image was scanned for vulnerabilities and attested for vulnerability findings. An important factor in vulnerability scans is the time when it was run. Since tools used to scan artifacts are continuously updated to detect new and emerging vulnerabilities, more recent scan results are more likely to be accurate and provide better assurance than results from the past. This technique enables DevOps teams to implement a proactive container security posture by ensuring that only verified container images are admitted into the environment and remain trusted during runtime [14]. Specifically, it should be possible to allow or block image deployment based on organization-defined policies.
- **DEPLOY-REQ-5:** The release build scripts should be periodically checked for malicious code. Specific tasks to be performed include:
    - A container image should be scanned for vulnerabilities as soon as it is built, even before it is pushed to a registry. The early scanning feature can also be built into local workflows.
    - The tools used to interact with repositories that contain container images and language packages should be capable of integration with CD tools, thus making all activities an integral part of automated CD pipelines.

5.2.1. Secure CD Pipeline — Case Study (GitOps)

All operations during and after a build in the CI/CD pipeline involve interacting with a central repository (e.g., Bitbucket, GitHub, and GitLab). The operations are collectively called GitOps, which is an automated deployment process facilitated by open-source tools, such as Argo CD and Flux. GitOps is carried out for both infrastructure code and application code and consist of commits, forking, and pull and push requests. The usage of GitOps covers the following [16]:

- Managing infrastructure as code
- Managing and applying cluster configurations
- Automating the deployment of containerized applications and their configurations to distributed systems.

The following SSC security tasks should be applied with respect to creating configuration data prior to deployment, capturing all data pertaining to a particular release, modifying software during runtime, and performing monitoring operations:

- **GitOps-REQ-1:** The process should rely on automation rather than manual operations. For example, manually configuring hundreds of YAML files to roll back a deployment on a cluster in a Git repository should be avoided.
- **GitOps-REQ-2:** Package managers that facilitate GitOps should preserve all data on the packages that were released, including the version numbers of all modules, all associated configuration files, and other metadata as appropriate for the software operational environment.

- **GitOps-REQ-3:** Changes should not be manually applied at runtime (e.g., kubectl). Instead, changes should be made to the relevant code, and a new release that incorporates those changes should be triggered. This ensures that Git commits remain the single source of truth for what runs in the cluster.
- **GitOps-REQ-4:** Since the Git repository contains the application definitions and configuration as code, it should be pulled automatically and compared with the specified state of these configurations (i.e., monitoring and remediation for drift). For any configurations that deviate from their specified state, the following actions may be performed:
    - Administrators can choose to automatically resync configurations to the defined state.
    - Notifications should be sent regarding the differences, and manual remediation should be performed.

5.3. SSC Security for CI/CD Pipelines — Implementation Strategy

The extensive set of steps needed for SSC security cannot be implemented all at once in the SDLC of all enterprises without a great deal of disruption to underlying business processes and operational costs. Rather, solutions that provide SSC security can be broadly classified into the following types [17]:

1. Solutions that ensure SSC security through features associated with each task in the DevSecOps pipelines:
    a. Verifying that the software is built correctly by ensuring tamper-proof build pipelines, such as by providing verified visibility into the dependencies and steps used in the build [18], since compromised dependencies or build tools are the greatest sources for poisoned workflows.
    b. Including features for the specification of checklists for each step of the delivery pipeline to provide guidance for implementation and to check and enforce controls for complying with checklists.
2. Solutions that ensure integrity and provenance through digital signatures and attestations
3. Strategy to ensure that running code is up to date, such as instituting a “build horizon” (i.e., code that is older than a certain time period should not be launched), to keep production as close as possible to the committed code in the repositories.
4. Securing CI/CD clients to prevent malicious code from stealing confidential information (e.g., proprietary source code, signing keys, cloud credentials), reading environment variables that may contain secrets, or exfiltrating data to an adversary-controlled remote endpoint.

6. Summary and Conclusions

This document provided an overview of strategies for integrating SSC security assurance measures into the various workflows associated with CI/CD pipelines, which is a methodology in the DevSecOps paradigm that is widely used for the development and deployment of cloud-native applications. However, no recommendations were provided with respect to the specific artifacts and frameworks associated with SSC security, such as SBOMs, code signing, and attestation. This is due to the fact that specifications and the standards associated with them are still evolving as part of projects in government institutions and industry forums. Further, NIST is aware of the emergence of a DevSecOps platform that provides an integrated set of services covering both CI and CD pipelines. Since this platform is not yet mature and there is a lack of consensus regarding the set of baseline features pertaining to it, the requirements for the secure use of this platform to carry out the activities in the CI/CD workflows are not discussed in this document.

References

[1] Chandramouli R (2022) Implementation of DevSecOps for a Microservices-based Application with Service Mesh. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204C. https://doi.org/10.6028/NIST.SP.800-204C

[2] Souppaya M, Scarfone K, Dodson D (2022) Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-218. https://doi.org/10.6028/NIST.SP.800-218

[3] EO 14028 (2021) *Improving the Nation's Cybersecurity*. Available at https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[4] Goud N (2021) *What is Solarigate.* Available at https://www.cybersecurity-insiders.com/what-is-solorigate/

[5] Berger A (2023) *What is Log4Shell?* Available at https://www.dynatrace.com/news/blog/what-is-log4shell/#:~:text=Log4Shell%20is%20a%20software%20vulnerability,logging%20error%20messages%20in%20applications

[6] Joint Task Force (2020) Security and Privacy Controls for Information Systems and Organizations. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-53, Rev. 5. Includes updates as of December 10, 2020. https://doi.org/10.6028/NIST.SP.800-53r5

[7] Lorenc D (2021) *Zero Trust Supply Chain Security*. Available at https://dlorenc.medium.com/zero-trust-supply-chain-security-e3fb8b6973b8

[8] Testify/Witness (2023) *Witness – Secure Your Supply Chain*. Available at https://github.com/testifysec/witness/

[9] Kennedy C (2021) *What is a Software Supply Chain Attestation – and Why do I need it?* Available at https://www.testifysec.com/blog/what-is-a-supply-chain-attestation/

[10] Gelb Y (2023) *Mass Scanning of Popular GitHub Repos for CI Misconfiguration*. Available at https://medium.com/checkmarx-security/mass-scanning-of-popular-github-repos-for-ci-misconfiguration-cd36ad6be788

[11] TUF V1.0.31 (2022) *The Update Framework Specification*. Available at https://theupdateframework.github.io/specification/latest/

[12] Malik Z, Sulakian M (2023) *Push Protection is generally available*. Available at https://github.blog/2023-05-09-push-protection-is-generally-available-and-free-for-all-public-repositories/?utm_source=thenewstack&utm_medium=website&utm_content=inline-mention&utm_campaign=platform

[13] GitHub Docs (2023) *Enabling Security Features for Multiple Repositories*. Available at https://docs.github.com/en/enterprise-cloud@latest/code-security/security-overview/enabling-security-features-for-multiple-repositories

[14] Cloud Build (2023) *Securing Image Deployments to Cloud Run and GKE*. Available at https://cloud.google.com/build/docs/securing-builds/secure-deployments-to-run-gke
[15] GitHub Docs (2023) *About dependency review*. Available at https://docs.github.com/en/enterprise-cloud@latest/code-security/supply-chain-security/understanding-your-software-supply-chain/about-dependency-review
[16] Williams A (2021) *A Blueprint for Supply Chain Security*. Published by Newstack
[17] Crane D (2023) *Five Stages for A Secure Software Supply Chain*. Available at https://danacrane.medium.com/five-stages-for-a-secure-software-supply-chain-f8420841cc3a
[18] CyRise (2023) *Supply Chain Security with Ensignia*. Available at https://medium.com/@cyrise/supply-chain-security-with-ensignia-483c1d872639
[19] GitLab Docs (2023) *Dependency Scanning*. Available at https://docs.gitlab.com/ee/user/application_security/dependency_scanning/index.html
[20] GitLab Docs (2023) *Secret Detection*. Available at https://docs.gitlab.com/ee/user/application_security/secret_detection/
[21] GitLab Docs (2023) *Push Rules*. Available at https://docs.gitlab.com/ee/user/project/repository/push_rules.html

Appendix A. Mapping of Recommended Security Tasks in CI/CD Pipelines to Recommended High-Level Practices in SSDF

Table 2. Mapping of recommended CI/CD pipeline security tasks to SSDF practices

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF
5.1.1 Secure Build — Policies for Build Process and Mechanisms to Enforce Policies **5.2 Securing Workflows in CD Pipelines**	Specify policies regarding the build. The policies include (a) the use of a secure isolated platform for performing the build, (b) the tools that will be used to perform the build, and (c) the authentication/authorization required for developers performing the build process. Enforce those build policies using an agent or some other means and a policy enforcement engine. **DEPLOY-REQ-1:** For code that is already in the repository and ready to be deployed, a security scanning sub-feature should be invoked to detect the presence of secrets in the code, such as keys and access tokens. In many instances, the repository should be scanned for the presence of secrets, even before being populated with code, since their presence in a repository can mean that the credentials are already leaked, depending on repository visibility. **DEPLOY-REQ-2:** Before merging pull requests, it should be possible to view the details of any vulnerable versions through a form of dependency review [15], [19]. **DEPLOY-REQ-3:** If a secure build environment and associated process have been established, it should be possible to specify that the artifact (i.e., container image) being deployed must have been generated by that build process in order to be cleared for deployment. **DEPLOY_REQ-4:** Check for evidence that the container image was scanned for vulnerabilities and attested for vulnerability findings. An important factor in vulnerability scans is the time when it was run. Since tools used to scan artifacts are continuously updated to detect new and emerging vulnerabilities, more recent scans results are more likely to be accurate and provide better assurance than results from the past. This technique enables DevOps teams to implement a proactive container security posture by ensuring that only verified container images are admitted into the environment and remain trusted during runtime [14]. Specifically,	**Define Security Requirements for Software Development (PO.1):** Ensure that the security requirements for software development are known at all times so that they can be considered throughout the SDLC, and the duplication of effort can be minimized. This includes requirements from internal sources (e.g., the organization's policies, business objectives, and risk management strategy) and external sources (e.g., applicable laws and regulations).

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF
	it should be possible to allow or block image deployment based on organization-defined policies. **DEPLOY-REQ-5:** Periodically check the release build scripts for malicious code. The tasks to be performed include: • A container should be scanned for vulnerabilities as soon as it is built, even before it is pushed to a registry. The early scanning feature can also be built into local workflows. • The tools used to interact with repositories that contain container images and language packages should be capable of integration with CD tools, thus making all activities an integral part of automated CD pipelines.	
5 Integrating SSC Security in CI/CD Pipelines	The prerequisites for activating CI/CD pipelines are: • Define roles for the actors operating the various CI/CD pipelines (e.g., application updaters, package managers, deployment specialists). • Identify the granular authorizations to perform various tasks, such as generating and committing code to SCMs, generating builds and packages, and checking various artifacts (e.g., builds and packages) into and out of the repositories.	**Implement Roles and Responsibilities (PO.2):** Ensure that everyone inside and outside of the organization involved in the SDLC is prepared to perform their SDLC-related roles and responsibilities throughout the SDLC.
5 Integrating SSC Security in CI/CD Pipelines	The entire CI/CD pipeline must be automated through the deployment of appropriate tools as a prerequisite for activating CI/CD pipelines. The driver tools for CI and CD pipelines are at a higher level and invoke a sequence of function-specific tools, such as those for code checkouts from repositories, edits and compilation, code commits, and testing (e.g., static application security testing [SAST], dynamic application security testing [DAST], and software composition analysis [SCA]. In general, the driver tools or build control plane execute at a higher level of trust than the individual functional steps, such as build.	**Implement Supporting Toolchains (PO.3):** Use automation to reduce human effort and improve the accuracy, reproducibility, usability, and comprehensiveness of security practices throughout the SDLC, as well as provide a way to document and demonstrate the use of these practices. Toolchains and tools may be used at different levels of the organization, such as organization-wide or project-specific, and may address a particular part of the SDLC, like a build pipeline.

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF
5.1.4 Secure Code Commits	Appropriate forms of testing should be performed before code commits, and the following requirements must be met: • SAST and DAST tools (covering all languages used in development) should be run in CI/CD pipelines with code coverage reports being provided to developers and security personnel. • If open-source modules and libraries are used, dependencies must be enumerated, understood, and evaluated for policy (potentially using appropriate SCA tools). The security conditions they should meet for their inclusion must also be tested. Dependency file detectors should detect all dependencies, including transitive dependencies with preferably no limit to the depth of nested or transitive dependencies that are to be analyzed [19].	**Define and Use Criteria for Software Security Checks (PO.4):** Help ensure that the software resulting from the SDLC meets the organization's expectations by defining and using criteria for checking the software's security during development.
5.1.1 Secure Build Policies for Build Process and Mechanisms for Enforcement of Policies	**Already covered under meeting requirements for PO.1.** In addition: 1. Environment attestation: Environment attestation involves an inventory of the system when the CI process happens. It generally refers to the platform on which the build process is run. This platform must be hardened, isolated, and secure.	**Implement and Maintain Secure Environments for Software Development (PO.5):** Ensure that all components of the environments for the SDLC are strongly protected from internal and external threats to prevent the environments or the software in them from being compromised. Examples of SDLC components include development, build, test, and deployment.
5.1.2 Secure PULL-PUSH Operations on Repositories	All forms of code used in the SDLC reside in repositories. Code is extracted from these repositories by authorized developers using a PULL operation, modified, and then put back into the repositories using a PUSH operation. To authorize these PULL-PUSH operations, two forms of checks are required: 1. The type of authentication required for developers authorized to perform the PULL-PUSH operations. The request made by the developer must be consistent with their role (e.g., application updater, package manager). Developers with "merge approval" permissions cannot approve their own merges.	**Protect All Forms of Code From Unauthorized Access and Tampering (PS.1):** Help prevent unauthorized changes to code, both inadvertent and intentional, that could circumvent or negate the intended security characteristics of the software. For code that is not intended to be publicly accessible, this helps prevent theft and may make it more difficult or time-consuming for attackers to find vulnerabilities in the software.

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF
	2. The integrity of the code in the repository can be trusted such that it can be used for further updates.	
5.1.3 Integrity of Evidence Generation During Software Updates (To provide assurance to acquirers that the software they get is legitimate, steps are taken to protect the integrity of evidence generation tasks)	1. The framework should provide protection against all known attacks on the tasks performed by the software update systems, such as metadata (hash) generation, the signing process, the management of signing keys, the integrity of the authority performing the signing, key validation, and signature verification. 2. The framework should provide a means to minimize the impact of key compromise by supporting roles with multiple keys and threshold or quorum trust (with the exception of minimally trusted roles designed to use a single key). The compromise of roles that use highly-vulnerable keys should have minimal impact. Therefore, online keys (i.e., keys used in an automated fashion) must not be used for any role that clients ultimately trust for files they may install [11]. When keys are online, exceptional care should be taken in caring for them, such as storing them in an HSM and only allowing their use if the artifacts being signed pass the policy defined in Sec. 5.1.1. 3. The framework must be flexible enough to meet the needs of a wide variety of software update systems. 4. The framework must be easy to integrate with software update systems.	**Provide a Mechanism for Verifying Software Release Integrity (PS.2):** Help software acquirers ensure that the software they acquire is legitimate and has not been tampered with.
5.2.1 Secure CD Pipeline — Case Study (GitOps)	The following SSC security tasks should be applied when creating configuration data prior to deployment, capturing all data pertaining to a particular release, modifying software during runtime, and performing monitoring operations: • **GitOps-REQ-2:** Package managers that facilitate GitOps should preserve all data on the packages that were released, including the version numbers of all modules, all associated configuration files, and other metadata as appropriate for the software operational environment.	**Archive and Protect Each Software Release (PS.3):** Preserve software releases in order to help identify, analyze, and eliminate vulnerabilities discovered in the software after release.

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF

5.1.2 Secure PULL-PUSH Operations on Repositories (Implements secure coding and build processes to improve security through various checks during PULL-PUSH operations)	• **PULL-PUSH_REQ-1:** The project maintainer should run automated checks on all artifacts covered in the pull request, such as unit tests, linters, integrity tests, security checks, and more.
	• **PULL-PUSH-REQ-2:** CI pipelines should only use external tools (e.g., Jenkins) when confidence is established in the trustworthiness of the source-code origin.
	• **PULL-PUSH-REQ-3:** The repository or source-code management system (e.g., GitHub, GitLab) should have built-in protection that incorporates a delay in CI workflow runs until they are approved by a maintainer with write access. This built-in protection should go into effect when an outside contributor submits a pull request to a public repository. The setting for this protection should be at the strictest level, such as "Require approval for all outside collaborators" [10].
	• **PULL-PUSH_REQ-4:** If there are no built-in protections available in the source-code management system, then external security tools with the following features are required:
	  ○ Functionality to evaluate and enhance the security posture of the SCM systems with or without a policy (e.g., OPA) to assess the security settings of the SCM account and generate a status report with actionable recommendations
	  ○ Functionality to enhance the security of the source-code management system by detecting and remediating misconfigurations, security vulnerabilities, and compliance issues
		Create Source Code by Adhering to Secure Coding Practices (PW.5): Decrease the number of security vulnerabilities in the software and reduce costs by minimizing vulnerabilities introduced during source-code creation that meet or exceed organization-defined vulnerability severity criteria.

5.1.1 Secure Build (Addresses the requirements for PW.6 through security requirements for the build platform)	Environment attestation: Environment attestation involves an inventory of the system when the CI process happens and generally refers to the platform on which the build process is run. The components of the platform (e.g., compiler, interpreter) must be hardened, isolated, and secure.	**Configure the Compilation, Interpreter, and Build Processes to Improve Executable Security (PW.6):** Decrease the number of security vulnerabilities in the software and reduce costs by eliminating vulnerabilities before testing occurs.

Section	Recommended Security Tasks in CI/CD Pipeline	Recommended High-Level Practice in SSDF
5.1.4 Secure Code Commits	Appropriate forms of testing should be performed before code commits, and the following requirements must be met: • Both SAST and DAST tools used in CI/CD pipelines must provide coverage for the different language systems used in cloud-native applications. • If open-source modules and libraries are used, dependencies must be detected using appropriate SCA tools, and the security conditions they should meet for their inclusion must also be tested.	**Test Executable Code to Identify Vulnerabilities and Verify Compliance With Security Requirements (PW.8):** Identify vulnerabilities so that they can be corrected before the software is released. Using automated methods lowers the effort and resources needed to detect vulnerabilities and improves traceability and repeatability. Executable code includes binaries, directly executed bytecode and source code, and any other form of code that an organization deems executable.
5 Integrating SSC Security into CI/CD Pipelines	Define CI/CD pipeline activities and associated security requirements for the development and deployment of application code; infrastructure as code, which contains details about the deployment platform; and policy as code and configuration code, which specify runtime settings (e.g., YAML files).	**Configure Software to Have Secure Settings by Default (PW.9):** Improve the security of the software at the time of installation to reduce the likelihood of the software being deployed with weak security settings, thus putting it at greater risk of compromise.

Appendix B. Justification for the Omission of Certain Measures Related to SSDF Practices in This Document

Table 3. Justification for the omission of certain SSDF practices

SSDF Practice	Justification for Omission
Produce Well-Secured Software (PW) PW1 through PW4, PW7	These practices pertain to secure software design, review of the design, and software reuse. CI/CD pipelines focus on setting up the environment for secure development and deployment in DevSecOps SDLC rather than software design.
Respond to Vulnerabilities (RV) RV1 through RV3	Vulnerability management strategies are at the organization policy level and are not specific to CI/CD pipelines.