UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Um Estudo sobre a Aplicação de Ferramentas para a Proteção de Cadeias de Suprimento de Software

Henrique Hideaki Koga

Orientador: Prof. Dr. Eduardo de Lucena Falcão

Natal, RN, 20 de janeiro de 2025

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Um Estudo sobre a Aplicação de Ferramentas para a Proteção de Cadeias de Suprimento de Software

Henrique Hideaki Koga

Orientador: Prof. Dr. Eduardo de Lucena Falcão

Trabalho de Conclusão de Curso de Graduação na modalidade Monografia, submetido como parte dos requisitos necessários para conclusão do curso de Engenharia de Computação pela Universidade Federal do Rio Grande do Norte (UFRN/CT).

Natal, RN, 20 de janeiro de 2025

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Koga, Henrique Hideaki.
Um estudo sobre a aplicação de ferramentas para a proteção de cadeias de suprimento de software / Henrique Hideaki Koga. - 2025.
48f.: il.

Monografia (Graduação) - Universidade Federal do Rio Grande do Norte, Departamento de Engenharia de Computação e Automação, Engenharia da Computação, Natal, 2025.
Orientação: Dr. Eduardo de Lucena Falcão.

1. Cadeia de Suprimentos de Software - Monografia. 2. Software Bill of Materials - Monografia. 3. Vulnerabilidades - Monografia. I. Falcão, Eduardo de Lucena. II. Título.

RN/UF/BCZM CDU 004

Elaborado por Raimundo Muniz de Oliveira - CRB-15/429

Um Estudo sobre a Aplicação de Ferramentas para a Proteção de Cadeias de Suprimento de Software

Henrique Hideaki Koga

Monografia aprovada em 20 de janeiro de 2025, pela banca examinadora composta pelos seguintes membros:

Prof. Dr. Eduardo de Lucena Falcão (orientador) . . . . DCA/UFRN

Prof. Dr. Carlos Manuel Dias Viegas . . . . DCA/UFRN

Prof. Dr. Ivanovitch Medeiros Dantas Silva . . . . DCA/UFRN

Resumo

Este trabalho apresenta um estudo sobre a aplicação de ferramentas para a proteção de cadeias de suprimentos de software, culminando no desenvolvimento de um *pipeline* de integração contínua (CI, abreviação para *Continuous Integration*). Por meio do uso de ferramentas como Trivy, Syft, Snyk e Cosign, o *pipeline* desenvolvido permite gerar e analisar a lista de materiais utilizados para a construção do *software* (SBOM, abreviação para *Software Bill of Materials*) automaticamente, bem como verificar a existência de vulnerabilidades em artefatos e aplicar assinaturas digitais para assegurar a integridade dos artefatos. A abordagem empregada consistiu no estudo de conceitos básicos de segurança, na experimentação das ferramentas e na implementação do *pipeline* de forma iterativa. O desenvolvimento da pesquisa permitiu obter como resultado uma solução eficaz no que concerne à mitigação de vulnerabilidades e contribuição para a rastreabilidade e segurança de artefatos, seguindo, deste modo, os preceitos do SLSA (*Supply Chain Levels for Software Artifacts*). Conclui-se, assim, que a abordagem proposta tem contribuição significativa para a proteção da cadeia de suprimento de software, oferecendo um processo seguro e confiável para a construção e implantação de sistemas.

Palavras-chave: cadeia de suprimentos de software, *pipeline* de CI, SBOM, vulnerabilidades.

Abstract

This work presents a study on the application of tools for the protection of software supply chains, culminating in the development of a continuous integration (CI) pipeline. Using tools such as Trivy, Syft, Snyk, and Cosign, the developed pipeline enables the automatic generation and analysis of the Software Bill of Materials (SBOM), as well as the identification of vulnerabilities in artifacts and the application of digital signatures to ensure artifact integrity. The approach involved the study of basic security concepts, experimentation with tools, and iterative pipeline implementation. The research resulted in an effective solution for mitigating vulnerabilities and contributing to artifact traceability and security, thus adhering to the principles of Supply Chain Levels for Software Artifacts (SLSA). It is concluded that the proposed approach significantly contributes to the protection of the software supply chain, offering a secure and reliable process for the construction and deployment of systems.

Keywords: software supply chain, CI pipeline, SBOM, vulnerabilities.

Agradecimentos

À minha mãe, Silvia Lumiko Ywanaga Koga, ao meu pai, Carlos Tomiuki Koga, ao meu irmão, Vinicius Massaki Koga, aos meus tios, tias, primos e primas, meu sincero agradecimento. À minha mãe e ao meu pai, minha eterna gratidão por todos os esforços e dedicação para me oferecer a melhor educação possível, proporcionando as bases para que eu alcançasse esta conquista.

Sumário

Sumário i

Lista de Figuras iii

Lista de Tabelas iv

1 Introdução 1

1.1 Contextualização e Motivação . . . 1

1.1.1 Incidentes . . . 2

1.2 Objetivos . . . 3

1.3 Metodologia . . . 3

1.4 Contribuições . . . 4

1.5 Organização do documento . . . 4

2 Fundamentação Teórica 5

2.1 Conceitos Básicos de Segurança . . . 5

2.2 Vulnerabilidades . . . 6

2.2.1 Common Vulnerability Scoring System . . . 6

2.2.2 Common Vulnerabilities and Exposures . . . 9

2.3 Cadeia de Suprimentos de Software . . . 10

2.3.1 Software Bill of Materials . . . 11

2.3.2 Ferramentas para Geração e Análise de SBOMs . . . 13

2.4 Supply Chain Levels for Software Artifacts (SLSA) . . . 14

2.4.1 Ataque à Cadeia de Suprimento de Software: Caracterização . . . 15

2.5 Trabalhos relacionados . . . 15

3 Proposta 18

3.1 Geração do SBOM . . . 18

3.2 Análise do SBOM . . . 20

3.3 Integridade do SBOM e da Imagem da Aplicação . . . 23

3.4 Considerações 26

4 Caso de Uso **27**
4.1 Validação do Pipeline 27
4.2 Atualização de Dependências e Redução de Vulnerabilidades 32

5 Conclusão **34**

Referências bibliográficas **35**

Lista de Figuras

2.1 Detalhes da Vulnerabilidade Log4j (Log4Shell) 8
2.2 Representação da Cadeia de Suprimentos de *software*. 10
2.3 SBOM-CycloneDX 12
2.4 SBOM-SPDX 14

3.1 Diagrama do pipeline 19
3.2 Geração do SBOM no Pipeline Proposto 20
3.3 Exemplo de uma vulnerabilidades identificada pelo Trivy. 21
3.4 Painel do Snyk com Recomendações de Mitigação 22
3.5 Validação Allowlist 23
3.6 Validação de Assinatura Digital do SBOM com o Cosign. 24
3.7 Assinatura e Verificação da Imagem Docker no Pipeline. 25
3.8 Processo de Assinatura e Verificação no Pipeline 26

4.1 Lista de Vulnerabilidades Detectadas no Caso de Uso. 28
4.2 Relatório Snyk Python. 29
4.3 Recomendação do Snyk para Atualização da Imagem Docker 29
4.4 Execução com Vulnerabilidades Permitidas 30
4.5 Interrupção do *Pipeline* 31
4.6 Redução de Vulnerabilidades Após Atualização de Dependências 33

Lista de Tabelas

4.1 Comparação de Vulnerabilidades Antes e Depois da Atualização 33

Capítulo 1

Introdução

Este capítulo começa destacando a importância de monitorar e proteger a cadeia de suprimentos de *software*, prevenindo a herança de vulnerabilidades existentes em suas dependências. Nesse contexto, o problema é introduzido, e os principais desafios são apresentados. Em seguida, detalham-se o objetivo, a metodologia e as contribuições do trabalho. Por fim, descreve-se a organização do documento.

1.1 Contextualização e Motivação

A segurança do software é um desafio multifacetado que abrange todas as etapas do seu ciclo de vida, desde a criação e desenvolvimento até sua execução em produção. Vulnerabilidades no código são particularmente prejudiciais, pois afetam diretamente a integridade do trabalho dos desenvolvedores, comprometendo o projeto antes mesmo de sua conclusão. Além disso, erros em configurações de infraestrutura introduzem riscos significativos durante a implantação e operação, expondo os sistemas a falhas e ataques potenciais. Em produção, as ameaças externas se tornam um risco constante, exigindo monitoramento e ações preventivas para proteger os sistemas. Como o ambiente de desenvolvimento e o de produção estão interligados, falhas em um deles podem rapidamente impactar o outro. Portanto, é essencial adotar uma abordagem estratégica e contínua que considere esses desafios de maneira integrada, garantindo a segurança em todas as fases do ciclo de vida do software.

Nesse cenário, a cadeia de suprimentos de *software* se destaca como um dos mais críticos. Ela é composta por diversas partes, como bibliotecas, ferramentas de compilação, plataformas de execução, integração, entre outras, que, juntas, formam a base de uma aplicação. Dado que elas frequentemente interconectam umas com as outras, bem como com sistemas externos, um ataque a uma delas se propaga rapidamente por toda a cadeia, comprometendo desde um pequeno módulo a diversos sistemas. Assim, a proteção deve

ser aplicada em todas as partes da cadeia, promovendo não só uma aplicação segura, mas a confiança de todos que a ela recorrem.

No cenário atual, nenhuma empresa pode ignorar a segurança de sua cadeia de suprimentos, uma vez que o impacto de vulnerabilidades dentro dela é considerável. Como tal, ignorar estes riscos já não é uma opção, e a adesão às práticas adequadas tornou-se uma necessidade estratégica. O presente trabalho destinou-se a contribuir para a integridade e segurança da cadeia de suprimentos de *software*, introduzindo e implementando um *pipeline* de integração contínua (CI, abreviação para o inglês *continuous integration*).

Essa iniciativa envolveu práticas de segurança no processo de concepção do sistema, incluindo a inspeção de vulnerabilidades e a implementação de soluções para corrigi-las. A solução sugerida foi projetada para ser pragmática e de fácil aplicação, oferecendo um ambiente de construção da aplicação (*build*) mais seguro e confiável.

1.1.1 Incidentes

Ataques à cadeia de suprimentos de *software* apresentam os perigos de comprometimento de componentes críticos usados na criação e distribuição de *software*. Ataques buscam explorar a confiança em bibliotecas, ferramentas e processos de entrega, resultando em graves e globais consequências.

Um dos incidentes mais notórios é o caso da empresa SolarWinds (2020). Atacantes implantaram um *malware*, chamado Sunburst, no processo de compilação do *software* Orion, tornando possível o provisionamento de atualizações infectadas em organizações governamentais e empresariais em vários países. Esse incidente evidenciou como a manipulação de um único ponto na cadeia de suprimentos, o processo de *build*, pode comprometer sistemas críticos em larga escala (U.S. Government Accountability Office 2022).

Outro exemplo clássico é a vulnerabilidade Log4Shell, descoberta em 2021 na biblioteca Log4J. A vulnerabilidade atingiu a adequada utilização de registros de *logs* onipresentes em aplicativos Java. Sua exploração permitiu que atacantes executassem o código remotamente nos servidores afetados (Kost 2025). O ataque teve um impacto global, principalmente devido ao amplo uso da biblioteca. Um caso semelhante é o do pacote NPM Event-Stream em 2018 (npm 2018), onde uma dependência maliciosa foi adicionada ao código para roubo de criptomoedas, demonstrando como alterações não autorizadas em bibliotecas amplamente utilizadas podem comprometer sistemas em larga escala.

Todos esses casos evidenciam a importância de práticas de monitoramento contínuo, validação de dependências e auditorias regulares para proteger a cadeia de suprimentos e mitigar riscos cibernéticos.

1.2 Objetivos

O principal objetivo desse trabalho é propor uma estratégia de monitoramento das vulnerabilidades na cadeia de suprimentos de *software*. O primeiro requisito da estratégia proposta é que ela seja simples o suficiente para ser incorporada nas etapas de desenvolvimento, teste e implantação de *software* arbitrário. O segundo requisito é que ela automatize o processo de detecção de vulnerabilidades introduzidas na cadeia de suprimentos do *software*.

1.3 Metodologia

As atividades fundamentais para a concepção de uma estratégia para detecção de vulnerabilidades na cadeia de suprimentos de *software* foram:

1. o estudo dos principais conceitos sobre cadeias de suprimentos de *software*;
2. uma revisão de literatura, de modo a verificar as práticas e ferramentas adotadas para o monitoramento da cadeia de suprimentos de *software*;
3. a experimentação das ferramentas de suporte ao monitoramento da cadeia de suprimentos de *software*;
4. o projeto e implementação de um *pipeline* de integração contínua para uma aplicação caso de uso, utilizando o Github.

A atividade 1 permitiu identificar os elementos essenciais de uma cadeia de suprimentos de *software*, bem como os principais desafios associados. A atividade 2 foi essencial para compreender a prática padrão no mercado e os casos de uso mais comuns de monitoramento da cadeia de suprimentos de *software*. Essa etapa ajudou a identificar as práticas estabelecidas como, por exemplo, a identificação da lista de materiais utilizados para a construção do *software* (ou SBOMs, do inglês *Software Bill of Materials*), a assinatura dos SBOMs e o uso de escâneres de vulnerabilidades. Uma vez conhecidas as principais práticas, a atividade 3 foi fundamental para aprofundar no funcionamento das ferramentas de monitoramento de vulnerabilidades e conhecer as principais funcionalidades e limitações delas. Finalmente, a atividade 4 consistiu no projeto e a implementação de um *pipeline* automatizado de integração contínua para uma aplicação caso de uso. Tal *pipeline* foi construído iterativamente, i.e., os requisitos foram refinados através da criação de protótipos intermediários, até alcançar a versão final proposta nesse documento.

1.4 Contribuições

Como contribuição deste trabalho, foi desenvolvido um *pipeline* que implementa práticas de segurança automatizadas no processo de desenvolvimento de *software*. O *pipeline* oferece benefícios significativos ao simplificar a integração com fluxos de desenvolvimento existentes, reduzindo o esforço necessário para sua adoção. Além disso, ele promove maior controle sobre a cadeia de suprimentos de *software* ao incorporar etapas como a geração, análise e validação de componentes, como SBOMs, e a análise contínua de vulnerabilidades. Essa abordagem contribui para a identificação e mitigação de riscos de forma proativa, aumentando a confiabilidade e a segurança dos artefatos gerados, ao mesmo tempo em que melhora a eficiência operacional das equipes de desenvolvimento.

1.5 Organização do documento

Este trabalho está organizado em cinco capítulos, conforme descrito a seguir:

1. **Introdução**: Apresenta a contextualização, motivação, objetivos, metodologia, contribuições e a organização do documento, fornecendo uma visão geral do trabalho.
2. **Fundamentação Teórica**: Discute conceitos essenciais relacionados à segurança em cadeias de suprimentos de software, incluindo vulnerabilidades, SBOM, ferramentas de análise e o modelo SLSA. Também aborda trabalhos relacionados ao tema.
3. **Proposta**: Detalha a construção do *pipeline* automatizado para a proteção da cadeia de suprimentos de software, explicando as etapas de geração, análise e validação de artefatos.
4. **Caso de Uso**: Apresenta a aplicação prática do *pipeline* em um cenário de teste, avaliando a eficácia na identificação e mitigação de vulnerabilidades e discutindo os resultados obtidos.
5. **Conclusão**: Resume os principais resultados do trabalho, destacando contribuições e limitações, e propõe possíveis direções para estudos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo inicia apresentando os conceitos básicos de segurança essenciais para a proteção de cadeias de suprimentos de *software*, tais como assinatura digital e sua verificação. Posteriormente, é apresentado o conceito de vulnerabilidade, juntamente com um *framework* que auxilia na avaliação das vulnerabilidades, além de uma base de dados oficial usada para catalogação de vulnerabilidades conhecidas. Por fim, são apresentados conceitos e práticas relacionadas à proteção da cadeia de suprimentos de *software*.

2.1 Conceitos Básicos de Segurança

A **assinatura digital** é um mecanismo de criptografia assimétrica que garante autenticidade, integridade e não-repudiação de uma mensagem ou documento eletrônico. Esse mecanismo funciona por meio do uso de um par de chaves: a privada, mantida em sigilo pelo remetente; e a pública, disponível para terceiros realizarem a checagem (Stallings et al. 2008).

A **verificação de assinatura** permite estabelecer confiança na integridade e legitimidade de uma mensagem. Para fazer isso, o destinatário: i) computa o *hash* da mensagem, ii) usa a chave pública para decodificar a assinatura realizada pelo remetente e recuperar o *hash* esperado, e iii) compara o *hash* decodificado da assinatura com o *hash* recém-calculado da mensagem. Se ambos forem idênticos, pode-se garantir que o remetente é genuíno, pois só ele possui a chave privada que corresponde a essa assinatura, e pode-se garantir que o conteúdo da mensagem não foi modificado desde a assinatura.

No contexto de segurança em cadeias de suprimentos de *software*, a assinatura digital desempenha um papel essencial na proteção da autenticidade e integridade de artefatos como pacotes, bibliotecas e contêineres, garantindo que os componentes do *software* sejam autênticos e livres de modificações maliciosas. Tal verificação pode ocorrer de forma integrada no processo de desenvolvimento e operação através de *pipelines* de *Dev-*

SecOps[1], reduzindo significativamente o risco de introdução de componentes inseguros e garantindo a conformidade com padrões de segurança e regulamentações. Além disso, tanto as dependências quanto o artefato que as lista (SBOM), bem como a própria aplicação (e.g., imagem de um contêiner) são assinados e verificados, permitindo rastrear a origem, validar a segurança desses elementos e fortalecer a confiança nos processos de desenvolvimento e entrega de *software*, mesmo em ambientes dinâmicos e complexos.

2.2 Vulnerabilidades

Uma vulnerabilidade é uma falha ou fraqueza em um sistema que pode ser explorado por agentes mal-intencionados para comprometer a confidencialidade, integridade ou disponibilidade de informações e recursos. As vulnerabilidades ocorrem como resultado de má configuração, falhas de projeto, codificação defeituosa e falta de atualizações. Devido ao fato de que essas lacunas são uma possível porta de entrada para ataques, encontrá-las e eliminá-las é um processo importante para garantir a segurança de sistemas e dados.

A classificação eficaz de vulnerabilidades é um elemento essencial da segurança em sistemas e cadeias de suprimentos de *software*. Quando se trata de mitigar os riscos da melhor maneira possível, é necessário um método para classificar as vulnerabilidades. A classificação pode ser feita com base em diversos fatores, incluindo a severidade da vulnerabilidade, a facilidade de exploração da vulnerabilidade por um atacante e o impacto da vulnerabilidade nos sistemas afetados.

Para atender a essa necessidade, o **Common Vulnerability Scoring System** (CVSS) foi criado. Trata-se de um padrão global para priorizar as vulnerabilidades com base em critérios objetivos e mensuráveis. É uma ferramenta que fornece orientação clara sobre onde é mais crítico focar tempo e recursos. Outro sistema que funciona em conjunto com CVSS é o **Common Vulnerabilities and Exposures** (CVE) – uma base de dados confiável que contém informações sobre as vulnerabilidades conhecidas. O CVSS e CVE são detalhados nas próximas sub-seções.

2.2.1 Common Vulnerability Scoring System

O *Common Vulnerability Scoring System* (CVSS) avalia a severidade de vulnerabilidades em sistemas com base em métricas padronizadas. Dividido em três grupos — métricas base, temporais e ambientais — ele orienta a priorização de ações mitigatórias. Métricas

[1] DevSecOps é uma metodologia que integra a segurança em todas as fases do desenvolvimento de *software*. O termo é uma combinação de *development*, *security* e *operations* (Amazon Web Services 2025).

como o vetor de ataque, complexidade do ataque e impacto em confidencialidade, integridade e disponibilidade são cruciais para a segurança da cadeia de suprimentos.

- **Métricas Base:**
  - **Vetor de Ataque (*Attack Vector* - AV):** Avalia o alcance da exploração, variando de físico (local) a remoto (via rede).
  - **Complexidade do Ataque (*Attack Complexity* - AC):** Mede a dificuldade de exploração, classificada como baixa ou alta.
  - **Privilégios Necessários (*Privileges Required* - PR):** Verifica o nível de acesso requerido, sendo mais crítico quando não há necessidade de credenciais.
  - **Impacto na Integridade (*Integrity Impact* - I):** Indica o potencial de manipulação ou alteração de informações.
  - **Impacto na Confidencialidade (*Confidentiality Impact* - C):** Avalia o risco de exposição de dados sensíveis.
- **Métricas Temporais:**
  - **Maturidade do Código de Exploração:** Mede a disponibilidade de códigos para explorar a vulnerabilidade.
  - **Nível de Remediação:** Verifica a existência de correções, como *patches* ou soluções alternativas.
- **Métricas Ambientais:**
  - **Relevância do Cenário:** Determina a criticidade da vulnerabilidade com base nos ativos estratégicos ou dependências críticas, como *pipelines* CI/CD.

Tomemos como exemplo o **Log4Shell**, vulnerabilidade do Log4j, com o registro **CVE-2021-44228**[2], para ilustrar a aplicação prática da estrutura do CVSS. O *CVSS Base Score* (Pontuação Base do CVSS) alcançou o valor total de 10.0, nível máximo de criticidade. No grupo de métricas de explorabilidade (*Exploitability Metrics Group*), o vetor de ataque (*Attack Vector*) recebeu uma classificação de remoto (**AV: *Network***), porque o ataque pode ser realizado via internet. A complexidade do ataque (*Attack Complexity*) foi avaliada como baixa (**AC: *Low***), indicando que não são necessárias condições específicas para que o ataque seja executado. O nível de privilégios exigidos (*Privileges Required*) foi considerado muito baixo (**PR: *None***), significando que o ataque pode ser realizado sem que o invasor tenha acesso a credenciais. A interação do usuário (*User Interaction*) foi nula (**UI: *None***), o que torna a invasão ainda mais perigosa, pois não depende de ações

[2] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228`

do usuário. O escopo (*Scope*) foi alterado (**S:** ***Changed***), indicando que o impacto do ataque se estende além do componente comprometido, afetando outros sistemas ou meios conectados.

CVE-2021-44228 Detalhe

MODIFICADO

Esta vulnerabilidade foi modificada desde a última análise feita pelo NVD. Ela está aguardando reanálise, o que pode resultar em mais alterações nas informações fornecidas.

Descrição

Apache Log4j2 2.0-beta9 até 2.15.0 (excluindo as versões de segurança 2.12.2, 2.12.3 e 2.3.1) Os recursos JNDI usados na configuração, mensagens de log e parâmetros não protegem contra LDAP controlado por invasor e outros endpoints relacionados a JNDI. Um invasor que pode controlar mensagens de log ou parâmetros de mensagens de log pode executar código arbitrário carregado de servidores LDAP quando a substituição de pesquisa de mensagens está habilitada. A partir do log4j 2.15.0, esse comportamento foi desabilitado por padrão. A partir da versão 2.16.0 (junto com 2.12.2, 2.12.3 e 2.3.1), essa funcionalidade foi completamente removida. Observe que essa vulnerabilidade é específica do log4j-core e não afeta log4net, log4cxx ou outros projetos do Apache Logging Services.

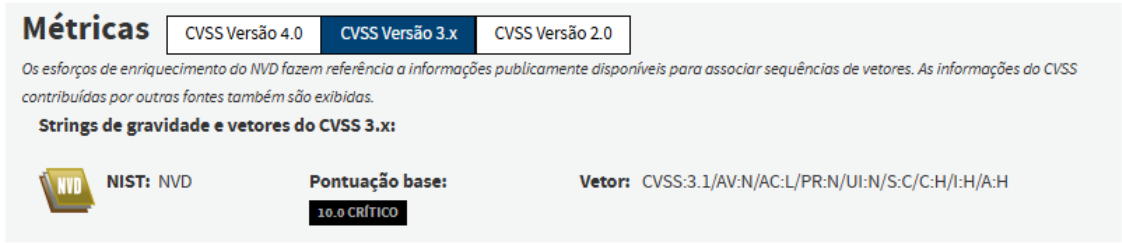

Figura 2.1: Detalhes da Vulnerabilidade Log4j (Log4Shell)
Fonte: (The MITRE Corporation 2025).

A Figura 2.1 apresenta detalhes sobre a vulnerabilidade CVE-2021-44228, conhecida como Log4Shell.

Já quanto às métricas de impacto, a vulnerabilidade atingiu níveis considerados altos em todas as dimensões. O impacto na confidencialidade (*Confidentiality Impact*) é alto e significa que dados confidenciais podem ser acessados sem autorização. O impacto na integridade (*Integrity Impact*) também é alto, indicando a possibilidade de ações maliciosas que podem alterar informações críticas. O impacto na disponibilidade (*Availability Impact*) é igualmente alto, denotando a potencial indisponibilidade de serviços essenciais.

Toda essa exemplificação faz do CVSS um *framework* completo para classificação e priorização de vulnerabilidades. Adicionalmente, quando utilizado em combinação com o CVE, é uma combinação extremamente eficaz para tornar a organização ciente de possíveis brechas críticas em seus sistemas e focar em ações mitigatórias efetivas.

2.2.2 Common Vulnerabilities and Exposures

O *Common Vulnerabilities and Exposures* (CVE) é uma autoridade que organiza e padroniza a maneira como as vulnerabilidades de segurança cibernética são disseminadas e descritas ao redor do mundo. O CVE foi criado em 1999 pela MITRE Corporation e fornece identificadores exclusivos – os CVE Identifiers , como é o caso do CVE-2021-44228, apresentado na Figura 2.1, que permite a todos catalogar e referenciar vulnerabilidades sempre da mesma forma (MITRE 2016).

Esse método uniforme possibilita que diferentes instituições, pesquisadores e ferramentas de segurança de qualquer parte do mundo referenciem todas as vulnerabilidades da mesma forma. Cada entrada no CVE é estruturada no formato de um dicionário, com três componentes principais: o identificador CVE , que segue o formato CVE-AAAA-NNNN, onde AAAA é o ano da atribuição e NNNN é um número sequencial; uma descrição breve da vulnerabilidade padronizada; e as referências, com *links* para relatórios profundos, avisos dos fornecedores e informações similares. No entanto, todos os CVEs por si só não possuem uma definição de quão severa é uma vulnerabilidade ou quão fácil é explorá-la (MITRE 2016).

A análise detalhada de diversos aspectos da vulnerabilidade é possibilitada pela combinação com o *framework* de CVSS. Enquanto o CVE é destinado a detalhar a vulnerabilidade, explicando quais componentes e dependências, e quais versões, a vulnerabilidade afeta, o CVSS se aprofunda na gravidade da vulnerabilidade. Para alcançar esta conclusão, recorre-se a uma pontuação numérica, variando de 0 a 10, conforme métricas de padronização. Tais métricas se referem ao quão fácil é explorar a vulnerabilidade, o dano que esta pode ocasionar e até mesmo a que circunstâncias permitem o sucesso de um ataque (Mell et al. 2006).

A combinação entre CVE e CVSS permite maior agilidade e assertividade na classificação das vulnerabilidades, identificando as vulnerabilidades críticas mais facilmente, e isso permite priorizar correções de forma inteligente. A título de exemplo, o CVE-2021-44228 trata-se da vulnerabilidade Log4Shell, uma falha significante no Log4j. Embora o CVE já proporcione uma descrição padronizada da vulnerabilidade, é o CVSS o responsável por revelar o nível ao qual o sistema está sujeito ao risco, citando a pontuação de 10.0. Essa alta pontuação decorre da facilidade em ser explorada, visto que o ataque pode se dar remotamente, e a criticidade no impacto da confidencialidade, integridade e disponibilidade do sistema e dados.

2.3 Cadeia de Suprimentos de Software

A cadeia de suprimentos de um *software* é o conjunto de processos, ferramentas, recursos e pessoas que participam do desenvolvimento, entrega e implantação do *software*. Da mesma forma que o fazem as cadeias de suprimentos tradicionais, que movem bens físicos desde a matéria-prima até o consumidor final, a cadeia de suprimentos do *software* articula o tráfego dos componentes digitais que dão suporte à criação e à operação de um sistema de *software*. A Figura 2.2 ilustra as etapas de uma cadeia de suprimento de um produto tradicional e de *software*.

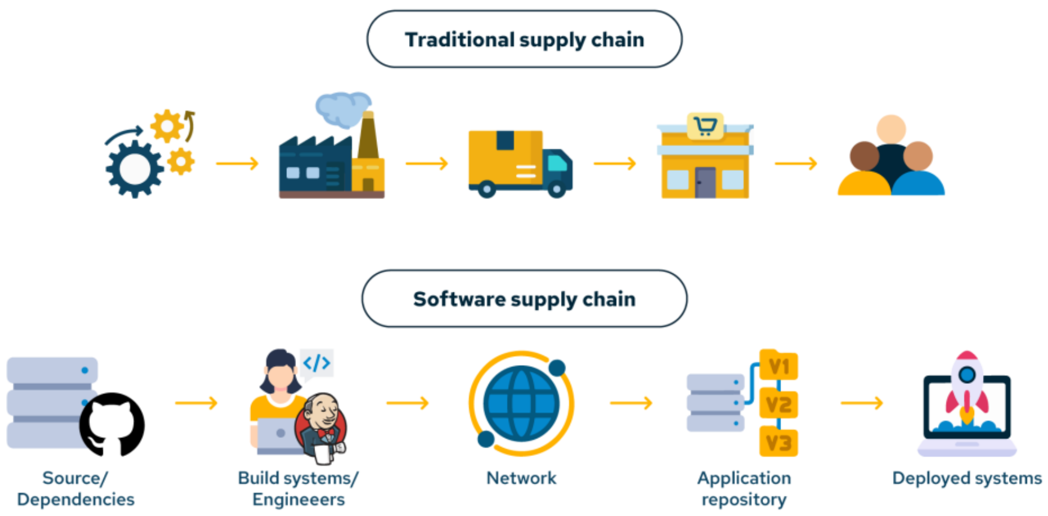

Figura 2.2: Representação da Cadeia de Suprimentos de *software*. Fonte: https://blog.convisoappsec.com/wp-content/uploads/2023/09/supply-chain-1024x526.png

A partir da Figura 2.2 é possível perceber que a cadeia de suprimentos de *software* inicia com as fontes e dependências. Tais fontes e dependências são bibliotecas de terceiros e *frameworks*.

Estes são usados por engenheiros e os sistemas de compilação, que os integram, testam o código e os compilam por meio de ferramentas automatizadas como *pipelines* de CI e entrega contínua (CD, do inglês *continuous deployment*). Após a compilação do código final, o *software* é transferido pela rede. Pode então ser distribuído ou testado antes de ser armazenado em repositórios de aplicações, os quais catalogam e armazenam diferentes partes do *software*. Finalmente, o *software* é implantado em sistemas de produção, e é a partir daí que os usuários finais podem usá-lo (DeFranco & Kshetri 2022).

Existem muitos riscos envolvidos na cadeia de suprimento de *software*, como dependências comprometedoras, falta de transparência e complexidade. Muitas bibliotecas de

terceiros e pacotes adicionam suas próprias vulnerabilidades que, se exploradas, podem comprometer um sistema inteiro.

A falta de rastreamento dificulta a identificação da origem dos componentes, torna difícil verificar a segurança de tais componentes, bem como a grande variedade de fornecedores e processos aumenta as possibilidades de falha. Para mitigar tais problemas, podem ser adotadas práticas como gestão de dependências, auditorias regulares e o uso de *Software Bill of Materials* (SBOMs). As ferramentas de análise e composição de *software* garantem a presença de vulnerabilidades em tais bibliotecas e componentes e as assinaturas digitais garantem a integridade do *software* ao mesmo tempo que o distribuem. A automação de processos com pipelines de CI reduz significativamente o risco de erro humano e é usada para construir testes de verificação de segurança diretamente no desenvolvimento de *software*. Essas medidas são fundamentais para garantir a segurança e integridade na cadeia de suprimento de *software* em um ambiente digital em constante mudança.

2.3.1 Software Bill of Materials

O **Software Bill of Materials** (SBOM), ou lista de materiais de software, é uma representação detalhada dos componentes de um software, incluindo suas dependências diretas e indiretas, inspirado em práticas de transparência das cadeias de suprimentos industriais. No contexto das cadeias de suprimentos de software, o SBOM fornece uma visão confiável e automatizada de todos os elementos integrados, permitindo rastreabilidade e controle sobre os componentes utilizados (National Telecommunications and Information Administration (NTIA) 2021).

```json
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:123e4567-e89b-12d3-a456-426614174002",
  "version": 1,
  "components": [
    {
      "type": "library",
      "name": "flask",
      "version": "2.0.3",
      "purl": "pkg:pypi/flask@2.0.3",
      "licenses": [
        {
          "license": {
            "id": "BSD-3-Clause"
          }
        }
      ]
    },
    {
      "type": "library",
      "name": "itsdangerous",
      "version": "2.1.2",
      "purl": "pkg:pypi/itsdangerous@2.1.2",
      "licenses": [
        {
          "license": {
            "id": "MIT"
          }
        }
      ]
    }
  ],
  "dependencies": [
    {
      "ref": "pkg:pypi/flask@2.0.3",
      "dependsOn": [
        "pkg:pypi/itsdangerous@2.1.2"
      ]
    },
    {
      "ref": "pkg:pypi/itsdangerous@2.1.2",
      "dependsOn": []
    }
  ]
}
```


Figura 2.3: Fonte: De autoria própria.

A Figura 2.3 apresenta um modelo de SBOM no formato CycloneDX.

Com atributos únicos como nome, versão e assinatura digital, o SBOM facilita a identificação de vulnerabilidades e a mitigação de riscos, reduzindo o tempo de resposta a incidentes. Além disso, promove a conformidade com padrões de segurança e requisitos legais, assegurando a integridade da cadeia de suprimentos e reforçando a confiança

na origem e no estado dos componentes (National Telecommunications and Information Administration (NTIA) 2021)

2.3.2 Ferramentas para Geração e Análise de SBOMs

Tanto as ferramentas de geração quanto as de análise de SBOM são peças importantes e complementares do quebra-cabeça de garantir a segurança de uma cadeia de suprimento de software. Conforme detalhado acima, elas fornecem alta visibilidade para cada componente e dependência, mas cada uma se destaca em aspectos diferentes. Por exemplo, o **Trivy** é uma ferramenta de varredura de segurança de código aberto que pode detectar vulnerabilidades em imagens Docker, repositórios e sistemas de arquivos. Além de rastrear vulnerabilidades, o Trivy também fornece um modelo SBOM que suporta formatos, como CycloneDX e SPDX, portanto, sendo de fácil integração e análise baseada no fluxo de CI / CD (Security 2025).

Por outro lado, o **Syft** é especializado exclusivamente na geração de SBOMs. Ele oferece uma visão completa de todos os componentes de imagens Docker ou sistemas de arquivos, descrevendo dependências e licenças nos formatos CycloneDX e SPDX. Embora o Trivy combine a geração de SBOMs com a análise de vulnerabilidades, o Syft foi projetado principalmente para a criação de inventários de software (Anchore 2025). O **Snyk** se destaca como uma plataforma completa de análise de vulnerabilidades. No entanto, ao contrário do Trivy e do Syft, que geram SBOMs, o Snyk aborda a identificação de problemas em dependências, infraestrutura como código e contêineres. Ele fornece relatórios detalhados e sugere correções, além de um painel fácil de usar para priorizar riscos. A integração avançada do Snyk com ferramentas de desenvolvimento o torna ideal para análise estendida durante o ciclo de vida do software (Ltd. 2025).

Já a ferramenta **Cosign** aborda uma necessidade diferente. Sua missão é garantir a integridade e autenticidade dos artefatos. O Cosign pode assinar e verificar imagens docker e SBOMs. Ele os criptografa para proteger a cadeia de fornecimento contra adulteração. O Cosign apresenta suporte à assinatura *keyless*, um método de assinatura digital que não exige o gerenciamento manual de chaves privadas, tornando-o um ponto a favor para fluxos de automação. CicloneDX e SPDX diferem nos padrões que apoiam. Como padrão da Linux Foundation, o SPDX é dedicado à conformidade de licenciamento e rastreabilidade de origem. Por outro lado, o padrão mantido pelo OWASP CicloneDX é centrado no problema da segurança dos componentes e análise de vulnerabilidades (OWASP 2025). Essa distinção faz com que o CicloneDX seja uma escolha melhor para equipes de segurança (Foundation 2025). A seguir, um exemplo de um SBOM no formato SPDX, o formato

CycloneDX pode ser visto na Figura 2.3: Sbom-CycloneDX já ilustrada anteriormente (cosign 2025).

```
SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
SPDXID: SPDXRef-DOCUMENT
DocumentName: MyPythonApp
DocumentNamespace: http://spdx.org/spdxdocs/mypythonapp-1.0.0
Creator: Tool: Syft v1.9.0
Created: 2025-01-13T12:00:00Z

##### Package Information #####
PackageName: requests
SPDXID: SPDXRef-Package-Requests
PackageVersion: 2.28.1
PackageDownloadLocation: https://pypi.org/project/requests/2.28.1/
PackageLicenseDeclared: Apache-2.0

PackageName: flask
SPDXID: SPDXRef-Package-Flask
PackageVersion: 2.1.3
PackageDownloadLocation: https://pypi.org/project/flask/2.1.3/
PackageLicenseDeclared: BSD-3-Clause
```

Figura 2.4: Fonte: De autoria própria.

A Figura 2.4 apresenta um modelo de SBOM no formato SPDX.

Assim, enquanto o Trivy é praticamente um gerador de SBOM e analisador de vulnerabilidades combinados, o Syft é mais especializado em inventários de software, o Snyk se expande para toda a infraestrutura, e o Cosign está preocupado com a integridade de artefatos. Ao combinar tais ferramentas com padrões como CycloneDX e SPDX, pode-se obter um ambiente seguro e eficaz que atende aos padrões de DevSecOps.

2.4 Supply Chain Levels for Software Artifacts (SLSA)

O **Supply Chain Levels for Software Artifacts (SLSA)** é uma especificação voltada para proteger o ciclo de vida de desenvolvimento e distribuição de software contra adulterações e vulnerabilidades. Ao organizar práticas em níveis progressivos, o SLSA promove confiança, rastreabilidade e integridade no uso de software (The Linux Foundation, SLSA Specification v1.0 2024).

A estrutura do SLSA é organizada em níveis, cada um introduzindo práticas específicas para melhorar a segurança:

- **Nível 1 - Rastreabilidade Inicial:** Estabelece a coleta de informações básicas sobre a origem e o processo de construção do software, criando uma base inicial de segurança.
- **Nível 2 - Provas de Integridade:** Garante que os artefatos de software correspondam ao código-fonte esperado, com comprovações de proveniência geradas automaticamente.
- **Nível 3 - Isolamento e Automação:** Exige que o ambiente de construção seja isolado de ameaças externas e que o processo seja completamente automatizado, reduzindo riscos de erros humanos.

Atualmente, o SLSA foca na trilha de construção (*Build Track*), abordando aspectos como produção de artefatos e verificação de sua proveniência. Futuras versões planejam expandir para áreas como gerenciamento de código-fonte, ampliando sua aplicação.

Com ataques cibernéticos cada vez mais sofisticados, o SLSA se destaca como uma ferramenta essencial para fortalecer a resiliência das cadeias de suprimento de software. Sua adoção não só aumenta a segurança e a transparência, mas também facilita a conformidade com regulamentações emergentes, protegendo tanto as operações das organizações quanto seus clientes (The Linux Foundation, SLSA Specification v1.0 2024).

2.4.1 Ataque à Cadeia de Suprimento de Software: Caracterização

Para ser classificado como um ataque à cadeia de suprimento, algumas características principais podem ser identificadas, como o comprometimento de componentes terceirizados, incluindo bibliotecas ou serviços utilizados no desenvolvimento e operação do software. Esses ataques também podem envolver a manipulação de processos de build, integração contínua ou distribuição, com potencial de causar impactos em larga escala devido à disseminação dos componentes comprometidos. Além disso, são difíceis de detectar, pois geralmente se apresentam como operações legítimas, sendo frequentemente descobertos somente após provocarem danos significativos.

2.5 Trabalhos relacionados

O trabalho de Faruk *et al.* (2022) analisa *frameworks* e práticas para proteger a cadeia de suprimento de software, destacando ataques como os da SolarWinds e Log4J, que comprometem produtos finais ao explorar vulnerabilidades em componentes e processos. O trabalho revisa taxonomias de ataques, incluindo métodos como confusão de dependências e injeção de código malicioso (Faruk et al. 2022).

Entre os *frameworks* analisados, estão modelos como o SSDF, que propõe integrar segurança ao ciclo de desenvolvimento, e o SLSA, que foca na integridade de processos. O estudo conclui que, embora existam abordagens robustas, a implementação prática ainda enfrenta desafios, como custos elevados e falta de automação. Recomenda maior colaboração entre fornecedores e consumidores para fortalecer a proteção da cadeia de suprimentos de *software* e mitigar riscos cibernéticos (Faruk et al. 2022).

O relatório ***Securing the Future: The State of Software Supply Chain Security in 2025***, da Xygeni Security (Security 2024), analisa os desafios e avanços na segurança da cadeia de suprimentos de software. O documento destaca o aumento de ataques em 2024, como a manipulação de bibliotecas amplamente utilizadas e pacotes maliciosos em repositórios de código aberto. Também aborda o impacto do *DORA (Digital Operational Resilience Act)*, uma regulamentação europeia que exige que empresas, especialmente do setor financeiro, adotem medidas rigorosas para monitorar fornecedores, relatar incidentes rapidamente e fortalecer a resiliência contra ataques à cadeia de suprimentos. Além disso, o relatório apresenta soluções práticas, como o monitoramento contínuo de dependências e a validação de segurança em todas as etapas do ciclo de vida do desenvolvimento de software.

A segurança da cadeia de suprimentos de software enfrenta desafios significativos devido a diversos vetores de ataque. ***Journey to the Center of Software Supply Chain Attacks***, de Ladisa *et al.* (2023), apresenta uma taxonomia com 117 vetores, destacando métodos como a subversão de pacotes legítimos (inserção de código malicioso em repositórios confiáveis), o abuso do mecanismo de resolução de dependências (priorização de dependências maliciosas) e a criação de pacotes maliciosos do zero. Essa classificação permite identificar e mitigar riscos em diferentes pontos da cadeia de suprimentos.

O estudo também propõe 33 salvaguardas organizadas por custo-benefício. Entre as mais relevantes estão: a geração e análise de *SBOMs* para rastrear e verificar dependências, autenticação segura para prevenir acessos não autorizados, ambientes de construção efêmeros para limitar a persistência de malwares e o isolamento de etapas de construção para minimizar impactos de pacotes maliciosos.

O trabalho ***SoK: A Defense-Oriented Evaluation of Software Supply Chain Security***, de Abu Ishgair *et al.* (2024) ., aborda conceitos diretamente relacionados à segurança em cadeias de suprimentos de software, propondo o modelo AStRA (*Artifacts, Steps, Resources, Principals*). Este modelo organiza a cadeia em quatro componentes principais: artefatos, como código-fonte e pacotes de software; etapas, que representam ações como construção e testes; recursos, incluindo ferramentas e infraestrutura; e agentes, como usuários e sistemas automatizados. Essa estrutura sistemática é empregada para mapear

dependências e identificar vulnerabilidades, promovendo estratégias de mitigação que integram segurança e confiabilidade ao longo de toda a cadeia de suprimentos.

O estudo destaca práticas de segurança distribuídas em níveis que abordam conceitos fundamentais de proteção em *pipelines*. O primeiro nível garante que as etapas sejam bem definidas e imutáveis, reforçando a integridade operacional. O segundo nível foca na rastreabilidade, registrando os agentes, recursos e operações envolvidos, enquanto o terceiro nível enfatiza a detecção de adulterações ou comportamentos inesperados nas etapas. Esses princípios reforçam a segurança integrada e a transparência, pilares essenciais para mitigar riscos e assegurar o controle de processos críticos.

O trabalho de (Gama et al. 2024) apresenta abordagens importantes para a proteção de cadeias de suprimento de software. Um dos principais pontos é o uso dos SBOMs e a integração com ferramentas como *Dependency Track* e *Trivy*, para viabilizar a análise contínua e automatizada de vulnerabilidades, contribuindo para a priorização de riscos com base em dados de fontes confiáveis, como o *National Vulnerability Database* (NVD). Outro aspecto relevante é a utilização de assinaturas digitais com ferramentas como *Cosign*, garantindo a integridade e a autenticidade de SBOMs e imagens Docker. Essa prática reforça a rastreabilidade e previne adulterações, atendendo aos padrões do SLSA. Contudo, uma diferença fundamental é que o monitoramento de vulnerabilidade acontece de forma contínua durante a execução das aplicações. Sempre que uma aplicação requisita uma identidade, essencial para a comunicação, a verificação do SBOM acontece de modo que a identidade só é emitida quando nenhuma vulnerabilidade for detectada na aplicação. Essa estratégia tende a ser mais eficiente pois independentemente da última vez que o CI executou, o monitoramento de vulnerabilidades permanece ativo durante a execução da aplicação.

Todos os trabalhos mencionados abordam estratégias para proteger as etapas e componentes de uma cadeia de suprimentos de software. Eles enfatizam a importância de identificar riscos, organizar processos e adotar práticas que garantam segurança e transparência em cada etapa. Essas análises fornecem uma base sólida para reduzir vulnerabilidades e fortalecer controles, contribuindo para o desenvolvimento de soluções integradas e confiáveis que protejam os sistemas de forma eficiente.

Capítulo 3

Proposta

Este capítulo apresenta a construção de um *pipeline* de integração contínua (CI) para monitorar e gerenciar vulnerabilidades, assegurando a integridade dos componentes da cadeia de suprimentos de *software*. O *pipeline* automatiza a geração e análise de SBOMs, além de implementar mecanismos de assinatura digital para garantir a autenticidade e rastreabilidade dos artefatos. O código-fonte do *pipeline* desenvolvido está disponível no repositório do GitHub de Henrique Hideaki, acessível em: `https://github.com/HenriqueHideaki/pipeline-ci-sbom`.

3.1 Geração do SBOM

O pipeline aplica Trivy, Syft, Snyk e Cosign para monitorar vulnerabilidades e garantir a integridade dos artefatos. Detalhes sobre as funcionalidades dessas ferramentas estão no capítulo 2.3.2. Ele é disparado automaticamente por meio de eventos como *push* ou *pull requests* na ramificação principal, garantindo que cada alteração no código seja analisada antes de ser integrada. Inicialmente, realiza-se o *checkout* do repositório para o ambiente de execução e uma verificação básica da estrutura do projeto, assegurando que os arquivos necessários estejam presentes. A Figura 3.1 apresenta as etapas principais do *pipeline* desenvolvido, destacando as atividades realizadas para garantir a segurança da cadeia de suprimentos de *software*.

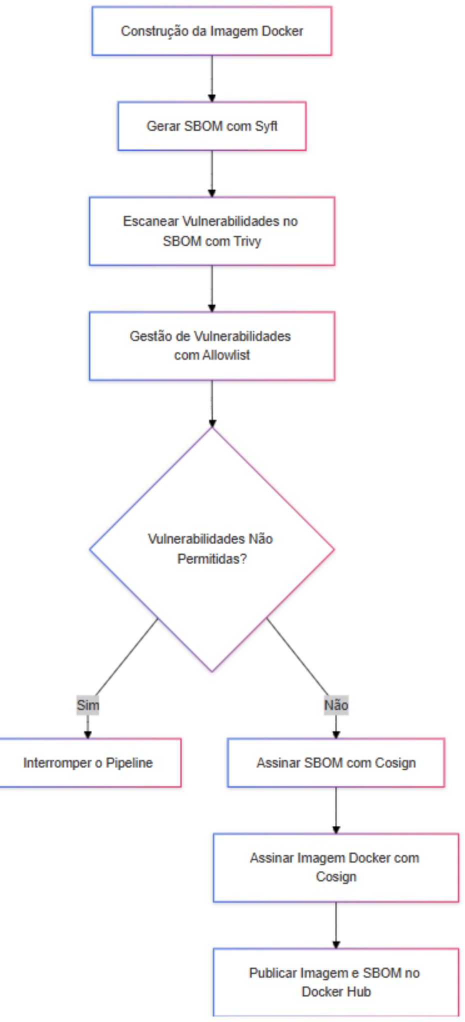

Figura 3.1: Diagrama do pipeline, Fonte: De autoria própria.

Após essa preparação, inicia-se a construção da imagem Docker. A imagem é criada localmente e então, inicia-se a geração do SBOM, utilizando a ferramenta Syft. O SBOM é exportado no formato CycloneDX. A Figura 3.2 apresenta a geração do SBOM durante a execução do *pipeline*.

```
name: Construção da Imagem Docker
       run: |
         echo "🐳 Construindo a imagem Docker..."
         docker build -t my-python-app .

     - name: Autenticar no Docker Hub
       run: |
         echo "${{ secrets.DOCKER_TOKEN }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin


     - name: Fazer Push da Imagem para o Docker Hub
       run: |
         docker tag my-python-app:latest henriquehideaki/tcc-pipeline-sbom:latest
         docker push henriquehideaki/tcc-pipeline-sbom:latest

     - name: Instalar Trivy
       run: |
         echo "🔧 Instalando o Trivy..."
         curl -sSfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin

     - name: Gerar SBOM com Trivy
       run: |
         echo "📄 Gerando SBOM para a imagem Docker..."
         trivy image --format cyclonedx --output docker-sbom.json my-python-app
```

Figura 3.2: Geração do SBOM no Pipeline Proposto. Fonte: De autoria própria.

3.2 Análise do SBOM

Com o SBOM gerado, o Trivy realiza um escaneamento para identificar vulnerabilidades conhecidas, utilizando bases de dados confiáveis como o *National Vulnerability Database* (NVD). As vulnerabilidades detectadas são categorizadas por severidade, facilitando a priorização por meio de uma visualização colorida. Essa visualização colorida agiliza o processo de priorização, permitindo que a equipe foque rapidamente nas falhas mais graves. A Figura 3.3 apresenta uma vulnerabilidades identificada pelo Trivy.

```
{
          "VulnerabilityID": "TEMP-0841856-B18BAF",
          "PkgID": "bash@5.2.15-2+b7",
          "PkgName": "bash",
          "PkgIdentifier": {
            "PURL": "pkg:deb/debian/bash@5.2.15-2%2Bb7?arch=amd64\u0026distro=debian-12.7",
            "UID": "ee9bfe8fa48be498",
            "BOMRef": "pkg:deb/debian/bash@5.2.15-2%2Bb7?arch=amd64\u0026distro=debian-12.7"
          },
          "InstalledVersion": "5.2.15-2+b7",
          "Status": "affected",
          "Layer": {},
          "SeveritySource": "debian",
          "PrimaryURL": "https://security-tracker.debian.org/tracker/TEMP-0841856-B18BAF",
          "DataSource": {
            "ID": "debian",
            "Name": "Debian Security Tracker",
            "URL": "https://salsa.debian.org/security-tracker-team/security-tracker"
          },
          "Title": "[Privilege escalation possible to other user than root]",
          "Severity": "LOW",
          "VendorSeverity": {
            "debian": 1
          }
        }
}
```


Figura 3.3: Exemplo de uma vulnerabilidades identificada pelo Trivy. Fonte: De autoria própria.

Após o escaneamento inicial, o Snyk complementa a análise de segurança ao oferecer uma avaliação detalhada das vulnerabilidades detectadas. Ele se destaca por apresentar recomendações práticas, como versões mais seguras de bibliotecas ou alternativas equivalentes para substituir componentes vulneráveis. O painel do Snyk desempenha um papel essencial nesse processo, sendo disponibilizado por meio de um link de acesso gerado durante a execução do pipeline no GitHub Actions. Nesse painel, informações detalhadas, como CVE, pontuação CVSS, maturidade da exploração e orientações de correção, são apresentadas de forma clara, facilitando a priorização e a tomada de decisões. Com essas informações integradas diretamente ao fluxo de trabalho, o processo de mitigação torna-se mais ágil e eficiente, fortalecendo a segurança do pipeline e da cadeia de suprimentos de software.

A Figura 3.4 apresenta o painel do Snyk com recomendações de mitigação de riscos.

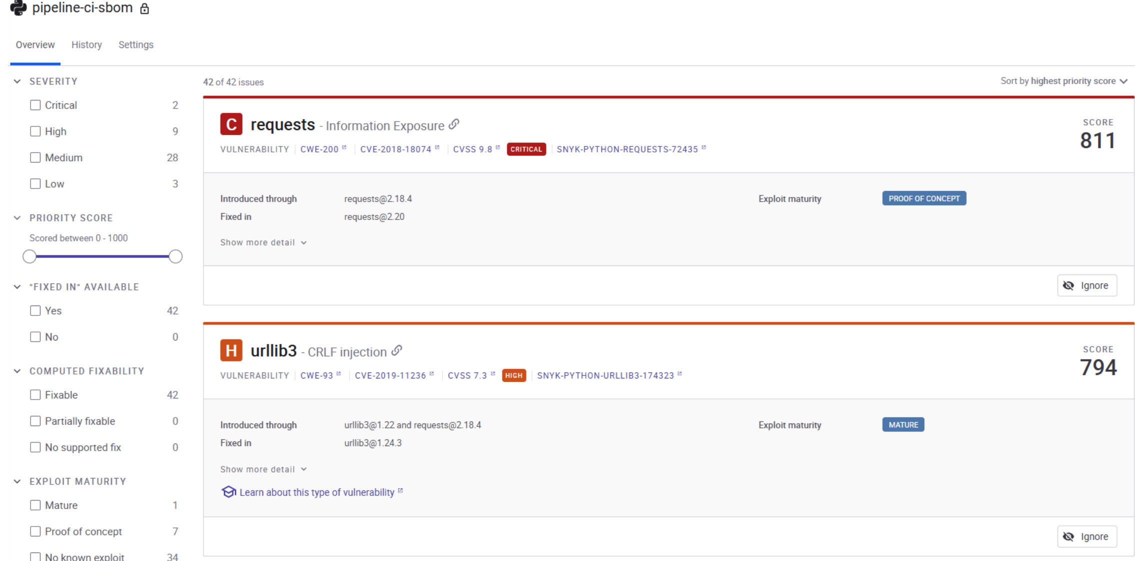

Figura 3.4: Painel do Snyk com Recomendações de Mitigação. Fonte: De autoria própria.

Para evitar interrupções desnecessárias no processo de CI foi implementado no *pipeline* um mecanismo de validação baseado em uma lista de permissões (*allow-list*). Essa lista de permissões é essencialmente composta por um arquivo de texto (formato .txt) que contém os identificadores CVE das vulnerabilidades previamente analisadas e consideradas de baixo risco. O *pipeline* cruza as vulnerabilidades encontradas com informações dessa lista: se apenas falhas registradas na lista de permissão forem detectadas, o *pipeline* prossegue normalmente; caso contrário, nas situações em que pelo menos uma das vulnerabilidades identificadas não esteja na lista de permissão, o *pipeline* é interrompido e a equipe é notificada. Essa abordagem garante que apenas problemas críticos e desconhecidos demandem atenção imediata.

A Figura 3.5 apresenta as vulnerabilidades toleradas e bloqueadas pelo *pipeline*.

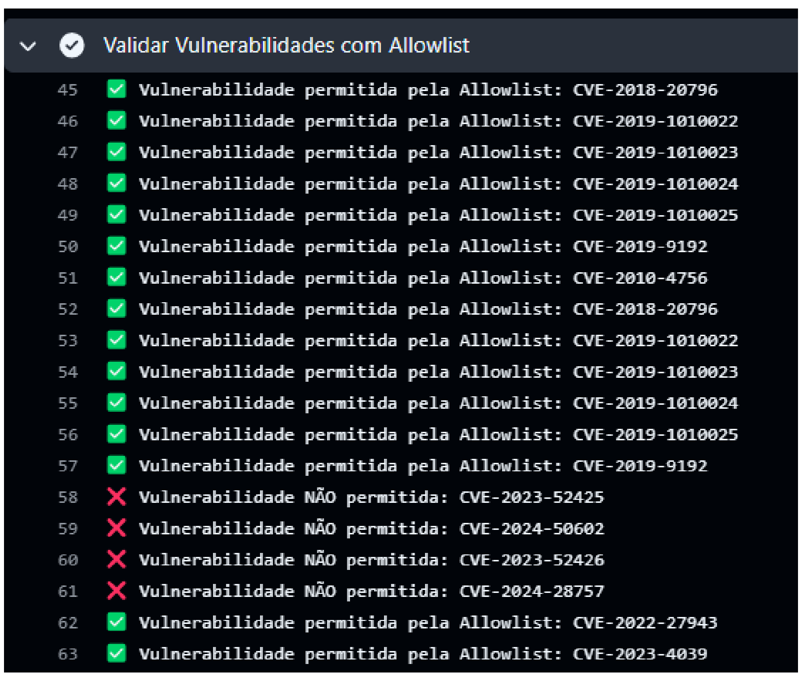

Figura 3.5: Validação Allowlist, Fonte: De autoria própria.

3.3 Integridade do SBOM e da Imagem da Aplicação

Após a geração, o SBOM é assinado digitalmente para garantir sua integridade e autenticidade. Este processo assegura que as dependências e componentes listados permanecem inalterados durante sua utilização. Com isso, evita-se a introdução de elementos maliciosos ou não autorizados na cadeia de suprimentos de software. Essa prática é essencial para promover rastreabilidade e confiança no uso dos artefatos em ambientes de desenvolvimento e produção, permitindo que organizações auditem e validem a segurança de seus componentes com maior precisão. Além disso, a assinatura possibilita a identificação rápida de alterações não autorizadas, facilitando respostas proativas a incidentes de segurança.

No *pipeline*, a assinatura do SBOM ocorre após sua geração e validação contra vulnerabilidades. O documento é assinado digitalmente e armazenado como um metadado adicional no Docker Hub, juntamente com a imagem Docker correspondente. Essa associação é feita por meio do *digest* (*hash*) único da imagem, garantindo que o SBOM permaneça vinculado à versão específica do artefato. Esse processo assegura que a integridade do SBOM seja mantida durante a distribuição e o consumo da imagem.

A verificação do SBOM ocorre no momento do consumo ou durante auditorias de segurança. Nessa etapa, consumidores ou auditores podem recuperar o SBOM do Doc-

ker Hub e validar sua assinatura digital. Isso garante que o documento não foi alterado desde sua criação e que as informações nele contidas são confiáveis. A verificação de integridade é uma medida crucial para assegurar que o SBOM é autêntico e seguro.

A Figura 3.6 apresenta a validação da assinatura digital do SBOM com o Cosign.

```
- name: Verificar a atestação do SBOM
  run: |
    echo "🔍 Verificando a atestação do SBOM..."
    cosign verify-attestation \
      --certificate-oidc-issuer=https://github.com/login/oauth \
      --type https://spdx.dev/Document \
      --certificate-identity="hideakikoga.hk@gmail.com" \
      docker.io/henriquehideaki/tcc-pipeline-sbom@sha256:e8ca7069704eb3717fd09a517d55b034a9babfadb0d403ad2c1336e3ffb7able | \
      jq -r .payload | \
      base64 -d | jq .predicate
```

Figura 3.6: Validação do SBOM. Fonte: De autoria própria.

Já a assinatura digital da imagem Docker, realizada pelo Cosign, assegura que o contêiner não sofreu alterações desde sua criação, vinculando-a ao *hash* da imagem. Isso é fundamental para proteger a cadeia de suprimentos de software, pois impede que imagens comprometidas sejam usadas em produção, minimizando riscos de ataques.

O Cosign também oferece a atestação *keyless*, que utiliza autenticação baseada na identidade do usuário, eliminando a necessidade de armazenar e gerenciar chaves privadas manualmente. Nesse processo, quando um usuário ou sistema precisa assinar um artefato, como uma imagem de contêiner, ele se autentica em um serviço de identidade confiável, como Google ou GitHub. A partir dessa autenticação, uma chave digital temporária é gerada automaticamente, usada para assinar o artefato e, em seguida, descartada. Para garantir a autenticidade e rastreabilidade, os detalhes da assinatura são armazenados em um registro público e seguro, permitindo que qualquer pessoa possa verificar sua validade a qualquer momento. Essa abordagem simplifica o processo de assinatura, garantindo segurança e rastreabilidade sem a complexidade de gerenciar chaves privadas, protegendo os artefatos contra adulterações ao longo de toda a cadeia de suprimentos de software.

A Figura 3.7 apresenta o processo de assinatura e verificação da imagem Docker.

```
- name: Assinar Imagem Docker com Cosign
  env:
    COSIGN_PASSWORD: ${{ secrets.COSIGN_PASSWORD }}
  run: |
    echo "🔏 Assinando a imagem Docker..."
    cosign sign --key cosign.key --upload=true docker.io/henriquehideaki/tcc-pipeline-sbom@sha256:e
    echo "✅ Imagem Docker assinada com sucesso."

- name: Verificar Assinatura da Imagem Docker
  run: |
    echo "🔍 Verificando assinatura da imagem Docker..."
    cosign verify --key cosign.pub             docker.io/henriquehideaki/tcc-pipeline-sbom@sha256:
    echo "✅ Assinatura da imagem Docker verificada com sucesso."
```

Figura 3.7: Assinatura da Imagem. Fonte: De autoria própria.

Um exemplo prático da importância da assinatura digital e verificação no processo CI/CD pode ser observado no uso de aplicações de código aberto amplamente utilizada, tal como, por exemplo, o sistema gerenciador de banco de dados MariaDB. Durante o *pipeline* de CI, o SBOM é gerado em seguida, tanto o SBOM quanto a imagem do contêiner do MariaDB são assinados digitalmente antes de serem enviados ao Docker Hub.

Essa assinatura garante que, ao recuperar esses artefatos, um desenvolvedor ou auditor possa verificar sua integridade. Por exemplo, ao implantar o MariaDB em produção, a assinatura do SBOM assegura que ele não foi adulterado desde sua criação. Nesse momento um especialista de segurança poderia analisar tal SBOM, sabidamente íntegro, para verificar a cadeia de suprimentos daquela imagem específica do MariaDB. Além disso, o *hash* da imagem é incluído no SBOM, e isso confirma que ele descreve precisamente aquela versão específica da imagem do MariaDB. Finalmente, a verificação da assinatura da imagem do contêiner reforça a proteção contra modificações maliciosas durante a transferência ou armazenamento.

A imagem 3.8 mostra dois procedimentos: o de assinatura (direita), onde o SBOM e a imagem do contêiner são gerados, assinados e publicados no Docker Hub, e o de verificação (esquerda), onde as assinaturas são validadas antes da implantação segura do MariaDB em produção. Fonte: De autoria própria.

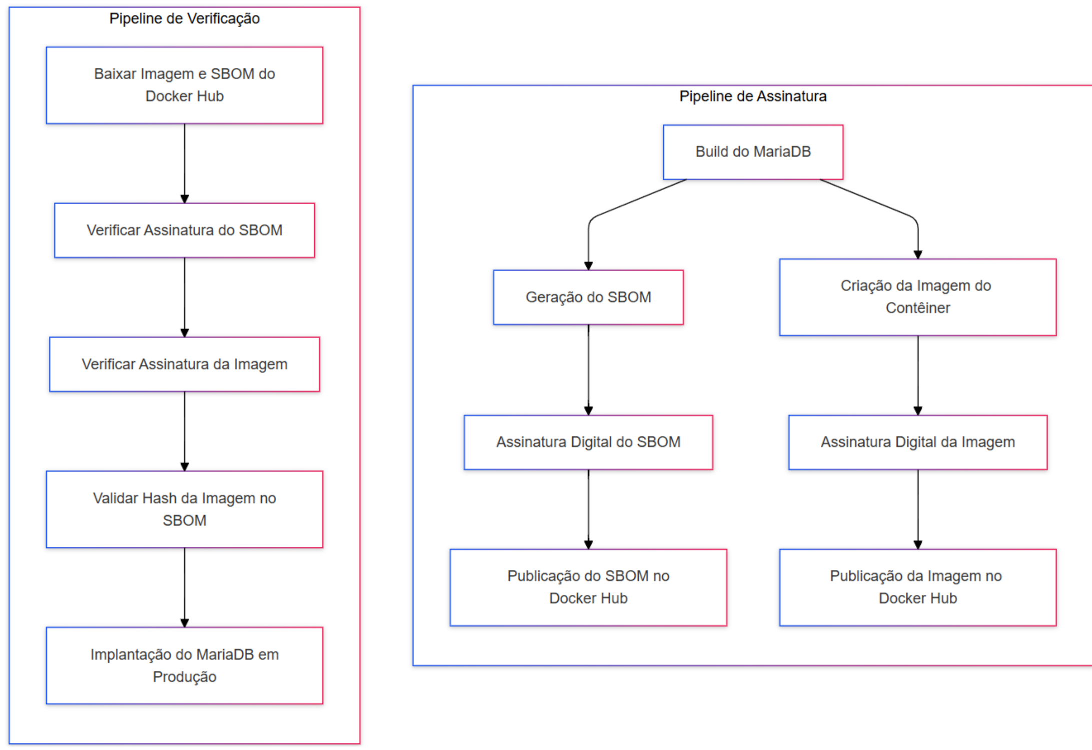

Figura 3.8: Processo de Assinatura e Verificação no Pipeline. Fonte: De autoria própria.

3.4 Considerações

O *pipeline* proposto é uma solução eficiente para proteger a cadeia de suprimentos de software. Ele integra ferramentas como Trivy, Syft, Snyk e Cosign, automatizando a detecção e correção de vulnerabilidades enquanto garante a rastreabilidade e segurança dos artefatos. Com a adoção de práticas como assinatura digital e gestão de vulnerabilidades por meio de uma lista de permissões, o *pipeline* garante que apenas componentes confiáveis sejam usados, reduzindo os riscos e protegendo o ambiente de produção. A integração dessas ferramentas facilita a identificação e a priorização de vulnerabilidades, permitindo que as equipes ajam rapidamente para corrigir problemas e aumentar a segurança. Dessa forma, o *pipeline* não só melhora a proteção da cadeia, mas também promove um desenvolvimento mais seguro e confiável.

Capítulo 4

Caso de Uso

Neste capítulo, são apresentados os principais resultados obtidos durante a implementação e execução do *pipeline* de para monitoramento de vulnerabilidades. Inicialmente, detalha-se o comportamento do *pipeline* ao lidar com uma aplicação configurada propositalmente com dependências vulneráveis, demonstrando sua capacidade de identificar, categorizar e priorizar vulnerabilidades com base em níveis de severidade. Em seguida, explora-se como as ferramentas integradas, como Trivy, Syft, Cosign e Snyk, contribuíram para fornecer informações detalhadas e recomendações práticas de mitigação. Por fim, são avaliados os benefícios alcançados, como a redução efetiva no número de vulnerabilidades, o aumento da rastreabilidade dos componentes utilizados e o impacto direto dessas práticas na segurança da cadeia de suprimentos de software.

4.1 Validação do Pipeline

A validação do *pipeline* proposto foi realizada com o objetivo de demonstrar sua eficácia na identificação e mitigação de vulnerabilidades, reforçando a segurança da cadeia de suprimentos de software. Para isso, utilizou-se uma aplicação de teste desenvolvida intencionalmente com bibliotecas vulneráveis, simulando cenários reais de riscos que poderiam comprometer o ambiente de produção.

- **Requests** (versão 2.18.4): Vulnerável a exposição de informações (CVE-2018-18074).
- **Urllib3** (versão 1.22): Afetada por CRLF Injection (CVE-2018-20060).
- **IDNA** (versão 2.6): Suscetível a falhas relacionadas à manipulação de domínios (CVE-2018-20250).
- **PyYAML** (versão 3.11): Vulnerável a execução de código remoto (CVE-2020-14343).

O *pipeline* demonstrou eficácia na identificação e categorização de vulnerabilidades nas dependências da aplicação de teste. Utilizando ferramentas como Trivy e Snyk, o **pipeline** analisou a imagem Docker gerada, identificando vulnerabilidades e destacando as mais críticas para priorização. As informações sobre categorização já discutidas no capítulo 2.2 foram aplicadas para definir as ações corretivas. Por exemplo, vulnerabilidades críticas, como a CVE-2023-45853s, foram identificadas com precisão, ressaltando a importância de abordá-las com prioridade.

A Figura 4.1 apresenta a lista de vulnerabilidades detectadas na aplicação de teste.

```
Listar Vulnerabilidades com Cores
118 Vulnerabilidade Baixa: CVE-2022-0563 (Severidade: LOW)
119 Vulnerabilidade Baixa: CVE-2022-0563 (Severidade: LOW)
120 Vulnerabilidade Crítica: CVE-2023-45853 (Severidade: CRITICAL)
121 Vulnerabilidade Alta: CVE-2023-30861 (Severidade: HIGH)
122 Vulnerabilidade Média: CVE-2020-28493 (Severidade: MEDIUM)
123 Vulnerabilidade Média: CVE-2024-22195 (Severidade: MEDIUM)
124 Vulnerabilidade Média: CVE-2024-34064 (Severidade: MEDIUM)
125 Vulnerabilidade Média: CVE-2024-56201 (Severidade: MEDIUM)
126 Vulnerabilidade Média: CVE-2024-56326 (Severidade: MEDIUM)
127 Vulnerabilidade Alta: CVE-2020-25659 (Severidade: HIGH)
128 Vulnerabilidade Alta: CVE-2023-0286 (Severidade: HIGH)
129 Vulnerabilidade Alta: CVE-2023-50782 (Severidade: HIGH)
130 Vulnerabilidade Média: CVE-2023-23931 (Severidade: MEDIUM)
131 Vulnerabilidade Média: CVE-2024-0727 (Severidade: MEDIUM)
```

Figura 4.1: Lista de Vulnerabilidades Detectadas. Fonte: De autoria própria.

O processo não se limitou apenas à detecção; o Snyk forneceu recomendações práticas para mitigar as vulnerabilidades, como a atualização para versões mais seguras de bibliotecas ou a substituição por alternativas equivalentes. Um exemplo significativo foi a redução no número total de vulnerabilidades após a atualização da imagem Docker com base nas recomendações fornecidas. Com isso, foi possível diminuir de 44 para 33 vulnerabilidades, evidenciando a eficiência do *pipeline* em aplicar correções que impactam diretamente na segurança do sistema. É importante ressaltar que para alguns casos a correção pode ser extremamente simples como, por exemplo, atualizar a versão da imagem da aplicação utilizada.

A Figura 4.2 apresenta o relatório do Snyk detalhando vulnerabilidades em dependências Python.

Rodar o Snyk para escanear vulnerabilidades no código Python

```
58   Upgrade flask@1.0.2 to flask@2.2.5 to fix
59   ✗ Information Exposure [High Severity][https://security.snyk.io/vuln/SNYK-PYTHON-FLASK-5490129] in flask@1.0.2
60     introduced by flask@1.0.2
61
62   Upgrade idna@2.6 to idna@3.7 to fix
63   ✗ Resource Exhaustion [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-IDNA-6597975] in idna@2.6
64     introduced by idna@2.6 and 2 other path(s)
65
66   Upgrade jinja2@2.10.1 to jinja2@3.1.5 to fix
67   ✗ Regular Expression Denial of Service (ReDoS) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-JINJA2-1012994] in jinja2@2.10.1
68     introduced by jinja2@2.10.1 and 1 other path(s)
69   ✗ Cross-site Scripting (XSS) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-JINJA2-6150717] in jinja2@2.10.1
70     introduced by jinja2@2.10.1 and 1 other path(s)
71   ✗ Cross-site Scripting (XSS) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-JINJA2-6809379] in jinja2@2.10.1
72     introduced by jinja2@2.10.1 and 1 other path(s)
73   ✗ Template Injection (new) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-JINJA2-8548181] in jinja2@2.10.1
74     introduced by jinja2@2.10.1 and 1 other path(s)
75   ✗ Improper Neutralization (new) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-JINJA2-8548987] in jinja2@2.10.1
76     introduced by jinja2@2.10.1 and 1 other path(s)
77
78   Upgrade requests@2.18.4 to requests@2.32.2 to fix
79   ✗ Information Exposure [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-REQUESTS-5595532] in requests@2.18.4
80     introduced by requests@2.18.4
81   ✗ Always-Incorrect Control Flow Implementation [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-REQUESTS-6928867] in requests@2.18.4
82     introduced by requests@2.18.4
83   ✗ Information Exposure [Critical Severity][https://security.snyk.io/vuln/SNYK-PYTHON-REQUESTS-72435] in requests@2.18.4
84     introduced by requests@2.18.4
85
```

Figura 4.2: Relatório Snyk Python. Fonte: De autoria própria.

A Figura 4.3 apresenta as sugestões do Snyk para atualização da imagem Docker.

Recommendations for upgrading the base image

	BASE IMAGE	VULNERABILITIES	SEVERITY
Current image	python:3.8-slim	44	1 C 1 H 0 M 42 L
Minor upgrades	python:3.14.0a1-slim	33	1 C 0 H 0 M 32 L

Show more upgrade types

View docs

Figura 4.3: Painel Snyk. Fonte: De autoria própria.

Os resultados obtidos ao utilizar o mecanismo de validação com a lista de permissões demonstram sua eficácia em gerenciar vulnerabilidades no pipeline. Em um cenário, vulnerabilidades previamente analisadas e classificadas como de baixo risco foram permitidas, permitindo que o pipeline continuasse sua execução sem interrupções. Isso evidencia como a lista de permissões é útil para evitar bloqueios desnecessários, mantendo o foco em vulnerabilidades críticas, como pode ser visto na imagem seguinte.

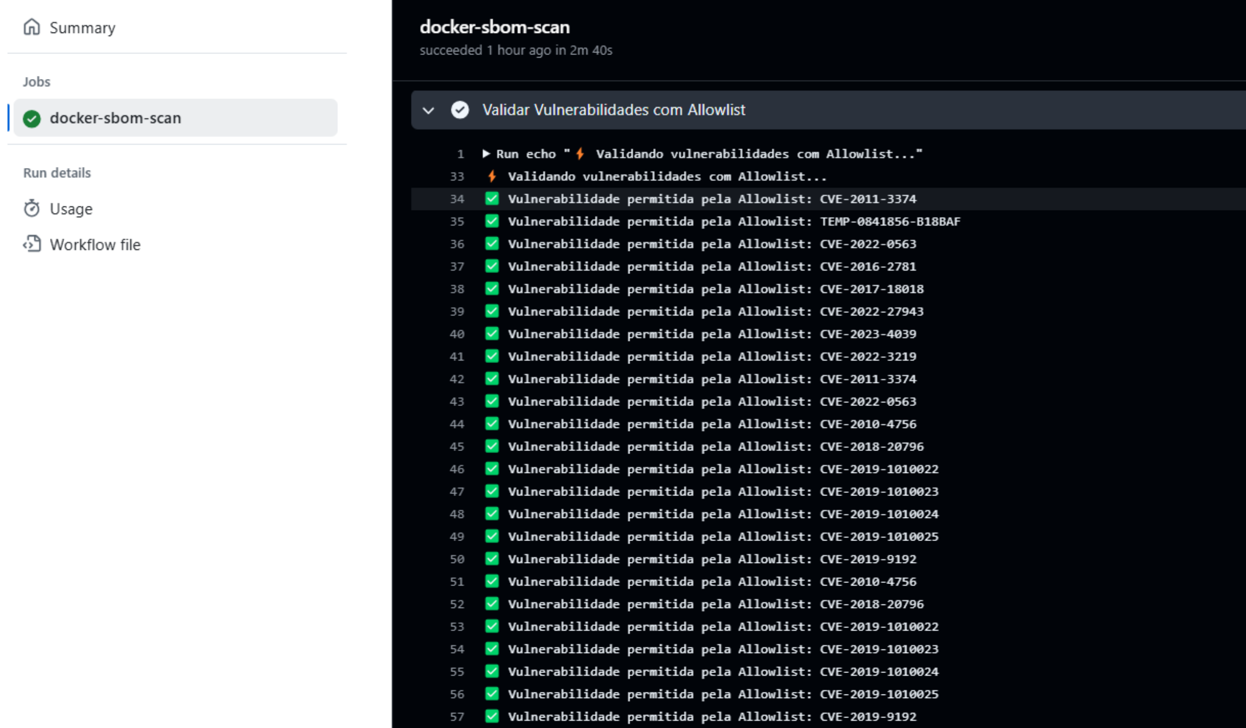

Figura 4.4: Execução com Vulnerabilidades Permitidas. Fonte: De autoria própria.

Em outro cenário, vulnerabilidades graves não registradas na lista de permissões resultaram na interrupção do *pipeline*, bloqueando a execução com uma mensagem clara indicando que essas falhas não estavam autorizadas. Esse comportamento assegura que somente componentes confiáveis avancem para as próximas etapas, protegendo o ambiente contra riscos significativos.

A Figura 4.5 apresenta o pipeline interrompido por vulnerabilidades críticas não autorizadas.

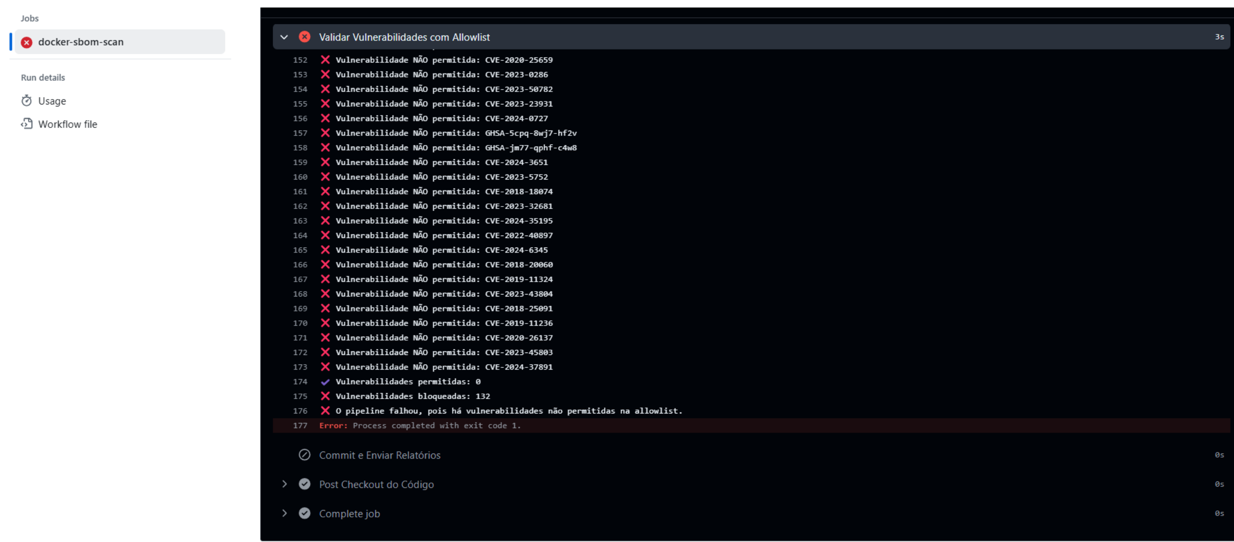

Figura 4.5: Interrupção do *Pipeline* por Vulnerabilidades Não Permitidas

Esses exemplos, apresentados com imagens anexadas, ilustram como a lista de permissões possibilita um controle flexível e seguro. Quando bem configurada, ela equilibra a continuidade do *pipeline* com a proteção contra vulnerabilidades críticas, ajustando-se às necessidades específicas do projeto.

O SBOM gerado na execução do *pipeline* foi fundamental para identificar e rastrear vulnerabilidades nas dependências do projeto. Ele permitiu mapear com precisão as bibliotecas utilizadas, associando cada vulnerabilidade às suas versões e origens, o que facilitou a priorização e a mitigação de riscos. Como resultado, vulnerabilidades críticas puderam ser detectadas, assegurando que nenhum componente vulnerável passasse despercebido. Essa rastreabilidade fortaleceu a segurança da cadeia de suprimentos de *software*, garantindo maior controle e confiabilidade nos artefatos utilizados.

Com a assinatura das imagens Docker, foi garantida a integridade dos contêineres desde sua criação, prevenindo alterações não autorizadas e assegurando que apenas artefatos confiáveis avançassem no processo. Isso contribuiu diretamente para aumentar a segurança em ambientes de produção, minimizando os riscos de ataques associados a imagens adulteradas. A assinatura digital do SBOM garantiu a autenticidade e a integridade das informações relacionadas às dependências do projeto, desde sua geração até o uso em ambientes de produção.

Esse mecanismo, aplicado tanto ao SBOM quanto às imagens Docker, reforçou a confiabilidade, autenticidade e integridade dos artefatos ao longo do ciclo de vida do *software*, garantindo que os dados de vulnerabilidades e componentes permanecessem consistentes e que apenas artefatos confiáveis avançassem no processo, promovendo maior segurança

e rastreabilidade.

4.2 Atualização de Dependências e Redução de Vulnerabilidades

No processo de melhoria contínua da segurança do *pipeline*, foi realizada a atualização das dependências com base nas recomendações geradas pelo Snyk. Durante a análise inicial, o *pipeline* utilizava bibliotecas desatualizadas, incluindo:

- **requests**: versão 2.18.4
- **flask**: versão 1.0.2
- **urllib3**: versão 1.22
- **idna**: versão 2.6
- **PyYAML**: versão 6.0.2
- **cryptography**: versão 2.3
- **jinja2**: versão 2.10.1

Essas versões apresentaram um total de 40 vulnerabilidades, incluindo 1 vulnerabilidade crítica, 9 vulnerabilidades altas e 27 vulnerabilidades médias, conforme destacado no relatório inicial. Após implementar as atualizações recomendadas pelo Snyk, as bibliotecas foram substituídas pelas seguintes versões mais recentes:

- **requests**: versão 2.32.2
- **flask**: versão 2.2.5
- **urllib3**: versão 1.26.19
- **idna**: versão 3.7
- **PyYAML**: versão 6.0.2 (mantida sem necessidade de atualização)
- **cryptography**: versão 42.0.8
- **jinja2**: versão 3.1.5

A atualização das dependências resultou na redução significativa de vulnerabilidades no *pipeline*, passando de 40 para apenas 2 (uma alta e uma média), essa redução representa uma diminuição de 95% na quantidade de vulnerabilidades, eliminando totalmente as críticas. Esse resultado destaca a importância de manter bibliotecas atualizadas para proteger a cadeia de suprimentos de *software*. Seguindo as recomendações do Snyk, foi

possível garantir maior segurança, confiabilidade dos artefatos gerados e proteção do ambiente de produção contra ataques. A Figura 4.6 destaca a redução significativa no número de vulnerabilidades após a atualização das dependências do *pipeline*.

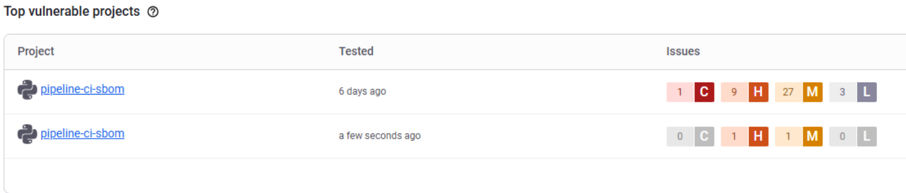

Figura 4.6: Redução de Vulnerabilidades

Tabela 4.1: Comparação de Vulnerabilidades Antes e Depois da Atualização

Severidade	**Antes da Atualização**	**Depois da Atualização**
Críticas	10	0
Altas	15	1
Médias	10	1
Baixas	5	0
Total	**40**	**2**

O *pipeline* implementado atende aos critérios do nível 3 do SLSA ao garantir controle confiável de ambientes e validação de artefatos. A assinatura digital aplicada às imagens Docker e ao SBOM assegurou a integridade e a autenticidade dos componentes gerados, eliminando o risco de alterações não autorizadas e reforçando a rastreabilidade até a origem do código. O uso do SBOM permitiu mapear dependências e identificar vulnerabilidades de forma precisa, enquanto as atestações geradas forneceram provas explícitas de que os artefatos foram criados de maneira segura. Além disso, o gerenciamento rigoroso de vulnerabilidades, com interrupção do *pipeline* em caso de falhas críticas, garantiu que apenas componentes confiáveis fossem aprovados. Esses elementos consolidam a conformidade com o nível 3 do SLSA, promovendo uma cadeia de suprimentos de software segura, rastreável e confiável.

Capítulo 5

Conclusão

A implementação do *pipeline* proposto demonstrou ser uma solução eficaz para proteger a cadeia de suprimentos de *software*, atendendo aos critérios do nível 3 do SLSA. Com o uso de ferramentas como Trivy, Syft, Snyk e Cosign, foi possível automatizar a geração de SBOMs, detectar vulnerabilidades e aplicar assinaturas digitais para garantir a integridade e autenticidade dos artefatos.

Os resultados obtidos evidenciaram uma redução significativa no número de vulnerabilidades, reforçando a importância de manter dependências atualizadas e de integrar práticas de segurança ao ciclo de vida do desenvolvimento. A adoção de uma lista de permissões permitiu gerenciar vulnerabilidades previamente analisadas, evitando interrupções desnecessárias e mantendo o foco em falhas críticas.

Além disso, a assinatura digital aplicada ao SBOM e às imagens Docker garantiu que apenas artefatos confiáveis fossem utilizados em ambientes de produção. Essa prática promoveu maior rastreabilidade, permitindo identificar e corrigir rapidamente problemas relacionados a vulnerabilidades ou alterações não autorizadas.

A abordagem proposta contribuiu significativamente para a segurança e a rastreabilidade na cadeia de suprimentos de *software*, proporcionando um processo confiável e alinhado às melhores práticas de DevSecOps. Para trabalhos futuros e possíveis melhorias, a aplicação de técnicas avançadas, como o uso de *machine learning*, pode contribuir para a detecção de ameaças ainda não catalogadas, ampliando a capacidade de proteção do *pipeline*. Outra possibilidade é a expansão do modelo para contemplar métricas de desempenho e conformidade regulatória, tornando o *pipeline* mais adaptável a ambientes tecnológicos dinâmicos e cada vez mais complexos. Além disso, o uso de modelos de linguagem de larga escala (*LLMs*) pode auxiliar na análise automatizada das vulnerabilidades encontradas, sugerindo correções e priorizando ações com base na criticidade das falhas.

Referências Bibliográficas

Abu Ishgair, Eman, Marcela S. Melara & Santiago Torres-Arias (2024), 'Sok: A defense-oriented evaluation of software supply chain security', arXiv preprint arXiv:2405.14993 . Accessed: January 2025.
URL: *https://arxiv.org/abs/2405.14993*

Amazon Web Services (2025), 'O que é devsecops?'. Accessed: 2025-01-14.
URL: *https://aws.amazon.com/pt/what-is/devsecops*

Anchore (2025), 'Syft - cli tool and library for generating sboms'. Accessed: 2025-01-13.
URL: *https://github.com/anchore/syft*

cosign, Sigstore (2025), 'Cosign - container signing, verification and storage in an oci registry'. Accessed: 2025-01-13.
URL: *https://github.com/sigstore/cosign*

DeFranco, Joanna F. & Nir Kshetri (2022), 'Software supply chains', IEEE Computer **55**(10), 16–17. Accessed: 2025-01-12.
URL: *https://csdl-downloads.ieeecomputer.org/mags/co/2022/10/09903815.pdf*

Faruk, Md Jobair Hossain, Masrura Tasnim, Hossain Shahriar, Maria Valero, Akond Rahman & Fan Wu (2022), Investigating novel approaches to defend software supply chain attacks, em '2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)', IEEE, pp. 283–288.
URL: *https://akondrahman.github.io/files/papers/issrew2022.pdf*

Foundation, Linux (2025), 'Spdx specification'. Accessed: 2025-01-13.
URL: *https://spdx.dev/specifications*

Gama, Diego, Andrey Brito, André Martin & Christof Fetzer (2024), Supporting continuous vulnerability compliance through automated identity provisioning, em '13th Latin-American Symposium on Dependable and Secure Computing (LADC 2024)', ACM, Recife, Brazil.
URL: *https://doi.org/10.1145/3697090.3697098*

Kost, Edward (2025), 'Log4shell: The log4j vulnerability emergency clearly explained', Online. Accessed: 7 January 2025.
URL: *https://www.upguard.com/blog/apache-log4j-vulnerability*

Ladisa, Piergiorgio, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez & Olivier Barais (2023), 'Journey to the center of software supply chain attacks', IEEE Security & Privacy **21**(2), 7–13.
URL: *https://dx.doi.org/10.1109/MSEC.2023.3302066*

Ltd., Snyk (2025), 'Snyk - developer security platform'. Accessed: 2025-01-13.
URL: *https://snyk.io/*

Mell, Peter, Karen Scarfone & Sasha Romanosky (2006), 'Common vulnerability scoring system', IEEE Security & Privacy **4**(6), 85–89.
URL: *https://ieeexplore.ieee.org/document/4042667*

MITRE, Corporation (2016), 'Common Vulnerabilities and Exposures (CVE) Introduction Handout'. Acesso em: 5 nov. 2024.
URL: *https://cve.mitre.org/docs/cve-intro-handout.pdf*

National Telecommunications and Information Administration (NTIA) (2021), 'Software Bill of Materials (SBOM)', `https://www.ntia.gov/page/software-bill-materials`. Acesso em: 5 dez. 2024.

npm, Inc. (2018), 'Details about the event-stream incident', Online. Accessed: 11 January 2025.
URL: *https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident*

OWASP (2025), 'Cyclonedx specification'. Accessed: 2025-01-13.
URL: *https://cyclonedx.org/specification/overview*

Security, Aqua (2025), 'Trivy documentation - sbom generation'. Accessed: 2025-01-13.
URL: *https://trivy.dev/v0.56*

Security, Xygeni (2024), Securing the future: The state of software supply chain security in 2025, Relatório técnico, Xygeni Security. Acesso em: 16 jan. 2025.
URL: *https://mautic.xygeni.io/asset/32:report-software-supply-chain-security-in-2025-pdf*

Stallings, William, Graça Bressan & Akio Barbosa (2008), Criptografia e segurança de redes ., Pearson Educacion.

The Linux Foundation, SLSA Specification v1.0 (2024), 'SLSA Specification v1.0', Online. Accessed: 5 December 2024.
URL: *https://slsa.dev/spec/v1.0/onepage*

The MITRE Corporation (2025), 'Common vulnerabilities and exposures (cve)', `https://cve.mitre.org/index.html`. Accessed: 2025-01-12.
URL: *https://cve.mitre.org/index.html*

U.S. Government Accountability Office (2022), Cybersecurity: Federal response to solarwinds and microsoft exchange incidents, Relatório técnico.
URL: *https://www.gao.gov/products/gao-22-104746*