CHAMPIONING TRUSTED RELEASES:

A SECURITY LEADER'S GUIDE TO ENSURING SOFTWARE INTEGRITY

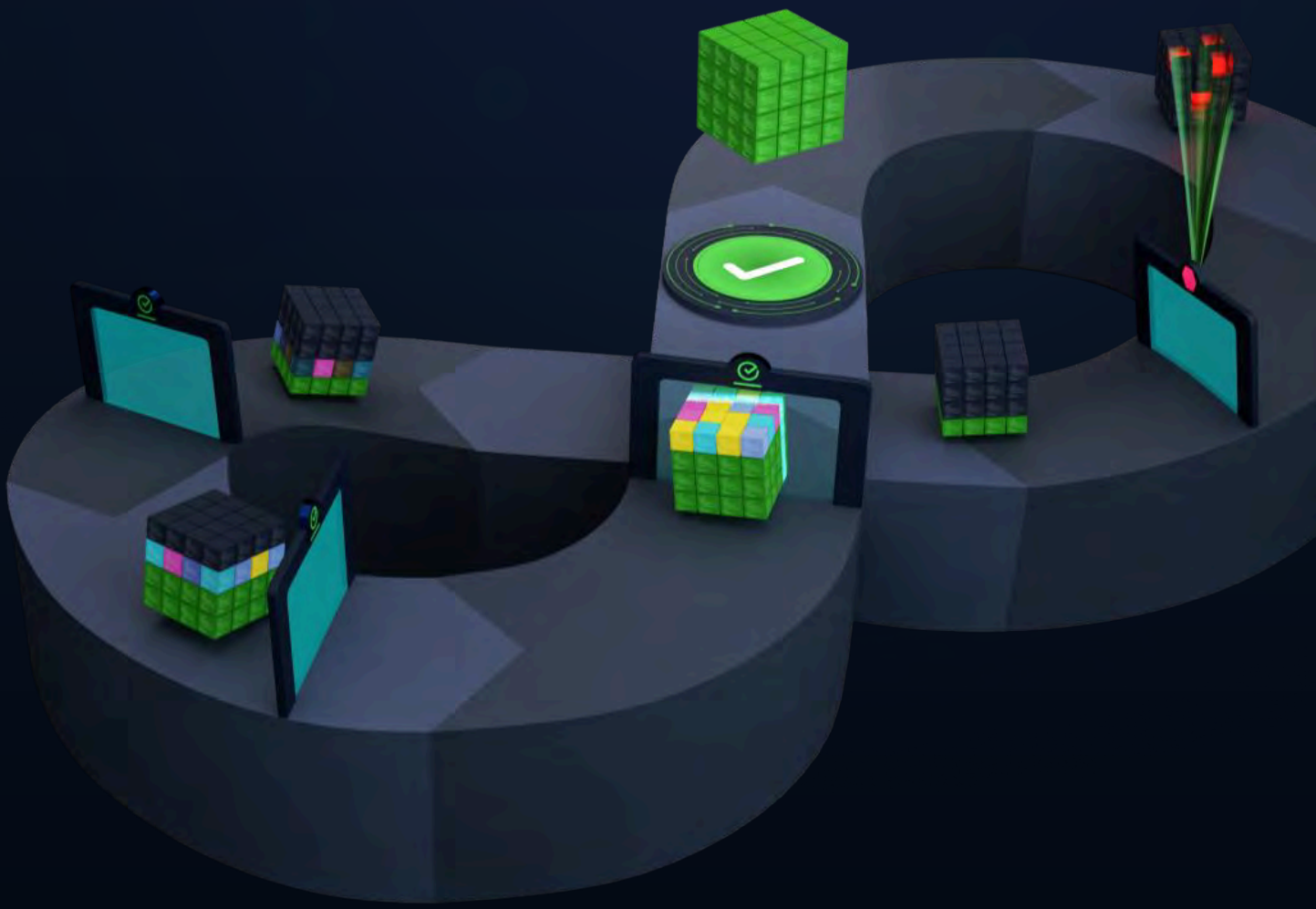


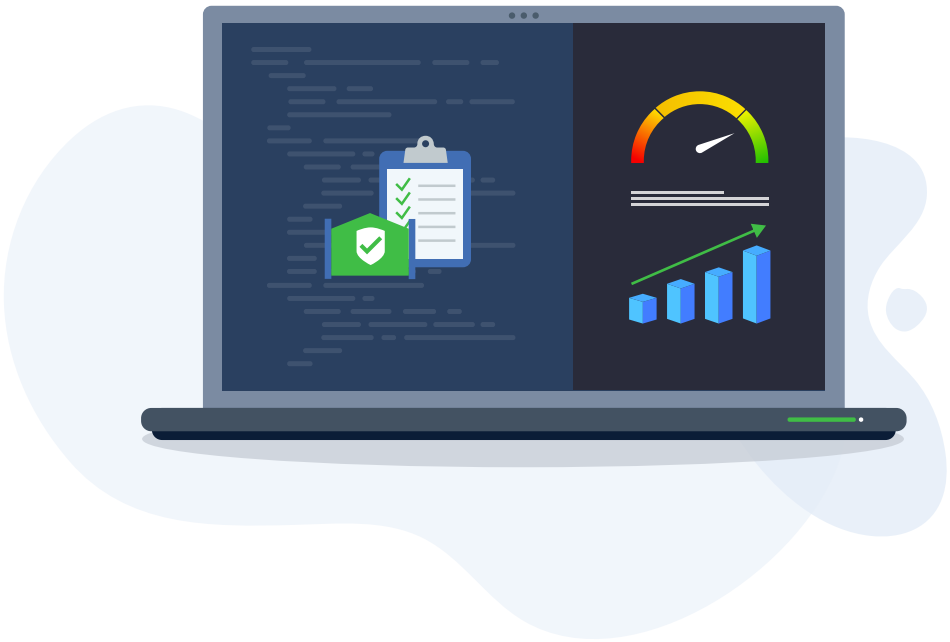
Table of Contents

EXECUTIVE SUMMARY

Enterprise security and compliance leaders and their development and operations counterparts know all too well that software applications represent one of the most difficult-to-defend attack surfaces in the enterprise. Cloud-native technologies, complex operations, prolific use of open source software (OSS), and the recent addition of AI/ML models, make today's software development lifecycle more difficult to secure and ensure the integrity of the applications within it.

This white paper serves as a guide for enterprise security and compliance leaders to:

- Better understand today's complex software supply chain and how it is more vulnerable than ever to security threats and other risks.
- Get a detailed overview of the types of evidence throughout the software supply chain that can be used to prove software integrity.
- Better govern the software development lifecycle through security gates and establish higher levels of trust in the organization's applications.



OVERVIEW

From a practical security standpoint, trusting an application means having verifiable assurance that it is secure, without hidden vulnerabilities, malicious code, or unintended functionality that could compromise data, systems, or user privacy. This trust extends beyond the application's immediate code to its entire lineage – every component, library, and dependency that has been used for every single release.

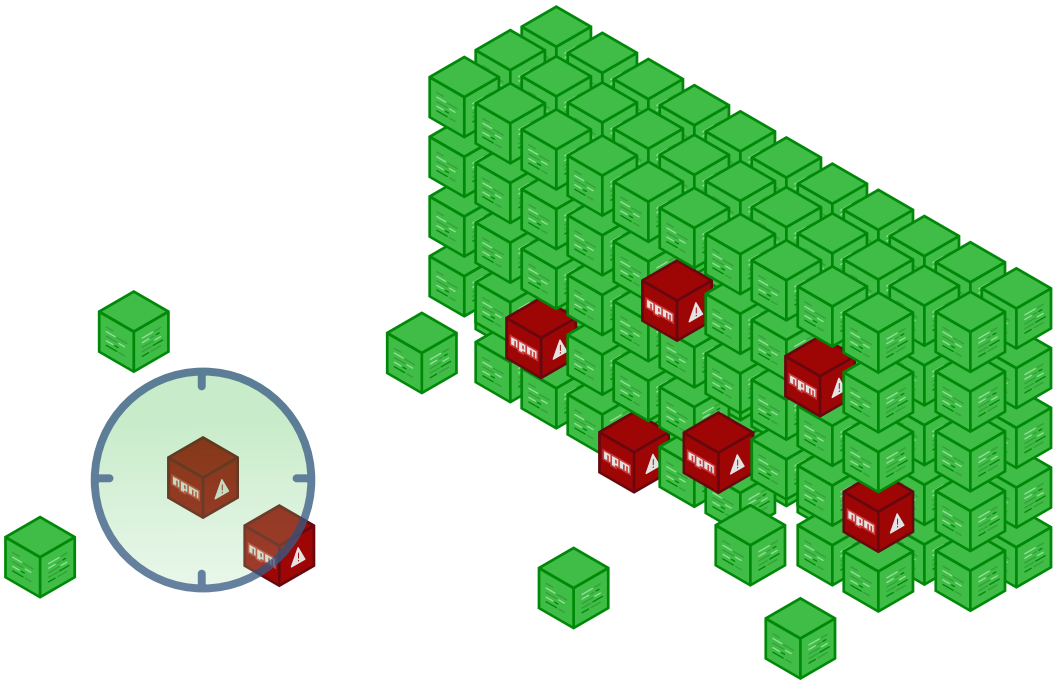
From a compliance perspective, trust translates into demonstrable adherence to a myriad of industry standards and governmental regulations, such as GDPR, HIPAA, PCI DSS, and emerging cybersecurity mandates. Organizations must prove, through rigorous audits, continuous monitoring and industry recognized documentation, that their applications meet the necessary security controls, protect sensitive data and maintain auditable tracking from development to deployment and through to runtime as well. Failure to do so can result in severe penalties, reputational damage, and loss of customer trust.

Ensuring the integrity of each and every application, however, presents a difficult challenge. The architectural shift towards microservices, containers, AI and machine learning models, and vast open-source ecosystems has introduced unprecedented complexity and dynamism. Applications are no longer monolithic entities built in isolated environments; they are distributed, ephemeral, and composed of hundreds, if not thousands, of interconnected components, many of which are third-party or open source. This fragmentation dramatically expands the attack surface, making it challenging for developers, who often operate under immense pressure to keep up with the release cadence, to inadvertently introduce vulnerabilities through misconfigurations, insecure coding practices, or incorporating compromised dependencies.

The rapid pace of continuous integration and continuous delivery (CI/CD) pipelines, while boosting agility, can also inadvertently accelerate the propagation of insecure code if security is not deeply embedded at every stage of development. Furthermore, the shared management responsibility found in cloud environments, often leads to ambiguity regarding security ownership, creating potential blind spots that attackers are eager to exploit.

Software supply chain breaches, such as Solarwinds and CodeCov, continue to proliferate– especially those which target open source software and the dependencies that developers rely heavily on. On September 8th, 2025, a malicious actor compromised the npm registry by publishing trojanized versions of 18 widely-used packages, after obtaining developers' tokens in a phishing attack. The injected payload was obfuscated with the popular

“javascript-obfuscator” library and contained a cryptocurrency stealer malware. The malware replaced methods of XMLHttpRequest class with its own, in order to monitor web3 traffic (for a deep technical analysis of this attack, read the detailed blog from JFrog’s Security Research team entitled, “**New compromised packages identified in largest npm attack in history (September 9th, 2025)**”.



The serious consequences of software supply chain attacks underscore why CISOs, Chief Compliance Officers, and security leaders are under immense and escalating pressure. They are on the front lines, tasked with defending increasingly complex digital perimeters against motivated and well-resourced adversaries. The "buck stops" with the CISO when a breach occurs, leading to intense scrutiny from boards, regulators, and the public. Beyond the immediate operational chaos and reputational damage, the legal ramifications for failing to prevent and report a security breach are severe and growing. Depending on the jurisdiction and the nature of the data compromised, organizations can face massive financial penalties (e.g., multi-million dollar fines under GDPR or CCPA), costly litigation from affected individuals or class-action lawsuits, and even criminal charges for executives in cases of gross negligence or intentional concealment. Furthermore, regulatory bodies can impose strict compliance obligations, revoke licenses, and mandate costly remediation efforts, all of which can cripple a business. The imperative for CISOs is clear: proactive, comprehensive security across the entire software supply chain is not just a best practice, but a legal and existential necessity.

THE SOFTWARE SUPPLY CHAIN

Today's software supply chain is a complex, interconnected ecosystem that encompasses every stage of an application's journey, from initial conception to coding, deployment and ongoing operations. It comprises every tool, service, code segment, and human interaction involved in developing and deploying an application, making it a prime target for threat actors seeking to compromise applications at scale.

The software supply chain can be categorized into two primary groups of elements:

- **Software Elements** - These are based on the people who code, use libraries, create applications and all the artifacts associated with those operations
- **Platform Elements** - These are the software tools that DevOps teams use to store, distribute, document and configure software components and the development environment.

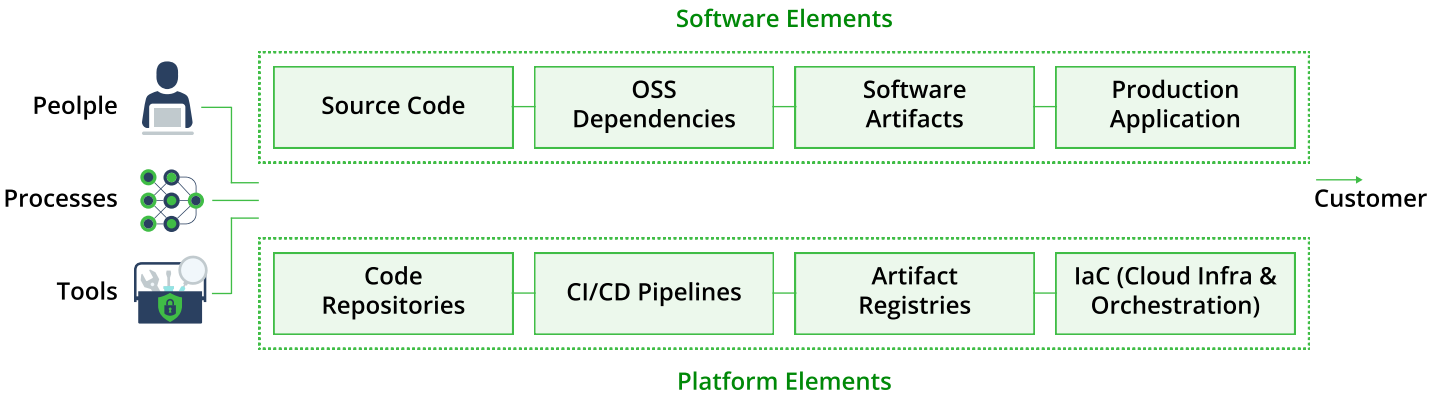


Fig. 1: The Software and Platform Elements of today's software supply chain

Platform Elements

Platform components refer to the infrastructure, tools, and systems that facilitate the development, testing, building, and deployment of software. Compromise at this layer can have far-reaching consequences, as it can affect all software flowing through the compromised platform.

Code Repositories

These systems manage changes to source code, track revisions, and enable collaboration among developers. They are the central repository for an application's intellectual property.

Common Vulnerabilities:

- **Weak Access Controls:** Insufficient authentication or authorization can allow unauthorized users to push malicious code, steal intellectual property, or tamper with existing code.
- **Compromised Credentials:** Stolen developer credentials can grant attackers direct access to repositories.
- **Lack of Branch Protection:** Unprotected branches can allow unreviewed or malicious code to be merged into production.
- **Exposed Secrets:** Sensitive information (API keys, passwords) accidentally committed to repositories.

Continuous Integration/Continuous Delivery (CI/CD) Pipelines

CI/CD pipelines are automated processes that integrate code changes, run tests, build artifacts, and deploy applications. As such, they play an essential role in ensuring that the latest versions are distributed and deployed.

Common Vulnerabilities:

- **Insecure Pipeline Configuration:** Misconfigured build steps, overly permissive permissions, or lack of input validation can allow injection of malicious commands or scripts.
- **Compromised Build Agents/Runners:** If a build agent is compromised, attackers can leverage its access to inject malware into compiled artifacts or exfiltrate data.
- **Dependency Confusion:** If not properly configured, CI/CD tools can fetch malicious packages from public registries instead of private ones.
- **Insufficient Logging and Monitoring:** Lack of visibility into pipeline execution makes it difficult to detect and respond to attacks.

Cloud Infrastructure and Orchestration

As applications migrate to the Cloud, code such as IaC is replacing hardware with software. This includes underlying cloud services and container orchestration platforms where applications are built, deployed, and run.

Common Vulnerabilities:

- **Misconfigurations:** Incorrectly configured security groups, IAM roles, storage buckets, or network policies can expose services and data.
- **Insecure API Endpoints:** Exposed or poorly secured APIs for managing cloud resources can act has entry points for malicious code.
- **Container Escapes:** Vulnerabilities in container runtimes or orchestration platforms can allow an attacker to break out of a container and access the host system.
- **Lack of Network Segmentation:** Flat networks in cloud environments can allow attackers to easily move laterally once a single component is compromised.

Software Elements

Software components are the building blocks of applications, and comprise the actual code, libraries, and frameworks that constitute the application itself. The increasing reliance on open-source software and third-party components significantly broadens the scope of potential vulnerabilities. According to JFrog's recent Software Supply Chain State of the Union 2025 report, fast-moving organizations are releasing one or more new packages every day.

Open-Source Libraries and Frameworks

Reusable code modules, often freely available, are used by developers and incorporated into their applications to accelerate development speed and increase reliability.

The fact is that Open Source Software (OSS) is a major component of today's software supply chain. Its prolific use allows developers to move fast in building applications, taking advantage of components and functions which have already been tested and are ready for deployment.

While this is certainly a positive step towards increasing development efficiency, the downside is that most vulnerabilities found in an organization's code base are unfortunately introduced via OSS. Finding and remediating OSS vulnerabilities, using techniques such as SCA to analyze direct and transitive dependencies, is critical, along with remediating vulnerabilities based on known CVEs, as early as possible in the development process.

Common Vulnerabilities:

- **Known Vulnerabilities (CVEs):** The most common issue, where a publicly disclosed vulnerability exists in a version of a library used by the application. (e.g., Log4j's Log4Shell).
- **Malicious Packages:** Attackers can inject malicious code directly into popular open-source projects or publish seemingly legitimate but compromised packages to public registries.
- **Outdated Dependencies:** Using old versions of libraries that contain unpatched vulnerabilities.
- **License Compliance Issues:** While not directly a security vulnerability, improper use of open-source licenses can lead to legal complications.

Third-Party APIs and Services

This refers to external services or APIs integrated into an application, such as payment gateways, authentication services, mapping APIs and AI/ML models.

Common Vulnerabilities:

- **Insecure API Keys/Credentials:** Hardcoding API keys or using weak authentication for external service calls.
- **Broken Authentication/Authorization:** Flaws in how the application authenticates or authorizes calls to third-party services, or vice-versa.
- **Data Exposure:** Third-party services mishandling sensitive data passed to them.
- **Rate Limitation Issues:** Lack of proper rate limiting on API calls can lead to denial-of-service or data enumeration attacks.

Proprietary (First-Party) Code

This is the unique code written by an organization's internal development teams. Common Vulnerabilities:

- **Common Weaknesses (OWASP Top 10):** Injection flaws (SQL, command), broken authentication, sensitive data exposure, XML external entities (XXE), broken access control, security misconfigurations, cross-site scripting (XSS), insecure deserialization, using components with known vulnerabilities, insufficient logging and monitoring.
- **Logic Flaws:** Errors in application logic that can be exploited such as bypassing business rules and privilege escalation.
- **Hardcoded Secrets:** Embedding sensitive credentials directly into the code.
- **Insecure Error Handling:** Revealing too much information in error messages that could aid an attacker.

Understanding these components and their associated vulnerabilities is the first step towards building a resilient software supply chain security strategy.

THE SDLC AND ITS ROLE IN SOFTWARE INTEGRITY

The software supply chain provides the foundational framework for ensuring that applications are not only functional and reliable, but also secure and compliant with business and regulatory requirements. It is what effectively enables your Software Development Lifecycle (SDLC). In the context of software integrity, integrating security into every phase of the SDLC, a practice often referred to as DevSecOps, is crucial. This proactive approach transforms security from a reactive, late-stage checkpoint into a continuous, data-driven effort.

SDLC Phases and Artifact Promotion Criteria

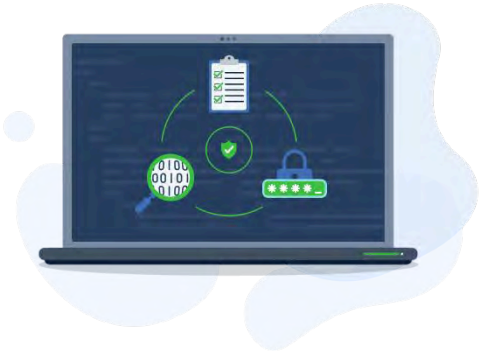
The key to a mature DevSecOps program is to define and enforce specific criteria, or "gates", that a software artifact must meet before it can be promoted to the next phase of the development cycle. These gates ensure that only verified and secure components proceed, creating a verifiable chain of trust. The sections below define each major SDLC phase and outline security and compliance criteria that software developing organizations are recommended to use in assessing whether an application component should be promoted to the next phase.

SDLC PHASE	RECOMMENDED CRITERIA FOR PROMOTION
 **Development and Coding** In this phase, developers write the source code. The focus is on producing high-quality, secure code while adhering to established standards.	■ The source code must pass Static Application Security Testing (SAST) with all high- and critical-severity vulnerabilities either remediated or formally accepted. ■ All third-party libraries and open-source components must be validated against a vulnerability database, and any known critical vulnerabilities must be addressed. ■ Code must be reviewed by a peer to ensure it meets both quality and secure coding standards.
 **Building and Integration** This phase compiles the source code and its dependencies into a single, deployable software application. This is a critical point for validating the integrity of all components included in a potential release	■ A complete and accurate Software Bill of Materials (SBOM) must be generated for the artifact, detailing all components, versions, and licenses. ■ The source code, binaries and all relevant dependencies must be scanned for known vulnerabilities, and all critical findings must be resolved before proceeding. ■ The build process must be reproducible and tamper-proof, with the final application cryptographically signed to prove its origin and integrity.

SDLC PHASE	RECOMMENDED CRITERIA FOR PROMOTION
 **Testing and Quality Assurance** In this phase the application is validated in a controlled environment to verify its functionality and security before it is released.	■ Dynamic Application Security Testing (DAST) must be performed on the running application to identify vulnerabilities that may not be visible in the source code. All critical findings must be addressed. ■ The application must pass a series of documented security test cases, confirming that it meets its security requirements. ■ If required, a formal penetration test must be conducted, with all findings remediated or documented as accepted risk.
 **Release and Deployment** This is the final and most important phase as it focuses on securely packaging and deploying the validated application to a production environment.	■ The software artifact must be cryptographically signed and attested, providing irrefutable proof that it is the same artifact that passed all previous gates. ■ All collected evidence—including SBOMs, SAST/DAST reports, and build logs—must be stored in a centralized, tamper-proof repository and linked to the final artifact. ■ The target production environment must meet all predefined security hardening and configuration standards.

PROVING SOFTWARE INTEGRITY

The strength of a software integrity claim depends on two major factors: (1) the extensiveness of the evidence you collect throughout the software development lifecycle and (2) how you govern the software supply chain based on that evidence.



This section offers a detailed overview of some of the more common types of evidence DevOps and application security teams should collect.

SLSA Attestations / Artifact Provenance

To combat the growing threat of supply chain attacks, software-producing organizations have rallied around a set of standards to ensure the integrity of software artifacts. One of the most prominent frameworks is SLSA, or Supply-chain Levels for Software Artifacts. Developed by the Open Source Security Foundation (OpenSSF), SLSA is a vendor-neutral, security-enhancing framework designed to prevent tampering and improve the integrity of software from its source code to its final, distributable artifact.

SLSA operates on a maturity model with four distinct levels, with each successive level building upon the previous one to provide greater assurance.

SLSA

SLSA LEVEL 1	SLSA LEVEL 2	SLSA LEVEL 3	SLSA LEVEL 4
A operates on a maturity model with four distinct levels, with each successive level building upon the previous one to provide greater assurance.	Demands that the build is hosted on a trusted service, where the build process is protected from tampering.	Adds stricter controls, such as a non-forgeable record of the build and a secure, hermetic build environment.	The highest level, requiring two independent, verified human reviewers and a fully hermetic, reproducible build process.

The Role of SLSA Attestations

The core mechanism for proving software integrity in the SLSA framework is the attestation. A SLSA attestation is a cryptographically signed, machine-readable statement that captures a detailed record of how a software artifact was produced. It's essentially a tamper-proof certificate for your software, providing all the critical information needed to verify its journey through the supply chain.

An attestation provides provenance—a verifiable chain of custody for the software. This provenance includes crucial details like:

- **The Source Code:** A definitive link to the exact commit hash in a version control system that was used to build the software.
- **The Build Process:** The specific CI/CD pipeline, build scripts, and commands that were executed.
- **The Build Environment:** Information about the environment where the build occurred, including its isolation, security controls, and the versions of tools used (e.g., compilers, dependencies).

When an organization consumes a software artifact, they can use the accompanying SLSA attestation to verify its integrity. By checking the cryptographic signature on the attestation, they can confirm that it was generated by a trusted party (e.g., the software vendor). They can then inspect the provenance to ensure the artifact was built from the expected source code in a secure, non-tampered-with manner. This process creates a powerful, auditable defense against many types of supply chain attacks, including those seen in the SolarWinds and CodeCov breaches, where malicious code was injected into the build or distribution process itself.

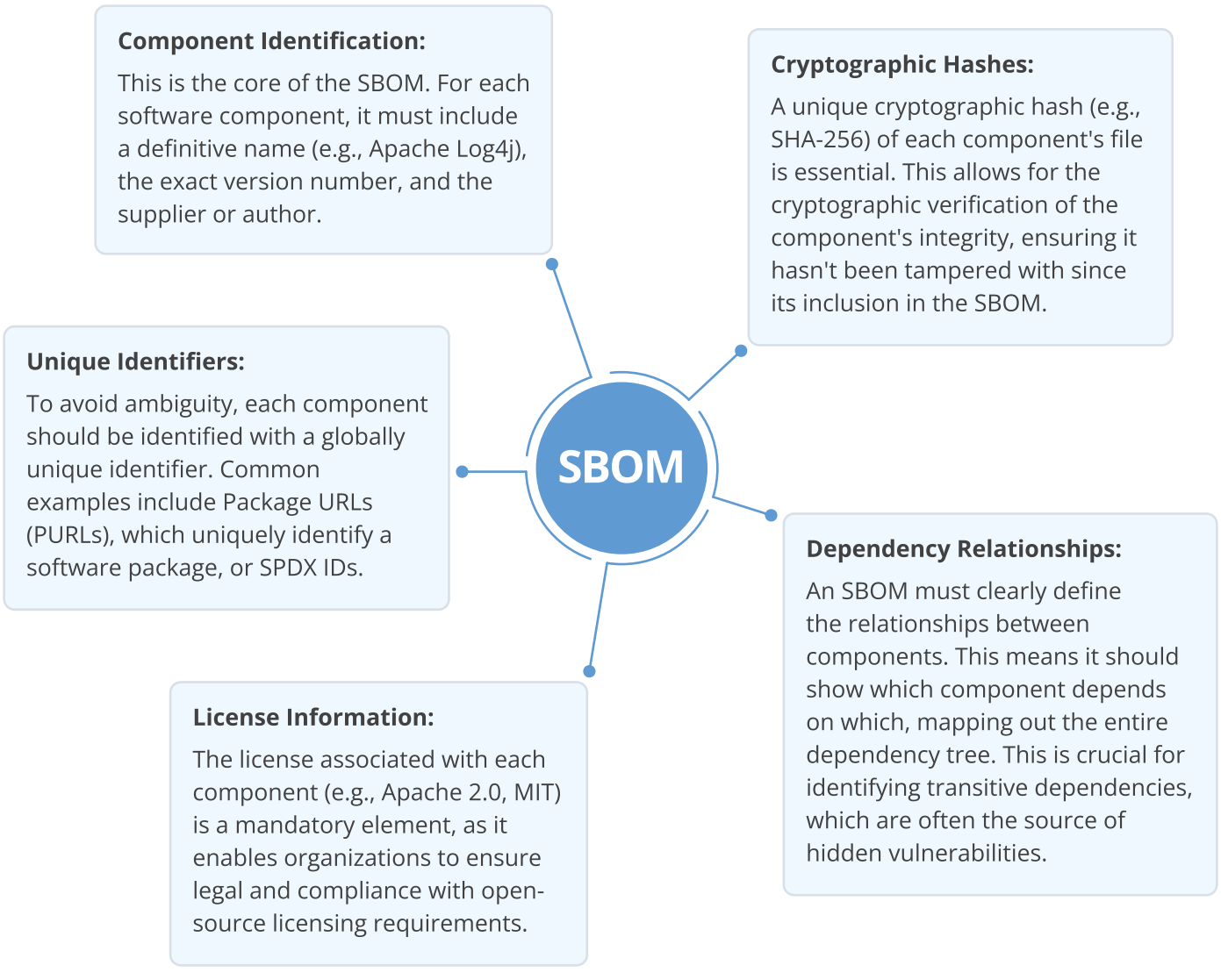
In essence, SLSA transforms the question of "Can I trust this software?" into a verifiable, data-driven process. By generating and consuming SLSA attestations, enterprises can confidently link the final software artifact to its secure source and build process, ensuring software integrity for their users and customers.

Software Bill of Materials (SBOM)

In the effort to secure the software supply chain, a foundational and indispensable tool is the Software Bill of Materials (SBOM). An SBOM is a formal, machine-readable list of ingredients that make up a software component. It provides a complete inventory of all the open-source and third-party components, libraries, and dependencies used to build a piece of software. This concept gained significant national importance with the issuance of Executive Order 14028 in 2021, which mandates that software producers provide SBOMs to their customers, particularly for those selling to the U.S. government. This order recognized the need for greater transparency to manage supply chain risks effectively.

What an SBOM Contains

A high-quality SBOM goes far beyond a simple list of software ingredients; it provides a detailed, structured dataset that allows security teams to manage risk proactively. The information it should contain is critical for its utility and can be broken down into several key categories:



Component Identification:

This is the core of the SBOM. For each software component, it must include a definitive name (e.g., Apache Log4j), the exact version number, and the supplier or author.

Cryptographic Hashes:

A unique cryptographic hash (e.g., SHA-256) of each component's file is essential. This allows for the cryptographic verification of the component's integrity, ensuring it hasn't been tampered with since its inclusion in the SBOM.

Unique Identifiers:

To avoid ambiguity, each component should be identified with a globally unique identifier. Common examples include Package URLs (PURLs), which uniquely identify a software package, or SPDX IDs.

Dependency Relationships:

An SBOM must clearly define the relationships between components. This means it should show which component depends on which, mapping out the entire dependency tree. This is crucial for identifying transitive dependencies, which are often the source of hidden vulnerabilities.

License Information:

The license associated with each component (e.g., Apache 2.0, MIT) is a mandatory element, as it enables organizations to ensure legal and compliance with open-source licensing requirements.

SBOM Quality

Not all SBOMs are created equal. The value of an SBOM is directly tied to the quality and thoroughness of the information it contains. Enterprise security leaders should evaluate an SBOM based on the following criteria:

- **Completeness:** A high-quality SBOM must list all components, including all ***transitive dependencies***. An SBOM that only lists top-level components is of limited use as it fails to expose the full attack surface.
- **Accuracy:** The information presented must be correct and free from errors. An inaccurate SBOM can provide a false sense of security and lead to incorrect risk assessments.
- **Timeliness:** An SBOM should be generated at the time of the build, ensuring it reflects the exact state of the software artifact being produced. An outdated SBOM is a significant liability.
- **Standardization:** The SBOM should be in a machine-readable format that can be easily parsed and integrated with security tools. Widely adopted standards include ***SPDX*** and ***CycloneDX.***

Proving Software Integrity With SBOMs

SBOMs are essential for proactive software supply chain security, enabling organizations to prove and maintain software integrity in several key ways. By consuming an SBOM, a security team can:

- **Perform Proactive Vulnerability Management:** In the event of a newly discovered vulnerability (e.g., the Log4j vulnerability), a security team can use the SBOM to immediately search for and pinpoint every application containing the vulnerable component, saving countless hours of manual effort.
- **Enhance Risk Analysis:** The dependency relationships and license information in an SBOM allow organizations to perform a comprehensive risk analysis of their third-party code, identifying risky or non-compliant components before they are deployed.
- **Establish a Runtime Baseline**: An SBOM can serve as a trusted baseline for a running application. Security tools can compare the components in the running application against the SBOM to detect unauthorized changes, malicious injections, or unexpected components that shouldn't be there.

Other Types of Evidence from Across The SDLC

While SLSA attestations and SBOMs provide critical insights into a software artifact's origin and components, they are not the only pieces of evidence required to prove integrity. A robust software supply chain security program collects a dossier of artifacts throughout the SDLC that, when combined, tell a complete story of an application's security posture. For a security leader, this collection of evidence is tangible proof of the organization's due diligence and proactive security culture as a software producer.

Source Code and Development Evidence

The integrity of a software artifact starts long before it's built. Evidence collected during the development phase validates that the code itself is secure and well-vetted.

- **Static Application Security Testing (SAST) Reports:** These reports prove that the source code has been automatically scanned for vulnerabilities, such as injection flaws or insecure cryptographic practices. A SAST report for a pull request, for example, can serve as a gating function. If the report shows no new critical vulnerabilities, it can be approved. This provides verifiable evidence that the code passed an automated security check at the earliest stage.
- **Code Review Logs:** Evidence of peer review from a version control system (like GitHub or GitLab) shows that the code was manually inspected. For a security practitioner, a code review log that includes a comment from a designated security reviewer, along with their approval, proves that the code was not only checked for functionality but also for security flaws.
- **Dependency Scanning Reports:** These are generated by tools that scan for known vulnerabilities in a project's open-source components. For example, a report from a built-in CI/CD scanner, showing a "clean" status for all dependencies, is a crucial piece of evidence that the application isn't inheriting a known vulnerability.

Build and Packaging Evidence

This phase focuses on the integrity of the build process itself, ensuring that the final artifact is precisely what was intended to be produced and has not been tampered with.

- **Immutable Build Records:** A record of the exact build environment, the commands executed, and the resulting artifact's hash proves that the build was hermetic and reproducible. This log file, stored in a secure location, provides a tamper-resistant record of how the final artifact came to be. For example, a practitioner can inspect a log from a CI/CD pipeline that shows the specific Docker image and shell commands used to build a container, and then verify that the final container image hash matches the hash in the log.

- **Cryptographic Signatures:** A digital signature applied to a final software artifact—such as a container image—provides irrefutable proof of its origin. This signature, verifiable with a public key, confirms that the artifact was produced by a trusted entity and has not been altered since it was signed. A security practitioner would use this signature to verify an image before deploying it to production, preventing the use of a maliciously tampered artifact.

Application Testing and Release Evidence

Before an application reaches production, it must be validated in a live environment. The evidence from this stage confirms that the deployed application behaves securely. Key assessments and reports include:

- **Runtime Security Assessment:** Running the application in a staging environment and monitoring it for security issues offers a means of generating substantial evidence, such as alerts, detailed event logs, exploit detection reports, and compliance-ready documentation.
- **Penetration Test Reports:** These are formal reports from external, third-party security firms. A penetration test report serves as powerful evidence of an application's security, as it represents a dedicated, manual effort to find vulnerabilities. The report, which includes details on findings and remediation, is often a mandatory compliance artifact for many enterprise customers.
- **Vulnerability Exception Reports:** Not every vulnerability can be fixed immediately. A vulnerability exception report is a formal document that provides an auditable record of an identified vulnerability, explaining why it was not fixed, what mitigating controls are in place, and a timeline for its eventual remediation. This proves that the security team is aware of the risk and is actively managing it, rather than simply ignoring it.

EVIDENCE MANAGEMENT THROUGHOUT THE SDLC

In the previous section, we detailed a broad array of different evidence types relevant to the different phases of the SDLC. For security and compliance leaders like you, managing this evidence means ensuring its integrity, accessibility, and utility for SDLC governance, audits, incident response, and continuous verification. This requires a shift from fragmented, manual processes to a centralized, automated system that provides an immutable, verifiable record of software integrity.

Storing and Securing Evidence

To prove software integrity, evidence must be stored in a way that is tamper-proof and easily auditable. A centralized evidence repository is a critical component, serving as a single source of truth for all security artifacts generated throughout the SDLC. This repository should be built on principles of immutability. This means that once evidence is documented, it cannot be altered or deleted. Technologies like content-addressable storage or immutable ledgers are ideal for this purpose.

The integrity of the evidence itself is secured through cryptographic hashing. Each piece of evidence - such as a Software Bill of Materials (SBOM), a vulnerability scan report, or a build log - is hashed, and a unique digital fingerprint is recorded. This hash is then included in a software integrity attestation, which is basically a digitally signed statement that binds the application artifact to its complete collection of evidence. Standards like **SLSA** (Supply Chain Levels for Software Artifacts) define these attestations, providing a framework for creating verifiable, tamper-proof metadata. By using these attestations, an organization can prove that a specific version of a software artifact was created by a trusted process and is accompanied by all the required security and quality evidence.

Retrieving and Using Evidence for Assurance

For security and compliance professionals, the primary goal of evidence management is to enable efficient and accurate retrieval for two key purposes: audits and incident response. The system must support automated, policy-based retrieval that ties evidence back to the software artifact and its components.

When a compliance audit is required, the team can use the software's unique cryptographic signature to query the evidence repository. The system then automatically retrieves the complete package of evidence—including the SBOM, vulnerability scan results, and build attestations—providing auditors with an irrefutable, transparent record of the software's secure build process. This eliminates the need for time-consuming, manual evidence gathering, significantly streamlining the audit process and demonstrating proactive security.

During a security incident, such as the discovery of a new critical vulnerability in an open-source library, a security professional can immediately query the repository to identify which applications use the vulnerable component. By retrieving the SBOMs associated with their software fleet, they can quickly pinpoint affected applications, assess risk, and prioritize remediation. This ability to instantly trace a component to every application that uses it is a fundamental pillar of modern vulnerability management and incident response. The evidence, therefore, serves as the foundation for both proving past integrity and enabling a swift and decisive response to future threats.

BUILDING TRUSTWORTHY SOFTWARE WITH JFROG

Piecing together all of the solutions and workflows needed to secure the software supply chain and ensure software integrity backed by extensive evidence is a complex, time consuming endeavor. JFrog offers a holistic approach of end-to-end software supply chain security that is fully-integrated with its comprehensive software artifact management platform. It allows application security teams, DevOps teams, and developers to secure software artifacts in the same place they are managed.

In addition, the JFrog Platform delivers a definitive means of proving software integrity and establishing trust through evidence-based governance of the SDLC.

JFrog AppTrust: Application Risk Governance

JFrog AppTrust is an application risk governance solution that enables organizations like yours to trust its software's security and drive compliant releases with evidence-based controls and contextualized insights. AppTrust gathers detailed evidence from across the software supply chain that allows you to enforce policies ensuring that security, compliance, quality, and performance criteria for your applications are met at each stage of the software development lifecycle. Only versions of your application that successfully passed all lifecycle "gates", verifying they comply with all your policies, receive a "Trusted Release" badge.

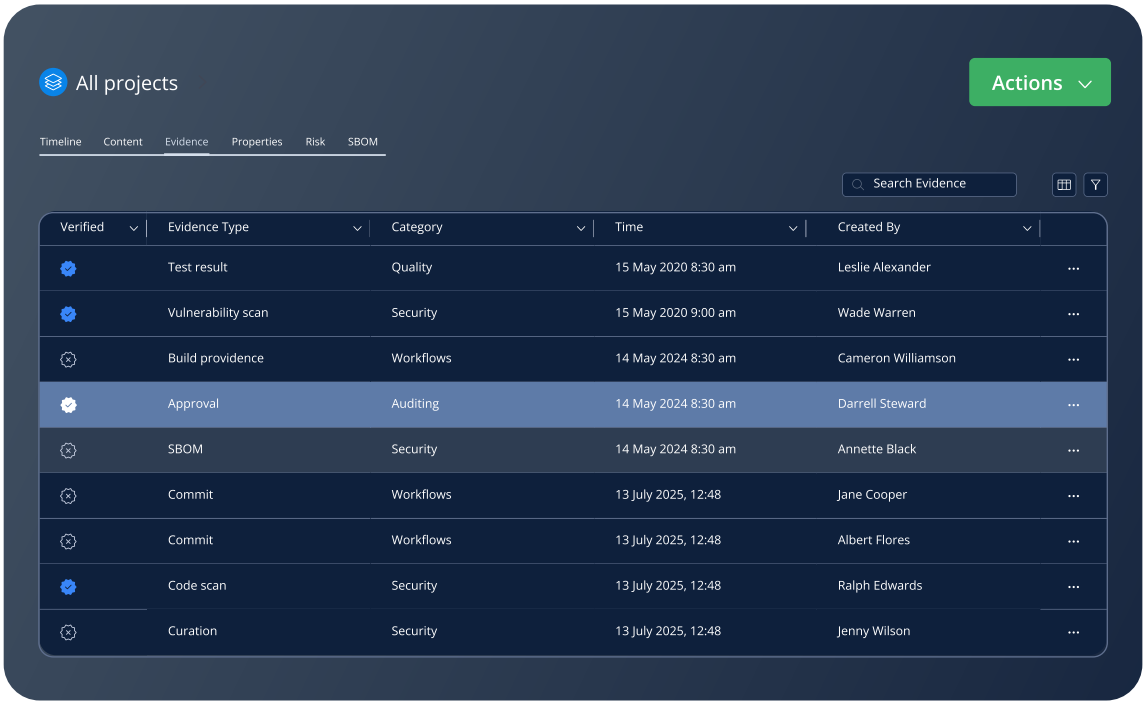


Fig. 2: AppTrust console showing applications' Trusted Release status

Evidence Through JFrog Ecosystem Integrations

Evidence collection throughout the many facets of the software supply chain is usually a manual, time-consuming process, often without any formal workflows across tools, stakeholders, processes.

JFrog offers a broad range of out-of-the-box technology integrations and a rapidly-growing evidence ecosystem that make it simple and seamless to capture key pieces of evidence spanning security, compliance, performance, and quality criteria that can be used to govern an organization's SDLC and ensure overall software integrity. Some of our key evidence partners are:

GitHub Actions build attestations will be converted into JFrog Evidence and remain stored alongside each software package indefinitely for compliance verification and policy enforcement.

ServiceNow will share its change requests, approvals, and vulnerability exceptions as signed evidence.

Sonar Source's SonarQube will create and share signed code quality, security scan findings, and code coverage attestations with JFrog Evidence.

For more information on JFrog AppTrust and JFrog Software Supply Chain Security, please visit https://jfrog.com/apptrust/, take an online tour or schedule a demo at your convenience.