International Journal of Technology, Management and Humanities Volume 11, Issue 3, 2025

Software Supply Chain Security in Cloud-Native Systems: A Zero Trust Framework for Securing APIs, Containers, Dependencies, and CI/CD Pipelines

Santosh Kumar Jadala

Cyber Security & Business Analysis Specialist Independent Researcher USA

Abstract

Software supply chain attacks have become a serious security concern in cloud-native environments, where applications depend on interconnected APIs, container images, open-source libraries, build systems, registries, and automated CI/CD pipelines. A weakness in any of these components can provide attackers with a path to alter source code, introduce malicious dependencies, steal credentials, compromise build processes, or deploy unauthorized software into production. Traditional perimeter-based security models are poorly suited to these distributed and highly automated environments because they often depend on implicit trust between internal systems and development stages. This article proposes a Zero Trust software supply chain security framework that applies continuous verification, least-privilege access, strong identity controls, software provenance, artifact integrity validation, and policy-based enforcement across the cloud-native software lifecycle. The framework addresses four major security domains: APIs, containers and cloud-native workloads, third-party dependencies, and CI/CD pipelines. It also incorporates software bills of materials, cryptographic signing, workload identity, secure build practices, runtime monitoring, and automated trust evaluation. A threat-control mapping and risk assessment model are used to examine how the proposed controls can reduce exposure to dependency poisoning, repository compromise, artifact tampering, credential theft, container attacks, and unauthorized deployment. The study provides a structured approach for integrating previously fragmented security controls into a unified supply chain security model and establishes a basis for future empirical evaluation in cloud-native production environments.

Keywords: Software Supply Chain Security, Zero Trust, Cloud-Native Security, API Security, Container Security, CI/CD Security, Dependency Security, Software Provenance, DevSecOps, Software Bill of Materials.

International Journal of Technology, Management and Humanities (2025)

Introduction

Background and Context

Modern software systems are increasingly developed and deployed through cloud-native architectures that combine microservices, containers, application programming interfaces, open-source libraries, automated build systems, container registries, infrastructure-as-code templates, and continuous integration and continuous delivery pipelines. This development model has improved deployment speed, scalability, portability, and operational flexibility. At the same time, it has expanded the number of components, identities, repositories, and automated processes that participate in the software lifecycle. As a result, the security of an application can no longer be determined solely by the quality of its internally developed source code. It also depends on the integrity of third-party packages, container images, build systems, deployment tools, APIs, cloud infrastructure, and the identities authorized to interact with these resources.

The software supply chain can therefore be understood as the collection of people, systems, processes, tools, dependencies, and artifacts involved in transforming source code into software running in a production environment. In cloud-native development, this chain is highly interconnected. A developer may retrieve an external dependency from a public package repository, commit code to a distributed version-control system, trigger an automated build pipeline, create a container image, store

Corresponding Author: Santosh Kumar Jadala, Cyber Security & Business Analysis Specialist Independent Researcher USA.

How to cite this article: Jadala, S.K. (2025). Software Supply Chain Security in Cloud-Native Systems: A Zero Trust Framework for Securing APIs, Containers, Dependencies, and CI/CD Pipelines. *International Journal of Technology, Management and Humanities,* 11(3), 171-196.

Source of support: Nil

Conflict of interest: None

the image in a registry, and deploy it to a Kubernetes cluster without direct manual intervention. Each stage creates a trust relationship that can be exploited if security controls are weak or inconsistently applied.

Research on open-source ecosystems has shown that dependency networks can create significant security exposure because vulnerabilities or malicious packages may propagate across many downstream applications. Zimmermann et al. (2019), for example, demonstrated that the npm ecosystem contains extensive dependency relationships capable of spreading security risks beyond the initially affected package. Similar concerns have been identified in studies of package managers, where attackers may exploit repository trust, dependency resolution mechanisms, or compromised packages to influence applications that indirectly depend on malicious code (Duan et al., 2021; Ohm et al., 2020).

Cloud-native systems further complicate this problem because application components are frequently distributed across containers, clusters, service meshes, APIs, and cloud services. Sultan et al. (2019) observed that containerization introduces a distinct set of security concerns involving image integrity, runtime isolation, registry protection, host security, and configuration weaknesses. Kubernetes deployments can also contain security misconfigurations that expose workloads or grant excessive privileges, particularly when configuration manifests are developed without consistent security validation (Rahman et al., 2023).

These conditions have shifted software supply chain security from a narrow software-development issue to a broader architectural and operational security concern. Protecting the cloud-native software lifecycle requires stronger mechanisms for validating identities, verifying software provenance, restricting privileges, securing automated pipelines, and continuously assessing the trustworthiness of software components.

The Software Supply Chain Security Problem

Software supply chain attacks seek to compromise trusted components, processes, or services that organizations rely upon during software development and deployment. Instead of attacking a production application directly, an adversary may target an earlier stage of the lifecycle where security controls are weaker or where a successful compromise can affect multiple downstream systems.

Ladisa et al. (2023) classify open-source software supply chain attacks across several stages, including source code, build processes, distribution mechanisms, and dependencies. Such attacks can involve malicious code insertion, compromised developer accounts, tampered build systems, dependency confusion, typosquatting, poisoned container images, stolen signing keys, and unauthorized modification of deployment pipelines. The attraction of these attack methods is that they exploit existing trust relationships. Once a compromised component is accepted as legitimate, malicious code may travel through automated development processes and reach production environments without triggering conventional perimeter defenses.

Dependency attacks represent one of the most visible examples of this problem. Modern applications can contain hundreds or thousands of direct and transitive dependencies. Developers may not be fully aware of every library executed within their applications. Decan et al. (2018) found that vulnerabilities within npm dependency networks can create exposure across large numbers of packages. Pashchenko et al. (2018) similarly showed that not every declared dependency presents the same practical risk, which highlights the importance of identifying which vulnerable components are actually reachable or used in the application.

Repositories and package managers also present a valuable attack surface. Cappos et al. (2008) demonstrated that weaknesses in software update and package-management systems can enable rollback, freeze, or malicious update attacks. Later work continued to examine mechanisms for securing software repositories and verifying the integrity of software distribution processes (Kuppusamy et al., 2016; Kuppusamy et al., 2017).

Cloud-native supply chains add further opportunities for compromise. An attacker who obtains CI/CD credentials may modify pipeline configurations, retrieve secrets, alter build artifacts, push malicious container images, or initiate unauthorized deployments. A compromised build system can be particularly dangerous because malicious changes may be introduced after source code has already passed review. This creates a situation in which trusted source code produces an untrusted artifact.

For this reason, software supply chain security must protect not only the application code but also the processes and infrastructure that transform that code into deployable software.

Limitations of Traditional Security Models

Traditional enterprise security architectures often rely on network boundaries and location-based trust. Once users, devices, or services gain access to an internal environment, they may receive broader access based on the assumption that internal systems are more trustworthy than external ones. Such assumptions are increasingly unsuitable for cloud-native environments.

Zero Trust architecture challenges this model by requiring explicit verification and limiting implicit trust regardless of network location. Rose et al. (2020) describe Zero Trust as an approach in which access decisions are based on continuously evaluated identity, device, resource, and contextual information rather than simple network membership. Syed et al. (2022) similarly emphasize that Zero Trust shifts security from perimeter-based assumptions toward identity-centered and policy-driven access control.

However, conventional implementations of Zero Trust often focus primarily on users, devices, networks, and

application access. Software supply chains require the same principles to be applied more broadly. A build artifact should not be trusted merely because it was produced by an internal CI/CD server. A dependency should not be accepted simply because it originates from a familiar package manager. A container image should not be deployed solely because it is stored in an approved registry. Trust must instead be supported by verifiable evidence such as identity, provenance, signatures, vulnerability status, policy compliance, and runtime behavior.

Traditional security also tends to separate application security, infrastructure security, dependency management, container security, and CI/CD security into different operational domains. This fragmentation can create gaps between individual controls. For example, a dependency scanner may identify vulnerable packages, while an image scanner evaluates containers, but neither tool may verify that the container being deployed actually originated from an approved build process. Software supply chain security therefore requires stronger connections between controls across the complete lifecycle.

Research Problem

The central research problem addressed in this study is the absence of a sufficiently integrated security architecture for enforcing continuous and evidence-based trust decisions throughout cloud-native software supply chains.

Existing practices frequently protect APIs, dependencies, containers, repositories, and CI/CD pipelines independently. Although each control contributes to security, isolated controls may fail to prevent attacks that move across multiple stages of the software lifecycle. An attacker may compromise a developer identity, inject malicious code into a repository, exploit a dependency-resolution process, manipulate a build pipeline, or replace a container image before deployment. Detecting one stage without validating the others does not provide complete assurance.

The problem is particularly important in cloud-native environments because automation reduces the number of manual checkpoints between code creation and production deployment. Automation can improve operational efficiency, but it can also accelerate compromise when malicious changes successfully enter the pipeline.

A broader Zero Trust model is therefore required in which every identity, dependency, build process, artifact, API interaction, container workload, and deployment action is treated as subject to verification.

Aim of the Study

The aim of this study is to develop a Zero Trust software supply chain security framework for cloud-native systems that integrates security controls across APIs, containers, third-party dependencies, build infrastructure, software artifacts, and CI/CD pipelines.

The proposed framework extends traditional Zero Trust concepts into the software lifecycle by applying continuous verification, least-privilege access, provenance validation, cryptographic integrity checking, identity-based controls, and automated policy enforcement to both human and machine actors.

Research Objectives

The study is guided by the following objectives

- To identify the principal attack surfaces and security risks affecting cloud-native software supply chains.
- To examine how Zero Trust principles can be extended to software artifacts, dependencies, workloads, APIs, and CI/CD infrastructure.
- To develop a layered Zero Trust framework for securing cloud-native software development and deployment.
- To map major software supply chain attack classes to preventive, detective, and responsive security controls.
- To establish measurable indicators for assessing security coverage, trust reduction, provenance assurance, and operational resilience.
- To provide a structured basis for future empirical validation of Zero Trust software supply chain security in production cloud environments.

Research Questions

The study addresses the following research questions

- RQ1: What are the principal threats and attack surfaces affecting cloud-native software supply chains?
- RQ2: How can Zero Trust principles be applied to APIs, containers, software dependencies, build systems, and CI/CD pipelines?
- RQ3: Which technical and governance controls can reduce the likelihood and impact of software supply chain compromise?
- RQ4: How can the effectiveness of a Zero Trust software supply chain security framework be assessed using measurable security indicators?

Research Contributions

This study makes several contributions to software supply chain security research. First, it develops a threat model that links cloud-native software development stages with common supply chain attack paths. Second, it extends Zero Trust beyond network and user access by applying explicit verification to code, dependencies, artifacts, workloads, pipelines, and automated services. Third, it proposes a layered security framework connecting repository protection, dependency assurance, secure builds, artifact provenance, container security, API protection, and runtime monitoring. Fourth, it maps major attack classes to corresponding Zero Trust controls. Finally, it provides an evaluation model that

can support future empirical testing and organizational security maturity assessments.

Literature Review and Theoretical Foundations

Software Supply Chain Security

Software supply chain security has become an important area of cybersecurity research because modern applications are assembled from a combination of internally developed code, third-party packages, open-source components, container images, build tools, cloud services, and automated deployment mechanisms. Each component can introduce vulnerabilities or create an opportunity for deliberate compromise.

Ohm et al. (2020) reviewed real-world open-source supply chain attacks and showed that attackers increasingly target trusted software distribution relationships rather than relying exclusively on direct exploitation of deployed applications. Ladisa et al. (2023) later provided a comprehensive taxonomy showing that software supply chain attacks can target development tools, source repositories, dependencies, build processes, distribution infrastructure, and software updates.

The security challenge is therefore broader than vulnerability management. A software component can be free of known vulnerabilities and still be malicious, tampered with, improperly built, or obtained from an untrusted origin. This distinction has encouraged greater emphasis on provenance, integrity, reproducible builds, package verification, and software bills of materials.

Torres-Arias et al. (2019) proposed in-toto as a mechanism for providing end-to-end guarantees about the steps performed during software supply chain processes. The approach illustrates the importance of verifying not only the final artifact but also the sequence of actions that produced it. Reproducible builds provide another form of assurance by making it possible to determine whether independent build processes generate consistent artifacts from the same source material. Lamb and Zacchiroli (2022) argue that reproducible builds can strengthen software supply chain integrity by reducing uncertainty around the build process.

Cloud-Native Architecture and Security Implications

Cloud-native applications differ significantly from traditional monolithic systems. They are commonly composed of distributed microservices running in containers and communicating through APIs. Workloads may be orchestrated using Kubernetes and integrated with cloud-based databases, identity platforms, message queues, secret stores, registries, and managed services.

Although this architecture provides scalability and flexibility, it also produces many machine-to-machine relationships that must be secured. Chandramouli and Butcher (2020) discuss the use of service-mesh architectures to secure microservice communication, particularly through identity-aware communication, encryption, and policy enforcement. Their work demonstrates how cloud-native security increasingly depends on service identity rather than traditional network location.

Containerization introduces further considerations. Container images may contain outdated packages, vulnerable dependencies, embedded secrets, or malicious modifications. Weak registry permissions can allow unauthorized users to replace or publish images, while excessive privileges may increase the consequences of container compromise. Sultan et al. (2019) identify isolation, image security, runtime protection, host security, and access control as central areas of container security.

Kubernetes security introduces another layer of complexity. Rahman et al. (2023) found that open-source Kubernetes manifests frequently contain security misconfigurations, illustrating the difficulty of consistently implementing least privilege and secure configuration in complex orchestration environments.

Cloud-native software supply chain security must therefore consider both pre-deployment and runtime trust. An artifact that was secure when built may later become vulnerable because of a newly disclosed vulnerability, configuration change, leaked credential, or compromised runtime environment.

Zero Trust Security Principles

Zero Trust provides the theoretical foundation for the proposed framework. Rose et al. (2020) define Zero Trust as an architectural approach that eliminates implicit trust based on physical or network location. Access should instead depend on explicit authentication, authorization, contextual information, and policy evaluation.

- Three principles are particularly relevant to software supply chains.
- The first is explicit verification. Every identity, request, artifact, dependency, and deployment action should be validated based on available evidence.
- The second is least-privilege access. Developers, build servers, service accounts, workloads, registries, and CI/CD pipelines should receive only the permissions necessary to perform their assigned functions.
- The third is assume breach. Security architectures should be designed under the assumption that a developer account, package, pipeline component, API credential, or workload may eventually be compromised.

Syed et al. (2022) explain that Zero Trust requires continuous assessment rather than a one-time authentication event. This principle is particularly important in software supply chains, where trust conditions may change rapidly. A dependency that was considered safe during development may later become vulnerable. A valid signing key may be stolen. A legitimate pipeline account may begin behaving abnormally.

Zero Trust for software supply chains should therefore be dynamic rather than binary.

API Security in Cloud-Native Systems

APIs are central to cloud-native architectures because they enable communication among microservices, cloud services, development tools, and external applications. Their central role also makes them attractive targets.

In traditional applications, internal functions may have been tightly coupled within a single application boundary. Microservice architectures expose these interactions through service interfaces, increasing the number of authentication and authorization decisions that must be performed. API security therefore requires strong identity verification, encrypted communication, fine-grained authorization, token protection, behavioral monitoring, and consistent policy enforcement.

Chandramouli et al. (2021) discuss attribute-based access control within microservices and service-mesh environments, showing how access decisions can incorporate contextual attributes rather than depending solely on static roles. Such capabilities align with Zero Trust because access can be continuously evaluated according to the identity of a service, requested resource, environmental conditions, and applicable security policies.

For software supply chain security, API protection is also relevant to development tooling. Source repositories, container registries, cloud control planes, CI/CD platforms, package repositories, and infrastructure management systems frequently expose APIs. Compromise of the credentials used to access these interfaces may allow attackers to alter critical supply chain resources.

Container and Kubernetes Security

Containers are now a major deployment unit for cloud-native applications. Their security begins before runtime because container images are created during build processes and commonly stored in registries before deployment.

Security controls should therefore cover image composition, vulnerability scanning, base-image selection, artifact signing, registry access, admission control, runtime configuration, and workload behavior. Container image signing can provide evidence that a deployment artifact came from an authorized source and has not been modified after signing.

Sultan et al. (2019) emphasize that container security requires protection across the entire container lifecycle rather than a single runtime control. This view aligns closely with software supply chain security because vulnerable or compromised container images may originate from earlier build stages.

Kubernetes admission controls can strengthen this process by preventing deployment of workloads that fail defined policies. Examples include unsigned images, vulnerable images, containers requesting prohibited privileges, or workloads that do not comply with organizational requirements.

Rahman et al. (2023) demonstrate that configuration weaknesses are common enough to warrant systematic and preferably automated enforcement. In a Zero Trust supply chain model, a container should not be deployed simply because it was successfully built. Deployment should depend on evidence that the artifact satisfies integrity, provenance, vulnerability, and policy requirements.

Open-Source and Third-Party Dependency Security

Modern software development depends heavily on open-source packages. This model accelerates development but transfers part of the application's security risk to external software ecosystems.

Dependency security includes at least four related challenges: known vulnerabilities, malicious packages, outdated components, and transitive dependencies. Kula et al. (2018) found that developers do not always update vulnerable libraries promptly, even when relevant security advisories are available. Decan et al. (2018) further demonstrated how vulnerabilities can propagate through dependency relationships in package ecosystems.

Supply chain attacks add a malicious dimension to this problem. Package attackers may use dependency confusion, typosquatting, account compromise, or package takeover to distribute hostile code. Duan et al. (2021) examined supply chain attacks against package managers for interpreted languages, highlighting the scale at which malicious packages may potentially affect developer ecosystems.

Dependency security therefore cannot be reduced to vulnerability scanning. A Zero Trust approach should also consider package origin, expected publisher, cryptographic integrity, repository source, package age, version changes, and provenance.

Pashchenko et al. (2018) additionally showed that dependency risk should account for whether vulnerable code is actually used or reachable. This suggests that future supply chain risk models should combine vulnerability data with application context.

CI/CD Pipeline Security

CI/CD pipelines are central to cloud-native development because they automate testing, building, packaging, signing, and deployment. Their extensive privileges make them a high-value target.

A compromised CI/CD system may have access to source repositories, secrets, signing keys, cloud credentials, registries, and production deployment mechanisms. A successful attack can therefore bypass several conventional controls.

Rajapakse et al. (2022) note that DevSecOps adoption faces challenges involving security integration, tooling, organizational responsibility, automation, and culture.

These findings are relevant to supply chain security because effective pipeline protection requires security controls to operate as part of normal development processes rather than as isolated manual reviews.

NIST guidance developed by Chandramouli et al. (2024) specifically addresses the integration of software supply chain security strategies within DevSecOps CI/CD pipelines. The guidance reinforces the need for secure development environments, provenance, integrity verification, access control, and automated policy enforcement.

Important CI/CD controls include ephemeral build environments, short-lived credentials, isolated build runners, secret management, signed artifacts, provenance generation, protected pipeline definitions, mandatory security scanning, and policy-based deployment gates.

Software Bills of Materials and Software Provenance

Software bills of materials have emerged as a major mechanism for improving transparency within software supply chains. An SBOM records the components, packages, and dependencies included within a software product, helping organizations identify exposure when new vulnerabilities are disclosed.

Xia et al. (2023) examined the state of SBOM adoption and identified both their value and implementation challenges. Zahan et al. (2023) similarly argued that SBOM requirements are becoming increasingly important while noting that implementation remains incomplete in many environments.

SBOMs are valuable, but they do not independently guarantee software integrity. An accurate inventory may show what components are present, but it does not prove that those components came from authorized sources or that the build process was uncompromised.

This is where provenance becomes important. Provenance records information about how, where, and under what conditions an artifact was produced. In-toto provides an example of a framework for verifying the sequence of software supply chain activities (Torres-Arias et al., 2019). Reproducible builds provide another mechanism for increasing confidence that a binary corresponds to a known source state (Lamb & Zacchiroli, 2022; Fourné et al., 2023).

Recent research also shows that SBOM effectiveness depends on the quality and completeness of the information produced. Stalnaker et al. (2024) examined stakeholder perspectives on SBOM use and highlighted practical challenges related to generation, consumption, and organizational integration. O'Donoghue et al. (2024) further explored how SBOM generation affects vulnerability detection, reinforcing the importance of treating SBOMs as operational security artifacts rather than merely compliance documents.

Existing Software Supply Chain Security Approaches and Research Gap

Several existing security approaches contribute to software supply chain protection, but each addresses different portions of the problem.

Secure software development frameworks focus on integrating security practices throughout development. NIST's Secure Software Development Framework provides recommendations for preparing organizations, protecting software, producing secure releases, and responding to vulnerabilities (Souppaya et al., 2022). DevSecOps similarly attempts to integrate security into development and operations processes.

SBOM-based security improves software component visibility. SLSA-related concepts and provenance frameworks focus on build integrity, attestation, and artifact traceability. Zero Trust architectures provide strong models for identity, authorization, policy enforcement, and continuous verification.

However, a research gap remains in integrating these ideas into a unified framework that addresses the complete cloud-native supply chain. Many existing approaches emphasize one domain, such as build integrity, dependency transparency, identity access, container security, or vulnerability management. Cloud-native environments require these controls to operate as a connected trust system.

The proposed framework addresses this gap by extending Zero Trust principles throughout the software lifecycle and linking identity, provenance, dependency assurance, container security, API protection, CI/CD integrity, runtime monitoring, and governance within a single security architecture.

Taken together, the reviewed literature indicates that software supply chain security is not adequately addressed by a single scanner, SBOM, authentication system, or secure build process. The security problem spans multiple technical and organizational boundaries. A stronger approach must establish verifiable trust throughout the path from source code and dependency acquisition to build, deployment, and runtime operation. This provides the theoretical basis for the Zero Trust software supply chain framework developed in the subsequent sections.

Research Methodology

Research Design

This study adopts a design-oriented conceptual research approach to develop a Zero Trust security framework for cloud-native software supply chains. The approach is appropriate because the study is concerned with identifying security problems, structuring threat relationships, and designing an architectural response that can be evaluated against known software supply chain risks. Rather than testing a single security product, the research integrates established security principles and controls into a unified

Table 1: Comparative Overview of Software Supply Chain Security Approaches

Security Approach	*Primary Focus*	*API Security*	*Container Security*	*Dependency Security*	*CI/CD Security*	*Provenance Support*	*Continuous Verification*	*Main Limitation*
Traditional DevSecOps	Security integration within development and operations	Moderate	Moderate	Moderate	High	Moderate	Moderate	Controls may remain tool-specific and fragmented
SBOM-Based Security	Software component visibility and dependency inventory	Low	Moderate	High	Moderate	Moderate	Low	Identifies components but does not by itself prove integrity
Provenance and Reproducible Builds	Build and artifact integrity	Low	Moderate	Moderate	High	High	Moderate	Concentrates mainly on artifact and build assurance
Conventional Zero Trust	Identity, access control, and resource protection	High	Moderate	Low	Moderate	Low	High	Usually focused on users, devices, services, and networks
Proposed Zero Trust Supply Chain Framework	End-to-end cloud-native software supply chain security	High	High	High	High	High	High	Requires integration across development, security, platform, and operational teams

framework covering source code, APIs, dependencies, containers, build infrastructure, and CI/CD pipelines.

The methodological process combines literature synthesis, threat modeling, security control mapping, and framework evaluation. Existing research on software supply chain attacks, Zero Trust architecture, DevSecOps, software provenance, container security, and dependency management provides the theoretical basis for identifying weaknesses and selecting appropriate controls. The Secure Software Development Framework also supports the methodological structure by emphasizing protection of software development environments, secure software production, and vulnerability response across the development lifecycle (Souppaya et al., 2022).

The research follows five stages: identification of software supply chain assets, analysis of attack surfaces, classification of likely attack paths, mapping of Zero Trust controls to identified threats, and evaluation of the proposed architecture using defined security indicators.

System Boundary and Unit of Analysis

The unit of analysis is the cloud-native software supply chain lifecycle, beginning with software development and ending with deployment and runtime operation. The study examines the following sequence:

Developer → Source Repository → Dependency Management → Build System → CI/CD Pipeline → Artifact Repository → Container Registry → Cloud or Kubernetes Deployment → APIs and Runtime Services

Each stage is treated as both a functional component and a potential security boundary. Trust is not assumed merely because a component belongs to the same organization or cloud environment.

The system boundary includes human identities, machine identities, source code repositories, package managers, build tools, CI/CD runners, container images, registries, Kubernetes clusters, APIs, secrets, signing keys, and deployment credentials. External package repositories and third-party software components are included because they directly influence the integrity of the final application.

This lifecycle view reflects the position taken by Chandramouli et al. (2024), who emphasize that software supply chain protection should be integrated throughout DevSecOps CI/CD processes rather than applied only after software has been built.

Threat Modeling Method

The threat model is developed by identifying four core elements: assets, threat actors, attack surfaces, and attack paths. Assets represent resources whose compromise could

affect software integrity, confidentiality, availability, or deployment trust. Threat actors include external attackers, insiders, compromised developers, malicious package maintainers, and actors that obtain access through stolen credentials.

Attack surfaces are identified at each stage of the supply chain. These include repository access, package acquisition, build execution, CI/CD configuration, container image creation, registry access, API communication, and cloud deployment.

The analysis is informed by the software supply chain attack taxonomy proposed by Ladisa et al. (2023), which demonstrates that attacks can occur across source code, build systems, dependencies, distribution channels, and deployment processes. Earlier research also shows that malicious software can enter trusted ecosystems through compromised or intentionally harmful open-source packages (Ohm et al., 2020).

The threat model focuses on how an attacker may move from an initial point of compromise to downstream systems. This approach is important because software supply chain attacks rarely remain isolated to the first compromised component.

Framework Development Procedure

The proposed framework is developed by mapping each identified threat to relevant Zero Trust principles and corresponding technical controls. Four control categories are considered:

- Identity controls, including strong authentication, workload identity, short-lived credentials, and least-privilege permissions.
- Integrity controls, including artifact signing, cryptographic verification, software provenance, and immutable build outputs.
- Preventive and detective controls, including dependency scanning, secret scanning, container inspection, admission policies, and runtime monitoring.
- Governance controls, including policy-as-code, audit trails, approval requirements, and incident response procedures.

The framework extends Zero Trust beyond user and device access. Code, packages, software artifacts, workloads, and deployment actions must also provide sufficient evidence of trust before progressing to the next lifecycle stage. This principle is consistent with the Zero Trust requirement that access decisions should be based on explicit verification rather than network location or prior trust (Rose et al., 2020).

Framework Evaluation Method

The proposed framework is evaluated conceptually using security control coverage and risk-reduction indicators. Evaluation focuses on whether the framework reduces implicit trust and increases verification across critical supply chain stages.

Key indicators include

- Percentage of verified software dependencies
- Percentage of cryptographically signed artifacts
- SBOM coverage
- CI/CD pipeline verification coverage
- Number of long-lived privileged credentials
- Percentage of workloads using verified identities
- Mean time to detect suspicious activity
- Mean time to contain compromised components
- Percentage of deployments passing provenance and policy checks
- Number of unauthorized deployment attempts blocked

The evaluation is intended to measure control coverage and security improvement rather than claim empirical performance. Future work can test the framework through controlled cloud-native environments, red-team exercises, simulated software supply chain attacks, and Kubernetes-based testbeds.

Cloud-Native Software Supply Chain Threat Model

Critical Software Supply Chain Assets

Cloud-native software supply chains contain many assets whose compromise can affect the integrity of deployed applications. The most important assets include source code, developer accounts, signing keys, API credentials, build infrastructure, software dependencies, container images, CI/CD configurations, artifact repositories, container registries, Kubernetes credentials, cloud service identities, and production workloads.

These assets are closely connected. A compromised developer account may provide access to a source repository. Modified source code may then trigger an automated build, create a container image, and reach production through a CI/CD pipeline. The security of the final workload therefore depends on the integrity of every stage that precedes deployment.

Software artifacts and build metadata are particularly important because attackers may alter software after source-code review. Provenance mechanisms can help establish evidence about how an artifact was created, which tools were involved, and whether expected build steps were followed (Torres-Arias et al., 2019).

Threat Actors

The threat model considers several categories of adversaries.

- External attackers may exploit vulnerable APIs, stolen credentials, exposed repositories, or insecure development infrastructure.
- Malicious insiders may intentionally alter code, leak credentials, modify build configurations, or bypass security controls.
- Compromised developers represent legitimate users whose credentials or development environments have

been taken over by attackers.

- Malicious or compromised package maintainers may introduce harmful code into software components consumed by downstream applications.
- Compromised vendors and service providers can create indirect access to organizations that trust their software or services.

Highly capable criminal or state-sponsored actors may also target build systems or widely used dependencies because compromising a single trusted component can provide access to many downstream organizations.

Major Attack Surfaces

Source Repository Compromise

Source repositories are central control points within modern development environments. Attackers who gain unauthorized access may alter application code, modify workflow definitions, insert malicious scripts, disable security checks, or steal sensitive information.

Repository compromise may result from weak authentication, exposed access tokens, excessive privileges, malicious insiders, or compromised developer accounts. Signed commits, protected branches, strong authentication, mandatory code review, and least-privilege access can reduce this risk.

Dependency and Package Ecosystem Attacks

Open-source dependencies create an important supply chain attack surface because applications often depend on large networks of direct and transitive packages.

Attackers may use dependency confusion, typosquatting, account takeover, or malicious package updates to introduce harmful code. Duan et al. (2021) demonstrated that package manager ecosystems can provide attackers with scalable opportunities to distribute malicious software through trusted dependency relationships.

Controls should therefore address both known vulnerabilities and package authenticity. Dependency pinning, software composition analysis, trusted repositories, package allowlists, provenance checks, and SBOM generation can reduce exposure.

Build Environment Compromise

Build systems transform trusted source code into deployable artifacts. If the build environment is compromised, an attacker may insert malicious code that is absent from the reviewed source repository.

This creates a serious integrity problem because the source code may appear legitimate while the final binary or container image has been modified. Build isolation, ephemeral build environments, signed outputs, reproducible builds, and provenance records can help detect or prevent such manipulation.

CI/CD Pipeline Compromise

CI/CD systems often possess privileged access to repositories, registries, cloud environments, deployment credentials, and signing keys. A compromised pipeline may therefore provide a direct route from development infrastructure into production.

Attackers may modify pipeline scripts, steal stored secrets, abuse deployment tokens, disable testing, alter build outputs, or trigger unauthorized releases. Secure CI/CD design should use isolated runners, protected pipeline configurations, secret vaults, short-lived credentials, approval gates, and continuous monitoring.

Container and Registry Attacks

Container images may contain vulnerable software, malicious files, exposed secrets, or unauthorized changes. Registries can also become targets because replacing a legitimate image with a malicious version may allow an attacker to reach production systems.

Sultan et al. (2019) identify image security, isolation, runtime protection, and access control as important areas of container security. Effective controls include container scanning, cryptographic signing, registry authorization, immutable tags where appropriate, and admission policies that prevent unverified images from running.

API and Service-to-Service Attacks

APIs connect microservices, cloud platforms, CI/CD systems, registries, and external services. Stolen tokens, weak authorization, excessive privileges, insecure endpoints, and poorly protected machine identities may allow attackers to gain unauthorized access to critical supply chain resources.

Service-to-service interactions are especially important because cloud-native workloads can generate large numbers of automated requests. Strong workload identity, mutual TLS, fine-grained authorization, token validation, and behavioral monitoring help reduce implicit trust between services.

Cloud and Kubernetes Credential Compromise

Cloud and Kubernetes credentials may provide broad access to clusters, workloads, secrets, storage resources, and deployment functions. Excessive permissions can increase the consequences of credential theft.

Rahman et al. (2023) found that Kubernetes configurations commonly contain security weaknesses, demonstrating the importance of configuration validation and least-privilege enforcement. Short-lived credentials, workload identities, role-based permissions, policy controls, and continuous monitoring should therefore form part of the supply chain security architecture.

Software Supply Chain Attack Paths

A defining feature of software supply chain attacks is the ability to move across multiple stages after the initial compromise. An attacker may begin with a developer

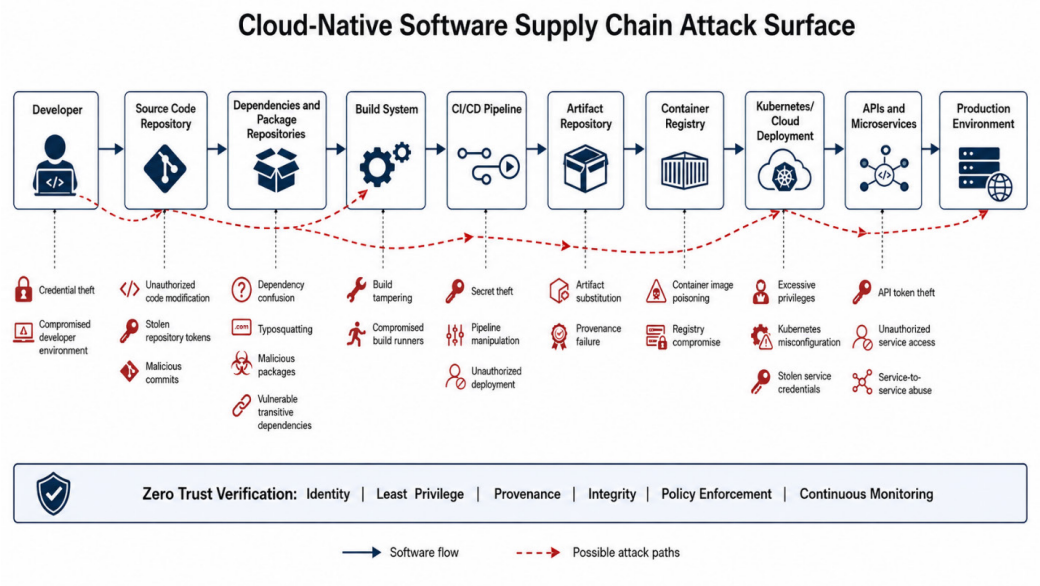


Fig 1: Cloud-native software supply chain attack surface and Zero Trust security controls.

account, alter a repository, manipulate a dependency or build configuration, produce a malicious artifact, publish it to a registry, and deploy it through an authorized CI/CD process.

Another attack path may begin with a malicious dependency. Once the package enters the application, it may execute during build or runtime, access stored credentials, communicate with external infrastructure, or modify downstream artifacts. Ohm et al. (2020) show that malicious open-source packages can exploit the trust placed in software ecosystems and their distribution mechanisms.

These attack paths demonstrate why isolated controls are insufficient. A vulnerability scanner may detect known weaknesses but cannot prove that an artifact came from an approved build process. Strong authentication may protect a repository but cannot guarantee the integrity of third-party dependencies. Container scanning may identify vulnerable packages but cannot determine whether the image was replaced after scanning.

The proposed Zero Trust framework therefore treats each transition in the software lifecycle as a security decision point. An artifact or workload progresses only when identity, integrity, provenance, vulnerability status, and policy requirements have been verified.

Zero Trust Principles for Software Supply Chain Security

Zero Trust provides a suitable security foundation for cloud-native software supply chains because it replaces implicit trust with continuous verification. In traditional software environments, internal repositories, build servers, service accounts, and deployment pipelines are often trusted once they are placed within an approved organizational boundary. This assumption is risky in cloud-native environments where development tools, open-source packages, APIs, container registries, and automated pipelines interact across distributed infrastructure. Rose et al. (2020) argue that Zero Trust decisions should be based on verified identity, resource state, policy, and context rather than network location. Applied to software supply chains, this principle means that code, dependencies, build artifacts, workloads, and deployment requests must provide evidence of trustworthiness before they are accepted.

From Network-Centric Zero Trust to Software Supply Chain Zero Trust

Conventional Zero Trust architectures primarily address users, devices, networks, applications, and resource access. Software supply chain security requires this model to be extended to the objects and automated processes involved in software production.

A source-code commit, for example, should not be trusted simply because it comes from an internal developer account. Its author, signature, repository permissions, and approval history should be verified. Similarly, a container image should not be deployed only because it exists in an approved registry. Its origin, signature, vulnerability status, build provenance, and policy compliance should also be checked.

The same requirement applies to dependencies, APIs, CI/CD workflows, and machine identities. Chandramouli and Butcher (2023) emphasize that cloud-native Zero Trust architectures require identity-aware and policy-based access control across distributed services. Extending this principle to the software lifecycle creates a model in which trust is granted conditionally and reassessed whenever security conditions change.

Verify Explicitly

Explicit verification is the central principle of the proposed security model. Every software supply chain transaction should be validated using relevant security evidence.

For human users, verification may include strong authentication, multifactor authentication, role validation, and contextual access decisions. For software components, verification may include cryptographic hashes, digital signatures, package provenance, SBOM records, vulnerability status, and approved repository sources.

For CI/CD pipelines, verification should confirm that build steps, credentials, source repositories, dependencies, and artifact outputs comply with organizational policies. Torres-Arias et al. (2019) demonstrate the importance of software provenance by showing how in-toto can verify whether expected supply chain steps were performed by authorized entities.

Explicit verification should therefore occur at every important lifecycle transition, particularly before code is merged, packages are introduced, artifacts are published, container images are deployed, and workloads receive access to production resources.

Least-Privilege Access

Least privilege limits the permissions granted to developers, CI/CD systems, build runners, service accounts, containers, APIs, and cloud workloads. Each identity should receive only the access needed for its assigned function.

This principle is particularly important for CI/CD systems because pipelines often have access to repositories, artifact stores, registries, cloud resources, secrets, and deployment environments. If these accounts receive broad privileges, a single credential compromise may provide an attacker with control over multiple stages of the supply chain.

Least privilege should therefore include restricted repository permissions, narrowly scoped API tokens, workload-specific identities, limited Kubernetes roles, short-lived deployment credentials, and separation between development and production permissions. Attribute-based and policy-based access controls can further restrict access according to identity, requested resource, environment, and security context (Chandramouli et al., 2021).

Assume Supply Chain Compromise

The proposed framework assumes that any individual supply chain component may eventually become compromised. This does not mean that every component is treated as malicious. Rather, the architecture should be designed so that compromise of one stage does not automatically provide trusted access to subsequent stages.

A compromised developer account should not be able to deploy directly to production. A malicious dependency should not automatically pass through build and deployment stages. A compromised registry should not be able to introduce unsigned images into a Kubernetes cluster. This principle supports segmentation, independent verification, artifact signing, approval gates, policy enforcement, and continuous monitoring. The purpose is to reduce the ability of attackers to convert an initial compromise into complete supply chain control.

Continuous Trust Evaluation

Trust within software supply chains should not remain permanent after an initial security check. Conditions change throughout the software lifecycle.

A package that was considered secure when added to a project may later receive a vulnerability disclosure. A legitimate signing key may be stolen. A container image may be modified after scanning. A service account may begin making unusual requests. Continuous evaluation allows trust decisions to change when new evidence becomes available.

Syed et al. (2022) describe continuous assessment as an important feature of Zero Trust architecture. In cloud-native software supply chains, this requires repeated verification of vulnerabilities, identities, configurations, provenance records, permissions, and runtime behavior.

Continuous trust evaluation may therefore trigger actions such as blocking a deployment, revoking a credential, quarantining an artifact, restricting an API session, or requesting additional approval.

Identity as the Foundation of Supply Chain Trust

Identity is central to Zero Trust because both human and machine actions must be attributable to known entities. Modern software supply chains contain a large number of machine identities, including CI/CD runners, service accounts, container workloads, APIs, automation tools, and cloud services.

These identities often operate without human involvement and may perform privileged operations at high frequency. Long-lived credentials or shared service accounts can make it difficult to determine which workload performed a particular action.

The proposed framework therefore prioritizes unique workload identities, short-lived tokens, certificate-based authentication, secrets management, and centralized identity governance. Strong identity attribution also supports accountability because repository changes, build actions, artifact publications, and production deployments can be linked to specific authorized entities.

Proposed Zero Trust Software Supply Chain Security Framework

Framework Overview

The proposed Zero Trust Software Supply Chain Security Framework establishes security controls across the complete cloud-native development lifecycle. It combines identity management, repository protection, dependency assurance,

build security, artifact verification, container protection, API security, runtime monitoring, and governance within a single architectural model.

The framework is organized into eight interdependent security layers. Each layer addresses a specific part of the software supply chain, while continuous verification, least privilege, provenance, and policy enforcement operate across the entire architecture. This structure reflects the secure development principles described by Souppaya et al. (2022) and the CI/CD supply chain controls proposed by Chandramouli et al. (2024).

Layer 1: Identity and Access Security

The first layer establishes trusted identities for developers, administrators, workloads, services, and automated pipelines.

- *Key controls include*

- Multifactor authentication
- Workload identities
- Short-lived credentials
- Privileged access management
- Role and attribute-based authorization
- Removal of shared accounts
- Automated credential rotation

This layer provides the foundation for determining who or what is permitted to interact with supply chain resources.

Layer 2: Source Code and Repository Security

The second layer protects source repositories from unauthorized changes and credential abuse.

- *Controls includ:*

- Branch protection
- Mandatory code review
- Signed commits
- Repository access restrictions
- Secret scanning
- Audit logging
- Protected workflow definitions

These controls help reduce the risk of malicious commits, unauthorized code modification, and repository takeover.

Layer 3: Dependency and Package Security

The dependency layer verifies the origin, integrity, and security condition of third-party software components.

- *Controls include*

- Software composition analysis
- Dependency pinning
- Trusted package repositories
- Package allowlists
- SBOM generation
- Provenance checks
- Vulnerability monitoring
- Transitive dependency analysis

This layer is necessary because package ecosystems can distribute risk across large dependency networks, particularly when malicious or vulnerable components are indirectly inherited by applications (Duan et al., 2021).

Layer 4: Build and CI/CD Security

The fourth layer protects build infrastructure and automated software delivery processes.

- *Controls include*

- Ephemeral build environments
- Isolated CI/CD runners
- Pipeline-as-code protection
- Secure secrets management
- Signed build outputs
- Provenance generation
- Deployment approval gates
- Short-lived pipeline credentials

Build and CI/CD systems should be treated as high-value security assets because they frequently possess permission to modify software and initiate deployment.

Layer 5: Container and Artifact Security

This layer protects software artifacts after the build process and before runtime execution.

Controls include

- Container image scanning
- Digital signing
- Artifact integrity validation
- Registry access controls
- Immutable artifacts
- Trusted base images
- Admission policies
- Artifact provenance verification

Lamb and Zacchiroli (2022) note that reproducible builds can improve software supply chain integrity by increasing confidence that deployed software corresponds to expected source code and build processes.

Layer 6: API and Service Security

The API security layer controls communication among microservices, pipelines, cloud systems, and external services.

- *Controls include*

API gateways
Mutual TLS
OAuth and OpenID Connect
Fine-grained authorization
Token validation
Rate limiting
Workload identity
API behavior monitoring

Service-to-service communication should be authenticated and authorized independently rather than trusted because services operate within the same cluster or cloud network.

Layer 7: Runtime and Cloud-Native Security

Security does not end when software reaches production. The runtime layer continuously assesses workloads and infrastructure after deployment.

Controls include

- Kubernetes RBAC
- Runtime threat detection
- Network segmentation
- Configuration monitoring
- Admission control
- Policy enforcement
- Cloud security posture assessment
- Workload behavior monitoring

Rahman et al. (2023) show that Kubernetes security misconfigurations are a significant concern, reinforcing the need for continuous configuration and privilege checks.

Layer 8: Monitoring, Response, and Governance

The final layer coordinates security visibility, decision-making, response, and accountability across the software supply chain.

- *Controls include*

- Centralized logging
- Security information and event management
- Automated incident response
- Policy-as-code
- Supply chain audit trails
- Compliance monitoring
- Risk reporting
- Artifact quarantine
- Credential revocation procedures

This layer provides a common view of security events across development and production environments. It also allows security teams to trace relationships between source changes, build events, artifacts, identities, and deployed workloads.

Continuous Verification Across the Framework

Continuous verification operates horizontally across all eight layers. Each major lifecycle transition acts as a policy decision point.

Before software progresses to the next stage, the framework checks whether relevant security requirements have been satisfied. For example, a container image may be required to have an approved SBOM, valid signature, verified provenance, acceptable vulnerability status, and authorized source before deployment.

Failure to satisfy these requirements may result in one of four responses:

Allow → Restrict → Quarantine → Deny

The decision depends on the risk associated with the software component, identity, or requested operation.

Trust Evidence and Policy Enforcement

Trust decisions within the framework are based on evidence rather than organizational location.

- *Important evidence includes*

- Verified identity
- Artifact signature
- Source repository status
- Dependency provenance
- SBOM information
- Vulnerability findings
- Build attestation
- Configuration compliance
- Runtime behavior
- Access history

This evidence can be evaluated by policy engines before access or deployment is granted. The approach reduces reliance on static trust assumptions and creates a documented basis for security decisions.

Framework Security Outcomes

The proposed architecture is intended to provide four primary outcomes.

First, it reduces implicit trust between development stages. Second, it improves traceability by linking software artifacts to identities, source code, dependencies, and build processes. Third, it limits the impact of credential or component compromise through least privilege and independent verification. Fourth, it improves detection and response by maintaining continuous visibility across the software lifecycle.

The framework does not assume that any single tool can secure the complete software supply chain. Its purpose is to connect existing technical controls through shared trust requirements, policy enforcement, and verifiable security evidence.

Securing APIs Under the Zero Trust Framework

APIs are fundamental to cloud-native systems because they connect microservices, CI/CD platforms, registries, cloud services, external applications, and administrative tools. This central role also makes them a significant attack surface. Weak authentication, excessive privileges, stolen access tokens, insecure service-to-service communication, and poorly governed interfaces can allow attackers to move between supply chain components or gain access to sensitive development and production resources.

A Zero Trust approach treats every API request as an independent access decision. Network location alone should not determine whether a request is trusted. Identity, authorization context, resource sensitivity, device or workload condition, and observed behavior should be evaluated before access is granted. Rose et al. (2020) emphasize that Zero

Trust architectures remove implicit trust and require explicit verification for access to protected resources.

API Authentication

Strong authentication provides the first control boundary for API access. Human users, services, workloads, and automated pipelines should use verifiable identities instead of shared accounts or static credentials.

- *Appropriate controls include*
- OAuth 2.0 and OpenID Connect
- Certificate-based authentication
- Workload identity
- Short-lived access tokens
- Mutual TLS
- Centralized identity providers

Long-lived API keys should be minimized because leaked credentials can remain usable for extended periods. Short-lived tokens reduce the window in which stolen credentials can be abused.

API Authorization

Authentication confirms identity, but authorization determines what that identity is permitted to do. Fine-grained access control is therefore essential.

Role-based access control can provide basic permission separation, while attribute-based access control can consider additional factors such as user role, service identity, resource type, location, workload status, and security risk. Chandramouli et al. (2021) show that attribute-based authorization can support more precise access decisions within microservice environments.

Administrative APIs, registry APIs, deployment endpoints, and CI/CD control interfaces should receive particularly strict authorization because compromise could affect the wider software supply chain.

Service-to-Service Security

Cloud-native applications often contain dozens or hundreds of services that communicate automatically. These internal interactions should not be assumed trustworthy simply because they occur within the same cluster.

Mutual TLS can provide bidirectional authentication and encryption between services. Service meshes can also support identity enforcement, certificate management, traffic policy, and access restrictions. Chandramouli and Butcher (2020) describe service mesh architecture as an important mechanism for securing communication among microservices.

Each workload should have a distinct identity, and access should be limited to the specific services required for its function.

API Discovery and Inventory

Organizations cannot adequately protect APIs that they do not know exist. Shadow APIs, outdated endpoints, development interfaces, and undocumented services can create unnoticed exposure.

API governance should therefore maintain an inventory containing

- API owner
- Purpose
- Authentication method
- Data classification
- Exposed operations
- Connected services
- Current version
- Security status

Unused or obsolete endpoints should be removed or disabled. Continuous discovery can also help detect unauthorized APIs introduced outside formal development processes.

API Monitoring and Threat Detection

Authentication and authorization controls should be supported by continuous monitoring. Suspicious activity may include repeated authentication failures, abnormal request volumes, unusual token use, unexpected service relationships, privilege escalation attempts, or access from previously unseen workloads.

API telemetry should be correlated with identity, CI/CD, cloud, and workload logs. This allows security teams to determine whether unusual API behavior is part of a broader software supply chain compromise.

Securing Containers and Cloud-Native Workloads

Containers provide portability and deployment consistency, but they also introduce security concerns involving image integrity, vulnerable packages, registry access, excessive privileges, insecure configuration, and runtime behavior. Container security must therefore begin during image creation and continue throughout deployment and operation.

Sultan et al. (2019) identify image security, isolation, access control, runtime protection, and host security as central aspects of container security. A Zero Trust model extends these controls by requiring evidence that a workload remains trustworthy before and after deployment.

Secure Container Image Development

Container security begins with the creation of the image. Images should use trusted base images and contain only the software required by the application.

- *Recommended practices include*
- Minimal base images
- Removal of unnecessary packages
- Non-root execution
- Secure handling of secrets

- Version pinning
- Controlled base-image repositories
- Reproducible build practices

Reducing unnecessary components limits the number of vulnerabilities and potential attack paths contained within an image.

Image Vulnerability Scanning

Container images should be scanned before being stored in a production registry and again before deployment. Scanning should identify operating-system vulnerabilities, vulnerable libraries, embedded secrets, malware, and prohibited software packages.

However, vulnerability scanning alone does not prove that an image is legitimate. Scanning should therefore operate alongside provenance and integrity verification.

Container Image Signing and Verification

Cryptographic signing allows an organization to verify whether an image came from an approved build process and whether it has changed since signing.

- *Before deployment, the platform should verify*
- The image signature
- Approved signer identity
- Artifact digest
- Build provenance
- Policy compliance
- Vulnerability status

Unsigned or improperly signed images should not be permitted to enter production environments.

Registry Security

Container registries are important trust points because they store deployable software artifacts. Attackers who gain registry access may replace legitimate images or publish malicious versions under trusted names.

Registry protection should include

- Strong authentication
- Least-privilege permissions
- Immutable repositories where appropriate
- Audit logging
- Image signing
- Access monitoring
- Separation between development and production registries

Credentials used by CI/CD pipelines to publish images should be narrowly scoped and short-lived.

Kubernetes Admission Control

Admission controls provide an opportunity to enforce security policy before workloads are accepted by a Kubernetes cluster.

A deployment may be rejected when

- The image is unsigned
- Provenance is missing
- The image contains critical vulnerabilities
- Privileged execution is requested
- Unapproved registries are used
- Security configuration requirements are not satisfied

This provides an important Zero Trust enforcement point between artifact storage and runtime execution.

Runtime Container Protection

Trust should continue to be evaluated after deployment. Runtime monitoring can detect activities that were not visible during image scanning, including abnormal process execution, unexpected file modifications, suspicious network connections, credential access, and privilege escalation.

Runtime controls should compare observed behavior with expected workload behavior and generate alerts when significant deviations occur.

Least-Privilege Kubernetes Configuration

Kubernetes workloads should receive only the permissions required for their operational role. Excessive privileges increase the impact of credential theft or workload compromise.

Rahman et al. (2023) found that Kubernetes manifests frequently contain security misconfigurations, highlighting the importance of policy enforcement and configuration review. Appropriate controls include restrictive RBAC policies, service-account separation, workload-specific permissions, network policies, and secure secret management.

Securing Open-Source and Third-Party Dependencies

Open-source and third-party components are deeply embedded in modern software development. Although they reduce development time and provide access to mature functionality, they also introduce security risks that organizations do not fully control.

Dependency risk includes vulnerable libraries, malicious packages, outdated software, compromised maintainers, dependency confusion, typosquatting, and hidden transitive dependencies. Duan et al. (2021) show that package-management ecosystems can provide attackers with opportunities to distribute malicious code through trusted software relationships.

A Zero Trust dependency model requires that packages be verified based on origin, integrity, security status, and organizational policy rather than automatically accepted because they appear in a recognized public repository.

Dependency Discovery and Inventory

Organizations should maintain a reliable record of direct and transitive dependencies used in their software.

The inventory should identify

- Package name

- Version
- Source repository
- Maintainer or publisher
- License
- Dependency relationship
- Known vulnerabilities
- Application usage

Without accurate dependency visibility, organizations may be unaware that a newly disclosed vulnerability affects deployed systems.

Software Composition Analysis

Software composition analysis can identify known vulnerabilities, outdated components, license concerns, and dependency relationships.

SCA should be integrated into development and CI/CD processes so that risk is identified before deployment. High-risk dependencies may require remediation, replacement, or additional review before software can progress through the pipeline.

SBOM-Based Dependency Visibility

Software bills of materials provide a structured record of components contained within an application. Xia et al. (2023) identify SBOMs as an important mechanism for improving software component transparency, while also noting challenges related to completeness, quality, and practical adoption.

- *SBOMs can support*
- Vulnerability impact analysis
- Incident response
- Dependency governance
- Regulatory reporting
- Software provenance

Supplier risk assessment

They should be generated automatically and updated whenever application dependencies change.

Dependency Provenance Verification

Knowing that a package exists is not enough. Organizations should also determine where the package came from and whether its origin can be trusted.

- *Provenance checks may consider*
- Repository source
- Publisher identity
- Package signature
- Build information
- Release history
- Approved package policy

This helps reduce the likelihood that malicious or substituted packages enter a trusted application.

Dependency Confusion and Typosquatting Prevention

Dependency confusion occurs when build systems retrieve an unintended external package instead of the approved internal dependency. Typosquatting relies on developers accidentally installing malicious packages with names similar to legitimate ones.

- *Controls should include*
- Private package repositories
- Explicit repository configuration
- Package namespace controls
- Dependency pinning
- Package allowlists
- Automated origin verification

These controls reduce reliance on package names alone as indicators of trust.

Vulnerability Prioritization and Patch Management

Not every vulnerability creates the same level of risk. Vulnerability management should consider exploitability, application exposure, dependency reachability, business importance, and runtime use.

Pashchenko et al. (2018) found that the presence of a vulnerable dependency does not necessarily mean that vulnerable code is actually used. This supports a more contextual approach to remediation rather than treating every dependency vulnerability as equally urgent.

Transitive Dependency Risk

Applications often inherit dependencies indirectly through other packages. These transitive dependencies may be difficult for developers to identify or manage.

Decan et al. (2018) demonstrated how vulnerabilities can propagate through package dependency networks. For this reason, dependency assessment should cover the entire dependency tree rather than only libraries explicitly selected by developers.

Securing CI/CD Pipelines and Build Infrastructure

CI/CD pipelines connect source code to production and often possess access to repositories, secrets, signing keys, container registries, cloud platforms, and deployment environments. This concentration of privilege makes CI/CD infrastructure one of the most sensitive components in the software supply chain.

Chandramouli et al. (2024) emphasize that software supply chain protections should be integrated directly into DevSecOps CI/CD pipelines. Security should therefore be enforced throughout code integration, dependency resolution, building, testing, artifact creation, and deployment.

Pipeline Identity and Authentication

Every CI/CD runner, automation process, and deployment service should have a distinct machine identity. Shared accounts make accountability difficult and increase the consequences of credential compromise.

Pipeline authentication should use

- Workload identity
- Short-lived tokens
- Certificate-based authentication
- Federated identity
- Restricted service accounts

Identity should be validated before access is granted to repositories, registries, cloud environments, or deployment systems.

Secrets and Credential Management

Secrets should not be embedded directly in source code, pipeline definitions, scripts, or container images.

Secure pipelines should use managed secret stores and provide credentials only when required. Controls should include automatic rotation, limited lifetime, restricted access, audit logging, and secret scanning.

If a credential is exposed, automated revocation should occur as quickly as possible.

Secure Build Environments

Build infrastructure should be isolated from unnecessary systems and should not retain state between unrelated builds.

Ephemeral build environments reduce the risk that malicious files or credentials remain available for later jobs. Build runners should also operate with limited network access and minimal privileges.

Reproducible build practices can provide further assurance by allowing independent verification that the produced artifact corresponds to the expected source and build instructions (Lamb & Zacchiroli, 2022).

Pipeline Code Protection

CI/CD workflow definitions are security-sensitive code because changes can alter testing, signing, deployment, and access to secrets.

Pipeline configurations should therefore receive protections similar to application source code, including

Branch protection
Peer review
Signed commits
Change approval
Version control
Audit logging

Unauthorized modifications to deployment logic should trigger alerts or block the pipeline.

Build Artifact Signing

Artifacts should be signed after successful completion of approved build steps. The signature provides evidence that the artifact was produced by an authorized process and has not been altered afterward.

Verification should occur before artifacts are accepted into registries or deployed to production.

Software Provenance and Attestation

Provenance provides information about how an artifact was produced, including the source repository, build environment, tools, dependencies, and workflow steps involved.

Torres-Arias et al. (2019) demonstrate how in-toto can provide verifiable evidence about software supply chain operations. Such evidence can help organizations detect unauthorized changes introduced between source development and final deployment.

Secure Deployment Gates

Deployment should depend on successful completion of defined security requirements.

A production release may require

- Successful code review
- Passed security testing
- Approved dependency status
- Valid SBOM
- Verified artifact signature
- Provenance attestation
- Container vulnerability checks
- Authorized release approval

If required conditions are not met, the deployment should be blocked or quarantined.

Continuous Pipeline Monitoring

CI/CD activity should be continuously monitored for abnormal behavior such as unusual repository access, unexpected pipeline changes, unauthorized build execution, large secret requests, altered artifacts, or unusual deployment activity.

Pipeline telemetry should be linked with cloud, identity, registry, API, and runtime logs. This makes it easier to reconstruct attack paths and respond when a compromise moves across several stages of the software supply chain.

Security Controls Against Major Software Supply Chain Attack Classes

Software supply chain attacks differ in technique, target, and impact, but most exploit weaknesses in trust relationships between developers, repositories, dependencies, build systems, registries, deployment platforms, and runtime services. Effective protection therefore requires controls that address both the initial point of compromise and the downstream paths available to an attacker. Ladisa et al.

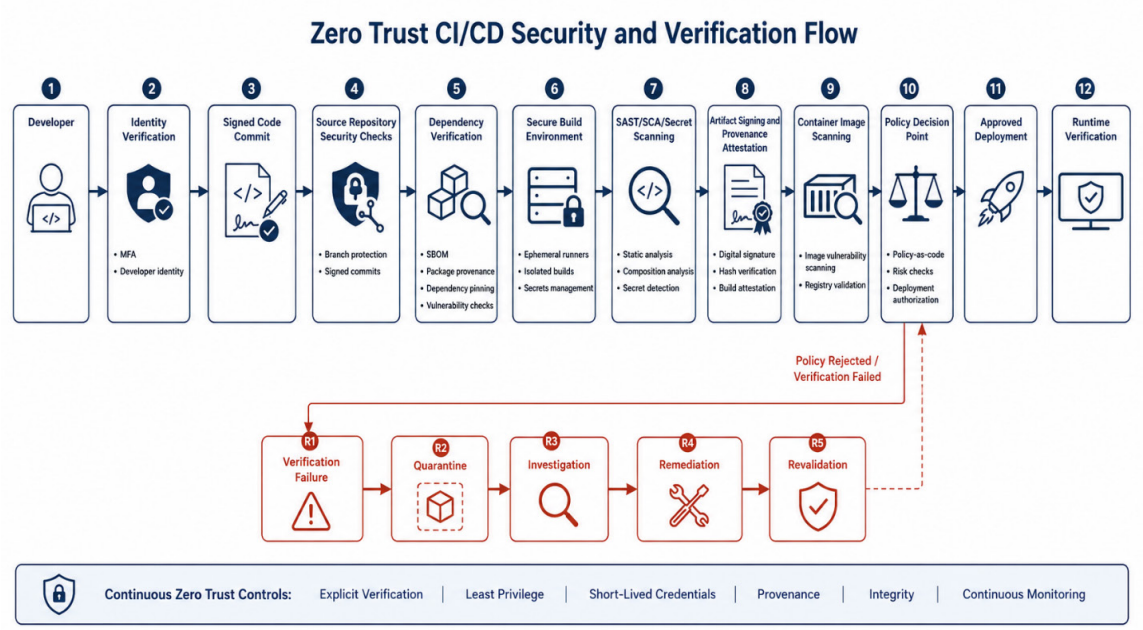


Fig 2: Zero Trust CI/CD security and verification flow for secure software delivery.

(2023) show that software supply chain attacks may target source code, dependencies, build environments, distribution mechanisms, and software delivery processes. The proposed Zero Trust framework addresses these attack classes through identity verification, least privilege, provenance validation, artifact integrity controls, policy enforcement, and continuous monitoring.

Dependency Confusion and Typosquatting

Dependency confusion exploits package-resolution behavior to cause a build system to retrieve an unintended external package instead of an approved internal dependency. Typosquatting relies on misleading package names that resemble legitimate software.

Controls should include private package repositories, namespace protection, package allowlists, dependency pinning, provenance verification, and software composition analysis. Package origin should be verified before installation rather than inferred from a package name or repository location. Duan et al. (2021) demonstrate that package-manager ecosystems can provide an effective distribution channel for malicious software when dependency trust is poorly controlled.

Repository and Source Code Compromise

Repository compromise may enable unauthorized code changes, malicious workflow modification, secret theft, or the insertion of backdoors into trusted projects. Strong developer authentication, protected branches, signed commits, mandatory peer review, limited repository permissions, and continuous audit logging can reduce this exposure.

Sensitive operations, including changes to build definitions or release configurations, should require additional authorization. A valid developer account should not automatically provide unrestricted authority over critical software delivery processes.

CI/CD Credential Theft

CI/CD systems commonly hold credentials for repositories, container registries, cloud environments, and deployment platforms. Stolen credentials can therefore provide access to several supply chain stages.

The proposed framework limits this risk through short-lived tokens, workload identity, managed secrets, automated rotation, least-privilege service accounts, and credential-use monitoring. Static secrets embedded within scripts or pipeline definitions should be prohibited.

Build System Compromise

A compromised build system may modify software after source-code review, allowing malicious behavior to appear only in the final artifact. This makes build integrity a central supply chain concern.

Controls include isolated and ephemeral build environments, signed build outputs, reproducible builds, provenance records, restricted network access, and verification of build tools. Torres-Arias et al. (2019) show that provenance mechanisms can provide evidence that expected software supply chain steps were performed by authorized entities.

Container Image Poisoning

Container image poisoning occurs when malicious or altered images are introduced into a registry or deployment process.

Table 2: Mapping Major Software Supply Chain Attacks to Zero Trust Controls

Attack Class	*Primary Target*	*Main Risk*	*Primary Zero Trust Control*	*Supporting Controls*
Dependency confusion	Package ecosystem	Execution of unintended malicious package	Provenance verification	Private registries, dependency pinning
Typosquatting	Dependency selection	Installation of deceptive package	Package allowlisting	SCA, namespace controls
Repository compromise	Source code	Unauthorized modification	Strong identity verification	Signed commits, branch protection
CI/CD credential theft	Pipeline	Unauthorized access and deployment	Short-lived credentials	Secrets vault, workload identity
Build compromise	Build environment	Creation of backdoored artifact	Isolated builds	Attestation, reproducible builds
Container image poisoning	Registry	Deployment of malicious image	Image signature verification	Admission control, scanning
API token theft	APIs	Unauthorized service access	Context-aware authentication	Token rotation, mTLS
Kubernetes privilege abuse	Cluster	Lateral movement and workload compromise	Least privilege	RBAC, workload identity
Artifact tampering	Artifact repository	Replacement or modification of software	Cryptographic integrity verification	Immutable storage, provenance
Secret leakage	Repository or pipeline	Credential compromise	Secret scanning and vaulting	Automated revocation and rotation

Protection requires image signing, registry access restrictions, immutable artifact practices, vulnerability scanning, and admission policies.

A Kubernetes platform should reject images that lack an approved signature, trusted provenance, or acceptable security status. Image identity should therefore be based on cryptographic evidence rather than image names or tags alone.

API Token Theft and Service Abuse

Stolen API credentials may provide unauthorized access to repositories, registries, cloud control planes, or internal microservices. API protection should use short-lived tokens, mutual TLS, workload identities, fine-grained authorization, and behavioral monitoring.

API access should also be reassessed when context changes, such as unexpected request locations, abnormal service interactions, or unusual privilege use.

Kubernetes Credential and Privilege Abuse

Kubernetes credentials can provide access to workloads, secrets, namespaces, and deployment functions. Excessive permissions may turn a single compromised service account into a wider cluster security incident.

Least-privilege RBAC, workload-specific identities, namespace restrictions, admission policies, and configuration monitoring should be used to limit lateral movement. Rahman et al. (2023) found that Kubernetes manifests frequently contain security misconfigurations, supporting the need for automated policy enforcement.

Artifact Tampering and Provenance Failure

Artifacts may be altered after a legitimate build but before deployment. Such attacks can bypass source-code security because the compromised software no longer corresponds to the reviewed source.

Digital signatures, cryptographic hashes, immutable artifact repositories, build attestations, and provenance verification can detect unauthorized changes. Provenance should form part of the deployment decision rather than being retained only for audit purposes.

Adaptive Trust Evaluation and Policy Enforcement

Static trust decisions are poorly suited to cloud-native software supply chains because security conditions can change rapidly. A dependency may become vulnerable after deployment, a signing key may be compromised, or a workload may begin behaving differently from its expected profile. The proposed framework therefore uses adaptive trust evaluation to reassess software components, identities, and deployment actions as new evidence becomes available.

Trust Scoring Model

A conceptual trust score can be used to combine several security indicators into a single decision input

- *where*

$T = w_1I + w_2P + w_3V + w_4C + w_5B$

- *T* = overall trust score
- *I*= identity confidence
- *P*= provenance confidence
- *V*= vulnerability posture
- *C*= configuration compliance
- *B*= behavioral trust
- $w_1 \ldots w_5$= weights assigned according to organizational risk priorities

Identity confidence may reflect authentication strength, credential age, and workload identity status. Provenance confidence may consider signature validity, approved build origin, and attestation completeness. Vulnerability posture reflects current security findings, while configuration compliance measures adherence to approved policies. Behavioral trust evaluates whether observed activity is consistent with the expected role of the identity or workload.

The model is conceptual and is intended to support structured decision-making rather than represent an empirically validated scoring algorithm.

Table 3: Illustrative Security Coverage Comparison

Security Dimension	*Traditional Approach*	*Proposed Zero Trust Framework*
API Security	60	92
Container Security	65	94
Dependency Security	50	91
CI/CD Security	55	95
Artifact Provenance	35	96
Identity Verification	60	94
Continuous Monitoring	55	93

Policy Decision Point

A policy decision point evaluates available trust evidence before granting access or allowing a software artifact to progress through the supply chain.

Four possible outcomes are proposed

- Allow: Security requirements are satisfied and risk is within an acceptable range.
- Restrict: Access is permitted with reduced privileges or additional controls.
- Quarantine: The component or artifact is isolated pending investigation.
- Deny: Access or deployment is blocked because trust requirements are not satisfied.

This structure is consistent with Zero Trust principles, where authorization is based on current identity, resource state, and contextual information rather than previous access alone (Rose et al., 2020).

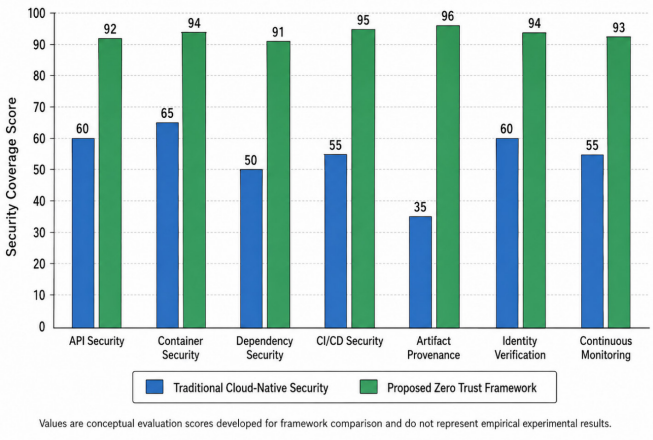
Continuous Reassessment

Trust should be recalculated when relevant conditions change. Reassessment may be triggered by

- Discovery of a new vulnerability
- Change in package provenance
- Credential compromise
- Failed signature verification

Table 4: Conceptual Risk Assessment Before and After Framework Controls

Threat	*Initial Risk*	*Main Framework Controls*	*Residual Risk*
Dependency poisoning	Critical	Provenance, SCA, allowlisting, SBOM	Medium
CI/CD credential theft	Critical	Short-lived identity, secrets vault, least privilege	Low
Container image tampering	High	Signing, hash verification, admission control	Low
API credential abuse	High	mTLS, short-lived tokens, contextual authorization	Medium
Build system compromise	Critical	Isolated builds, attestation, reproducible builds	Medium
Repository compromise	Critical	MFA, signed commits, branch protection	Low
Kubernetes privilege abuse	High	RBAC, workload identity, policy enforcement	Medium



Graph 1. Comparative security coverage of traditional cloud-native and proposed Zero Trust software supply chain frameworks.

- Unexpected artifact modification
- Policy violation
- Abnormal runtime behavior
- New threat intelligence
- Changes in workload configuration

For example, a container image that previously passed deployment checks may need to be reassessed when a critical vulnerability is later identified in one of its dependencies. Similarly, a workload using a compromised certificate should lose access even if the certificate was valid at the time of deployment.

Continuous reassessment prevents trust from becoming permanent and supports rapid response to changes in the software supply chain.

Framework Evaluation and Risk Assessment

The proposed framework is evaluated using security control coverage, verification capability, and expected risk reduction across major software supply chain stages. Because the study is conceptual, the evaluation does not claim measured production performance. Instead, it establishes a structured basis for comparing conventional security practices with the proposed Zero Trust model.

Evaluation Criteria

Evaluation focuses on indicators that can be measured in future experimental or operational deployments. Suitable metrics include

- Percentage of verified software dependencies
- Percentage of signed build artifacts
- SBOM coverage
- Provenance attestation coverage
- Percentage of deployments using verified identities
- Number of long-lived privileged credentials
- Percentage of container images passing admission controls
- CI/CD security check coverage
- Unauthorized deployment attempts blocked
- Mean time to detect suspicious activity
- Mean time to contain compromised components

These indicators measure whether the framework improves verification and reduces opportunities for implicit trust.

Comparative Security Coverage

A conceptual comparison can be made between a traditional cloud-native security approach and the proposed Zero Trust software supply chain framework. Traditional architectures may provide strong protection in selected areas while leaving gaps between development, build, deployment, and runtime controls. The proposed model seeks more consistent security coverage across the entire lifecycle.

The values are illustrative framework evaluation scores on a 0 to 100 scale and should not be interpreted as empirical findings. Their purpose is to demonstrate how the proposed framework can be evaluated once implemented within a controlled testbed or production environment.

Risk Reduction Analysis

Risk can be represented using a simple relationship

- R = L x I
- where:
- R= risk level
- L= likelihood of successful compromise
- I= potential impact

The proposed controls primarily seek to reduce the likelihood that an attacker can move from one compromised supply chain component to another. In some cases, least privilege and segmentation also reduce the potential impact of a successful compromise.

The residual risk values remain conceptual because no security architecture can eliminate software supply chain risk entirely. The objective is to reduce both the probability of successful compromise and the attacker's ability to propagate through trusted systems.

Research on reproducible builds, SBOMs, and software provenance supports this emphasis on verifiable evidence. Fourné et al. (2023) highlight the importance of reproducible builds for software supply chain assurance, while Xia et al. (2023) show that SBOMs can improve component visibility when implemented with sufficient completeness and quality. Together, these mechanisms support stronger risk assessment by improving knowledge of what software contains and how it was produced.

Interpretation of Framework Evaluation

The principal advantage of the proposed framework is not the presence of any single control, but the connection between

controls across the software lifecycle. Identity information can be linked with source changes, dependency records, build attestations, artifact signatures, deployment approvals, and runtime behavior.

This linkage improves traceability and reduces the possibility that a compromised component will continue through the supply chain simply because earlier stages were trusted. It also provides clearer evidence for incident investigation and recovery.

Future empirical evaluation should test these assumptions through controlled attack scenarios involving dependency poisoning, build tampering, credential theft, registry compromise, and unauthorized Kubernetes deployment.

Governance and Operational Management

Technical controls alone cannot secure a software supply chain. Effective protection also depends on clear ownership, enforceable policies, auditability, and coordinated response across development, security, platform, and operations teams. Governance provides the structure needed to ensure that Zero Trust principles are applied consistently rather than only within isolated tools or projects.

Software Supply Chain Governance

Organizations should establish formal governance for source repositories, dependencies, CI/CD systems, container registries, APIs, cloud workloads, and production deployments. Policies should define approved development practices, trusted software sources, minimum security checks, access requirements, and release conditions.

The Secure Software Development Framework emphasizes that secure development requires organizational preparation, protection of development environments, secure software production, and structured vulnerability response (Souppaya et al., 2022). These principles support the governance model proposed in this study.

Governance policies should address

- Approved repositories and package sources
- Dependency review requirements
- SBOM generation
- Artifact signing
- Build provenance
- Secrets management
- CI/CD access controls
- Container security
- API authorization
- Production deployment approval
- Incident response

Security requirements should be incorporated into development workflows so that compliance becomes part of routine software delivery.

Roles and Responsibilities

Clear responsibility is necessary because software supply chain security crosses several organizational functions.

Development teams should maintain secure code, review dependencies, protect credentials, and follow repository policies. DevSecOps teams should integrate security checks into CI/CD pipelines and ensure that build and deployment processes meet defined requirements. Platform and cloud security teams should manage Kubernetes controls, workload identities, registries, and runtime policies.

Security operations teams should monitor suspicious activity and coordinate incident response. Risk and compliance teams should maintain governance requirements, audit evidence, and security reporting.

Shared responsibility should not result in unclear accountability. Each critical supply chain asset should have a defined owner who is responsible for its security condition and remediation decisions.

Policy-as-Code

Policy-as-code allows security requirements to be expressed in machine-readable form and automatically enforced during development and deployment.

Examples include policies that

- Block unsigned container images
- Reject unapproved dependencies
- Prevent privileged containers
- Require valid provenance
- Enforce approved registries
- Restrict insecure Kubernetes configurations
- Prevent deployment of artifacts with critical vulnerabilities

Automated enforcement reduces dependence on manual review and helps maintain consistent controls across large development environments.

Software Supply Chain Auditability

Auditability is necessary for accountability, compliance, and incident investigation. Organizations should maintain evidence linking software from development through deployment.

Relevant records include

- Source-code changes
- Commit signatures
- Build logs
- Dependency inventories
- SBOMs
- Artifact signatures
- Provenance attestations
- Registry activity
- Deployment approvals
- API access logs
- Workload identity records

Such records help security teams determine where a

compromised component entered the supply chain and which downstream systems may have been affected.

Incident Response and Recovery

Software supply chain incidents require rapid containment because compromised components can spread across multiple environments.

A structured response process should include

- Identification of the affected component or identity.
- Isolation of compromised artifacts or workloads.
- Revocation of affected credentials and signing keys.
- Identification of downstream deployments.
- Removal or blocking of malicious packages and images.
- Rebuilding software from verified sources.
- Revalidation of provenance and signatures.
- Redeployment of trusted artifacts.
- Review of the control failure that enabled the incident.

The response process should also update policies where necessary so that the same attack path cannot be reused.

Discussion

The findings of this study support the view that software supply chain security is fundamentally a problem of managing trust across interconnected development and deployment processes. Cloud-native applications rely on many components that are created, stored, modified, and executed by different systems. A security weakness in one stage can affect later stages even when those later systems are correctly configured.

This creates a limitation for security models that focus mainly on network boundaries or user access. Zero Trust provides a stronger basis because it requires explicit verification and assumes that internal systems may also become compromised. Rose et al. (2020) emphasize that trust should not be granted solely because an entity is located within an organizational network. The same principle applies to software artifacts, workloads, dependencies, and automated pipelines.

The proposed framework extends this reasoning across the software lifecycle. A dependency is evaluated according to origin, integrity, and vulnerability status. A build artifact is assessed through provenance and signing evidence. A container image must satisfy policy before deployment. APIs and services are authenticated independently, while runtime workloads remain subject to continuous monitoring.

One important implication is that software provenance becomes a security control rather than only an auditing mechanism. Provenance allows organizations to determine whether software was produced by an expected build process and whether required stages were completed. Torres-Arias et al. (2019) demonstrate how supply chain metadata can support verification of software production activities.

The framework also highlights the importance of machine identities. Automated pipelines, service accounts, Kubernetes workloads, registries, and APIs frequently perform operations without direct human involvement. These identities may possess substantial privileges. Weak governance of machine credentials can therefore undermine otherwise strong development controls.

Another important finding is that no single security technology can provide complete software supply chain protection. SBOMs improve visibility, but they do not prove integrity. Vulnerability scanners identify known weaknesses, but they may not detect malicious packages. Artifact signing verifies integrity, but it does not guarantee that the build process itself was trustworthy. Effective protection depends on combining these mechanisms and applying them at appropriate decision points.

The framework therefore complements DevSecOps rather than replacing it. DevSecOps integrates security into development workflows, while Zero Trust provides a stronger trust model for deciding whether identities, artifacts, workloads, and deployment actions should be accepted. Together, these approaches can reduce fragmented security practices and improve control across cloud-native software delivery.

Challenges and Limitations

Although the proposed framework provides broad coverage across cloud-native software supply chains, several practical and methodological limitations remain.

First, implementing Zero Trust across existing CI/CD environments can be complex. Many organizations operate development platforms that were not designed for short-lived credentials, artifact provenance, workload identity, or continuous policy evaluation. Introducing these controls may require significant architectural change.

Second, security tooling can increase operational overhead. Continuous scanning, provenance generation, signing, admission checks, and runtime monitoring consume processing resources and may increase pipeline execution time. Organizations must therefore balance security requirements with development speed and system performance.

Third, SBOM quality remains inconsistent. An SBOM is valuable only when it accurately represents the software being used. Missing transitive dependencies, inconsistent package identifiers, or outdated inventories may reduce its usefulness. Xia et al. (2023) identify several practical challenges in SBOM adoption, including generation quality and operational use.

Fourth, vulnerability detection is imperfect. Security tools may generate false positives or fail to identify malicious software that does not match known vulnerability signatures. Software composition analysis should therefore be combined with provenance verification, behavioral monitoring, and package governance.

Fifth, software provenance also has limitations. Valid provenance confirms information about how software was

produced, but it does not necessarily prove that every tool involved in the build environment was uncompromised. Build infrastructure therefore remains a high-value target.

Sixth, Zero Trust policy design can become difficult in complex Kubernetes environments. Excessively strict policies may disrupt legitimate workloads, while permissive policies may weaken security. Policy tuning requires accurate knowledge of application behavior and operational dependencies.

The study also has a methodological limitation because the proposed framework is conceptual. The comparative scores and residual risk assessments presented earlier are illustrative rather than experimentally measured. They are intended to provide a basis for future testing and should not be interpreted as empirical evidence of security improvement.

Future Research Directions

Automated Software Supply Chain Trust Scoring

Future research should develop and test quantitative methods for assessing the trustworthiness of software artifacts, dependencies, workloads, and machine identities. Such models could combine identity confidence, provenance quality, vulnerability exposure, configuration status, and runtime behavior.

Empirical work would be needed to determine appropriate weighting factors and decision thresholds.

Runtime Software Provenance Verification

Current provenance mechanisms are generally strongest during build and release stages. Future research should investigate methods for verifying whether a running workload still corresponds to the approved and signed artifact from which it originated.

This could improve detection of post-deployment modification and runtime tampering.

Security of AI-Generated Software and Dependencies

The growing use of code-generation tools introduces additional software supply chain questions. Generated code may reference unapproved libraries, insecure packages, or outdated dependencies. Future studies should examine how automated code generation affects dependency risk, provenance, code review, and secure development practices.

Machine Identity Security in CI/CD Environments

Machine identities deserve greater research attention because automated systems often possess privileged access to repositories, registries, APIs, cloud environments, and deployment tools.

Future work should examine

- Workload identity lifecycle management
- Automated certificate issuance
- Credential rotation
- Privilege minimization
- Identity-based anomaly detection
- Compromise containment

Context-Aware Dependency Risk Prioritization

Current dependency management often relies heavily on vulnerability severity scores. Future models should incorporate exploitability, code reachability, application exposure, runtime use, business importance, and dependency provenance.

Pashchenko et al. (2018) show that vulnerable dependencies do not always present equal practical risk, supporting the need for more contextual prioritization.

Empirical Framework Validation

The proposed framework should be validated through controlled experiments and production case studies.

Potential evaluation environments include

- Kubernetes testbeds
- Cloud-native microservice applications
- Git-based repositories
- Container registries
- Automated CI/CD platforms
- Simulated dependency attacks
- Build-system compromise scenarios
- Credential theft exercises

Future experiments should measure detection rates, policy enforcement effectiveness, deployment delay, false positives, recovery time, and residual risk.

Conclusion

Cloud-native software development has increased the speed and flexibility of software delivery, but it has also expanded the number of trust relationships that attackers can exploit. APIs, open-source dependencies, containers, build systems, registries, machine identities, and CI/CD pipelines now form a connected software supply chain in which compromise at one stage may affect multiple downstream systems.

This study proposed a Zero Trust Software Supply Chain Security Framework that applies explicit verification, least privilege, provenance validation, artifact integrity, identity-based access control, policy enforcement, and continuous monitoring across the software lifecycle. The framework integrates controls for source repositories, dependencies, build infrastructure, CI/CD pipelines, containers, APIs, Kubernetes environments, and runtime workloads.

The central argument is that software should not be trusted merely because it originates from an internal repository, approved pipeline, or recognized registry. Trust should be supported by verifiable evidence concerning

identity, origin, integrity, vulnerability status, configuration, and behavior. SBOMs, signed artifacts, build attestations, workload identities, short-lived credentials, and policy-based deployment gates provide important mechanisms for establishing this evidence.

The framework also emphasizes governance, accountability, and auditability as necessary complements to technical controls. Although further empirical evaluation is required, the proposed model provides a structured basis for reducing implicit trust and strengthening resilience against modern software supply chain attacks in cloud-native environments.

References

[1] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). *Zero trust architecture* (NIST Special Publication 800-207). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-207.

[2] Chandramouli, R., & Butcher, Z. (2023). *A zero trust architecture model for access control in cloud-native applications in multi-location environments* (NIST Special Publication 800-207A). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-207A.

[3] Syed, N. F., Shah, S. W., Shaghaghi, A., Anwar, A., Baig, Z., & Doss, R. (2022). Zero trust architecture (ZTA): A comprehensive survey. *IEEE Access, 10*, 57143–57179. https://doi.org/10.1109/ACCESS.2022.3174679.

[4] Chandramouli, R., & Butcher, Z. (2020). *Building secure microservices-based applications using service-mesh architecture* (NIST Special Publication 800-204A). National Institute of Standards and Technology.

[5] Potla, R. (2023). Designing a BTP-centric integration mesh for shop-floor IoT, MES and ERP in discrete manufacturing. Journal of Artificial Intelligence, Machine Learning and Data Science, 1(2), 1-8.

[6] Chandramouli, R., Butcher, Z., & Chetal, A. (2021). *Attribute-based access control for microservices-based applications using a service mesh* (NIST Special Publication 800-204B). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-204B.

[7] Chandramouli, R., Kautz, F., & Torres-Arias, S. (2024). *Strategies for the integration of software supply chain security in DevSecOps CI/CD pipelines* (NIST Special Publication 800-204D). National Institute of Standards and Technology.

[8] Souppaya, M., Scarfone, K., & Dodson, D. (2022). *Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities* (NIST Special Publication 800-218). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-218.

[9] Ladisa, P., Plate, H., Martinez, M., & Barais, O. (2023). SoK: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)* (pp. 1509–1526). IEEE. https://doi.org/10.1109/SP46215.2023.10179304.

[10] Okafor, C., Schorlemmer, T. R., Torres-Arias, S., & Davis, J. C. (2022). SoK: Analysis of software supply chain security by establishing secure design properties. In *Proceedings of the 2022 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (pp. 15–24). Association for Computing Machinery. https://doi.org/10.1145/3560835.3564556.

[11] Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., & Cappos, J. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)* (pp. 1393–1410). USENIX Association.

[12] Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 23–43). Springer. https://doi.org/10.1007/978-3-030-52683-2_2.

[13] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., & Lee, W. (2021). Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2021)*. Internet Society. https://doi.org/10.14722/ndss.2021.23055.

[14] Vu, D. L., Pashchenko, I., Massacci, F., Plate, H., & Sabetta, A. (2020). Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2093–2095). Association for Computing Machinery. https://doi.org/10.1145/3372297.3420015.

[15] Cappos, J., Samuel, J., Baker, S. M., & Hartman, J. H. (2008). A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (pp. 565–574). Association for Computing Machinery. https://doi.org/10.1145/1455770.1455841.

[16] Potla, R. B. (2024). Optimizing extended warehouse management for make-to-order plants: Slotting, wave picking, and yard orchestration at scale. Journal of Computer Science and Technology Studies, 6(3), 181-192.

[17] Samuel, J., Mathewson, N., Cappos, J., & Dingledine, R. (2010). Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (pp. 61–72). Association for Computing Machinery. https://doi.org/10.1145/1866307.1866315.

[18] Kuppusamy, T. K., Torres-Arias, S., Diaz, V., & Cappos, J. (2016). Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (pp. 567–581). USENIX Association.

[19] Kuppusamy, T. K., Diaz, V., & Cappos, J. (2017). Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (pp. 673–688). USENIX Association.

[20] Kunaparaju, C. (2024). Detecting and Preventing State-Sponsored Cyber Attacks Using Deep Learning and Behavioral Analytics. Journal of Electrical Systems, 20, 5471-5486.

[21] Zimmermann, M., Staicu, C. A., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)* (pp. 995–1010). USENIX Association.

[22] Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering, 23*(1), 384–417. https://doi.org/10.1007/s10664-017-9521-5.

[23] Decan, A., Mens, T., & Constantinou, E. (2018). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (pp. 181–191). Association for Computing Machinery. https://doi.org/10.1145/3196398.3196401.

[24] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., & Massacci, F.

(2018). Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Article 42, pp. 1–10). Association for Computing Machinery. https://doi.org/10.1145/3239235.3268920.

[25] Lamb, C., & Zacchiroli, S. (2022). Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software, 39*(2), 62–70. https://doi.org/10.1109/MS.2021.3073045.

[26] Fourné, M., Wermke, D., Enck, W., Fahl, S., & Acar, Y. (2023). It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *2023 IEEE Symposium on Security and Privacy (SP)* (pp. 1527–1544). IEEE. https://doi.org/10.1109/SP46215.2023.10179320.

[27] Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2630–2642). IEEE. https://doi.org/10.1109/ICSE48619.2023.00219.

[28] Jadala, S. K. (2024). Zero Trust Architecture and Identity Threat Detection for Securing Cloud, IoT, and Hybrid Enterprise Systems. International Journal of Humanities and Information Technology, 6(04), 141-185.

[29] Zahan, N., Lin, E., Tamanna, M., Enck, W., & Williams, L. (2023). Software bills of materials are required. Are we there yet? *IEEE Security & Privacy, 21*(2), 82–88. https://doi.org/10.1109/MSEC.2023.3237100.

[30] Stalnaker, T., Wintersgill, N., Chaparro, O., Di Penta, M., German, D. M., & Poshyvanyk, D. (2024). BOMs away! Inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (pp. 517–529). Association for Computing Machinery. https://doi.org/10.1145/3597503.3623347.

[31] Potla, R. B. (2024). A SOX/ITAR-aligned global ERP template for multi-plant manufacturers: Governance patterns and controls. J Artif Intell Mach Learn & Data Sci, 2(2), 3222-3232.

[32] O'Donoghue, E., Boles, B., Izurieta, C., & Reinhold, A. M. (2024). Impacts of software bill of materials (SBOM) generation on vulnerability detection. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (pp. 67–76). Association for Computing Machinery. https://doi.org/10.1145/3689944.3696164.

[33] Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE Access, 7*, 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732.

[34] Rahman, A., Shamim, M. S. I., Bose, D. B., & Pandita, R. (2023). Security misconfigurations in open source Kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology, 32*(4), Article 99, 1–36. https://doi.org/10.1145/3579639.

[35] Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology, 141*, 106700. https://doi.org/10.1016/j.infsof.2021.106700