UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO ALVES GAMA

SUPPORTING CONTINUOUS VULNERABILITY COMPLIANCE THROUGH AUTOMATED IDENTITY PROVISIONING

CAMPINA GRANDE
2024

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Supporting continuous vulnerability compliance through automated identity provisioning

Diego Alves Gama

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Sistemas de Computação

Nome do Orientador

(Andrey Elísio Monteiro Brito)

Campina Grande, Paraíba, Brasil

G184s Gama, Diego Alves.
Supporting continuous vulnerability compliance through automated identity provisioning / Diego Alves Gama. – Campina Grande, 2024.
41 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2024.
"Orientação: Prof. Dr. Andrey Elísio Monteiro Brito".
Referências.

1. Computer Systems. 2. Continuous Compliance. 3. Zero Trust Architecture. 4. Vulnerability Management. 5.Incident Response. 6. Identity Provisioning. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU 004.031(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secpg@computacao.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

DIEGO ALVES GAMA

SUPPORTING CONTINUOUS VULNERABILITY COMPLIANCE THROUGH AUTOMATED IDENTITY PROVISIONING

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 12/09/2024

Prof. Dr. ANDREY ELÍSIO MONTEIRO BRITO, UFCG, Orientador

Prof. Dr. JOÃO ARTHUR BRUNET MONTEIRO, UFCG, Examinador Interno

Prof. Dr. FÁBIO JORGE ALMEIDA MORAIS, UFCG, Examinador Interno

Prof. Dr. ANDRÉ MARTIN, TU-Dresden, Examinador Externo

Referência: Processo nº 23096.061197/2024-12 SEI nº 4791629

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secpg@computacao.ufcg.edu.br

REGISTRO DE PRESENÇA E ASSINATURAS

ATA Nº 020/2024 (DISSERTAÇÃO N° 734)

Aos doze (12) dias do mês de setembro do ano de dois mil e vinte e quatro (2024), às oito horas e trinta minutos (08:30), no Auditório Mário Hattori, da Universidade Federal de Campina Grande - UFCG, nesta cidade, e de forma online, através da plataforma Google Meet, reuniu-se a Comissão Examinadora composta pelos Professores ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG, Orientador, funcionando neste ato como Presidente, JOÃO ARTHUR BRUNET MONTEIRO, Dr., UFCG, FÁBIO JORGE ALMEIDA MORAIS, Dr., UFCG, ANDRÉ MARTIN, Dr. da(o) TU-Dresden, este com participação por videocoferência. Constituída a mencionada Comissão Examinadora pela Portaria Nº 031/2024 do Coordenador do Programa de Pós-Graduação em Ciência da Computação, tendo em vista a deliberação do Colegiado do Curso, tomada em reunião de 13 de Agosto de 2024 e com fundamento no Regulamento Geral dos Cursos de Pós-Graduação da Universidade Federal de Campina Grande - UFCG, juntamente com o Sr(a) DIEGO ALVES GAMA, candidato(a) ao grau de MESTRE em Ciência da Computação, presentes ainda professores e alunos do referido centro e demais presentes. Abertos os trabalhos, o(a) Senhor(a) Presidente da Comissão Examinadora anunciou que a reunião tinha por finalidade a apresentação e julgamento da dissertação "SUPPORTING CONTINUOUS VULNERABILITY COMPLIANCE THROUGH AUTOMATED IDENTITY PROVISIONING", elaborada pelo(a) candidato(a) acima designado, sob a orientação do(s) Professor(es) ANDREY ELÍSIO MONTEIRO BRITO, com o objetivo de atender as exigências do Regulamento Geral dos Cursos de PósGraduação da Universidade Federal de Campina Grande - UFCG. A seguir, concedeu a palavra, ao (a) candidato(a), o qual, após salientar a importância do assunto desenvolvido, defendeu o conteúdo da dissertação. Concluída a exposição e defesa do(a) candidato(a), passou cada membro da Comissão Examinadora a arguir o(a) mestrando sobre os vários aspectos que constituíram o campo de estudo tratado na referida dissertação. Terminados os trabalhos de arguição, o(a) Senhor(a) Presidente da Comissão Examinadora determinou a suspensão da sessão pelo tempo necessário ao julgamento da dissertação. Reunidos, em caráter secreto, no mesmo recinto, os membros da Comissão Examinadora passaram à apreciação da dissertação. Reaberta a sessão, o(a) Presidente da Comissão Examinadora anunciou o resultado do julgamento, tendo assim, o(a) candidato(a) obtido o Conceito APROVADO. Na sequência, o(a) Presidente da Comissão Examinadora anunciou o resultado do julgamento, tendo a seguir encerrado a sessão, da qual lavrei a presente ata, que vai assinada por mim, Paloma Nascimento Porto, pelos membros da Comissão Examinadora e pelo(a) candidato(a). Campina Grande, 12 de Setembro de 2024.

Documento assinado eletronicamente por **ANDREY ELISIO MONTEIRO BRITO**, **PROFESSOR 3 GRAU**, em 13/09/2024, às 07:44, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Referência: Processo nº 23096.061197/2024-12 SEI nº 4791627

Abstract

Most applications will exhibit vulnerabilities that impact their availability, integrity, or confidentiality during their life cycle. Nevertheless, the leading cause for such vulnerabilities is not the application itself but its dependencies. Continuous compliance processes often perform vulnerability assessment to prevent compliance breaches during a CI/CD pipeline. However, current proposals do not extend beyond the pipeline and thus do not take into account incident response when dynamic aspects change, such as newfound vulnerabilities on deployed applications. In this work, we leverage Zero Trust to continuously assess vulnerability compliance and isolate workloads that do not conform to a minimum vulnerability posture. Our approach builds on top of SPIRE, a selective identity provider, and integrates incident response caused by dynamic aspects to continuous compliance.

Resumo

A maioria das aplicações exibirá vulnerabilidades durante seu ciclo de vida, as quais podem afetar sua disponibilidade, integridade e confidencialidade. Entretanto, a origem principal para tais vulnerabilidades não é a aplicação em si, mas suas dependências. Processos de conformidade contínua frequentemente executam verificações de vulnerabilidade para prevenir violações de conformidade durante *pipelines* CI/CD. Entretanto, propostas atuais não se estendem além do *pipeline*, e portanto não consideram resposta a incidente de acordo com aspectos dinâmicos, tal qual o surgimento de vulnerabilidades em aplicações já implantadas. Este trabalho busca utilizar Zero Trust para verificar conformidade de vulnerabilidade de forma contínua e isolar cargas de trabalho que não respeitam uma postura mínima de vulnerabilidades. A abordagem proposta aproveita o SPIRE, um provedor seletivo de identidades graduado pela CNCF, e integra resposta a incidentes de violação de conformidade ao modelo de conformidade contínua.

Agradecimentos

Desejo agradeçer a Deus e à vida como um todo por me apoiar com tantas excelentes oportunidades. Sou grato porque cada desafio e obstáculo me ajudou a ter mais experiência, melhor compreensão, e me ajudou como profissional e como estudante.

Primeiramente, em um patamar muito pessoal, eu agradeço com todo o meu amor à minha família, cujo apoio foi essencial para que eu pudesse ter foco e empenho. Meus pais e meu irmão, que mesmo não entendendo muito da área, sempre respeitaram e incentivaram minha decisão de perseguir uma pós graduação. Graças a essa validação em dias de dúvida eu consegui continuar. Fui muito abençoado em nascer e crescer nesse lar, que nunca deixará meu coração.

Quero agradecer pela minha maravilhosa esposa, a quem sou extremamente grato pelo apoio incessante, e com quem tive o privilégio de nutrir meu segundo lar, nosso lar. Graças a ela sempre tive motivação e clareza para o próximo passo, desde a graduação, até a pós graduação. Em dias caóticos me escutou, em dias desanimados me reergueu. Sem seus encorajamentos e conselhos talvez não tivesse chegado até aqui. Minha estrela guia.

Finalmente, gostaria de agradecer nesse trabalho ao professor Andrey Brito, pela sua orientação, suas ideias e discussões que permitiram a culminação desta pesquisa. Agradeço também ao projeto SSIP (*Secure and Scalable Identity Provisioning*), realizado graças à colaboração entre Hewlett Packard Enterprise Brasil e unidade EMBRAPII UFCG-CEEI que me possibilitou a chance de alinhar a pesquisa às necessidades da indústria, de maneira a otimizar sua validade e impacto. Por consequência, agradeço aos membros do projeto, especialmente Laerson Veríssimo pelas discussões sobre *pipelines* CI/CD e *Dependency Track*, e a Jan Paulo de Lima e Raiff Silva pelas discussões acerca de conformidade.

Contents

List of Symbols

CI - *Continuous Integration*

CD - *Continuous Delivery*

CC - *Continuous Compliance*

CNCF - *Cloud Native Computing Foundation*

CVSS - *Common Vulnerability Scoring System*

DAST - *Dynamic Application Security Testing*

EPSS - *Exploitability Prediction Scoring System*

FIRST - *Forum of Incident Response and Security Teams*

NIST - *National Institute of Standards and Technology*

NVD - *National Vulnerability Database*

SAST - *Static Application Security Testing*

SCA - *Software Component Analysis*

SLSA - *Supply-chain Levels for Software Artifacts*

SBOM - *Software Bill of Materials*

TLS - *Transport Layer Security*

VEX - *Vulnerability Exploitability eXchange*

ZTA - *Zero Trust Architecture*

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Currently, the industry faces an emergent awareness regarding software supply chain attacks and their impacts on deployed products. From 2019 to 2021, attacks in supply chains around the world suffered a surprising growth of 742% [30], with examples such as SolarWind's Orion Platform and *XZ Utils*. Such attacks seek to weaken some part of the supply chain to inject vulnerable or malicious dependencies in order to exploit them after the product is delivered or deployed. This allows attackers to bypass security measures focused on outsider attacks and infect the ecosystem from the inside. This problem is intensified due to the prevalence of open-source projects being used as dependencies, allowing attackers to potentially infect multitudes of closed-source projects via successfully attacking some fundamental dependencies. As an example, in 2022, 245,000 open-source malicious dependencies were found, doubling the total amount of discovered malicious dependencies in all previous years combined [30].

Considerable effort has been made to develop tools and strategies to mitigate issues of supply chain security. DevSecOps best practices are among them, such as the use of SAST (Static Application Security Testing) during a CI/CD pipeline to help prevent local vulnerabilities in source code. Adopting SBOM (Software Bill of Materials) is also recommended to improve the transparency of the supply chain by describing all of the software components and dependencies used. SBOM is then frequently used to power SCA (Software Component Analysis) tools that look for known vulnerabilities in said dependencies, be they direct or transitive. The resulting output can be used to halt the release of a vulnerable system.

Many compliance regulations such as PCI DSS [7] and FedRAMP [2] include requirements about vulnerability detection and mitigation, which accentuates 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 [28]. 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 [1; 26].

1.1 The problem

The problem of vulnerable products is not necessarily mitigated if the code is compliant before a release. New vulnerabilities can be discovered at any time. The absence of known vulnerabilities in the product at release does not mean the product is completely free from them. A new CVE (Common Vulnerabilities Enumeration) entry can be reported as related to a dependency that was previously considered safe, as was the case for the recent *XZ Utils* disclosure that enabled a backdoor, affecting various Linux distributions. Recent studies have also shown that generative AI such as ChatGPT-4 can automatically generate exploits for CVEs [21], making new entries instantaneously exploitable. These high impact and short time to exploit are evidence of the need for continuous compliance approaches that find new vulnerabilities as quickly as possible.

The problem does not stop at vulnerabilities. Although not as pressing as immediate vulnerability exploitation, security controls requiring strict licenses for third-party dependencies as well as evidence of good risk management can also be violated during runtime. A previously reliable open-source project could change licenses, or could even be abandoned. This poses a problem to current continuous compliance strategies of always preventing, but not considering sudden violation as an incident.

In this scenario, ZTA (Zero-Trust architecture), defined by NIST (National Institute of Standards and Technology) [29], shows principles to help solve these problems, as it holds that by default, any person, event, or device inside and outside a network and information system is untrustworthy before sufficient authentication. Zero-Trust then poses authentication as based on specific evidence to meet criteria, which therefore defines trust as dynamic rather than static [15]. That also means that if trust can change over time, it must be checked continuously. Thus, only after continuous authentication can a service communicate with others within the network.

This idea of "never trust, always verify" diametrically opposes the notion that the product is trustworthy if it originated from a trusted DevSecOps pipeline. For this reason, we propose leveraging ZTA to extend continuous compliance beyond deployment in a zero-trust environment, by defining vulnerability tolerance as a policy for trust assessment to isolate untrusted services and prevent vulnerability exploitation.

1.2 Objectives

The main goal of this work is to build upon the state of practice on continuous compliance by defining vulnerability management as a trust policy. In order to achieve this goal, we enumerated the following specific objectives:

1. Identify threat points on general continuous compliance approaches where vulnerability status can be exploited for supply chain attacks.
2. Design an architecture for integrating DevSecOps and continuous compliance best practices to useful ZTA principles to mitigate found threats.
3. Implement the designed architecture using identified tools on state of practice.
4. Analyse the implementation's impact on resources and security as well as its limitations.

1.3 Contributions

In this work, we present the following three contributions:

1. We show how integrating vulnerability assessment as a policy on a ZTA's trust engine can isolate a non-conforming application within an environment after failing authentication. This can prevent vulnerability exploitation and can be considered an immediate response to compliance violations.

2. We implement this approach through a new custom workload attestor plugin for SPIRE, an open-source selective identity provider. It triggers SCA for each authentication attempt and then checks if found vulnerabilities are tolerated for each configured identity.

3. We evaluate the solution's impact on performance and security and confirm its practicality regarding added performance and resource costs. Furthermore, we discuss how the approach impacts organizations by considering the roles of operations, security, and development teams.

1.4 Structure

This work is organized as follows. Section 2 reviews the background needed to understand our approach and implementation, alongside related work and our gap analysis. Section 3 explains threats related to vulnerability posture found in the current state of practice, and lays out the requirements for solving the problem. Section 4 provides an overview of a proposed architecture for mitigating said threats, and Section 5 details its implementation using well-known tools. In Section 6, we evaluate our solution regarding performance and security. Section 7 explains the threats to the research's validity as well as employed mitigations, and finally, Section 8 concludes our work with some final considerations and future work directions.

Chapter 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 is the adoption of DevSecOps best practices. These practices can include using security frameworks such as SLSA (Supply-chain Levels for Software Artifacts). Security frameworks are guidelines for achieving security guarantees on a software supply chain. SLSA in particular specifies incremental levels for artifact security to attain better security guarantees, such as hardened builds and non-forgeable provanance [8]. 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 also often exploit vulnerabilities present in code when the product is delivered. That means that even if the CI/CD pipeline itself is secure, the same may not be true about the source code of a project. Vulnerabilities can be found directly in source 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 [34]. Such vulnerabilities can come from direct and transitive dependencies (i.e., dependencies included in dependencies recursively).

Security advisories like NIST and GHSA (GitHub Security Advisory) disclose vulnerabilities in the form of 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) [25], which is managed by NIST and kept up to date with CVEs from multiple advisories. A CVE describes, among other details, affected versions of software. 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 with the product.

SCA tools can automate this process and search these databases for vulnerabilities in a project's dependencies. Examples of tools are *Trivy* from AquaSecurity or *Snyk*, both able to search on popular dependency files like the Maven POM (Project Object Model) or NPM `package.json`, depending on the language and package manager used. As a powerful alternative, they can ingest SBOMs for a transparent inventory of dependencies, regardless of which technology the project was developed upon. Another example is OWASP's (Open Web Application Security Project) *Dependency Track*, which 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 only helpful while 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, artifact signing to demonstrate provenance can help discriminate a fake or tampered artifact from a legitimate one.

A popular option for signing, verifying, and attaching artifacts to a container image is *Cosign* [9], from the *Sigstore* framework. Attached artifacts can then be stored alongside the product's image in an OCI registry in the form of *in-toto* attestation, a fixed, lightweight format to describe supply chain metadata, including SBOMs [27]. 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 [9]. By using the *Sigstore* framework or similar tools as a support for artifact trustworthiness, an operator can reliably apply SCA to a supply chain.

After acquiring knowledge of vulnerabilities, teams must remediate them in order to keep the software secure. However, vulnerability remediation can be a high-effort task. As shown in a study made by Kenna Security and Cyentia [19], hundreds of companies remediated only a monthly rate of 15.5% of known vulnerabilities on average, while a quarter had an even lower rate of 6.6%. To assist with prioritization, there are well-used scoring systems that help estimate how vulnerable the current state of a product is. CVSS (Common Vulnerability Scoring System) is a very dependable and robust method used to describe the severity of a CVE (i.e., how critical it is) [25].

CVSS estimates how vulnerable an application is based 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 $[9.0, 10.0]$. By using CVSS, teams can choose to remediate the most critical vulnerabilities, and might even compromise by disregarding CVEs with low severity.

Another way to describe the potential impact of a vulnerability is to use EPSS (Exploit Prediction Scoring System) [24], which expresses the likelihood between 0.0 and 1.0 of a CVE being exploited within the next 30 days. This metric is updated daily for every public CVE reported, and is maintained by FIRST [3] (Forum of Incident Response and Security Teams). Therefore, unlike CVSS, EPSS does not measure severity, but risk. It is not interchangeable with CVSS, but is also widely used, and can help further prioritize CVEs as well as define remediation for low risk ones.

Finally, to standardize a developer's stance towards known vulnerabilities, CISA [6] (Cybersecurity and Infrastructure Security Agency) specified VEX (Vulnerability Exploitability eXchange) [17]. This format allows a formal statement about vulnerabilities and is readable by both machines and humans. VEX is useful for improving transparency and formally ignoring non-applicable vulnerabilities. For instance, a vulnerability that comes from a library could have high EPSS and CVSS values, but the vulnerable code might be unreachable in the context of a specific application, making the threat harmless. The company can then use VEX as a means to declare the vulnerability and issue a `NOT AFFECTED` status. As another example, if a medium EPSS vulnerability could affect the product, the company can declare that it will be fixed in the next patch cycle using an `AFFECTED` status. This makes VEX a powerful format to enhance transparency and further improve trustworthiness between stakeholders.

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 [26]. 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. Because expensive auditing is needed to provide evidence of fulfilling compliance, vendors strive to keep internal compliance, often manually, to avoid failing an audit and needing to request another one [26].

Upholding compliance continuously has thus become relevant for DevSecOps [28]. As described by Ramaj et al. [28], works related to continuous compliance seek to automate general security activities, like SCA, for compliance assessment. For instance, compliance-specific tools (like *OpenScap*, *UpGuard1* and CIS-CAT) are discussed as useful to verify if a software conforms to specific regimes, while non-specific tools are pointed as helpful when assessing general compliance requirements common to many regimes.

There is also the notion of CaC (Compliance as Code), which proposes to formalize security controls into code so they can be validated pragmatically [28]. Some approaches include integrating common compliance controls into automated testing [13; 26], and others aim to parameterize controls so that compliance can be checked in a data-driven testing architecture [32]. 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 [35].

Although promising, these solutions fall into what Nygard called "pipeline compliance" [1]: 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 allows for the pipelines to be separated based on each component, and different teams could specialize in building secure components for a secure ecosystem. This, however, introduces the threat of a previously compliant component violating one or more compliance controls and then compromising 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 no confusion of ownership between developers and security officers. Note that this method does not necessarily replace "composite compliance", and could also be distributed if needed.

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. Causes for this include the discovery of a new vulnerability that exceeds severity tolerance or even risk recalculation for an old one due to recent exploits. 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 absence of consideration for new vulnerabilities leaves a gap within the state of the art. If continuous compliance cannot address the possibility of sudden compliance violations, then the responsibility of mitigating them falls to the people involved in the compliance process. Whether they are the developers providing the now vulnerable images, the security team defining compliance, or operators managing the environment is unclear, which rekindles the aforementioned ownership problem.

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 [33]. NIST (National Institute of Standards and Technology) 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. [16] 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 [15]. In orchestrated, containerized environments such as Kubernetes, mTLS can be empowered by Service Meshes like *Istio* to remove overhead from applications and secure traffic between microservices [20]. Trust in ZTA is also never static, but rather dynamic [15]. This results in needing to authenticate a component before establishing trust, and since this trust is not static, authentication must be continuous, which is a strong pillar of ZTA [23].

X.509 certificates can be provided traditionally, by manual procedure or using a tool, which amounts to creating a CSR (Certificate Signing Request) and having a single CA (Certificate Authority) or the chain of CAs sign it. In this traditional process, this process must be repeated in order to rotate the certificates before they expire. If they do expire, the TLS handshake will fail at some point, breaking the communication.

To help operationalize this, as well as avoid other security issues like credential theft, secretless authentication allows for an intermediary tool to manage the secret and its rotations and deliver it to the application. The intermediary tool identifies the application and issues the most up-to-date secrets it needs. Identity providers like *Kerberos* [11] and *WS02 Identity Server* [12] do that by granting identities to applications, and embedding secrets in those identities. Said secrets can take the form of X.509 certificates to power mTLS and secure communication between identified parties. In this approach, a client or application must perform authentication, in which the identity provider will return the appropriate session key. After this session key expires, the application or client must once again request it in order to keep its identity.

There is also SPIFFE [14], an identity provisioning specification that proposes that the 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). The SVID contains signatures to prove that its SPIFFE ID is valid. Currently, the standard defines JWT and X.509 as valid SVID formats[1]. SVID in the X.509 format is very similar to identities from other identity providers and can be used as certificates to power mTLS. The SPIFFE standard is most famously implemented by SPIRE, a tool that graduated from CNCF (Cloud Native Computing Foundation) 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 [10].

SPIRE differs from other tools because it requires a secret-less authentication process before issuing an identity, and uses short-lived keys for these identities. In SPIRE, this authentication is known as *attestation*. Whenever an identity is defined in SPIRE, it must contain a set of selectors. Selectors are the properties that need to be satisfied by the workload that is requesting the identity. During attestation, SPIRE collects selectors from the workload and compares them with each entry in its database. If a workload with those selectors is eligible for some identity, that identity is granted to it embedded in the SVID of the requested type (most often, a set composed of a X.509 certificate, the associated private key, and the set of CAs needed for validation).

SPIRE has two main components to support this attestation-based selective provisioning: the SPIRE Server and the SPIRE Agent. The server acts as the one source of truth for the Trust Domain. Considering a distributed cloud architecture of many nodes or instances, a SPIRE Agent represents a node. An Agent and, therefore, a node is not trustworthy by default. Before it can be trusted, it is subjected to an attestation, which can consider different properties of a node, such as where it is running and properties from its software or hardware stack. If successful, the Server issues the Agent's identity, enabling it to attest its local workloads (i.e., services running on the node). Afterward, workloads can try to attest with

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

the Agent to receive their identity. The SVIDs for each workload and Agent are typically short-lived and can be rotated easily, implying a reattestation. Hence, if a workload loses mTLS clearance because its SVID expired (depending on the TLS implementation), that can only mean it could not reattest and, therefore, is rightfully isolated in the network.

As a result of its selective identity provisioning, SPIRE also supports another principle 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. [15] 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. [23], 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 [18]. This research gap appears, therefore, to be a missing link for connecting ZTA to continuous compliance.

Chapter 3

Threat model and requirements

Here we explain our assumptions for an environment where our approach of integrating ZTA to continuous compliance applies. After that, we detail threats that can be found in such a workflow that uses the current state of practice for continuous compliance and ZTA and find requirements to evaluate a solution to the problem.

3.1 Assumptions

We assume a typical environment where developers and operators develop and maintain cloud-native applications. Following the DevOps (or DevSecOps) movement, developers and operators try to cooperate but are not specialists in each other's work. That means that while there is collaboration, responsibilities are defined well enough so there are no ownership issues. Therefore, being responsible for the system's long-term operation, operators have to 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.

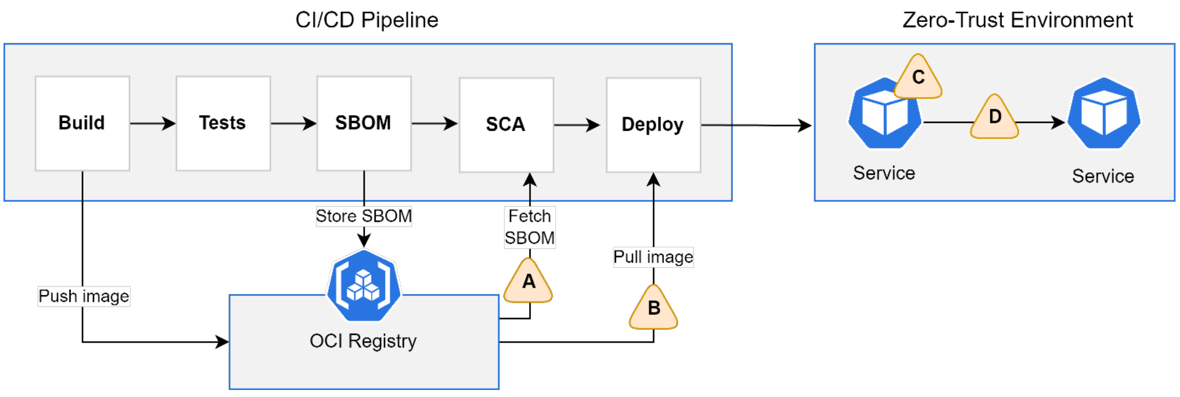
We also assume developers are mostly benign but may not have the necessary knowledge to build services that are secure in the long term. For this reason, they may use libraries and modules that are not mature enough. However, we assume that the tests used in the CI/CD pipeline follow best practices and include security tests. 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. Additionally, companies interested in maintaining compliance can employ discussed approaches when building their pipelines. Consequently, we assume that there are no known vulnerabilities in the application at the initial state of a release, and thus, it is compliant.

Finally, we assume that ZTA is implemented in an effort to protect the environment with dynamic trust assessment. Therefore, all services have unique identities, and their communication is authenticated. As a result, if a service does not have a valid identity, it will be unable to communicate with other services and will be effectively isolated. Isolating a service should trigger termination (e.g., due to failing health checks) and trigger alerts in monitoring tools.

3.2 Considered threats

Due to these assumptions, we are concerned with threats to compliance after deployment, specifically regarding changes in dynamic aspects. In this work, we consider only vulnerability posture as a dynamic aspect because of its direct impact on security. Figure 3.1 illustrates these assumptions and indicates our considered threat points.

Figure 3.1: Assumed workflow and considered threats



Firstly, it is possible that a third-party attacker can intercept the release and switch it with a tampered image if the OCI container registry storing it is not safe enough. Although we do not concern ourselves with the registry integrity, the image's provenance and its SBOM are a concern. If the provenance of either artifact is compromised, there would not be a way of reliably assessing vulnerability. That means a more sophisticated attacker who knows a vulnerable release will fail the pipeline can even try to switch the original SBOM with one that masks the vulnerable state in a way that will be accepted. This can be summarized as ***Threat A*** *– use of a compromised image* and ***Threat B*** *– use of a compromised SBOM*.

Moreover, as discussed before, an initially compliant release can later violate compliance on account of dynamic aspects. A new vulnerability could be exploited by an attacker at any time, and the likelihood of this happening will increase as long as the release remains available. If incident response is manual, this means an attacker could benefit from a delay in incident response. This creates ***Threat C*** *– exploit of a newfound vulnerability*.

Lastly, if a service is compromised (e.g., because of an exploit), it could compromise other services through communication. Although ZTA is implemented, since vulnerability posture is not taken into consideration for assessing trust, then the compromised service is trusted. This means the final ***Threat D*** *– compromise propagation*, is not correctly mitigated.

3.3 Requirements for a solution

Considering these assumptions and threats, the solution must provide a way to identify if compliance is upheld. If, at some point, compliance is violated due to new vulnerabilities, the solution must breach the gap left by continuous compliance strategies and apply an automatic incident response. This 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. This requires mapping compliance into policy, also breaching the gap left by current ZTA research.

Such a solution should not only be able to mitigate listed threats but also allow operators to work with developers and compliance officers to decide the best way to remediate the sudden unavailability of certain services.

Chapter 4

Enforcing continuous compliance

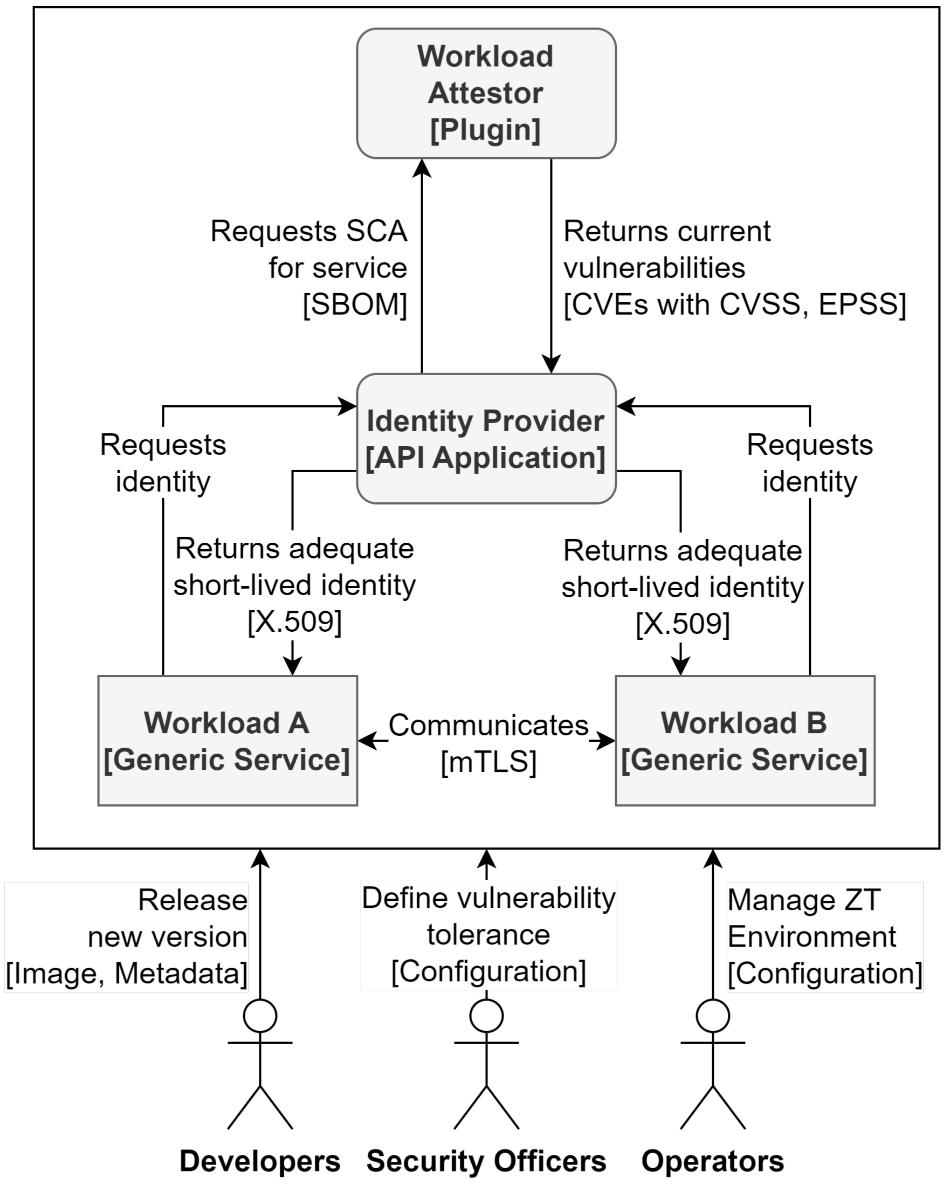
Compliance must be present during the entire life cycle of the application in order to keep it compliant. As mentioned before, this can be made possible by adopting DevSecOps best practices, SCA, and a supply chain security framework during a CI/CD pipeline, as well as employing continuous compliance. By implementing these known practices, a company can enforce an initial state of compliance by making sure the static part of a release (its build and attached artifacts) follows compliance rules. SLSA-compliant build processes further help with this and, in its level 3, guarantee artifact provenance and hardened builds [8].

However, to continually ensure compliance, it is necessary to continue to perform SCA, even after deployment or delivery. To operationalize this, we propose integrating workload attestation using vulnerability tolerance as a minimum security posture criterion for selective identity provisioning. Figure 4.1 illustrates our architecture.

We propose a workload attestor that can perform SCA based on the same parameters as previously performed during the pipeline. The yielded results should be the same. That means that if the workload keeps its initial compliant state, it must also satisfy the vulnerability tolerance criterion, and thus should be eligible for an adequate identity. If the results have changed even with the same static parameters, than changes in dynamic properties were detected and captured during SCA. For our architecture, we consider the SBOM as the source of truth for this operation.

The workload attestor should work as an extension (or plugin) to an identity provider, where the attestor is responsible for returning the current vulnerability state and the provider decides whether or not the state is acceptable. This architecture is agnostic to how the com-

Figure 4.1: Proposed architecture


Workload Attestor [Plugin]
Requests SCA for service [SBOM]
Returns current vulnerabilities [CVEs with CVSS, EPSS]
Identity Provider [API Application]
Requests identity
Requests identity
Returns adequate short-lived identity [X.509]
Returns adequate short-lived identity [X.509]
Workload A [Generic Service]
Communicates [mTLS]
Workload B [Generic Service]
Release new version [Image, Metadata]
Define vulnerability tolerance [Configuration]
Manage ZT Environment [Configuration]
Developers
Security Officers
Operators

munication between both components happens, and which protocols to employ. As long as the identity provider can request the workload attestor to perform SCA based on an SBOM, in a synchronous manner, and the attestor can respond to the found vulnerability state, the functionality is successfully implemented. Said state should be measured by CVSS and EPSS (which are attached to CVEs) due to them being the prevalent vulnerability scoring systems. The identity provider should be able to check if the results satisfy the criteria for any identity, and if so, return an X.509 certificate that represents it.

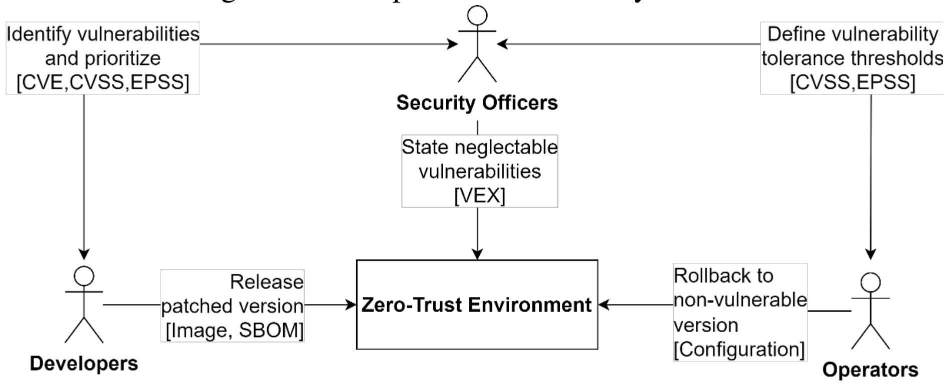
By receiving the certificates, contemplated applications can use mTLS communication to leverage Zero Trust further by imposing communication security. How mTLS is implemented in the environment is not in the scope of our proposal, but Service Meshes like *Istio* could remove the overhead of implementing mTLS between microservices [20]. Regardless, if such identities are short-lived, periodic reattestation is a natural necessity in order to renew them. Given that the mean of new CVE reports per day in 2023 for NVD is 79.18, which amounts to around 3.29 per hour [22], reattesting more than once per hour increases the chance of denying a workload its identity as soon as it is no longer compliant. In time, the workload will lose communication privileges and be isolated, creating an availability incident.

This workflow assumes the authentication between any given workload and the identity provider is secretless. That is, the workload must not contain the certificate beforehand and must only come into its possession through the identity provider. The reasons for that are: 1) the identity provider can be the sole source of certificates and thus the root of trust in the Zero-Trust environment, and 2) always having short-lived identities represents that the workload continues compliant with vulnerability tolerance rules.

If secrets were within the workload in some way, they would need to be long-lived, which would mean that continued compliance would not be a guarantee. If secrets were given to the workload, but through another component, then for the secrets to be trusted their source would need to be assessed, and there would not be a single root of trust. Additionally, the extra component would challenge the role of the identity provider. Either both components would need to co-exist, adding unnecessary complexity to achieve the same results, or the identity provider would be entirely replaced by this other component. Replacing the identity provider would mean that the extra component needs to manage the secrets since would become the root of trust. To make the secrets short-lived, it would need to manage them for the workloads, basically returning to secretless authentication.

Considering this architecture, the post-deployment process also follows Nygard's proposal for separation of responsibilities on "point of change compliance" [1]. That is, security officers keep working together to define compliance criteria as before, as well as developers continue in charge of fulfilling compliance, but now operators are responsible for ensuring that both the pipeline and the attestor can perform SCA on the same terms by configuring the Zero-Trust environment.

Figure 4.2: Response to availability incident


Identify vulnerabilities and prioritize [CVE,CVSS,EPSS]
Security Officers
Define vulnerability tolerance thresholds [CVSS,EPSS]
State neglectable vulnerabilities [VEX]
Developers
Release patched version [Image, SBOM]
Zero-Trust Environment
Rollback to non-vulnerable version [Configuration]
Operators

As a consequence of this architecture, in the case of an availability incident caused by compliance violation, each role may only act accordingly, as shown in Figure 4.2. The security officers could issue a VEX so that SCA ignores some vulnerabilities, developers can patch vulnerabilities and release new versions to be attested, and finally, operators are able to roll back to previous non-vulnerable versions of the service. Since isolation is a response to violation, the maximum CVSS/EPSS requirements for issuing an identity should be defined by security officers considering the trade-off of availability *versus* exploitation prevention.

Chapter 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*, an OWASP tool, 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. Other mentioned tools, such as *Trivy* and *Snyk* were considered, but since they lack these features, using them instead of Dependency Track would require more complexity as additional integration to other APIs would be needed to yield similar results.

For the identity provider, we chose SPIRE due to its openness, flexibility, and native workload-attesting capabilities. In addition, it is already used to implement ZTA's communication security and continuous authentication principles. We nevertheless expand SPIRE in regards 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.

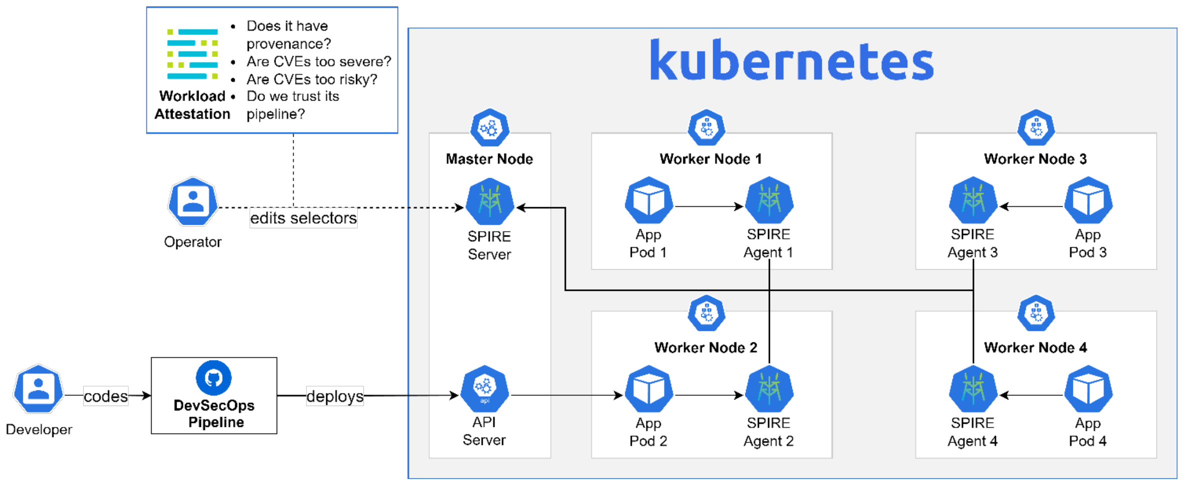
Two preconditions are necessary for the plugin features: (1) it needs to have access to the image information (i.e., its complete identification) to tell which workload is attesting, and (2) it must also have access to that image's SBOM. With this information, it can call *Dependency Track* to feed it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

As illustrated by Figure 5.1, developers and operators can collaborate independently on the attestation process. Developers input the code into a DevSecOps pipeline that deploys an initially compliant image into Kubernetes as a Pod, which is the logical group of containers and the smallest and deployable units of computing Kubernetes manages[1]. After the Pod starts, it needs to attest itself to a SPIRE Agent before acquiring its identity. The SPIRE Agent is already attested to the SPIRE Server, and therefore can issue identities and knows the required selectors for each one. Those selectors are defined by an operator that manages SPIRE, and should implement the rules and thresholds specified by the security officer.

Similar approaches for satisfying our two preconditions could be implemented in other contexts. For example, if we assume that microservices run in micro-VMs orchestrated by a system such as OpenStack[2], the image information could be retrieved from the image service and the SBOM embedded in the image metadata.

Figure 5.1: Proposal of compliance attestation within SPIRE



[1]https://kubernetes.io/docs/concepts/workloads/pods/
[2]https://www.openstack.org/

5.1 Embedding the basic information

To ensure the product possesses an SBOM describing it, its build pipeline should contain a step that generates and saves this artifact before 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 where the product images are kept. The latter is efficient 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 in 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 pipeline platforms with at least SLSA level 2 or, preferably, level 3 guarantees. SLSA level 2 means that the tool provides signed provenance evidence that the image was built on that pipeline. This does not require the platform to protect secret material used to sign the provenance, and thus the provenance could be forged. In addition to this, SLSA level 2 does not include integrity guarantees for the build platform, which means that there is no mitigation against tampering. This is why SLSA level 3 is preferable, as it requires the platform to be hardened against integrity attacks and prevent provenance secrets from being accessible outside of signing procedures, which in turn means provenance is non-forgeable [8].

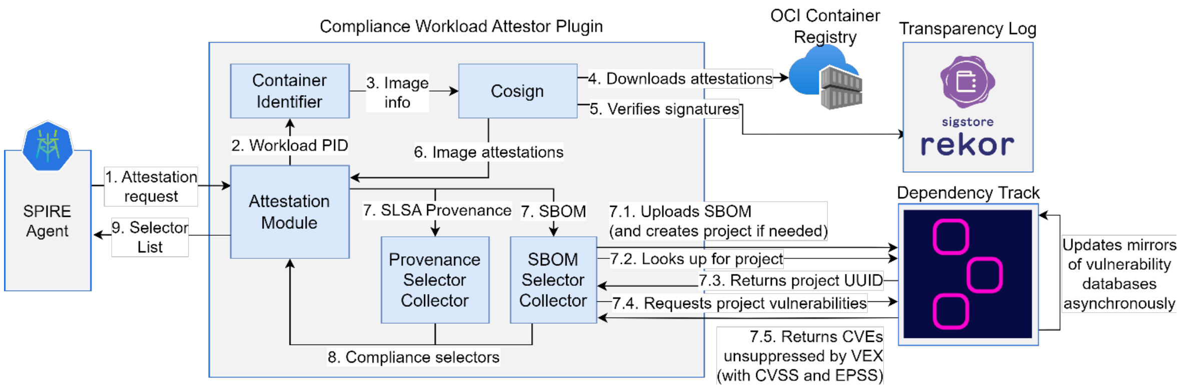
If available, the workload attestor plugin will use SLSA Provenance to report the image's origins in addition to vulnerability data. This allows administrators to restrict the origin of their images (e.g., the CI pipeline that produced it) and the source code repository and branch used.

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 5.2 illustrates the workflow for the plugin.

Figure 5.2: Compliance workload attestor plugin


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

When it starts, the plugin immediately collects information about the running image. For our scope of this work, it queries the Kubernetes API regarding the Pod and its 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 application in *Dependency Track* if there is no entry for this image version using digest 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. Using these selectors should help an operator define vulnerability tolerance.

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. These selectors should be applied to guarantee that SPIRE considers only a specific origin for the image.

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 selector about them will be built, and thus, no identity that requires such selectors will be issued. Table 5.1 lists all available selectors for the compliance 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 in 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`
`contains-cvss-severities`	The list of CVSS severities tolerated for the application	`LOW` or `LOW,MEDIUM` or `LOW,MEDIUM,HIGH` or `LOW,MEDIUM,HIGH,CRITICAL`
`contains-epss-risks`	The list of EPSS risks tolerated for the application	`LOW` or `LOW,MEDIUM` or `LOW,MEDIUM,HIGH` or `LOW,MEDIUM,HIGH,CRITICAL`

Table 5.1: Selectors for the workload attestor plugin

Listing 5.1 illustrates how to register an entry for an identity to the SPIRE Server using the defined selectors. The URI for the identity is defined in the snippet as `-spiffeID`, where `example.org` is the Trust Domain and `example-service/main/severity-high/risk-high` is the full identity name. As for the selectors, they are defined by using the `-selector` argument, and all of them are prefixed by `cc`, which signalizes for SPIRE that this selector comes from the continuous compliance plugin. Via these selectors, this entry imposes that `example-service` should only tolerate CVEs if they are not `CRITICAL`, both in severity (CVSS) or in 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 in another platform, or if employing reusable pipelines.

Listing 5.1: SPIRE entry creation example

```
spire-server entry create \
    -spiffeID spiffe://example.org/example-service/main/severity-high/
       risk-high \
    -parentID spiffe://example.org/ns/spire/sa/spire-agent \
    -selector cc:has-sbom:true \
    -selector cc:contains-epss-risks:LOW,MEDIUM,HIGH \
    -selector cc:contains-cvss-severities:LOW,MEDIUM,HIGH \
    -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
```

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 [4], to fetch SVIDs and configure non-SPIRE-aware workloads to use them.

It is important to note the format of both `contains-cvss-severities` and `contains-epss-risks` selectors. 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 does.

The reasoning behind it being a comma-separated string 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 comma-separated strings are a workaround for this limitation. If a `HIGH` severity is found, the attestor returns that all preceding levels were achieved (`LOW,MEDIUM,HIGH`). To make this viable to EPSS, we needed to map the values on a similar scale and then provide classes the same way. Without an official definition for EPSS risk classes, we allow every organization to configure the intervals for each class in plugin settings.

Chapter 6

Evaluation

Because SPIRE is a graduate project at CNCF, it is already considered a stable, production-ready system [5]. Therefore, instead of evaluating the use of SPIRE, we chose to focus on the plugin itself. Additionally, given that CVEs and their scoring systems CVSS and EPSS are well established in the state of the art as sources of vulnerability intelligence, we also do not include performance analysis on these systems.

We separate our evaluation into two aspects: (1) a quantitative analysis of performance, to evaluate the impact of our plugin and supporting architecture on a running SPIRE environment; and (2) a qualitative security analysis to verify how the established threats are mitigated by adopting our plugin.

6.1 Performance costs

To consider the performance cost, we should first consider the necessary resource allocation for running *Dependency Track*, which is responsible for analyzing the SBOM. As per official guidance, the sum of the *Dependency Track* Docker containers requires a minimum of $4.5\ GiB$ of memory and 2 CPU cores, and a recommended allocation of $16\ GiB$ for memory and 4 CPU cores [31]. In a production environment concerned with compliance, a vulnerability assessment tool should be used. Hence, these resources should be a necessary expense.

We run this tool on top of Kubernetes and measure the resources used through a controlled workload to evaluate the adequacy of these recommendations to various degrees of stress. Our specific cluster had 2 Worker Nodes, both running SPIRE Agents, while the SPIRE Server ran on the Master Node. One of them hosted *Dependency Track*, and the other hosted a dummy application to try and attest to SPIRE. Because we wanted to test the recommended resources, we allocated $16\ GiB$ of RAM and 4 CPU cores for each Node. Other cluster configurations could apply to the context, as long as *Dependency Track* is isolated on one Node to avoid sharing resources, and the allocated RAM and CPU for this specific node is greater or equal to the recommended amount.

Apart from the use of resources, because SCA runs entirely on *Dependency Track* and is completely parallel to the attestation process, the only possible 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 one hour of longevity by default, reattestation will usually be made every 30 minutes. As mentioned before, because the mean of new CVEs per hour in 2023 for NVD is 3.29 [22], checking for new vulnerabilities frequently is advised. Therefore, to test the solution with a realistic workload, we imposed the following levels of demand to *Dependency Track*: 1 request per second, 50 requests per second, 100 requests per second, and 250 requests per second. This means achieving the lowest considered load would require $1\,800$ workloads renewing their identities (and checking for vulnerabilities). Similarly, to achieve the highest workload case, $900\,000$ workloads would be needed.

The requests to the *Dependency Track* API were made to check on an application whose SBOM contained over 800 dependencies, above the average of 526 [34] per application. The chosen SBOM is among the examples in the CycloneDX GitHub repository[1]. Each experiment had 30 iterations that lasted 20 minutes and were executed in a Kubernetes cluster within a private, on-premises OpenStack infrastructure. In our case, this meant that one Node is a Virtual Machine provided by Nova[2]. For each experiment, we monitored resource usage

[1]`https://github.com/CycloneDX/bom-examples/blob/master/SBOM/keycloak-10.0.2/bom.xml`

[2]`https://docs.openstack.org/nova/latest/`

for the *Dependency Track* REST API container within the Kubernetes Pod using *Cadvisor*[3], a sidecar (i.e., a secondary container within a Kubernetes Pod) that analyzes container metrics and exposes them. The metrics were collected by a local *Prometheus*[4] and exported for analysis in the granularity of 20 seconds per data point. Metric collection ignored *Dependency Track*'s front-end container counterpart due to not being relevant for the experiments.

For each experiment, we normalized the timestamp for each measurement as the number of minutes elapsed since the start of the experiment. We grouped the measurements by experiment and minute timestamp and took the mean of each measurement in order to analyze the general resource usage over time.

Figure 6.1: *Dependency Track* CPU usage per request rate

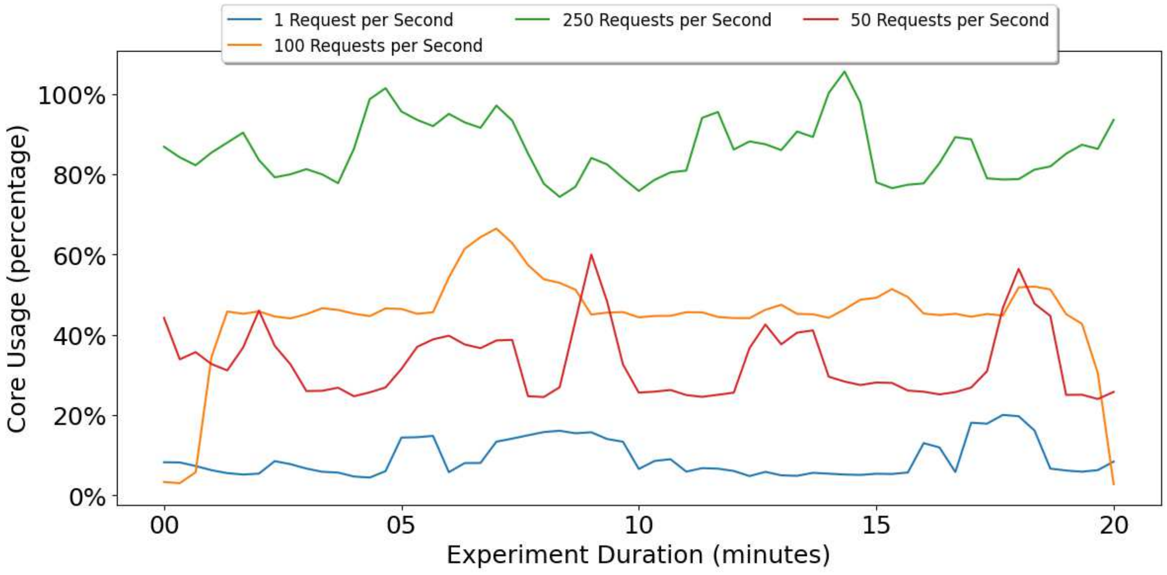
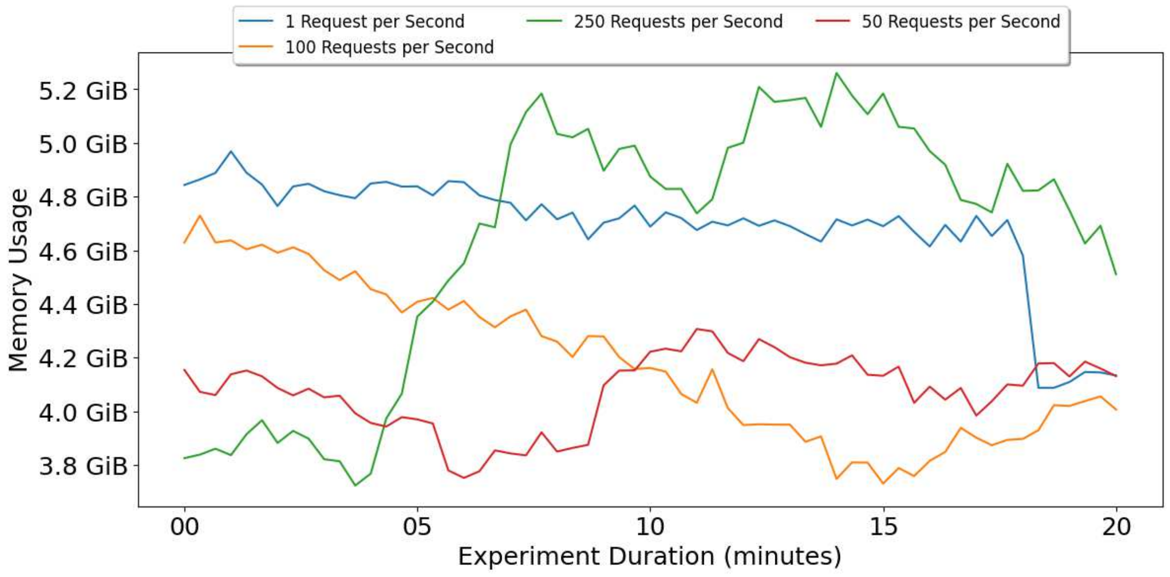


Figure 6.1 illustrates the mean CPU usage for each experiment over its duration. We can see that although processing time changes with higher demands, it remains well under the required 4 CPU cores, as observed peaks reach just above 100% of one core's worth of time. Figure 6.2 exhibits a different behavior regarding memory usage. While it is true that even the worst load stays well below the 16 GiB recommendation, it is not as stable. This correlates to *Dependency Track* updating its database mirrors periodically, parallel to synchronous requests. As a result of this independent update, *Dependency Track* appears to

[3]`https://github.com/google/cadvisor`
[4]`https://prometheus.io/`

Figure 6.2: *Dependency Track* memory usage per request rate



withstand both sudden peaks in demand, as well as high stable demands.

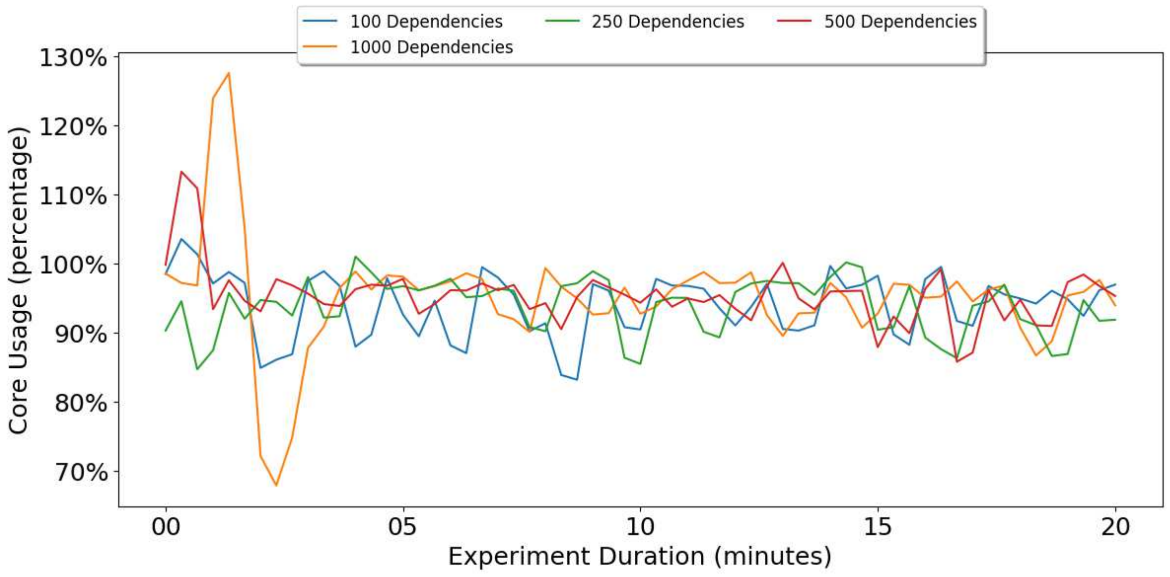
To verify if resource use changes with the number of dependencies to track, rather than how many requests are made, we further experimented on escalating numbers of dependencies: 100 dependencies, 250 dependencies, 500 dependencies, and finally 1000 dependencies. For each amount, we used different SBOMs based on the SBOMs publicly available at CycloneDX's GitHub repository[5].

Figure 6.3 shows an unstable usage of CPU, which is very different than shown in Figure 6.1. Firstly, the number of dependencies does not seem to impact much, except at the beginning. At the start of every experiment, there is a peak that seems proportional to the number of dependencies. After that, CPU usage fluctuates, but in general, stays below the initial peak. The same can be said when accounting for memory usage, as displayed by Figure 6.4. But in the case of memory, there seems to be a similar behavior as in Figure 6.2, in that after each peak there is a gradual release of memory.

In general, this consistent initial peak suggests that the most resource-intensive phase is also the least frequent one. That is, this peak will be as infrequent as service deployment since the number of dependencies should change only once in the lifetime of a service version. Additionally, the number of dependencies is not necessarily linear to the number of

[5]`https://github.com/CycloneDX/bom-examples/tree/master/SBOM`

Figure 6.3: *Dependency Track* CPU usage per number of dependencies



services, since in a scenario where applications use the same development stack they also share many dependencies. Therefore, the worst-case scenario for resource usage on *Dependency Track* happens when multiple unique services, with many dependencies and different stacks, are executed in the same environment.

Even in the explored worst-case scenarios, resource usage stays below the official recommendation, allocating a little under 7 GiB of RAM and using 2 CPU cores worth of time. These requirements can be satisfied with a large general-purpose instance from a public cloud vendor, such as AWS with `t2.large`[6] or DigitalOcean with one of their Basic Droplets[7]

6.1.2 Impact on latency

Regarding added latency, there are two major points of interest. The first is latency due to *Sigstore* related requests, and the second is due to *Dependency Track* related requests. While the latter are requests to only one component, the former are comprised of 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 public default 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*

[6]`https://aws.amazon.com/pt/ec2/instance-types/`

[7]`https://www.digitalocean.com/pricing/droplets`

Figure 6.4: *Dependency Track* memory usage per number of dependencies

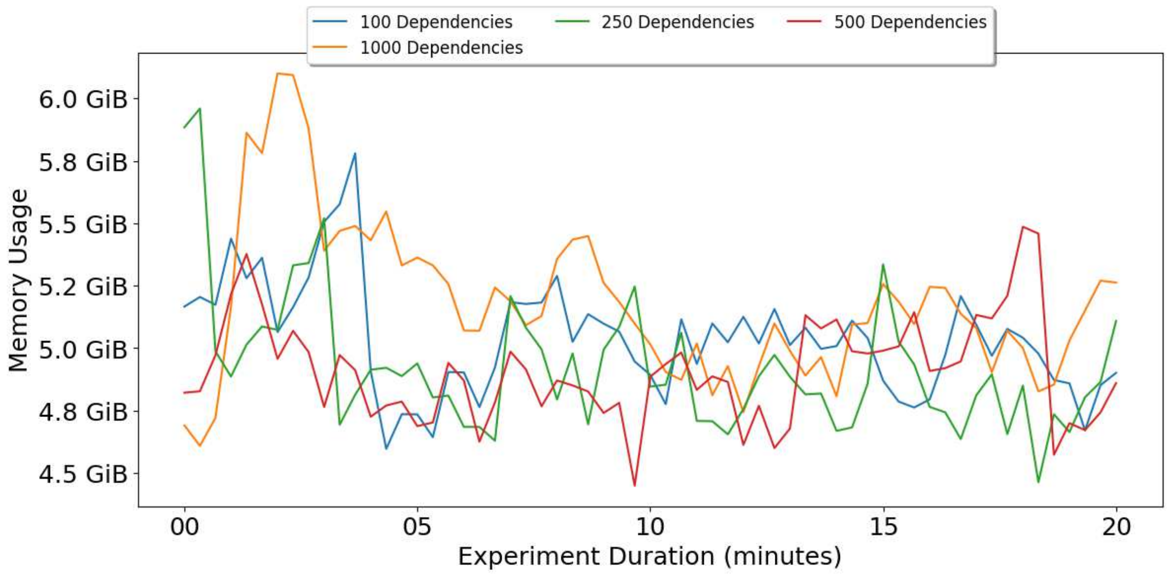


Figure 6.5: Distribution of latency in *Sigstore* requests

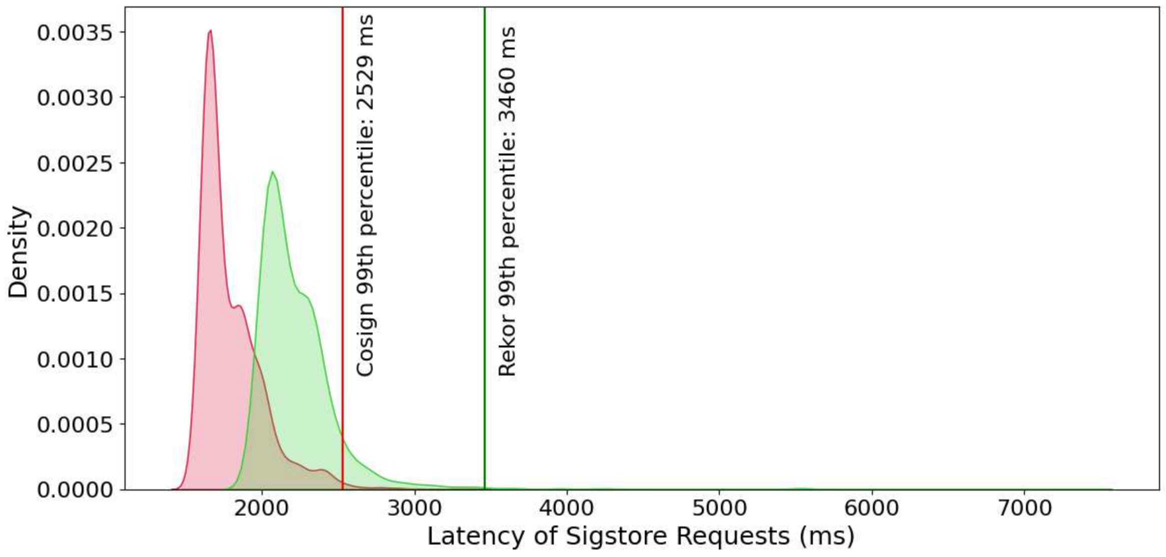
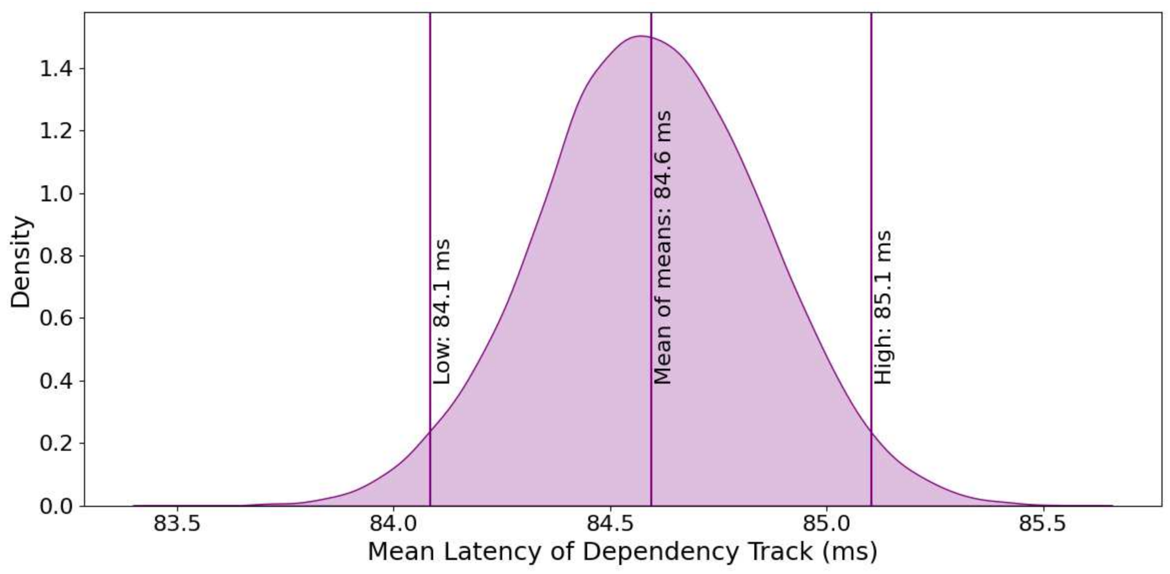


Figure 6.6: Distribution of latency in *Dependency Track* requests



requests, for each attestation. The workload we used to attest was a simple application called *SPIFFE Helper*[8], a simple utility that can be used as a sidecar to communicate to SPIRE in case the main application is not SPIRE-aware. In our case, we used it as a standalone service, and only to trigger attestations.

Figure 6.5 shows the latency distribution for each type of *Sigstore* component. Although the distributions are highly skewed, we can see by their 99^{th} percentiles that usually the attestations are 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 in an attestation. As for *Dependency Track*, since the distribution is log-normal, we bootstraped the MLE (Maximum Likelihood Estimation) of the mean of latency on $5\,000$ re-samples. The resulting mean of means, as shown in Figure 6.6, is 84.6 ms, with a Confidence Interval of $[84.1\ ms, 85.1\ ms]$ for a confidence level of 95%.

When adding the impact of both *Dependency Track* and *Cosign* in latency, the total additional processing time per attestation is less than 6 seconds in most cases. This latency does not significantly affect an attestation attempt, since they by default are tried twice every hour.

[8]`https://github.com/spiffe/spiffe-helper`

6.2 Security evaluation

To evaluate the security aspect of the proposed plugin, we review the workflow for workload attestation and check how it mitigates threats defined in Section 3.

Firstly, due to periodic reattestation through SPIRE, workloads are continuously being analyzed by workload attestor plugins. By adding our vulnerability compliance plugin, this periodic reattestation performs SCA and, because of *Dependency Track's* self-update, the plugin always returns the current vulnerability posture from the workload. Therefore, changes in this posture are tracked at each reattestation, extending continuous compliance beyond when the deployment happened.

Because vulnerability posture is mapped as a policy via SPIRE selectors, changes in this posture cause identities to no longer be issued. Consequently, mTLS connections to the non-compliant workload will cease after the previous certificate expires. This effectively implements compliance as a policy for trust assessment, helping to fill the gap we found in ZTA's treatment of compliance while applying automatic incident response.

Since we assume that isolated services are terminated, availability is damaged in exchange for preventing exploitation. This mitigates ***Threat C*** *– exploit of a newfound vulnerability*. Even if termination is not triggered by health checks or similar mechanisms, isolation will mitigate ***Threat D*** *– compromise propagation*.

The failed workload is free to retry attestation *ad infinitum*, but the only way the result can be changed is through outside forces, such as databases recalculating CVSS and/or EPSS values, or introduction of VEX into *Dependency Track*. In both cases, the workload has become compliant again because either its vulnerability state changed or it has been manually attenuated due to internal officers' intervention.

Additionally, since the SBOM is the source of truth for vulnerability assessment, its provenance is also considered. In case of 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. And due to SLSA 3 guaranteeing that forging the provenance is beyond the capabilities of most adversaries, we can avoid ***Threat B*** *– use of compromised SBOM*. Furthermore, as a result of SLSA Provenance also representing the origin of the image, ***Threat A*** *– use of compromised image* is also mitigated.

As a result of fulfilling these requirements, the threats pointed out in Section 3 are also mitigated. Table 6.1 encapsulates the discussed mitigations for each threat.

The consequence of an automated isolation of non-compliant service is, of course, denial of service. This can aggravate the applicability of our solution for non-critical applications. In such

Threat	**Cause of mitigation**
Threat A: *use of compromised image*	Employing non-forgeable provenance through SLSA 3
Threat B: *use of compromised SBOM*	Employing non-forgeable provenance through SLSA 3
Threat C: *exploit of newfound vulnerability*	Reattestation failure and automated termination of vulnerable service
Threat D: *propagation of compromise*	Reattestation failure and isolation of communication

Table 6.1: Threat and mitigation strategies

scenarios, a new CVE may be considered a threat to compliance, but availability might be more important. On the one hand, the security team could increase vulnerability tolerance to make attestation possible, but this would be a bad practice. On the other hand, developers could issue a VEX stating that the new CVE does not affect the product, but this would not be true. For such situations, the developers or operators could issue a VEX indicating that the vulnerability has been considered and a fix will be provided, but this does not impact the attestor analysis.

A solution for this problem would be adding a configurable grace period for new vulnerabilities, either based on severity (or risk) or based on statements within VEX. This would create an exploitation prevention *versus* availability trade-off that would facilitate the solution's applicability and give more time for developers to patch new CVEs before service isolation. However, this is not part of this work's scope, and extending the solution to support non-critical applications could be a future work direction.

In summary, the implemented plugin, through the help of *Sigstore* and *Dependency Track*, can map selective identity provisioning with vulnerability posture rules for compliance. This effectively isolates non-compliant services, even if they were previously considered compliant at some point, and prevents threat exploits as soon as possible without human intervention.

Chapter 7

Threats to validity

There are some aspects of the research that need to be highlighted due to the impact on its validity. Firstly, this research was conducted with support from HPE (Hewlett Packard Enterprise) Brazil, which are potential users for the results yielded from it. Through periodic meetings with experts from HPE, we collected insights and directions that helped bring confidence in the practicality of the plugin, and consequently to its external validity.

There is, however, one problem with its practicality due to a SPIRE limitation. The plugin cannot produce numerical selectors because the SPIRE Agent compares the set of returned selectors with the set of expected selectors with string equality. This mostly impacts the selectors for CVSS (`contains-cvss-severities`) and EPSS scores (`contains-epss-risks`), which are known numerical metrics and also forces the user to configure expected results less intuitively.

We have consulted with SPIRE's open-source community about the possibility of supporting numeric selectors, but this is not currently aligned with their vision of identity selectors, which sees them as categorical properties that a node or workload has or does not have. This was therefore not included in our implementation and is not included in the list of future avenues of research.

Another threat that could be pointed out for internal validation. This regards the OpenStack infrastructure used as an environment for performance testing. Because the compute used for the instances that compose the Kubernetes cluster is a shared one, one could argue that resource usage could have been impacted by fluctuations caused by third-party usage. We mitigated this, however, by performing each experiment with 30 iterations. This not only increased our sample but also removed bias by reducing possible noise from the environment.

Lastly, the choice of tools could also be questioned and seen as a threat. This research is supported by the selection of prestigious tools, and based on related work and expert suggestions. The correct mitigation of our found threats pertaining to vulnerability status strongly relies on the correctness of such tools. This in turn depends on the moderation of their respective open-source communities. While it is possible for such tools to loose quality or become inadequate over time, this is mitigated due to the involvement of renowned organizations like CNCF and OWASP.

Chapter 8

Conclusion

In this work, we proposed a workflow to continuously assess workloads for compliance regarding their provenance and vulnerability state. This solution addresses the gap of previous continuous compliance approaches, which disregarded vulnerabilities after deployment. Our approach leverages Zero-Trust environments so applications are explicitly and continuously authenticated. Doing so also addresses the lack of examples of compliance as a policy for trust assessment on ZTA.

8.1 Results

In addition to assuming continuous compliance is used in the DevSecOps context and ZTA is used in the deployment environment, our solution has two additional requirements: (1) the CI/CD pipeline produces compliance evidence (namely, an SBOM and, ideally, information on the build pipeline and source-code repository information such as SLSA Provenance); and (2) an identity provisioning tool that periodically renews the identities used by service in the Zero-Trust context (which will effectively isolate workloads that cannot renew their identities). We understand these assumptions and requirements align with the good practices of DevSecOps.

By considering these requirements, we found four main threats that could allow attackers to enact supply chain attacks: use of compromised image and use of compromised SBOM (both to bypass compliance assessment), exploitation of newfound vulnerability before remediation, and propagation of compromise.

Our solution was to implement a new plugin for the CNCF SPIRE framework as a means to integrate vulnerability posture to ZTA. The plugin leverages popular tools, such as the *Sigstore* framework for downloading and verifying evidence of compliance and OWASP's *Dependency Track* to perform

SCA. Employing our plugin allows SPIRE's authentication to consider vulnerability posture and isolate non-conforming workloads as soon as possible. This extends continuous compliance assessment after deployment and helps mitigate compromise propagation. Since isolation should also lead to termination due to health checks, vulnerability exploitation can also be mitigated. Lastly, by verifying the origins of SBOM and image through non-forgeable SLSA Provenance we can also eliminate the threat of bypassing or misguiding compliance assessment.

We evaluated that the performance impact of our plugin does not add significant latency to SPIRE's attestation and that the resources necessary to employ our implementation amount only to the requirements to run *Dependency Track* in a scalable way. This means that in an environment concerned with compliance, such a tool or a similar one would be needed regardless; therefore, these requirements would be a necessary expense. Finally, adopting the plugin does not complicate the development or operation of modern cloud-native applications, since it does not add ownership issues and allows collaboration between developers, operators, and security officers.

8.2 Future work

As future work, we envisage designing and implementing additional compliance metrics and refining our plugin usability and interfaces in conjunction with SPIRE's open-source community. For instance, other dynamic properties like licensing restrictions could also be monitored in our approach by adding low complexity while expanding selectors. One example of a dynamic property that could be monitored is the Scorecard of each dependency.

Another improvement we envisage is adding a grace period configuration for different CVSS severities. With such support, non-critical applications can better benefit from the solution without automatic loss of availability.

Finally, we also believe in combining such an approach with technologies like confidential computing. For example, confidential computing ensures the integrity of the running binary but does not provide transparency into the security of its building process, which could be achieved using provenance artifacts. Combining both approaches would reduce the need for trust in the infrastructure and service operators at the same time as it ensures continuous compliance.

Bibliography

[1] Compliance in a DevOps Culture — martinfowler.com. `https://martinfowler.com/articles/devops-compliance.html`, 2021. [Accessed 19-07-2024].

[2] Documents & Templates | FedRAMP.gov — fedramp.gov. `https://www.fedramp.gov/documents-templates/`, 2024. [Accessed 19-07-2024].

[3] Exploit Prediction Scoring System (EPSS) — first.org. `https://www.first.org/epss/`, 2024. [Accessed 19-07-2024].

[4] 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. `https://github.com/spiffe/spiffe-helper`, 2024. [Accessed 19-07-2024].

[5] Graduated and Incubating Projects — cncf.io. `https://www.cncf.io/projects/`, 2024. [Accessed 19-07-2024].

[6] Home Page | CISA — cisa.gov. `https://www.cisa.gov/`, 2024. [Accessed 19-07-2024].

[7] Official PCI Security Standards Council Site. `https://east.pcisecuritystandards.org/`, 2024. [Accessed 19-07-2024].

[8] Security levels — slsa.dev. `https://slsa.dev/spec/v1.0/levels`, 2024. [Accessed 19-07-2024].

[9] Signing — docs.sigstore.dev. `https://docs.sigstore.dev/signing/overview`, 2024. [Accessed 19-07-2024].

[10] spire/ADOPTERS.md at main · spiffe/spire — github.com. `https://github.com/spiffe/spire/blob/main/ADOPTERS.md`, 2024. [Accessed 19-07-2024].

[11] The Kerberos ticket — IBM Documentation. `https://www.ibm.com/docs/en/sc-and-ds/8.4.0?topic=concepts-kerberos-ticket`, 2024. [Accessed 19-07-2024].

[12] The Leading Open-Source IAM Solution — wso2.com. `https://wso2.com/identity-server/`, 2024. [Accessed 19-07-2024].

[13] Vikas Agarwal, Chris Butler, Lou Degenaro, Arun Kumar, Anca Sailer, and Gosia Steinder. Compliance-as-code for cybersecurity automation in hybrid cloud. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 427–437, 2022.

[14] Andrew Babakian, Pere Monclus, Robin Braun, and Justin Lipman. A retrospective on workload identifiers: From data center to cloud-native networks. *IEEE Access*, 10:105518–105527, 2022.

[15] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, and Torsten Eymann. Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust. *Computers Security*, 110:102436, 2021.

[16] Baozhan Chen, Siyuan Qiao, Jie Zhao, Dongqing Liu, Xiaobing Shi, Minzhao Lyu, Haotian Chen, Huimin Lu, and Yunkai Zhai. A security awareness and protection system for 5g smart healthcare based on zero-trust architecture. *IEEE Internet of Things Journal*, 8(13):10248–10263, 2021.

[17] CISA. When to Issue VEX Information, Nov 2023. `https://www.cisa.gov/resources-tools/resources/when-issue-vex-information/`.

[18] CISA. Zero trust maturity model v2.0. `https://www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf`, 2023. [Accessed 19-07-2024].

[19] KENNA SECURITY CYENTIA INSTITUTE. *Prioritization to Prediction Volume 8: Measuring and Minimizing Exploitability*. 2022.

[20] Catherine de Weever and Marios Andreou. Zero trust network security model in containerized environments. *University of Amsterdam: Amsterdam, The Netherlands*, 2020.

[21] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. Llm agents can autonomously exploit one-day vulnerabilities, 2024.

[22] Jerry Gamblin. 2023 CVE Data Review — jerrygamblin.com. `https://jerrygamblin.com/2024/01/03/2023-cve-data-review/`, 2024. [Accessed 19-07-2024].

[23] Yuanhang He, Daochao Huang, Lei Chen, Yi Ni, Xiangjie Ma, and Yan Huo. A survey on zero trust architecture: Challenges and future trends. *Wirel. Commun. Mob. Comput.*, 2022, jan 2022.

[24] Jay Jacobs, Sasha Romanosky, Benjamin Edwards, Idris Adjerid, and Michael Roytman. Exploit prediction scoring system (epss). *Digital Threats*, 2(3), jul 2021.

[25] Pontus Johnson, Robert Lagerström, Mathias Ekstedt, and Ulrik Franke. Can the common vulnerability scoring system be trusted? a bayesian analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):1002–1015, 2018.

[26] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 511–523, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Aditya Sirish (NYU) and Tom Hennen (Google) representing the in-toto Community. in-toto and SLSA — slsa.dev. `https://slsa.dev/blog/2023/05/in-toto-and-slsa`, 2024. [Accessed 19-07-2024].

[28] Xhesika Ramaj, Mary Sánchez-Gordón, Vasileios Gkioulos, Sabarathinam Chockalingam, and Ricardo Colomo-Palacios. Holding on to compliance while adopting devsecops: An slr. *Electronics*, 11(22), 2022.

[29] Scott Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. Zero trust architecture, oct 2020.

[30] Sonatype. 8th Annual State of the Software Supply Chain. `https://www.sonatype.com/resources/state-of-the-software-supply-chain-2022/introduction`, 2022. [Accessed 19-07-2024].

[31] Steve Springett. Deploying Docker Container — docs.dependencytrack.org. `https://docs.dependencytrack.org/getting-started/deploy-docker/`, 2024. [Accessed 19-07-2024].

[32] Andreas Steffens, Horst Lichter, and Marco Moscher. Towards data-driven continuous compliance testing. In *Software Engineering*, 2018.

[33] Naeem Firdous Syed, Syed W. Shah, Arash Shaghaghi, Adnan Anwar, Zubair Baig, and Robin Doss. Zero trust architecture (zta): A comprehensive survey. *IEEE Access*, 10:57143–57179, 2022.

[34] Synopsys. Open Source Security & Risk Analysis Report (OSSRA) | Synopsys — synopsys.com. `https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html`, 2024. [Accessed 19-07-2024].

[35] Kennedy A. Torkura and Christoph Meinel. Towards vulnerability assessment as a service in openstack clouds. In *2016 IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops)*, pages 1–8, 2016.