Journal of Internet Services and Applications, 2026, 17:1, doi: 10.5753/jisa.2026.5922

Leveraging zero trust and risk indicators to support continuous vulnerability compliance

Diego Gama [**Universidade Federal de Campina Grande** | ***diegogama@lsd.ufcg.edu.br***]
Carlos Fuch [**Universidade Federal de Campina Grande** | ***carlos.fuch@lsd.ufcg.edu.br***]
Andrey Brito [**Universidade Federal de Campina Grande** | ***andrey@computacao.ufcg.edu.br***]
André Martin [**Technische Universität Dresden** | ***andre.martin@tu-dresden.de***]
Christof Fetzer [**Technische Universität Dresden** | ***christof.fetzer@tu-dresden.de***]

Laboratório de Sistemas Distribuídos, Universidade Federal de Campina Grande, Rua Aprigio Veloso, 882, Universitário, Campina Grande, PB, 58429-900, Brazil.

Received: 15 April 2025 • **Accepted:** 10 October 2025 • **Published:** 21 January 2026

Abstract

Open source dependencies are the leading source of vulnerabilities in applications and are often exploited in software supply chain attacks. Efforts to assess vulnerabilities are employed during DevSecOps pipelines in order to keep a system compliant with security regimes. However, current strategies for continuous compliance are limited to preventing issues before deployment, and thus do not address changes in dynamic aspects such as newfound vulnerabilities, let alone how to respond to such incidents. In this work, we leverage zero-trust to enable continuous, post-deployment vulnerability compliance assessment, isolating workloads that fail to meet a minimum security posture. This approach balances exploitation prevention with application availability — a fundamental trade-off for critical use cases. The solution is built on top of SPIRE, a robust open-source identity provider based on workload attestation, and implements a custom plugin that responds to compliance violations driven by dynamic aspects exposed by OWASP's Dependency Track, an open-source tool for monitoring software components and their dependencies for vulnerabilities. To enhance flexibility in the security-availability trade-off, we introduce a grace period mechanism, enabling organizations to defer enforcement of newly identified vulnerabilities based on workload criticality, thus supporting availability for non-critical workloads without compromising long-term security. Finally, we evaluate the performance impact of this approach on a SPIRE environment, showing that the added resource usage reliably remains within the recommended 16 GiB of RAM and 4 vCPUs to run Dependency Track in production. We also show that the plugin adds less than 6 seconds of latency to the attestation process, which is insignificant given its default frequency of twice per hour. Moreover, the results confirm that the approach successfully prevents vulnerability exploitation by prioritizing security, while enabling controlled flexibility in less critical contexts.

Keywords: Continuous Compliance, Vulnerability Management, Zero Trust Architecture, Incident Response, Identity Provisioning, Supply Chain, SPIRE

1 Introduction

The software supply chain is facing a critical security challenge, underscored by a staggering 742% increase in attacks between 2019 and 2021 [Sonatype, 2022]. High-profile incidents, such as the breaches involving SolarWinds' Orion Platform [IBM, 2024b] and the backdoor in XZ Utils [NVD, 2025], reveal the unprecedented scale of risk modern development practices entail.

Software supply chain attacks aim to inject malicious or vulnerable code into a final product through its dependencies. By embedding compromised components, attackers effectively bypass traditional security perimeters that focus on external threats, allowing them to infect an entire ecosystem from within.

This threat is magnified by the software industry's heavy reliance on open-source projects. A successful attack on a single, popular open-source library can cascade through countless downstream projects that depend on it. This danger is not theoretical; in 2022 alone, over 245 000 malicious open-source packages were discovered, more than doubling the total from all previous years combined [Sonatype, 2023]

While DevSecOps best practices, such as using Static Application Security Testing (SAST) in a CI/CD pipeline, are crucial for identifying vulnerabilities in locally written source code, they are not enough to secure the supply chain. The primary defense against dependency-based threats starts with two key components: Software Bill of Materials (SBOM) and Software Composition Analysis (SCA).

An SBOM is a detailed inventory of every software component within a system, including libraries, packages, and their dependencies [CycloneDX Core Working Group, 2024]. An SBOM creates essential transparency by mapping out the entire software supply chain. Then, using an SBOM as its foundation, SCA tools automatically scan for known vulnerabilities within all direct and transitive dependencies. The analysis output enables teams to take decisive action, such as blocking the deployment of a release containing critical vulnerabilities, thereby preventing a compromised component from ever reaching production.

Compliance regulations such as PCI DSS [PCI Security Standards Council, 2024] and FedRAMP [GSA, 2024] include vulnerability detection and mitigation requirements, highlighting the importance of SCA in the development lifecycle. As regional and international markets push compliance as a requirement for software, continuous compliance has become a focal point for DevSecOps [Ramaj *et al.*, 2022]. Approaches to ensure continuous compliance include building on top of DevSecOps best practices to integrate CaC (Compliance as Code), which can potentially automate checking if security controls are satisfied during a pipeline [Nygard, 2021; Kellogg *et al.*, 2021].

Unfortunately, even the best practices during development cannot mitigate all issues. The absence of known vulnerabilities in the product at release does not mean it is free from them. New vulnerabilities will likely be discovered after release, and a new CVE (Common Vulnerabilities Enumeration) entry can appear on a previously safe dependency. Moreover, these vulnerabilities can be critical, as was the case for the recent *XZ Utils* disclosure, which enabled a backdoor and affected various Linux distributions. Recent studies have also shown that generative AI such as ChatGPT-4 can automatically generate exploits for CVEs [Fang *et al.*, 2024], making new entries instantaneously exploitable. The high impact and short time to exploit prove the need for continuous compliance approaches that react to new vulnerabilities as quickly as possible.

This paper proposes an approach that leverages the recent Zero-Trust Architecture (ZTA) paradigm to enable continuous compliance. ZTA, as defined by NIST (National Institute of Standards and Technology) [Rose *et al.*, 2020], aims to minimize the attack surface in modern distributed systems. ZTA holds that by default, any person, event, or device is untrustworthy before sufficient authentication, even if it is within a local perimeter. In Zero-Trust environments, trust is dynamic rather than static [Buck *et al.*, 2021] and continuous authentication is required between communicating parties. This idea of "never trust, always verify" diametrically opposes the idea that the product is trustworthy if it left a trusted DevSecOps pipeline. Thus, as it requires continuous re-evaluation of the components' identities, implementing ZTA can support runtime security enforcement.

This work proposes leveraging ZTA to extend compliance beyond deployment. Our approach has demonstrated the following contributions:

1. We show how integrating vulnerability assessment as a ZTA's trust engine policy can isolate a non-conforming application. Isolation can prevent vulnerability exploitation and be considered an immediate response to compliance violations.
2. We validate this approach through a new custom workload attestor plugin for using SPIRE, a robust, open-source selective identity provider. The implemented plugin transparently triggers the dependency analysis during each identity renewal, confirming the feasibility of integrating continuous compliance in existing environments.
3. We evaluate the solution and confirm its practicality regarding added performance and resource costs. Furthermore, we discuss how the approach relies only on well-established security mechanisms.
4. We discuss compliance on both critical and non-critical applications and how adopting our approach impacts organizations, considering the roles of operations, security, and development teams.

We organize this paper as follows. Section 2 reviews the background needed to understand our approach and implementation, alongside related work and our gap analysis. Section 3 explains our threat model and outlines the problem's requirements. Section 4 provides an overview of a proposed architecture, and Section 5 details its implementation using well-known tools. In Section 6, we evaluate our solution regarding performance and security. Finally, Section 7 concludes our work with some final considerations and future directions.

2 Background and related work

This section reviews the concepts of continuous compliance and ZTA, as well as related work on these lines of research. We also explain relevant technology for the state of practice of these concepts while explaining the technology used for the solution.

2.1 Continuous compliance and supply chain security

Among the efforts to protect a supply chain and prevent attacks, one is adopting DevSecOps best practices. These practices can include using security frameworks such as SLSA (Supply-chain Levels for Software Artifacts). SLSA specifies incremental levels of artifact security to improve security guarantees, such as hardened builds and non-forgeable provenance [SLSA Specification, 2025]. The main point of such frameworks is to propose a secure way to produce software in a DevSecOps process, aiming to make a pipeline impervious to direct attacks.

However, attacks often also exploit vulnerabilities present in code when the product is delivered. That means that even if the CI/CD pipeline itself is correctly configured, the same may not be said about the source code of a project. Vulnerabilities can be found directly in code, but are even more frequently found in its dependencies, such as its libraries, frameworks, and other tools. Synopsy's BDSA (Black Duck Security Advisories) analysis report for 2024 states that most vulnerabilities found in audits were associated with JavaScript libraries [Synopsys, 2024]. Such vulnerabilities can come from direct and transitive dependencies (i.e., dependencies included in dependencies recursively).

Many security advisories such as NIST and GHSA (GitHub Security Advisory) disclose CVEs (Common Vulnerabilities and Exposure). CVEs are records stored in vulnerability databases that often provide APIs for consultation, the most effective of them being NVD (National Vulnerability Database) [Johnson *et al.*, 2018], which is managed by NIST and kept up to date with CVEs from various advisories. A CVE describes affected versions of software, and by frequently querying databases such as NVD, it is possible to verify if there is any reported vulnerability for a given software version before shipping it within the product.

SCA tools can automate this process and search these databases for vulnerabilities in a project's dependencies. Examples of tools are *Trivy*[1] or *Snyk*[2], which can ingest SBOMs for a transparent inventory of dependencies. Another example is OWASP's (Open Web Application Security Project) *Dependency Track*. It not only supports NVD but can also be integrated with many other public and private data sources. This makes it possible to aggregate more knowledge and allows companies with private vulnerability intelligence to combine their information with open-source vulnerability information.

Nonetheless, SBOM is as helpful as it is reliable. Given the popularity of software supply chain attacks for exploiting vulnerabilities, an attacker could try to tamper with an unprotected SBOM to change dependency versions. This could potentially misguide SCA into outputting a reduced list of CVEs to mask a known vulnerability. To mitigate this, tools employ artifact signing to attest provenance and help discriminate a fake or tampered artifact from a legitimate one.

Cosign, from the *Sigstore* framework, is a popular option for signing, verifying, and attaching artifacts as *in-toto* attestations [Sigstore, 2024]. An *in-toto* attestation is a fixed, lightweight format to describe supply chain metadata, including SBOM [Sirish and Hennen, 2024]. To both sign and verify signatures on attestations, *Cosign* uses *Rekor*, another component of the *Sigstore* framework that works as a transparency log, providing an auditable record of when a signature was created [Sigstore, 2024].

Vulnerability remediation can be a high-effort task, with hundreds of companies remediating only a monthly rate of 15.5% of known vulnerabilities on average [Cyentia Institute and Kenna Security, 2022]. To assist with prioritization, there are well-used scoring systems that help estimate how vulnerable the current state of a product is. The Common Vulnerability Scoring System (CVSS) is very prominently used to describe the severity of a CVE. CVSS is considered a dependable and robust method for rating the severity of a vulnerability [Johnson *et al.*, 2018].

For CVSS, the estimation of how vulnerable an application is relies on exploitability and impact properties and ranges from 0.0 to 10.0, usually discretized in classes. The classes and their respective closed intervals are LOW for $[0.1, 3.0]$, MEDIUM for $[4.0, 6.9]$, HIGH for $[7.0, 8.9]$, and CRITICAL for $[9.0, 10.0]$. Another way to describe the relevance of a vulnerability is to use the Exploit Prediction Scoring System (EPSS) [Jacobs *et al.*, 2021], which expresses the likelihood between 0.0 and 1.0 of a CVE being exploited within the next 30 days. This risk metric is updated daily for every public CVE reported [FIRST, 2024]. Both CVSS and EPSS are widely used, mainly to help prioritize vulnerability remediation.

To standardize a developer's stance towards known vulnerabilities, the Vulnerability Exploitability eXchange (VEX) format was specified [CISA, 2023a]. This format allows a formal statement about vulnerabilities and is readable by both machines and humans. VEX is useful for improving transparency and ignoring non-applicable vulnerabilities. For instance, a vulnerability that comes from a library could have high EPSS and CVSS scores, but the vulnerable code might be unreachable in the context of a specific application, making the threat harmless. As another example, if a vulnerability could affect the product, the company can declare that will fix it in the next patch cycle. This makes VEX a powerful format to enhance transparency and further improve trustworthiness between participants.

This level of effort to improve software quality and prevent security issues is needed to comply with certain regimes. Many vendors are required by organizations or governments to abide by one or more compliance regimes. These can be defined as a set of encoded best practices, such as guidelines for data encryption, storage management, and vulnerability management [Kellogg *et al.*, 2021]. Regimes like PCI DSS and FedRAMP consist of many requirements, and for each requirement, there is usually some sort of control, a rule defined by industry standards for fulfilling that requirement. As expensive auditing is needed in order to provide evidence of compliance, vendors strive to keep internal compliance, often manually, to avoid failing an audit [Kellogg *et al.*, 2021].

Upholding continuous compliance has become relevant for DevSecOps. As described by Ramaj *et al.* [2022], works related to continuous compliance seek to automate general security activities, like SCA, for compliance assessment. While the usefulness of compliance-specific tools like *OpenScap*, *UpGuard1*, and *CIS-CAT* is discussed in his work, non-specific tools can be useful for fulfilling general compliance requirements common to many regimes. There is also the notion of CaC, where some approaches include defining compliance through integrating common compliance controls into automated testing [Agarwal *et al.*, 2022; Kellogg *et al.*, 2021], and others aim to parameterize controls so that compliance can be checked in a data-driven testing architecture [Steffens *et al.*, 2018]. Finally, there is research in assessing vulnerabilities in cloud infrastructure and automatically producing security checks so the next pipeline iteration can further avoid these vulnerabilities [Torkura and Meinel, 2016].

Although promising, these solutions fall into what Nygard [2021] called "pipeline compliance": embedded in a CI/CD pipeline as a set of functions whose results are validated before release. He states that if the entire responsibility of internal audit sits within a pipeline, frequent changes may reduce correctness and cause ownership issues. He then proposes a form of "composite compliance" to distribute responsibility by assuming that if the parts of a system are compliant, then the system as a whole is also compliant. This, in turn, poses the problem of a previously compliant component violating compliance and then tainting the whole system. This is especially concerning when considering ecosystems comprised of many microservices and their replicas.

As a final solution, the author proposes "point of change compliance", an architecture where a security team defines compliance requirements, and pipelines built for these baselines run without security checks while producing security evidence. Before deployment, the final product is analyzed by an admission controller, which checks if the gathered evidence satisfies the requirements. If not, the pipeline fails. Otherwise, the product can be trusted because it is compliant, and is consequently deployed. Through this method, there is

[1]https://trivy.dev/
[2]https://snyk.io/

no confusion of ownership among developers and security officers.

While these forms of continuous compliance solve the issue of avoiding the launch of a non-compliant service, compliance is violated if the vulnerability status of a service changes. Since this situation cannot be completely prevented, and in such cases, the component was already admitted and is therefore trusted, it can potentially endanger the rest of the ecosystem.

This leaves a gap within the state of the art. If continuous compliance cannot address incidents due to sudden compliance violations, then the responsibility of incident response falls completely to the team providing the software (i.e., developers).

2.2 Zero-trust principles and tools

The problem of trusting a component indefinitely because of an initial state of compliance can be solved by leveraging ZTA. The Zero Trust Architecture is an approach that seeks to protect data in its various states, be it at rest or in transit [Syed *et al.*, 2022]. NIST defines ZTA as not a single network architecture achievable using one technology, but a set of many guiding principles that must be strategically implemented to secure enterprise assets. This gives flexibility for ZTA to be applied to many contexts, such as in the work of Chen *et al.* [2021], where the authors also leverage ZTA in order to add security awareness to healthcare devices in a 5G network.

One important principle of Zero Trust is that of *communication security*. It declares that communication needs to be secured regardless of its location. In other words, there should never be a communication that, due to taking place in a certain perimeter, is considered safe enough to be unauthenticated or unencrypted. This means that just because a component was able to be deployed, that does not imply that it is safe to communicate with it insecurely. A standard solution for this is the use of a software-defined perimeter (instead of a firewall-defined perimeter), frequently through mTLS. In orchestrated, containerized environments such as Kubernetes, mTLS can be powered by Service Meshes like Istio to remove overhead from applications and secure traffic between microservices [de Weever and Andreou, 2020]. Additionally, trust in ZTA is never static, but rather dynamic [Buck *et al.*, 2021]. This results in authenticating a component before establishing trust, and since this trust is not static, authentication must be continuous, which is a strong pillar of ZTA [He *et al.*, 2022].

Identity providers like *Kerberos* [IBM, 2024a] and *WS02 Identity Server* [Inc., 2024] can help with both establishing trust and securing communication by granting identities to applications. Such identities will be issued only to registered applications, thus establishing trust, and can take the form of X.509 certificates to power mTLS, thus securing communication between identified parties.

Another known specification for trust and identity issuing is SPIFFE [Babakian *et al.*, 2022], which proposes that identities assigned within a certain Trust Domain (a logical perimeter for an ecosystem) should be identifiable and verifiable. Identities, here called SPIFFE IDs, are merely semantic URIs but always come embedded in an SVID (SPIFFE Verifiable Identity Documents). SPIFFE specifies that the SVID must always contain signatures from a trusted party to prove that its SPIFFE ID is valid. Currently, the standard defines JWT and X.509 as valid SVID formats[3], making the SPIFFE specification equivalent to other identity providers in terms of establishing trust and securing communication.

However, the aforementioned solutions for authentication do not establish dynamic trust, but rather static trust. That is, one must statically define which applications or parties to trust so that the provider can issue or deny an identity. To help satisfy this requirement, among other goals, SPIRE was created to implement the SPIFFE standard with a hierarchical and dynamic authentication process.

SPIRE is a tool that graduated from the Cloud Native Computing Foundation (CNCF) and was adopted by many companies such as Netflix, Pinterest, and Uber. SPIFFE and SPIRE have also been receiving contributions from big tech companies such as VMWare, Google, and Hewlett-Packard Enterprise [SPIFFE Project, 2025]. SPIRE differs from other tools due to requiring a secret-less authentication process, known as *attestation*, before issuing an identity. Whenever an identity is defined in SPIRE, it must contain a set of selectors. These values are criteria that need to be satisfied by the application, referred to as *workload*, that is requesting the identity. During attestation, SPIRE collects selectors from the workload and compares them with each admin-defined entry in its database. Only if a workload with those selectors is eligible for some identity, an SVID with that identity with the requested type (most often a short-lived X.509 certificate) is issued to the application. If the application fails the attestation, it does not receive a new identity, which means that even if it previously had an identity, said identity will eventually expire and cease to power any mTLS-required connections within that Trust Domain.

This required attestation process to obtain and renew identities effectively establishes trust as dynamic, because instead of looking for specific workloads, it searches for their eligibility, regardless of who they are. Trust can be defined to be strict or lenient depending on the identities' configurations. A certain SPIFFE ID can be set to only match specific selectors, guaranteeing that only the desired application will bear that identity. However, the opposite can be achieved by declaring multiple entries with the same SPIFFE ID, increasing the attestation possibilities.

To support this attestation-based selective provisioning, SPIRE employs two main components in a defined hierarchy: the SPIRE Server and the SPIRE Agent, as depicted in Figure 1. The server acts as the single source of truth for the entire Trust Domain, which should encompass a logical perimeter for an ecosystem of applications. Because it is the place where identities are registered and managed, it must be protected, lest a penetration attack compromise the ecosystem. Meanwhile, the SPIRE Agent is a proxy for the SPIRE Server so that local applications can be attested without direct access to the source of trust. For example, considering a distributed cloud architecture of many nodes or instances, a SPIRE Agent represents a node, while the SPIRE Server resides on a specially protected node, and workloads are applications that run

[3]https://github.com/spiffe/spiffe/tree/main/standards

in each separate node, able only to communicate with their local node.

Because the source of trust is the Server, no Agent, and therefore no node, is trusted by default. Before it can be trusted to be a proxy, it is subjected to a node attestation, much like the workloads. Selectors for node attestation can consider different properties of a node, such as its location on the infrastructure, its runtime environment, and properties from its software or hardware stack. If successful, the Server issues an identity to the Agent, enabling it to attest its local workloads (i.e., services running on the node). Afterwards, workloads can try to attest to their local Agent to receive their identity.

An important characteristic of this hierarchy is that, by default, workloads are not meant to attest to all Agents. This is enforced by SPIRE via the structure of the identity. When an admin enters an identity into the SPIRE Server, they not only specify the selectors and SPIFFE ID, but also which parent SPIFFE ID (i.e., Agent identity) is responsible for issuing it. This means that a workload that successfully contacts an external Agent will only be able to attest to it if that Agent has that specific parent SPIFFE ID. Additionally, communication between Agents and workloads happens via local sockets, both because it is insecure in nature since the workload is not initially trusted, but also to enforce that only local Agents are approachable. There are benefits to discriminating the node based on its characteristics (e.g., restricting certain critical applications on high-protected nodes). But if an admin wishes to treat all nodes equally, entries can be configured so that all Agents receive the same SPIFFE ID.

Finally, once bearing their SVIDs, Agents can communicate with the Server and attest workloads. Once workloads acquire their SVID, they can use it to employ mTLS or other authentication mechanisms. As said before, because these are usually short-lived, they must be renewed via reattestations. Thus, if a workload loses mTLS clearance because its SVID is no longer valid, that can only mean it failed to reattest because it no longer matches the required selectors and is, therefore, rightfully isolated in the network.

As a result of its selective identity provisioning, SPIRE also supports another tenet of ZTA: *Continuous Authentication*. Because identities are short-lived and depend on periodic attestation, if two services communicate through SVID-powered mTLS connections, they will implicitly authenticate each other continuously. Finally, although SPIRE does not support every attestation use case with its built-in components, it is extensible with its plugin architecture, allowing for customization by implementing new plugins, including workload attestation plugins. Thus, it is a viable alternative for assessing compliance as a criterion for determining trust.

Although compliance is seen as important, efforts to establish good practices of ZTA do not detail the role of compliance. As part of their multivocal review of both academic and gray literature, Buck *et al.* [2021] describe the trust assessment process of a ZTA implementation as needing two components: a Trust Engine and a zero-trust PEP (Policy Enforcement Point). While the Trust Engine assesses trust based on policy, the PEP enforces the decision by providing secure communication. Compliance can be used as policy in theory, but no work is shown by the authors to explore this possibility despite its relevancy. The same can be said by the survey from He *et al.* [2022], who point out that further attention on continuous diagnostics and mitigation systems is needed to integrate industrial compliance into ZTA. Finally, industry standards for achieving maturity in Zero Trust also lack guidance on integrating security posture while at the same time recommending vulnerability management [CISA, 2023b]. This research gap appears, therefore, to be a missing link for connecting ZTA to continuous compliance.

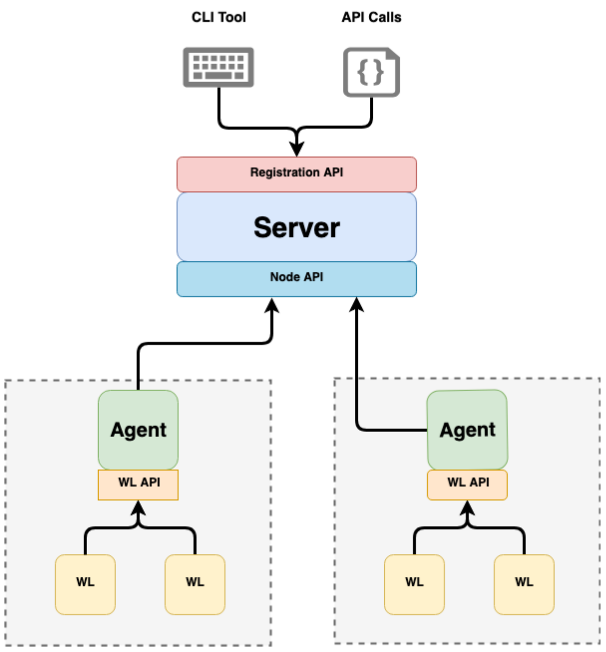

Figure 1. SPIRE Hierarchy (Source: [SPIFFE, 2025b])

3 Threat model and solution requirements

Understanding our threat model requires thinking about some roles people may assume and the tools used in the environment.

Starting with the human roles, we assume a typical environment where developers and operators develop and maintain cloud-native applications. In addition, security officers (or the operators themselves, in frequent cases) define security policies for the organization. Security officers know the organization's obligations and understand the functional components of security management (e.g., PCI DSS [PCI Security Standards Council, 2024], FedRAMP [GSA, 2024], ITU-T Security Requirements [ITU, 2020]). Thus, security officers understand the incident response strategies and the development team's deadlines to fix vulnerabilities.

Following the DevOps (or DevSecOps) movement, developers and operators try to cooperate but are not specialists in each other's work. Therefore, being responsible for the system's long-term operation, operators must understand system security and want bug-free applications, but do not get deeply involved in the development process. Nevertheless, we assume operators are benign and do not represent an internal threat.

We also assume developers are mostly benign but may not have the necessary knowledge to build long-term secure services. Therefore, they may use libraries and modules that are not mature enough. However, we assume that the most experienced developers build the tests used in the CI/CD pipeline. Thus, if some developers are not trusted, such as in an open-source community, there must be a process that forces reviews by selected community members, and evidence of this process is collected. Consequently, we assume that the application has no known vulnerabilities at the initial state of a release. Thus, all workloads composing an application are compliant at that moment.

Regarding tools, our environment adopts three types of tools: (i) CI/CD tools, which generate artifacts such as SBOMs and images; (ii) zero-trust tools, which implement the identity provisioning workflow and the proxies that control mTLS connections; (iii) vulnerability management tools (Dependency Track in our implementation), which track vulnerability databases and serve local queries about vulnerabilities.

We assume that the CI/CD pipeline implements good practices. Nevertheless, it is possible that a third-party attacker can intercept a software release and substitute it with a tampered image if the OCI container registry storing it is not secure enough. Similarly, an attacker can substitute the SBOM. Therefore, attackers can replace SBOMs and images but cannot falsify the signatures of trust entities. SPIRE can already check image signatures and, consequently, this is outside our scope.

Next, the application runs within a ZTA. ZTA brings dynamic trust assessment to the environment. Therefore, all workloads have unique identities and authenticate each other in all communication. Although not strictly necessary for our approach, our implementations assume mutual TLS communications. As a result, workloads have identities in the form of X.509 certificates.

As a consequence of the ZTA, workloads that do not have valid identities will be unable to communicate with other workloads and will be effectively isolated. Isolating a workload should trigger termination (e.g., due to failing health checks) and trigger alerts in monitoring tools. Tools such as Envoy [Envoy Project, 2025] manage mutual TLS connections, and SPIRE generates workload identity certificates. We assume that ZTAs and the SPIRE installations and operations are correct. For example, attackers cannot access servers or the private key from the local certificate authority to register malicious identities or generate certificates. Similarly, preventing attackers from changing critical configurations in the environment is out of scope. Our focus is then on automating first responses to vulnerability compliance violations in deployed applications, preventing exploits, and overcoming the limitations of pipelines that only verify security before deployment.

Considering these assumptions, the solution must provide a way to monitor compliance continuously. If, at some point, compliance is violated due to new vulnerabilities, the solution must bridge the gap left by continuous compliance strategies and apply an automatic incident response. This automated response can lessen the overhead and responsibilities of teams to detect and enact an emergency intervention. To achieve this while integrating with a ZTA environment, selective identity provisioning must be implemented so that non-compliant services are denied their identities. Accurately blocking the generation and renewal of identities requires mapping compliance into policy, also bridging the gap left by current ZTA research. In the end, the approach should help operators work smoothly with developers and compliance officers to decide trade-offs between their applications' availability and security.

4 Enforcing continuous compliance

Compliance must be present during the entire life cycle of the application. As mentioned before, this can be made possible by adopting DevSecOps best practices, SCA, and a supply chain security framework during a CI/CD pipeline. By implementing these known practices, a company can enforce an initial state of compliance by ensuring the immutable part of a release (its build and attached artifacts) follows compliance rules. SLSA-compliant build processes further help with this and, at its level 3, guarantee artifact provenance and hardened builds [SLSA Specification, 2025].

However, to ensure continuous compliance, it is necessary to continue to perform SCA, even after deployment or delivery. To operationalize this, we propose to integrate workload attestation using vulnerability tolerance as a minimum security posture criterion for selective identity provisioning. Figure 2 illustrates our architecture. By providing short-lived identities to power mTLS between applications, we can leverage Zero Trust by imposing communication security that requires frequent reattestation to renew.

The resulting isolation of a workload could be mitigated by (1) a VEX issued by security officers so that SCA ignores some vulnerabilities, (2) a new release without said vulnerabilities, or (3) operators rolling back to previous non-vulnerable versions. Since isolation is a response to a compliance violation, the maximum CVSS and EPSS requirements for issuing an identity should be defined by security officers, considering the trade-off between availability and exploitation prevention. On top of CVE scoring requirements, a grace period could be put in place to filter out young vulnerabilities, measured by the day of their first known disclosure until the current day. Configuring thresholds for grace periods could provide more flexibility in said trade-off.

5 Implementation

In this section, we discuss the implementation of the proposal explained in the previous section. To perform continuous SCA, we selected *Dependency Track*, a tool from the Open Worldwide Application Security Project (OWASP), due to its openness, robustness, and integration possibilities. Dependency Track already yields CVSS and EPSS for every vulnerability in an SBOM's dependency list. As for its integration, it supports the NVD database by default and allows additional private data sources that extend the relevance of this approach. Finally, it provides a REST API to help automate its use.

For the identity provider, we chose SPIRE due to its openness, flexibility, and native workload-attesting capabilities.

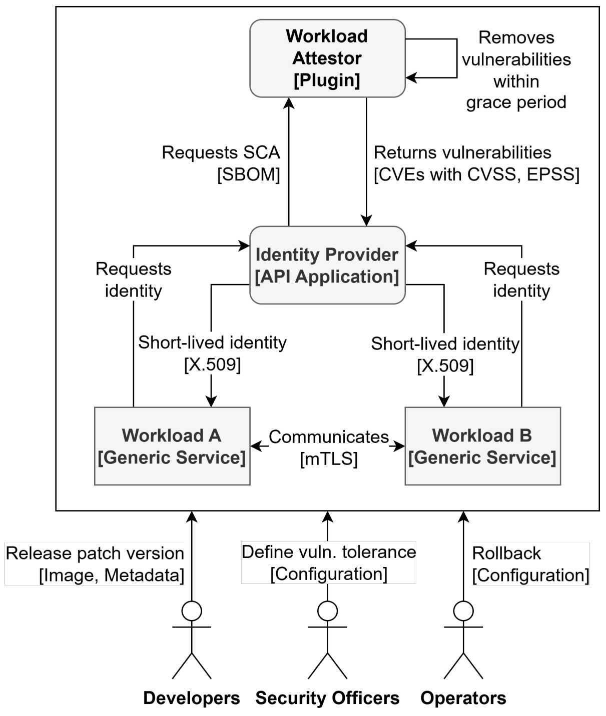
Workload Attestor [Plugin]
Removes vulnerabilities within grace period
Requests SCA [SBOM]
Returns vulnerabilities [CVEs with CVSS, EPSS]
Identity Provider [API Application]
Requests identity
Requests identity
Short-lived identity [X.509]
Short-lived identity [X.509]
Workload A [Generic Service]
Communicates [mTLS]
Workload B [Generic Service]
Release patch version [Image, Metadata]
Define vuln. tolerance [Configuration]
Rollback [Configuration]
Developers
Security Officers
Operators

Figure 2. Proposed architecture

In addition, it is already being used to implement ZTA's communication security and continuous authentication principles. We expand SPIRE in regard to its available selectors. This is done by implementing a workload attestor plugin that will be used during a workload attestation. Our custom plugin connects to a *Dependency Track* instance to return the SCA results and uses this information to provide the workload properties to the SPIRE agent as selectors.

Two preconditions are necessary for the plugin features: (1) it needs to have access to the image information (i.e., its complete identification) in order to tell which workload it is attesting, and (2) it must also have access to that image's SBOM. With this information, it can call *Dependency Track* to feed it the SBOM and then gather the results.

These conditions can be provided in any containerized environment. However, in this work, we use Kubernetes as an orchestration tool due to its popularity. The following subsections will respectively propose how to embed the evidence so it can be accessible, and then detail how the plugin can collect the evidence to use SCA. Figure 3 illustrates our proposal, and how operators and developers can collaborate on the attestation process.

Similar approaches could be implemented in other contexts. For example, if we assume that microservices run in micro-VMs orchestrated by a system such as OpenStack[4], the image information could be retrieved from the image service and the SBOM embedded in the image metadata.

5.1 Embedding the basic information

To make sure the product possesses an SBOM describing it, its build pipeline should contain a stage that generates and saves this artifact prior to release. Ideally, this information is open so that clients and other interested stakeholders can access it for transparency reasons. This can be done by making the SBOMs available in a repository, a public artifact registry of some kind, or within the same OCI registry the product images are kept. The latter is especially practical because if a client or other interested stakeholder has access to the image, they also have access to the SBOM. As mentioned before, *Cosign* can be helpful to both sign and attach the SBOM to the image, storing it to the registry and making it available for the future.

SBOMs come in various formats. *CycloneDX* is a format also created by OWASP, with high interoperability due to high adoption, and is required by *Dependency Track*. Many tools can produce *CycloneDX* formatted SBOMs, such as the aforementioned *Trivy* and *Snyk*.

In addition to the SBOM, another compliance evidence that can be used is a provenance artifact. The provenance can prove that the image's origins are trustworthy. That means it came from a trusted, quality pipeline, managed by a specialized or otherwise trusted party. An example of this artifact is a SLSA Provenance, that can be generated by adhering to tools with at least SLSA level 2 or, preferably, level 3 guarantees. SLSA level 2 means that the tool provides a signed provenance evidence that the image was built on that pipeline, while on top of that SLSA level 3 means that forging the provenance is beyond the capabilities of most adversaries and the build platform is hardened against tampering [SLSA Specification, 2025].

If available, the workload attestor plugin will use SLSA Provenance to report the image's origins in addition to vulnerability data. This allows administrators to restrict the origin of their images (e.g., the CI pipeline that produced it) and the source code repository and branch used.

Lastly, to make sure the SBOM and provenance are trustworthy, the same authority should sign both. This way, we can discriminate if the *in-toto* attestations come from the same pipeline as the built image and tell apart a legitimate artifact from one forged by an attacker.

5.2 Framework for evidence collection

The plugin implementation follows the desired workflow specified by SPIRE for a workload attestor. It is triggered by the SPIRE Agent when a workload tries to fetch an identity. When it does so, the Agent begins the workload attestation process, which triggers all installed workload attestor plugins, including our custom one. Figure 4 illustrates the workflow for the plugin.

When it starts, the plugin immediately collects information about the running image. For the scope of this work, it queries the Kubernetes API regarding the Pod (i.e., the Kubernetes set of running containers). This information will include the image source and its hash digest. After it discovers which container started the attestation, the plugin tries to fetch all attached evidence using *Cosign* and looks among them for SBOM and SLSA Provenance. It then checks their signature using *Rekor* to build selectors regarding who issued the artifacts (the pipeline that generated them) and who signed them. Then, it proceeds to process both artifacts.

Firstly, it uses the SBOM to feed *Dependency Track's* SCA via its REST API. It registers the workload in *Dependency Track* if there is no entry for this image version using digest,

[4]https://www.openstack.org/

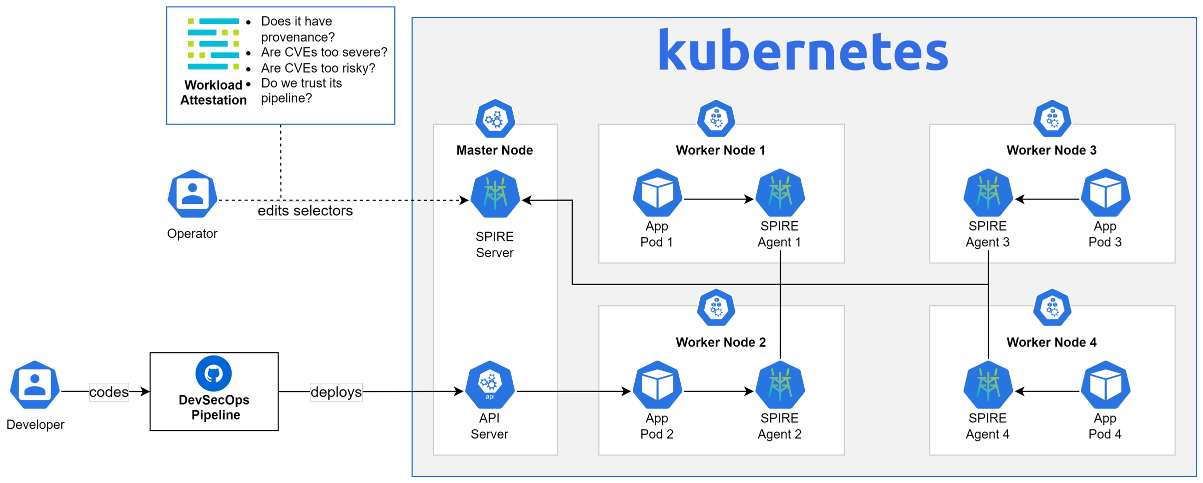

Figure 3. Proposal of compliance attestation within SPIRE

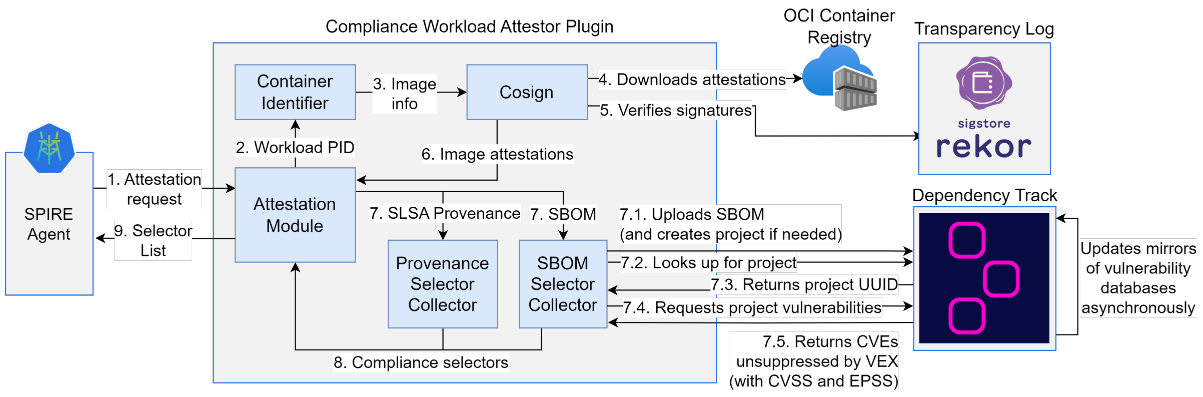
Compliance Workload Attestor Plugin
OCI Container Registry
Transparency Log
sigstore rekor
Container Identifier
3. Image info
Cosign
4. Downloads attestations
5. Verifies signatures
2. Workload PID
6. Image attestations
SPIRE Agent
1. Attestation request
9. Selector List
Attestation Module
7. SLSA Provenance
7. SBOM
Provenance Selector Collector
SBOM Selector Collector
8. Compliance selectors
Dependency Track
7.1. Uploads SBOM (and creates project if needed)
7.2. Looks up for project
7.3. Returns project UUID
7.4. Requests project vulnerabilities
7.5. Returns CVEs unsuppressed by VEX (with CVSS and EPSS)
Updates mirrors of vulnerability databases asynchronously

Figure 4. Compliance workload attestor plugin

and then triggers component analysis. Following that, the plugin will request *Dependency Track* all of that image's known CVEs, alongside their CVSS scores and EPSS likelihoods. Then, the CVE list will be processed to return the highest CVSS severity and EPSS risk scores and build them as selectors.

Secondly, the SLSA Provenance will be inspected to find the repository's location and the build pipeline used. It will include the repository and the reference version (i.e., branch or tag) used to build the image in the selectors.

After all selectors are built, they are returned so that the SPIRE Agent can compare the results found with the criteria defined for the identities in its database. If one or more artifacts are not found during attestation, no selectors about them will be built, and thus, no identity that requires such selectors will be issued. Table 1 lists all available selectors for the compliance workload attestor plugin.

To make sure the communication with *Dependency Track* is protected, it also uses mTLS powered by SPIRE so that only attested SPIRE Agents can communicate with *Dependency Track*, preventing unauthorized or illegitimate Agents to deposit SBOMs or consume analysis results. Furthermore, since *Dependency Track* does not have native support for SPIRE, we use an official utility sidecar, named SPIFFE Helper [SPIFFE, 2025a], to fetch SVIDs and configure non-SPIRE-aware workloads to use them.

One important note about the selectors is the format of both highest-cvss-severity and highest-epss-risk. It would certainly be more intuitive if thresholds could be represented as a number. For instance, CVSS could be represented as the actual value, providing more control for operators. EPSS would benefit the most from this, as it does not contain official classes like CVSS.

The reasoning behind using a category instead of a number is that SPIRE does not natively support numeric selectors; they are all used as strings. To be more specific, to compare selectors during attestation, SPIRE checks if the set of expected selectors is a subset of returned selectors, and element comparison is done by string equality. This way, if numbers were used, they would have no inherent numerical value or order. Using them more semantically would require contributions to the selector comparison logic within SPIRE, and this is not currently aligned with the community vision of selectors, which sees the selectors as properties that a node or workload has or does not have.

The categories are a workaround for this limitation. If a high severity is the highest CVSS found, the attestor returns only this information, and if a medium risk is found as the highest EPSS, only this is returned on the selector. This requires additional entries, but is also useful to create different identities for different levels of vulnerabilities, enabling operators to create slightly different SPIFFE IDs for different risk levels. To make this viable to EPSS, we needed to map the values on a similar scale and then provide classes in the same way. Without an official definition for EPSS risk classes, we allow every organization to configure the intervals for each

Table 1. Selectors for the workload attestor plugin

Selector	**Semantics**	**Example**
attestation-certificate-identity	The identity that generated the attestations (i.e., workflow that produced the evidence)	https://github.com/company/trusted-workflows/.github/workflows/devsecops-pipeline.yml@refs/heads/main
attestation-certificate-oidc-issuer	The OIDC issuer that signed the attestations (i.e., GitHub OIDC Issuer, which signed on the pipeline's behalf)	https://token.actions.githubusercontent.com
has-provenance	The image possesses a SLSA Provenance	True or False
source-code-uri	The public URI for the repository that produced the image	https://github.com/repository.git
source-code-version	The version (i.e., branch or tag) of the source code	main
has-sbom	The image has an SBOM	True or False
highest-cvss-severity	The list of CVSS severities tolerated for the workload	LOW or MEDIUM or HIGH or CRITICAL
highest-epss-risk	The list of EPSS risks tolerated for the workload	LOW or MEDIUM or HIGH or CRITICAL

class in plugin settings.

5.3 Managing different security requirements

The main intended consequence of using the plugin is the unavailability in the presence of an unacceptable vulnerability. In other words, in order to uphold compliance beyond deployment, it may be preferable to stop the workload rather than to allow it to be susceptible to exploitation.

This, of course, is not the case for many noncritical workloads. For example, security officers may decide that a simple web application that does not handle sensitive information should remain available while developers patch vulnerable dependencies. In this scenario, a CVE with a critical CVSS might be an unacceptable threat, while a high CVSS vulnerability could await mitigation without damaging SLA. Such differentiation is aligned with strategies such as CISA's Stakeholder Specific Vulnerability Categorization (SSVC), which aims to improve prioritization [CISA, 2025].

To enable this behavior, the plugin can be configured to apply a grace period for newly identified CVEs. With this configuration in place, when a new CVE is identified in the project or in its dependencies, instead of immediately reacting by including the CVE in the analysis, the plugin can disregard its existence (when evaluating the scores for the selectors) until the assigned grace period expires. The practical consequence is that, while the grace period granted to a certain CVE remains valid, the attestation plugin will tolerate it. In this context, workloads are allowed to continue running alongside newer vulnerabilities for a limited time, according to the organization's policies.

To achieve this effect, an admin could set the length of this period in the plugin configuration for every CVSS and EPSS class. This way, the plugin can better consider the context of the workload when reporting its findings to the SPIRE Agent. The state machine in Figure 5 illustrates the evaluation of each CVE found after SCA, including the initial grace period filter.

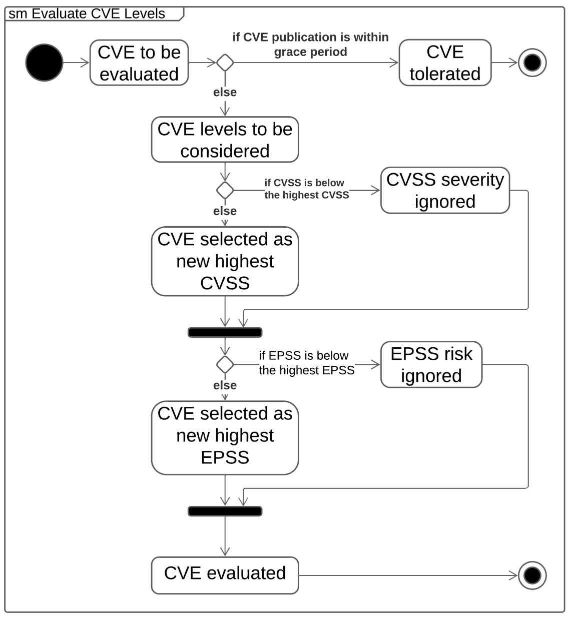
sm Evaluate CVE Levels
CVE to be evaluated
if CVE publication is within grace period
CVE tolerated
else
CVE levels to be considered
if CVSS is below the highest CVSS
CVSS severity ignored
else
CVE selected as new highest CVSS
if EPSS is below the highest EPSS
EPSS risk ignored
else
CVE selected as new highest EPSS
CVE evaluated

Figure 5. State machine of CVE evaluation

Unfortunately, this approach alone is not applicable to an environment containing both critical and noncritical workloads. For instance, consider the same example as before with the web application, except now it has a separate component in charge of authentication and authorization. This component is naturally more critical than the web application, since exposing it to a vulnerability could allow unauthorized access, potentially resulting in a sensitive data breach. Considering the configuration explained before, the plugin would apply the same grace period rules to both workloads, which would be either too lenient or too restrictive.

To avoid this problem, the configuration is further developed into a table that maps what grace period behavior to consider for each level of criticality. Following the semantics kept so far, they also use the same classes defined in CVSS and EPSS, namely, LOW, MEDIUM, HIGH, and CRITICAL; each level represents a degree of stringency regarding the tolerance allowed by the grace period. Below, we provide a sample definition based on the examples discussed in this paper. It is important to note, however, that this is not a prescriptive guideline. In practice, the interpretation of each level and the corresponding behavior should be defined by the organization through a collaborative effort involving security officers, infrastructure operators, and development teams.

- **LOW**: The workload is associated with a low security concern, allowing for extended grace periods before remediation is required. This includes auxiliary or low-impact services, such as static content servers, telemetry exporters, monitoring agents with read-only access, and non-sensitive background tasks. These workloads typically have limited privileges and minimal impact in case of compromise.
- **MEDIUM**: The workload presents a moderate security concern. Grace periods remain relatively long but are shorter than those defined for the LOW level. Typical examples are internal microservices processing non-sensitive business logic, authenticated APIs with limited scope, or batch jobs that handle controlled data. These workloads may have broader access or serve user requests, but are not critical to core system security.
- **HIGH**: Security concerns for this workload are high. Grace periods are shorter, but still allow for a measured response. This applies to services that authenticate users, manage permissions, or handle personally identifiable information or regulated data, such as financial records. These workloads require tighter response windows due to their potential impact.
- **CRITICAL**: The workload is deemed critical from a security standpoint. Grace periods are minimized to reduce the exploitation window. Examples include certificate authorities, secrets managers, encryption key storage, ingress controllers exposed to public traffic, or components with privileged cluster access. Exploitation of vulnerabilities in these workloads would likely result in severe or cascading consequences.

Tables 2, 3, 4, and 5 present example configurations that define grace periods based on the workload's criticality, and then the match between the CVSS severity of a CVE and its EPSS risk level. For instance, suppose a CVE with a high CVSS is found in a workload classified with low criticality, as found in Table 2. If the same CVE has a low EPSS, its grace period would be of 60 days. For a different CVE of this same workload, if it had a medium EPSS and a critical CVSS, the grace period would be of 20 days. Until this period elapses, both CVEs will be ignored by the plugin. On the very next day after the expiry date, they will be included in the analysis.

As another example, consider a CVE with a critical CVSS and a high EPSS in a critical workload, as described in Table 5. The plugin will only suppress it for 18 hours (i.e., 0.75 days) before returning it in selectors. Finally, for workloads with no declared criticality, the plugin's default behavior is applied, disabling consideration of the grace period entirely and reacting to CVEs immediately.

Table 2. Criticality: LOW

	EPSS Score			
CVE Score	LOW	MEDIUM	HIGH	CRITICAL
LOW	240	220	210	180
MEDIUM	135	120	105	90
HIGH	60	50	40	30
CRITICAL	25	20	15	10

Table 3. Criticality: MEDIUM

	EPSS Score			
CVE Score	LOW	MEDIUM	HIGH	CRITICAL
LOW	135	120	105	90
MEDIUM	60	50	40	30
HIGH	25	20	15	10
CRITICAL	8	7	6	5

Table 4. Criticality: HIGH

	EPSS Score			
CVE Score	LOW	MEDIUM	HIGH	CRITICAL
LOW	135	120	105	90
MEDIUM	25	20	15	10
HIGH	8	7	6	5
CRITICAL	3.5	3	2.5	2

Table 5. Criticality: CRITICAL

	EPSS Score			
CVE Score	LOW	MEDIUM	HIGH	CRITICAL
LOW	25	20	15	10
MEDIUM	8	7	6	5
HIGH	3.5	3	2.5	2
CRITICAL	1.25	1	0.75	0.5

Since the plugin cannot discern the criticality of a given workload during the attestation process, we need to input this information in some way. For this, we chose to leverage Kubernetes as our choice of environment. Kubernetes manifests can contain labels and annotations used by operators, both for semantics and sometimes for the configuration of Kubernetes tools. The plugin calls the Kubelet to collect annotations of the workload being attested and searches specifically for the `spire.io/criticality` label. The string value should be either LOW, MEDIUM, HIGH or CRITICAL. This allows the plugin to determine which grace period behavior, if any, to apply.

Another way to implement this is to embed this information within the SBOM from the start. One could argue that since the SBOM is trusted due to its verifiable provenance, this information should be discovered during the SCA step. Moreover, it could be set in CycloneDX's properties section, made specifically for the format's extensibility [CycloneDX Core Working Group, 2024].

However, this is only appropriate for applications that should retain a single criticality in all environments and use cases. This is not always the case. For instance, the same database management system can be used in one context to keep low-sensitivity data accessible by multiple components, while in another context it might store only high-sensitivity information that should have strict access rules. The former use case could receive a low criticality, while the latter is more akin to a high criticality value; despite being the same application, its criticality could be set as low. This could happen for a variety of reasons, from the priority on security to the available workforce dedicated to patching vulnerabilities. Since each version of an application should produce a single SBOM, such an application would only have a single criticality. This would force different organizations to define their criticalities based on the application settings, instead of their own semantics as discussed before.

Therefore, we chose to decouple this information from the application itself and set it into its configuration in the runtime environment (i.e., Kubernetes manifest, in our case).

5.4 Plugin usage and configuration

Following the configuration pattern defined in SPIRE, the plugin is configured in HCL (HashiCorp Configuration Language) according to Listing 2. These settings declare how to communicate with *Dependency Track* and *Cosign*, as well as acceptable identity issuers and owners for the *in-toto* attestations.

For instance, the snippet shows that the only trusted identity issuer is GitHub Actions and that the only identity to be trusted is that of a specific CI/CD pipeline. This means that the plugin will consider invalid any attestations with different credentials, and thus will not perform their respective analysis, which will result in no relevant selectors. In order for the identities to be taken into consideration, they must be trusted in the first place. Therefore, all trusted and expected credentials should be included, as the plugin will verify every possible combination between identity and issuer for each attestation.

Additionally, the grace-periods section defines the behavior explained in Section 5.3. The outermost levels represent the CVSS severity classes, and the innermost levels represent the workload criticality classes. Each element in the classified lists, from left to right, represents the EPSS levels from LOW to CRITICAL. If there are fewer than 4 elements in any list, no grace period will be applied for that combination of CVSS, Criticality, and EPSS classes.

This configuration should be included in the deployment of the plugin within the SPIRE Agent configuration[5].

Workloads deployed in the system must contain the spire.io/criticality annotation in their manifests, as exemplified in Listing 1. Otherwise, as explained before, the plugin will not enable any grace periods for this workload and will instead consider all vulnerabilities it finds.

Listing 3 illustrates how to register entries for identities to the SPIRE Server using the defined selectors. The URI for an identity is defined in the snippet as -spiffeID, where example.org is the Trust Domain and example-service/main/low-severity/low-risk is the full name of the first identity, and example-service/main/no-severity/no-risk is the full name of the second identity, both for the same workload.

As for the selectors, they are defined by using the -selector argument, and all of them are prefixed by cc, which signals to SPIRE that this selector comes from the continuous compliance plugin.

Via these selectors, this first entry imposes that the example-service should only tolerate CVEs with low values for both severity (CVSS) and risk (EPSS). The entry also restricts the provenance of the product. By defining the source-code-uri selector, it will only match workloads that come from that specific Git repository, and by defining attestation-certificate-identity it restricts the pipeline that built the workload. In this example, the pipeline is not in the same location as the repository, which is not the default but can be the case if the pipeline runs on another platform, or if employing reusable pipelines.

The second entry is stricter, as it defines that no vulnerabilities are tolerated (by using NONE, because if any CVSS or EPSS values are found at all, the returned category in either selector cannot be NONE.

The semantics of defining two identities for the same workload help specify that this application tolerates either no vulnerabilities or low-risk and low-severity ones. If no vulnerabilities are found, the identity example-service/main/no-severity/no-risk will be issued, while if low-risk and low-severity vulnerabilities are found, then the identity example-service/main/low-severity/low-risk will be issued.

If both identities had the same SPIFFE IDs, which is possible, then a neighboring workload that accepts that ID will allow communication regardless of which identity was issued. However, if they have different SPIFFE IDs, the neighboring workload can decide if it trusts all IDs prefixed by example-service or not. This allows the neighboring workload to evaluate the risk of trusting that same workload when its vulnerability posture changes over time.

Listing 1: Deployment manifest example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  selector:
    ...
  template:
    metadata:
      labels:
        ..
      annotations:
        spire.io/criticality: "HIGH"
    spec:
      ...
```

[5]https://spiffe.io/docs/latest/deploying/spire_agent/

Listing 2: Plugin configuration

```
WorkloadAttestor "cc" {
    plugin_data {
      dependency_track_host = "dependency-track.example.org.com"
      dependency_track_port = "8080"
      dependency_track_x_api_key = "X_API_KEY"
      cosign_login_registry = "ghcr.io"
      cosign_login_username = "registryusername"
      cosign_login_password = "registrytoken"
      trusted_certificate_oidc_issuers = ["https://token.actions.githubusercontent.com"]
      trusted_certificate_identities = ["https://github.com/example-company/trusted-pipelines/.github/
          workflows/devsecops-pipeline.yml@refs/heads/main"]
      grace_periods = {
            "LOW" = {
                "LOW" = [240, 220, 210, 180],
                "MEDIUM" = [135, 120, 105, 90],
                "HIGH" = [60, 50, 40, 30],
                "CRITICAL" = [25, 20, 15, 10] }
            "MEDIUM" = {
                "LOW" = [135, 120, 105, 90],
                "MEDIUM" = [60, 50, 40, 30],
                "HIGH" = [25, 20, 15, 10],
                "CRITICAL" = [8, 7, 6, 5] }
            "HIGH" = {
                "LOW" = [60, 50, 40, 30],
                "MEDIUM" = [25, 20, 15, 10],
                "HIGH" = [8, 7, 6, 5],
                "CRITICAL" = [3.5, 3, 2.5, 2] }
            "CRITICAL" = {
                "LOW" = [25, 20, 15, 10],
                "MEDIUM" = [8, 7, 6, 5],
                "HIGH" = [3.5, 3, 2.5, 2],
                "CRITICAL" = [1.25, 1, 0.75, 0.5] }
          }
    }
}
```

Listing 3: SPIRE entry creation example

```
spire-server entry create \
    -spiffeID spiffe://example.org/example-service/main/low-severity/low-risk \
    -parentID spiffe://example.org/ns/spire/sa/spire-agent \
    -selector cc:has-sbom:true \
    -selector cc:highest-epss-risk:LOW \
    -selector cc:highest-cvss-severity:LOW \
    -selector cc:attestation-certificate-identity:https://github.com/example-company/trusted-pipelines
        /.github/workflows/devsecops-pipeline.yml@refs/heads/main \
    -selector cc:has-provenance:true \
    -selector cc:source-code-uri:https://github.com/example-company/example-service \
    -selector cc:source-code-version:vd.1.4 \
    -selector cc:attestation-certificate-oidc-issuer:https://token.actions.githubusercontent.com

spire-server entry create \
    -spiffeID spiffe://example.org/example-service/main/no-severity/no-risk \
    -parentID spiffe://example.org/ns/spire/sa/spire-agent \
    -selector cc:has-sbom:true \
    -selector cc:highest-epss-risk:NONE \
    -selector cc:highest-cvss-severity:NONE \
    -selector cc:attestation-certificate-identity:https://github.com/example-company/trusted-pipelines
        /.github/workflows/devsecops-pipeline.yml@refs/heads/main \
    -selector cc:has-provenance:true \
    -selector cc:source-code-uri:https://github.com/example-company/example-service \
    -selector cc:source-code-version:vd.1.4 \
    -selector cc:attestation-certificate-oidc-issuer:https://token.actions.githubusercontent.com
```

5.5 Grace Period Impact on Compliance Enforcement

As shown in Section 5.3, when the grace period configuration is enabled, the CVSS and EPSS Scores of a newly identified CVE will only be considered as potential compliance violations after the expiry of the period for that CVE. If, after the grace period filters tolerated vulnerabilities, the highest CVSS or EPSS among the remaining CVEs still exceeds the thresholds defined in the workload's identity selectors, the attestation will fail, and an SVID is not issued to the workload.

To illustrate this behavior, consider a workload deployed with a manifest such as the one shown in Listing 1. This means it would have a high criticality value and would be represented by Table 4, as set in the plugin's configuration in Listing 2. Also, consider the registration entries in Listing 3. These selectors define the workload as accepting only vulnerabilities with low severity and low risk. Table 6 summarizes four base scenarios involving a CVE identified in this context:

- **SC1: CVE is not tolerated, but is within the grace period**. A CVE published 15 days ago is identified with a CVSS higher than the allowed threshold. However, since it is still within the configured grace period, its levels are temporarily tolerated, and the attestation succeeds.
- **SC2: CVE is not tolerated, and the grace period expired**. 11 additional days have passed since SC1, causing the grace period to expire. The same CVE now has its severity and risk levels evaluated, and since they exceed the allowed thresholds, the attestation fails.
- SC3: Tolerated CVE within the grace period. A recently published CVE (3 days ago) is identified, with both CVSS and EPSS Scores within the accepted low thresholds. Although its scores are acceptable, it is disregarded by the plugin since it is still in its grace period. The attestation succeeds.
- SC4: Tolerated CVE outside of the grace period. The same low CVSS and low EPSS CVE has expired its grace period, but its scores remain within acceptable thresholds. The attestation still succeeds.

In other words, the grace period mechanism does not alter the core compliance enforcement logic implemented by the plugin — it simply delays the evaluation of CVEs during a configured window of time. This deferral enables organizations to implement measured responses to newly discovered vulnerabilities, instead of immediate enforcement. The criticality defined for each workload influences this behavior directly: the higher the criticality, the lower the **tolerance to potential exposure**, reflected in shorter grace periods conceived for the CVEs.

6 Evaluation

SPIRE is a graduate project at CNCF, it is already considered a stable, production-ready system [CNCF, 2024]. In addition, SPIRE generates cryptographically robust identities in the form of X.509 certificates and these certificates are used by also mature libraries and proxies (e.g., Envoy [Envoy Project, 2025]). Therefore, instead of evaluating the resources and security of SPIRE and tools related to zero-trust implementation, we focus on the proposed plugin itself. We separate our evaluation into three aspects: (1) a performance analysis to assess the impact of our plugin and its supporting architecture on a running SPIRE environment, (2) a security analysis to verify if the motivating problems are solved by adopting our plugin, and (3) an analysis of the threats to validity to provide a transparent overview of the study's limitations and the factors that may influence the general applicability of our results.

6.1 Performance costs

To evaluate the performance cost, we must first consider the resource allocation for running *Dependency Track*, which is responsible for analyzing the SBOM. According to official guidance, the sum of the *Dependency Track* Docker containers requires a minimum of 4.5 GiB of memory and 2 CPU cores, with a recommended allocation of 16 GiB of memory and 4 CPU cores [Springett, 2024]. In a production environment concerned with compliance, a vulnerability assessment tool would already be a necessary expense. Hence, these resource costs are inherent to the security posture itself, not an exclusive overhead of our solution.

We deployed *Dependency Track* on a Kubernetes cluster and measured its resource consumption through a controlled workload to validate these recommendations. Additionally, because the SCA runs entirely on *Dependency Track* and is completely parallel to the attestation process, the only potential overhead imposed by integrating *Dependency Track* would be the latency of REST API communications.

6.1.1 Impact on resources

By default, a SPIRE workload attestation occurs at half of the certificate's expiration time. Since a short-lived certificate has a default longevity of one hour, reattestation happens approximately every 30 minutes, or twice per hour. Given that the daily mean of CVE reports in 2023 for NVD is 79.18, which amounts to 3.29 per hour [Gamblin, 2024], checking for new vulnerabilities frequently is a recommended practice.

To test the solution's impact on performance and scalability, we simulated environments with multiple workloads. Because we are interested in seeing how well the system handles stress when accommodating highly scalable applications, we chose certain bursts of attestation requests per second to represent a high number of workloads that might request attestation at the same time. The bursts start from only 1 request per second, then go to 50, then to 100, and stop at 250. Realistically, workloads from multiple applications will not all attest simultaneously, so these bursts should not represent the total number of workloads in a single environment, but rather the volume of incidental concurrent requests. While higher stress levels could be tested, the likelihood of such a large number of applications initiating attestation at the same moment – unless deliberately coordinated – is minimal. For this reason, we capped our evaluation at 250 simultaneous requests.

Table 6. Attestation result for CVEs in different conditions

Scenario	CVE Severity	CVE Risk	Grace Period	Publication Time	Attestation Result
SC1	MEDIUM	LOW	25d	15d	Succeed
SC2	MEDIUM	LOW	25d	26d	Fail
SC3	LOW	LOW	135d	3d	Succeed
SC4	LOW	LOW	135d	136d	Succeed

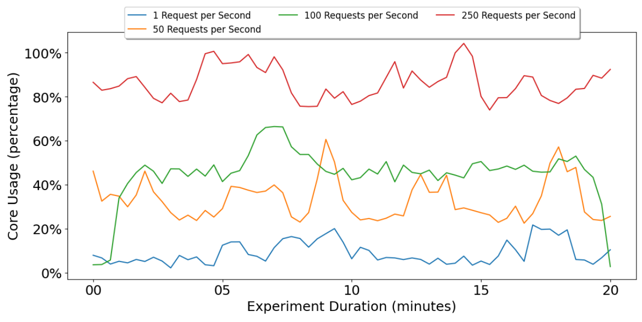

Figure 6. Dependency Track's CPU Usage per Burst

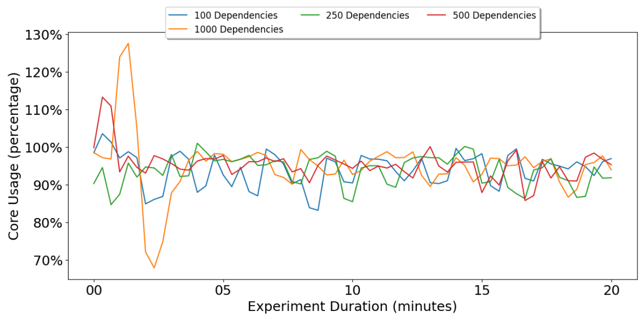

Figure 8. Dependency Track CPU Usage per SBOM Size

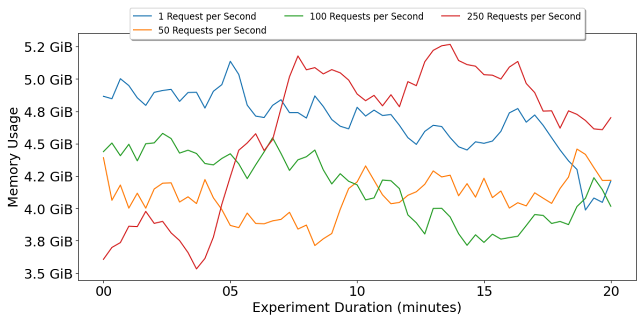

Figure 7. Dependency Track Memory Usage per Burst

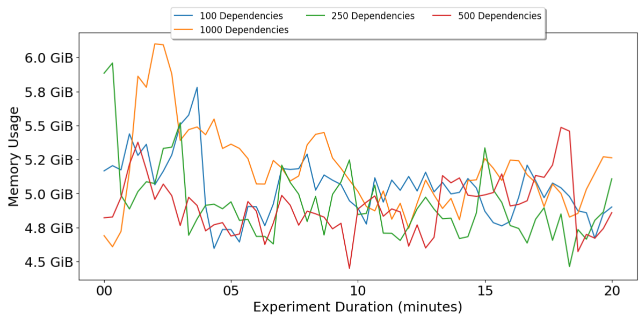

Figure 9. Dependency Track Memory Usage per SBOM Size

To fire these request bursts, we used the same application with an increasing number of replicas. This approach is equivalent to using distinct applications, given that each replica will nonetheless try to attest independently and that the workload attestor does not have any applicable cache logic that could cause interference. The application's SBOM contained over 800 dependencies, above the average of 526 per application [Synopsys, 2024]. For each burst number, the workload requested attestation from the SPIRE Agent, which prompted the plugin to query Dependency Track for a vulnerability report and to verify the grace period before returning the selectors found. Each burst number was executed for 20 minutes and repeated 30 times to minimize the interference of infrastructure on the experiment.

Figure 6 illustrates the CPU usage for each burst level. We can see that although processing time changes with higher demands, it remains well below the recommended 4 CPU cores, with observed peaks just exceeding 100% of a single core's worth of time. Figure 7 exhibits a different behavior regarding memory usage. While it is true that even the heaviest load remains well below the 16 GiB recommendation, the usage is not as stable. This fluctuation correlates to Dependency Track updating its database mirrors periodically, in parallel to synchronous requests. As a result of this independent update, Dependency Track appears to withstand both sudden demand peaks and high, stable loads.

Since all burst cases use the same application, the SBOM is the same across all cases. Given that Dependency Track analyzes each dependency in the SBOM to perform SCA, further experimentation is required to assess the impact of SBOM size on resource usage. This is particularly relevant because different applications within the same ecosystem might use different technology stacks, resulting in widely varying numbers of dependencies (be they direct or transitive). To address this, we extended our experiments to cover four different SBOM sizes: 100, 250, 500, and 1000 dependencies. In order to better isolate the impact of the SBOM's size, we stabilized the number of requests per second to 1.

Figure 8 shows that, apart from an initial peak, the number of dependencies does not dictate CPU usage. After this peak, usage fluctuates with no strong pattern across all SBOM sizes. A similar behavior is displayed in Figure 9, concerning memory usage: an initial peak that is influenced by SBOM size, followed by a general fluctuation that does not strongly correlate with size changes.

Another observation is that the initial peak displays the most resource-intensive part of the attestation process: the first time a workload is ever attested. This corresponds to step 7.1 in Figure 4, where the project is created in Dependency Track if it does not yet exist. This means SCA is being performed for the first time, which is understandably a more intensive task than updating an internal database of the most recent vulnerabilities.

In conclusion, our findings suggest that the recommended resource allocation can be more than sufficient, even for high-stress situations involving numerous simultaneous attestation

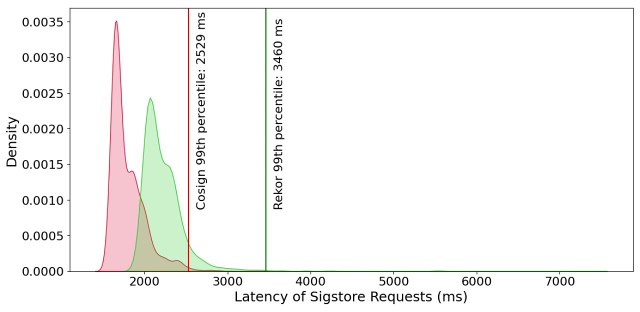

Figure 10. Distribution of Latency in Cosign Requests

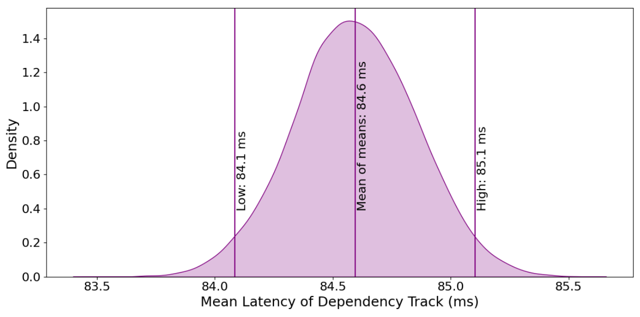

Figure 11. Distribution of Latency in Dependency Track Requests

requests and varying SBOM sizes. Furthermore, if attestation resources become a concern, simply reconfiguring the identity's lifetime (e.g., to be longer than one hour) or the renewal margins (e.g., renewing at 75% of lifetime instead of 50%) would considerably reduce the attestation load.

6.1.2 Impact on latency

Regarding added latency, there are two primary points of interest: the latency from *Sigstore*-related requests and the latency from *Dependency Track*-related requests. While the latter involves requests to a single component, the former comprises requests to both *Cosign* (to download attestations) and *Rekor* (to validate the signature and its trustworthiness).

To test this, we performed over 1 600 attestations on a test environment using the default public remote of *Sigstore* and a *Dependency Track* instance running in the same Kubernetes cluster as SPIRE. We measured the individual latency for each type of *Sigstore* request, as well as *Dependency Track* requests, during each attestation.

Figure 10 illustrates the latency distribution for each *Sigstore* component. Despite the distributions being highly skewed due to network variability, their 99^{th} percentiles show that attestations are typically downloaded in 2 529 ms or less and are then verified in 3 460 ms or less. This amounts to just above 6 seconds of added delay for a single attestation.

As for *Dependency Track*, because the distribution is log-normal, we bootstrapped the MLE (Maximum Likelihood Estimation) of the mean of latency on 5 000 re-samples. The resulting mean of means, as shown in Figure 11, is 84.6 ms, with a Confidence Interval of [84.1 ms, 85.1 ms] for a confidence level of 95%.

When combining the latency impacts of *Dependency Track* and the *Sigstore* components, the total additional processing time per attestation is less than 6 seconds in most cases. This latency does not significantly hinder an attestation process, given that attempts are made by default twice every hour.

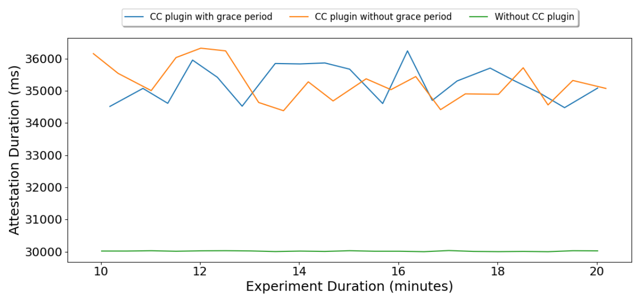

Figure 12. Comparison of total attestation latency without plugin, with plugin but without grace period, and with plugin and grace period

6.1.3 Grace period impact on evaluation

The introduction of the grace period does not significantly affect the performance of the plugin. In practical terms, it performs a lookup on the local configuration and decides whether to omit a CVE or not, in case the grace period is still active. Consequently, this feature builds directly on the result of the already existing *Dependency Track* communication, without introducing new interactions or altering the established communication flow in any way.

Considering that no impact on external components can be caused by the plugin, assessing its true performance overhead requires measuring its effect on the attestation process itself. To this end, we compared the general attestation response time of the SPIRE Agents in three different scenarios: (1) the default SPIRE Agent with the compliance plugin disabled, (2) the SPIRE Agent with the plugin enabled but with the grace period feature disabled, and (3) the SPIRE Agent with all compliance plugin features enabled. For this comparison, we performed a 20-minute execution for each configuration, using the SPIFFE Helper as the workload performing the attestation attempts. The experiment ignores a warm-up phase, focusing on the steady state, and is shown in Figure 12.

As seen in the figure, the total attestation time with the plugin, with and without the grace period filtering, is around 6 seconds longer than the base attestation time. Also noteworthy is that the base attestation is very stable, while the continuous compliance plugin shows some variation. This is a consequence of the base attestation using local information, while the plugin still needs to query the Dependency Track, which inevitably adds some network and processing jitter.

6.2 Security evaluation

As defined in our threat model from Section 3, our security objectives do not include an analysis of failing to implement SPIRE's or ZTA's guidelines or how trustworthy and complete the CVEs provided by NVD are to the Dependency Check tool. Therefore, we first analyze the workload attestation workflow and then review how this workflow satisfies our requirements defined in Section 3.

The periodic re-attestation performed by SPIRE analyzes workloads periodically. An SPIRE Agent calls the attestation plugins, and they use direct and indirect means to derive the selectors (i.e., properties) of a workload. Our vulnerability compliance plugin adds SCA to this routine. Then, because of *Dependency Track's* self-update, the plugin always returns the current vulnerabilities associated with the workload's dependency. Therefore, changes in this status are tracked at each re-attestation, extending continuous compliance beyond when the deployment occurred.

Next, as SPIRE selectors will reflect the expected vulnerability posture of a workload, deviations cause identities to no longer be issued. Consequently, mTLS connections to the noncompliant workload will cease after the previous certificate expires. Thus, workloads or the zero-trust proxies that implement mTLS connections correctly will transparently ensure vulnerability posture compliance. Such support addresses the gap we found in ZTA's treatment of compliance while applying a first, automatic response to incidents.

The failed workload is free to retry attestation *ad infinitum*. However, the only way to receive an actual valid identity certificate is through outside forces, such as updates in CVSS or EPSS Scores in the CVEs related to its SBOM. Alternatively, developers and operators can interact to insert a VEX entry that voids that specific vulnerability to that specific project into Dependency Track. In both cases, the workload has become compliant again because either its vulnerability state changed or it was considered irrelevant, or at least not urgent, due to internal officers' intervention.

As explained in our threat model, we are also concerned about the provenance of an SBOM, since it is the source of truth for our vulnerability assessment. Two cases may take place. First, if there is an attempt to alter or forge the SBOM, *Cosign* and *Rekor* can easily use the SLSA Provenance to discriminate the origin of the trustworthiness of the SBOM. As SLSA 3 focuses on signed SBOMs, this guarantees that forging the provenance is beyond the capabilities of most adversaries. We can avoid the threat of illegitimate SBOM masking vulnerabilities. Second, it may be the case that some critical applications require a reviewing board. In this case, if only specific entities should sign the SBOM for some application, the SPIRE entries for the identities of the workloads will include the OIDC issuer field as a selector (as detailed in Table 1). This selector will force SPIRE to check this field in the SBOM, and the use of invalid issuers will block workloads from getting the identity.

Once workloads have no identity, they will be isolated. In practice, as the workload identity should be the same as that used to respond to health checkers, the workload will be terminated. Workload termination should then generate alerts on a monitoring system, which is also a well-established practice for production systems.

The consequence of isolating a noncompliant workload is, of course, reduced availability. Isolating non-critical applications due to minor security issues reduces our solution's applicability. To mitigate this, we provide a flexible grace period option, so that stakeholders can decide the best prevention *versus* availability trade-off, considering the isolation's impact on the provided workloads and the vulnerability mitigation deadlines of the development team.

In summary, the implemented plugin, through the help of *Sigstore* and *Dependency Track*, can map selective identity provisioning with vulnerability posture rules for compliance. Tying vulnerability posture rules to X.509 identities that are (directly or indirectly) the base for all communications effectively isolates noncompliant workloads, even if they were previously considered compliant at some point. This isolation prevents threat exploits as soon as organizational policies dictate, even without human intervention.

6.3 Threats to validity

This work proposes integrating vulnerability assessment into ZTA to enable continuous compliance enforcement post-deployment. In order to assess vulnerability, SCA should be continuously performed and its results used as evidence of an application's security posture. Consequently, this approach heavily depends on the completeness and accuracy of vulnerability databases.

Such databases merely contain the CVEs disclosed by CNAs (CVE Numbering Authorities) — third-party organizations authorized by the CVE Program to disclose and assign CVEs CVE [2025]. Examples include the MITRE organization, the GitHub Security Advisory, as well as software vendors such as Netflix Inc and Atlassian [6]. These CNAs are also responsible for devising appropriate mitigations for the vulnerabilities published by them. However, some CNAs take longer than others to disclose, and their severity assessments might diverge, potentially introducing inconsistencies or biases in CVSS and EPSS scoring Lin *et al.* [2023]. This can lead to false positives or false negatives in vulnerability scores, which in turn might negatively impact the workload attestor's decision.

While this risk also impacts this paper, it is a well-recognized, field-wide challenge that cannot be mitigated at the methodological level of this work. In fact, it is imperative for organizations employing vulnerability assessments to remain aware of this and implement complementary strategies to mitigate its impact.

Another significant factor is the accuracy of the SBOM. Since the SBOM is the required basis for SCA, incomplete or incorrect data results in an inaccurate vulnerability list. This highlights the importance of choosing a reliable SBOM generation tool, such as Trivy and Snyk. This is not to be seen as a recommendation, however, as there are many open-source tools as well as proprietary ones available. Such a choice is at the discretion of the organization, and this paper does not aim to guide on choosing tools, but rather only to point out their importance.

Additionally, the choice of tools also represents a potential threat. This work is supported by the selection of prestigious tools, based on related work and relevant, renowned organizations. The correct mitigation of the threat model in Section 3 relies on the chosen tools (i.e., SPIRE and Dependency Track) and their respective open-source communities' continuous maintenance. While such tools can lose quality or become inadequate over time, this risk is mitigated by the backing of well-regarded organizations such as the CNCF and OWASP, ensuring ongoing relevance and security.

With the introduction of the grace period mechanism, compliance assurance also depends on accurately translating an organization's policy into the values added by the SPIRE operator to the Workload Attestor's configuration file. Since these policies are defined and updated through human decision-making, they are subject to change due to executive strategies

[6]https://www.cve.org/PartnerInformation/ListofPartners

and may not always be accurately reflected in the configuration. Outdated, incorrect, or improperly entered grace period values can result in attestation outcomes misaligned with organizational intent. To avoid undesired behavior, when deciding for configuring a grace period policy, the organization must also establish a strict process to make sure these values have been correctly entered by the SPIRE operator and reconfigured whenever the policies are updated.

Lastly, the very existence of a grace period, if not strategically employed, may introduce a threat by fostering a false sense of security. Vulnerabilities might appear absent when, in fact, their assessment has merely been deferred. In this sense, even though the Workload Attestor responds as intended, the mechanism fails to fulfill its purpose, and instead only allows the system to be available with unaddressed vulnerabilities. This availability window should only be permitted under a clear strategy. Therefore, it is important to monitor the vulnerability records logged by the Workload Attestor, and use the afforded time to prepare for remediation or for a potential downtime once the grace period expires, rather than simply overlooking the vulnerabilities.

7 Conclusion

In this work, we present a comprehensive workflow designed to continuously evaluate workloads for compliance related to their provenance and vulnerability status. This solution fills a significant gap in existing continuous compliance methodologies, which often overlook vulnerabilities post-deployment. Our approach is grounded in Zero-Trust principles, where applications undergo explicit and ongoing authentication. It is built upon two fundamental requirements: (1) a CI/CD pipeline that generates compliance evidence, specifically a Software Bill of Materials (SBOM) and, ideally, provenance information of the source code, such as SLSA Provenance; and (2) an identity provisioning tool that regularly updates the identities utilized within a Zero-Trust framework. This tool effectively isolates workloads that fail to renew their identities. We believe these requirements are in line with modern, well-established practices in development and operations.

To ensure the solution is applicable for both critical and non-critical workloads, we introduced the concept of a configurable grace period for newly discovered CVEs. By deferring immediate enforcement, the grace period provides the organization with a critical window to assess the risk, plan mitigation steps, and prepare for any necessary downtime. The possible duration values for the grace period vary according to the characteristics of the CVE and the application in which it was identified, based on the following factors: (1) the severity of the impact of the CVE being exploited, as indicated by its CVSS score; (2) the likelihood of exploitation within the next 30 days, as estimated by the EPSS score; and (3) the criticality of securing the affected application within the system's context, as specified by the infrastructure operator. By explicitly encoding these criteria into a policy, organizations can better align vulnerability management with their risk tolerance and operational constraints.

We implemented this approach as a new plugin for the CNCF[7] SPIRE framework, integrating simple and popular tools, such as the *Sigstore* framework and OWASP's[8] *Dependency Track*. Our evaluation demonstrated that the plugin does not hinder the deployment or operation of modern cloud-native applications. Specifically, the performance analysis showed that it does not add significant latency to SPIRE's attestation process, and enabling the grace period feature had negligible runtime cost. The necessary resources to run our implementation are primarily those required to run *Dependency Track* in a scalable way – an expense already justified in any compliance-conscious environment. These results confirm that our approach is practical for real-world cloud-native deployments without compromising operational performance.

Finally, the extensibility of SPIRE serves as a fertile ground for simple-to-adopt verification mechanisms. For example, we envisage that additional compliance metrics can be implemented together with SPIRE's open-source community.

Acknowledgements

This work has been financed through the Secure and Scalable Identity Provisioning (SSIP) and CloudAI projects, collaborations between Hewlett Packard Enterprise Brazil and the EMBRAPII unit UFCG-CEEI (Universidade Federal de Campina Grande) with the incentive of the Informatics Law (Law 8.248 from October 23rd, 1991). We thank the other members of the projects, especially Laerson Veríssimo, for the discussions about CI/CD pipelines and Dependency Track, and to Jan Paulo Filho and Raiff Silva for the discussions about compliance.

This manuscript represents an extended version of our previously published paper, entitled 'Supporting continuous vulnerability compliance through automated identity provisioning,' which appeared in the Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing (LADC '24). In this extended work, we evolve the plugin via the grace period functionality to allow flexibility for organizations with both critical and non-critical applications. We also include better configuration guidelines so organizations can better adopt this proposal.

Funding

This research was funded by a collaboration between Hewlett Packard Enterprise Brazil and the EMBRAPII unit UFCG-CEEI (Universidade Federal de Campina Grande) with the incentive of the Informatics Law (Law 8.248 from October 23rd, 1991).

Authors' Contributions

Diego Gama and Andrey Brito are the main contributors to this work, from the research leading to the plugin up to its implementation, as well as its experimentation. Carlos Fuch led the efforts to implement the grace period functionality, and both André Martin and Christof Fetzer contributed with their expertise in cybersecurity, providing insights and feedback on the ideas and on the manuscript.

Competing interests

The authors declare they have no competing interests.

[7]https://www.cncf.io/about/who-we-are/
[8]https://owasp.org/about/

References

Agarwal, V., Butler, C., Degenaro, L., Kumar, A., Sailer, A., and Steinder, G. (2022). Compliance-as-code for cybersecurity automation in hybrid cloud. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 427–437. DOI: 10.1109/CLOUD55607.2022.00066.

Babakian, A., Monclus, P., Braun, R., and Lipman, J. (2022). A retrospective on workload identifiers: From data center to cloud-native networks. *IEEE Access*, 10:105518–105527. DOI: 10.1109/ACCESS.2022.3211293.

Buck, C., Olenberger, C., Schweizer, A., Völter, F., and Eymann, T. (2021). Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust. *Computers Security*, 110:102436. DOI: 10.1016/j.cose.2021.102436.

Chen, B., Qiao, S., Zhao, J., Liu, D., Shi, X., Lyu, M., Chen, H., Lu, H., and Zhai, Y. (2021). A security awareness and protection system for 5g smart healthcare based on zero-trust architecture. *IEEE Internet of Things Journal*, 8(13):10248–10263. DOI: 10.1109/JIOT.2020.3041042.

CISA (2023a). When to issue vex information. Available at: https://www.cisa.gov/resources-tools/resources/when-issue-vex-information/ Last accessed on April 13th, 2025.

CISA (2023b). Zero trust maturity model v2.0. vailable at: https://www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf Last accessed April 13th, 2025.

CISA (2025). Stakeholder-specific vulnerability categorization (ssvc). Available at: https://www.cisa.gov/stakeholder-specific-vulnerability-categorization-ssvc Last accessed April 13th, 2025.

CNCF (2024). Graduated and incubating projects — cncf.io. Available at: https://www.cncf.io/projects/ Last accessed April 13th, 2025.

CVE (2025). Cve numbering authorities (cnas). Available at: https://www.cve.org/programorganization/cnas Last accessed August 06th, 2025.

CycloneDX Core Working Group (2024). Cyclonedx: Authoritative guide to sbom. Available at: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf Last accessed April 13th, 2025.

Cyentia Institute and Kenna Security (2022). Prioritization to prediction volume 8: Measuring and minimizing exploitability. Technical report, Cyentia Institute. Available at: https://library.cyentia.com/report/report_008756.html Last accessed on April 13th, 2025.

de Weever, C. and Andreou, M. (2020). Zero trust network security model in containerized environments. *University of Amsterdam: Amsterdam, The Netherlands*. Available at: https://rp.os3.nl/2019-2020/p01/report.pdf.

Envoy Project (2025). Envoy proxy. Avalable at: https://www.envoyproxy.io/ Last accessed August 06th, 2025.

Fang, R., Bindu, R., Gupta, A., and Kang, D. (2024). Llm agents can autonomously exploit one-day vulnerabilities. DOI: 10.48550/arXiv.2404.08144.

FIRST (2024). Exploit prediction scoring system (epss) — first.org. Available at:https://www.first.org/epss/ Last accessed April 13th, 2025.

Gamblin, J. (2024). 2023 cve data review — jerrygamblin.com. Available at: https://jerrygamblin.com/2024/01/03/2023-cve-data-review/ Last accessed April 13th, 2025.

GSA, U. (2024). Documents & templates | fedramp.gov — fedramp.gov. Available at: https://www.fedramp.gov/documents-templates/ Last accessed April 13th, 2025.

He, Y., Huang, D., Chen, L., Ni, Y., Ma, X., and Huo, Y. (2022). A survey on zero trust architecture: Challenges and future trends. *Wirel. Commun. Mob. Comput.*, 2022. DOI: 10.1155/2022/6476274.

IBM (2024a). The kerberos ticket — ibm.com. Available at: https://www.ibm.com/docs/en/sc-and-ds/8.4.0?topic=concepts-kerberos-ticket Last accessed April 13th, 2025.

IBM (2024b). Solarwinds orion (cve-2020-10148) — ibm.com. Available at: https://www.ibm.com/docs/en/randori?topic=2022-solarwinds-orion-cve-2020-10148 Last accessed April 13th, 2025.

Inc., W. (2024). The leading open-source iam solution — wso2.com. Available at: https://wso2.com/identity-server/ Last accessed April 13th, 2025.

ITU (2020). Security requirements of public infrastructure as a service (iaas) in cloud computing (recomendation itu-t x.1605). Available at: http://handle.itu.int/11.1002/1000/14094.

Jacobs, J., Romanosky, S., Edwards, B., Adjerid, I., and Roytman, M. (2021). Exploit prediction scoring system (epss). *Digital Threats*, 2(3). DOI: 10.1145/3436242.

Johnson, P., Lagerström, R., Ekstedt, M., and Franke, U. (2018). Can the common vulnerability scoring system be trusted? a bayesian analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):1002–1015. DOI: 10.1109/TDSC.2016.2644614.

Kellogg, M., Schäf, M., Tasiran, S., and Ernst, M. D. (2021). Continuous compliance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 511–523, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3324884.3416593.

Lin, J., Adams, B., and Hassan, A. E. (2023). On the coordination of vulnerability fixes: An empirical study of practices from 13 cve numbering authorities. *Empirical Software Engineering*, 28(6):151. DOI: 10.1007/s10664-023-10403-x.

NVD (2025). Nvd - cve-2024-3094 — nvd.nist.gov. Available at: https://nvd.nist.gov/vuln/detail/cve-2024-3094 Last accessed April 13th, 2025.

Nygard, C. (2021). Compliance in a devops culture — martinfowler.com. Available at: https://martinfowler.com/articles/devops-compliance.html Last accessed on April 13th, 2025.

PCI Security Standards Council (2024). Official pci security standards council site. Available at: https://east.pcisecuritystandards.org/ Last accessed April 13th, 2025.

Ramaj, X., Sánchez-Gordón, M., Gkioulos, V., Chockalingam, S., and Colomo-Palacios, R. (2022). Holding on to compliance while adopting devsecops: An slr. *Electronics*, 11(22). DOI: 10.3390/electronics11223707.

Rose, S., Borchert, O., Mitchell, S., and Connelly, S. (2020). Zero trust architecture. DOI: 10.6028/NIST.SP.800-207.

Sigstore (2024). Signing — docs.sigstore.dev. Available at: https://docs.sigstore.dev/cosign/signing/overview/ Last accessed April 13th, 2025.

Sirish, A. and Hennen, T. (2024). in-toto and slsa — slsa.dev. Available at: https://slsa.dev/blog/2023/05/in-toto-and-slsa Last accessed April 13th, 2025.

SLSA Specification (2025). Slsa v1.0 security levels. Available at: https://slsa.dev/spec/v1.0/levels Last accessed April 13th, 2025.

Sonatype (2022). 8th state of the software supply chain. Available at: https://www.sonatype.com/resources/state-of-the-software-supply-chain-2022/introduction Last accessed April 13th, 2025.

Sonatype (2023). 9th state of the software supply chain. Available at: https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-and-demand Last accessed April 13th, 2025.

SPIFFE (2025a). Github - spiffe/spiffe-helper: The spiffe helper is a tool that can be used to retrieve and manage svids on behalf of a workload — github.com. Available at: https://github.com/spiffe/spiffe-helper Last accessed April 13th, 2025.

SPIFFE (2025b). Spire concepts. Available at: https://spiffe.io/docs/latest/spire-about/spire-concepts/ Last accessed August 6th, 2025.

SPIFFE Project (2025). spire/adopters.md at main · spiffe/spire — github.com. Available at: https://github.com/spiffe/spire/blob/main/ADOPTERS.md Last accessed April 13th, 2025.

Springett, S. (2024). Deploying docker container — docs.dependencytrack.org. Available at: https://docs.dependencytrack.org/getting-started/deploy-docker/ Last accessed April 13th, 2025.

Steffens, A., Lichter, H., and Moscher, M. (2018). Towards data-driven continuous compliance testing. In *Software Engineering*. Available at: https://api.semanticscholar.org/CorpusID:3818261.

Syed, N. F., Shah, S. W., Shaghaghi, A., Anwar, A., Baig, Z., and Doss, R. (2022). Zero trust architecture (zta): A comprehensive survey. *IEEE Access*, 10:57143–57179. DOI: 10.1109/ACCESS.2022.3174679.

Synopsys (2024). Open source security & risk analysis report (ossra) | synopsys — synopsys.com. Available at: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html Last accessed April 13th, 2025.

Torkura, K. A. and Meinel, C. (2016). Towards vulnerability assessment as a service in openstack clouds. In *2016 IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops)*, pages 1–8. DOI: 10.1109/LCN.2016.022.