UNIVERSIDADE DO ESTADO DA BAHIA (UNEB)
DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA, CAMPUS I
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ERNESTO SOUZA MENEZES NETO JÚNIOR

UM PROTOCOLO PARA IDENTIFICAÇÃO E MITIGAÇÃO DE VULNERABILIDADES EM APLICAÇÕES WEB DE BIOINFORMÁTICA CONTEINERIZADAS EM DOCKER

SALVADOR
2025

ERNESTO SOUZA MENEZES NETO JÚNIOR

UM PROTOCOLO PARA IDENTIFICAÇÃO E MITIGAÇÃO DE VULNERABILIDADES EM APLICAÇÕES WEB DE BIOINFORMÁTICA CONTEINERIZADAS EM DOCKER

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do Departamento de Ciências Exatas e da Terra (DCET) - Campus I, da Universidade do Estado da Bahia (UNEB), como requisito à obtenção do grau de bacharel em Sistemas de Informação.
Área de concentração: Bioinformática.

Orientador: Prof. Dr. Alexandre Rafael Lenz
Coorientador: Prof. Dr. Vagner de Souza Fonseca

SALVADOR
2025

ERNESTO SOUZA MENEZES NETO JÚNIOR

UM PROTOCOLO PARA IDENTIFICAÇÃO E MITIGAÇÃO DE VULNERABILIDADES EM APLICAÇÕES WEB DE BIOINFORMÁTICA CONTEINERIZADAS EM DOCKER

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do Departamento de Ciências Exatas e da Terra (DCET) - Campus I, da Universidade do Estado da Bahia (UNEB), como requisito à obtenção do grau de bacharel em Sistemas de Informação.
Área de concentração: Bioinformática.

Aprovada em: 09/12/2025

BANCA EXAMINADORA

Prof. Dr. Alexandre Rafael Lenz
Orientador

Prof. Dr. Vagner de Souza Fonseca
Coorientador

Prof. Dr. Diego Gervásio Frias Suarez
Examinador interno (DCET-I/UNEB)

Profa. Dra. Maria Inês Valderrama Restovic
Examinador interno (DCET-I/UNEB)

AGRADECIMENTOS

À minha família, expresso minha mais profunda gratidão. À minha avó, Maria Batista Figueira, por todo o incentivo que me fez chegar até aqui. Ao meu orientador, Prof. Dr. Alexandre Rafael Lenz, e ao meu coorientador, Prof. Dr. Vagner de Souza Fonseca, pelo apoio e orientação neste trabalho. Aos professores e colegas de curso, pela jornada compartilhada, pelos desafios superados e pelo conhecimento adquirido.

"Segurança não é um produto, mas um processo."
(Bruce Schneier)

RESUMO

O uso de contêineres Docker consolidou-se em sistemas web de bioinformática por promover a reprodutibilidade; no entanto, essa prática introduziu desafios de segurança, uma vez que processos de manutenção existentes, como o BioDockFlow, focam na estabilidade operacional, mas carecem de validação de segurança integrada. Este trabalho identificou como problema a necessidade de evoluir processos de manutenção científica para incluírem a gestão contínua de vulnerabilidades, tendo como objetivo geral desenvolver e validar uma extensão de segurança para o processo BioDockFlow, incorporando verificações automatizadas de segurança ao fluxo de Integração e Entrega Contínuas (CI/CD). A metodologia adotada foi a Design Science Research (DSR), materializando o artefato como um conjunto de pipelines de CI/CD que integram ferramentas de Teste Estático de Segurança de Aplicações (SAST) e Teste Dinâmico de Segurança de Aplicações (DAST) para a detecção automatizada de falhas. A validação ocorreu no sistema Cogumelos Luminescentes (LUMM), onde a eficácia da nova versão do processo foi mensurada pela métrica Escore Ponderado de Vulnerabilidades (EPV), proposta neste trabalho. Os resultados demonstraram que a extensão identificou 102 vulnerabilidades iniciais e validou a eficácia das mitigações aplicadas, reduzindo o risco mensurado pelo EPV em aproximadamente 64% no backend e 100% no frontend. Conclui-se que a extensão de segurança do BioDockFlow é uma solução funcional que fortalece a segurança de sistemas científicos ao impor critérios técnicos de qualidade sem comprometer a reprodutibilidade.

Palavras-chave: Docker; segurança da informação; CI/CD seguro; bioinformática; escaneamento de vulnerabilidades.

ABSTRACT

The use of Docker containers has become consolidated in bioinformatics web systems for promoting reproducibility; however, this practice has introduced security challenges, since existing maintenance processes, such as BioDockFlow, focus on operational stability but lack integrated security validation. This work identified the need to evolve scientific maintenance processes to include continuous vulnerability management as the main problem, aiming to develop and validate a security extension for the BioDockFlow process, incorporating automated security checks into the Continuous Integration and Continuous Delivery (CI/CD) workflow. The methodology adopted was Design Science Research (DSR), materializing the artifact as a set of CI/CD pipelines that integrate Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools for automated fault detection. Validation took place in the Luminescent Mushrooms (LUMM) system, where the efficacy of the new process version was measured by the Weighted Vulnerability Score (EPV) metric, proposed in this work. The results demonstrated that the extension identified 102 initial vulnerabilities and validated the efficacy of the applied mitigations, reducing the risk measured by the EPV by approximately 64% in the backend and 100% in the frontend. It is concluded that the BioDockFlow security extension is a functional solution that strengthens the security of scientific systems by imposing technical quality criteria without compromising reproducibility.

Keywords: Docker; information security; secure CI/CD; bioinformatics; vulnerability scanning.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API Interface de Programação de Aplicações (*Application Programming Interface*). 33, 69

BEM Cogumelos Comestíveis Brasileiros (*Brazilian Edible Mushrooms*). 46

CD Entrega contínua (*Continuous delivery*). 32, 37, 41, 43, 65, 67

cgroups Grupos de controle (*Control groups*). 18

CI Integração contínua (*Continuous integration*). 7, 32, 35–37, 40, 41, 43, 54, 59, 67

CI/CD Integração Contínua e Entrega Contínua (*Continuous Integration / Continuous Delivery*). 14, 15, 17, 21, 25, 27, 29, 45

CLI Interface de linha de comando (*Command Line Interface*). 19

CVE Vulnerabilidades e Exposições Comuns (*Common Vulnerabilities and Exposures*). 13, 20–22, 29, 34, 45

CVSS Sistema Comum de Pontuação de Vulnerabilidades (*Common Vulnerability Scoring System*). 26, 27

DAST Teste Dinâmico de Segurança de Aplicações (*Dynamic Application Security Testing*). 14, 21, 22, 25, 32, 33, 37, 40–43, 46, 69, 71

DevSecOps Desenvolvimento, Segurança e Operações (*Development, Security and Operations*). 13–15, 20, 21, 25, 26, 29, 43–45

DSR Pesquisa em Design Científico (*Design Science Research*). 15, 24, 29, 42, 43

EPV Escore Ponderado de Vulnerabilidades. 15, 26, 27, 32, 35–39, 41–44, 46, 54, 73

G2BC Grupo de Pesquisa em Bioinformática e Biologia Computacional da UNEB. 16, 22, 33, 46

GHCR Registo de Contêineres do GitHub (*GitHub Container Registry*). 33, 37, 65

HTTP Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*). 22, 29, 40, 71

LUMM Cogumelos Luminescentes (*Luminescent Mushrooms*). 14, 16, 22, 26, 27, 29, 33, 35, 38, 39, 42–44, 51, 52

NVD Banco de Dados Nacional de Vulnerabilidade (*National Vulnerability Database*). 21, 45

OSV Vulnerabilidades de código aberto (*Open Source Vulnerabilities*). 21, 45

OWASP Projeto Aberto Mundial para Segurança de Aplicações (*Open Worldwide Application Security Project*). 29, 33, 38, 46

SAST Teste Estático de Segurança de Aplicações (*Static Application Security Testing*). 14, 21, 25, 31, 40–43

XSS *Cross-Site Scripting*. 29

SUMÁRIO

1 INTRODUÇÃO

O avanço das tecnologias de virtualização e a crescente necessidade de ambientes computacionais escaláveis, portáteis e reprodutíveis transformaram a conteinerização em uma das principais abordagens na engenharia de software moderna (Foundation, 2023). Nesse contexto, o Docker consolidou-se como a ferramenta amplamente utilizada, permitindo o empacotamento e a execução de aplicações de maneira isolada, eficiente e previsível (Merkel, 2014; Baresi *et al.*, 2023; Foundation, 2023). A capacidade de encapsular dependências e garantir consistência entre diferentes ambientes tornou a conteinerização essencial tanto para a indústria quanto para a pesquisa científica (Wilson *et al.*, 2025).

Entre os domínios que mais se beneficiam dessa tecnologia, destaca-se a bioinformática, cuja natureza interdisciplinar demanda a integração de dados biológicos, algoritmos e infraestrutura computacional heterogênea (Boettiger, 2015). Aplicações web de bioinformática exigem não apenas desempenho e escalabilidade, mas também reprodutibilidade e confiabilidade, características fundamentais em experimentos científicos. Nesse sentido, o uso de contêineres tem viabilizado a padronização de *pipelines* de análise e a distribuição de ferramentas científicas complexas em larga escala (Grüning *et al.*, 2018). Contudo, a adoção dessa abordagem traz também desafios relacionados à segurança da informação, uma vez que a modularidade e a facilidade de compartilhamento de imagens podem introduzir vulnerabilidades e dependências inseguras (Baresi *et al.*, 2023; Haque; Babar, 2022).

Estudos como Haque e Babar (2022), Alghawli e Radivilova (2024), Jarkas *et al.* (2025) apontam que uma parcela significativa das imagens Docker disponíveis publicamente contém Vulnerabilidades e Exposições Comuns (CVEs), resultantes de bibliotecas desatualizadas, configurações incorretas ou más práticas de desenvolvimento. Tais vulnerabilidades comprometem a integridade, a confidencialidade e a disponibilidade dos sistemas, podendo afetar diretamente a confiabilidade de aplicações científicas que processam dados sensíveis. Essa problemática torna evidente a necessidade de integrar práticas de segurança ao ciclo de vida das aplicações conteinerizadas, alinhando-se ao paradigma de Desenvolvimento, Segurança e Operações (*Development, Security and Operations*) (DevSecOps), que preconiza a segurança como parte fundamental e contínua do processo de desenvolvimento.

Apesar da evolução das ferramentas de segurança voltadas para ambientes conteinerizados, como Trivy, Clair e Grype, ainda há dificuldades na adoção prática dessas soluções em *pipelines* científicos (Security, 2025; Quay, 2025; Anchore, 2025). Em geral, as equipes de bioinformática priorizam a reprodutibilidade e a estabilidade do ambiente em detrimento da atualização de pacotes e bibliotecas, o que cria um dilema entre manutenção e segurança (Boettiger, 2015; Haque; Babar, 2022). A carência de protocolos acessíveis e

bem documentados que conciliem esses dois aspectos evidencia uma lacuna que precisa ser preenchida para fortalecer a segurança dos ecossistemas científicos conteinerizados.

Visando reduzir esse hiato, Benevides (2024) propôs o BioDockFlow, um processo de manutenção baseado em contêineres Docker voltado a aplicações web heterogêneas de bioinformática. O BioDockFlow introduziu um modelo para a manutenção de sistemas científicos, estruturando o ciclo de vida de aplicações conteinerizadas sob princípios de modularidade e reprodutibilidade. A proposta alcançou resultados relevantes ao garantir a padronização e estabilidade de ambientes, mas não contemplava mecanismos específicos de segurança, como escaneamento de vulnerabilidades e mitigação automatizada.

Diante dessa limitação, este trabalho propõe um protocolo de segurança, uma extensão do processo BioDockFlow que incorpora etapas de segurança, escaneamento e mitigação de vulnerabilidades em imagens Docker. O protocolo é fundamentado nos princípios de DevSecOps e na integração contínua de ferramentas de análise automatizada, buscando identificar, avaliar e corrigir vulnerabilidades antes que cheguem ao ambiente de produção. A proposta introduz uma camada adicional ao fluxo original, composta por três fases principais: (i) escaneamento automatizado de imagens, (ii) avaliação das vulnerabilidades por meio da métrica Escore Ponderado de Vulnerabilidades (EPV) e (iii) mitigação manual ou automatizada conforme o nível de severidade. Essa integração visa garantir um ciclo de desenvolvimento e manutenção mais seguro, sem comprometer a eficiência e a reprodutibilidade já alcançadas pelo BioDockFlow.

O objetivo geral deste trabalho é propor, implementar e avaliar experimentalmente uma extensão de segurança para o processo de manutenção BioDockFlow, integrando ferramentas de análise estática e dinâmica ao fluxo de Integração Contínua e Entrega Contínua (*Continuous Integration / Continuous Delivery*) (CI/CD) para automatizar a detecção e sistematizar o fluxo de correção de vulnerabilidades em contêineres.

Como objetivos específicos, destacam-se:

- Mapear as lacunas de segurança nas fases originais do processo BioDockFlow;
- Integrar ferramentas de análise estática Teste Estático de Segurança de Aplicações (*Static Application Security Testing*) (SAST) e dinâmica Teste Dinâmico de Segurança de Aplicações (*Dynamic Application Security Testing*) (DAST) ao fluxo de trabalho do BioDockFlow via *pipelines* de CI/CD;
- Estabelecer critério de bloqueio (*security gate*) baseados na severidade das vulnerabilidades para impedir a integração de artefatos inseguros;
- Validar a eficácia técnica da extensão de segurança proposta para o BioDockFlow, mensurando a redução da superfície de ataque no sistema Cogumelos Luminescen-

tes (*Luminescent Mushrooms*) (LUMM) através da métrica Escore Ponderado de Vulnerabilidades (EPV).

A relevância do protocolo manifesta-se em duas dimensões complementares. No âmbito prático, o protocolo oferece uma solução baseada em ferramentas de código aberto e padronizadas, mitigando barreiras de custo para equipes de pesquisa que utilizam Docker, ao passo que promove a redução da superfície de ataque por meio da gestão contínua de vulnerabilidades conhecidas. No âmbito científico, o trabalho contribui para a sistematização da segurança automatizada em *pipelines* de CI/CD para aplicações científicas, adaptando conceitos técnicos de DevSecOps às especificidades da bioinformática. Além disso, reforça a viabilidade de incorporar requisitos de segurança ao fluxo de manutenção preservando a estabilidade do ambiente operacional.

Este trabalho está estruturado em seis capítulos. O Capítulo 1 apresenta a introdução e contextualização da pesquisa, com destaque para o problema, objetivos e justificativas. O Capítulo 2 expõe o referencial teórico, abordando os fundamentos de conteinerização, segurança da informação e o processo BioDockFlow. O Capítulo 3 descreve a metodologia adotada, baseada na Pesquisa em Design Científico (*Design Science Research*) (DSR). O Capítulo 4 detalha o desenvolvimento do protocolo. O Capítulo 5 apresenta e discute os resultados obtidos na aplicação e avaliação do protocolo. Por fim, o Capítulo 6 apresenta as considerações finais e sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta os fundamentos teóricos e conceituais que sustentam esta pesquisa, abrangendo os temas relacionados às aplicações web de bioinformática, à reprodutibilidade computacional, à engenharia de software, à conteinerização com Docker e às práticas de segurança integradas ao ciclo de desenvolvimento. O objetivo é demonstrar como esses fundamentos se inter-relacionam, construindo uma base conceitual que justifica a necessidade de protocolos unificados que conciliem reprodutibilidade, manutenção, identificação e mitigação de vulnerabilidades em sistemas científicos.

2.1 Aplicações Web de Bioinformática

A bioinformática é uma área interdisciplinar que combina biologia, ciência da computação e estatística para a análise e interpretação de dados biológicos em larga escala. O avanço dessa área tem sido impulsionado pelo desenvolvimento de aplicações web, que permitem a execução de fluxos de trabalho complexos em ambientes distribuídos e acessíveis, facilitando a colaboração entre grupos de pesquisa (Grüning *et al.*, 2018; Boettiger, 2015). Essas aplicações reúnem bancos de dados, algoritmos e interfaces gráficas, permitindo que pesquisadores executem tarefas como montagem genômica, anotação e análise filogenética diretamente em navegadores.

A reprodutibilidade computacional tornou-se um dos pilares da ciência moderna, permitindo que experimentos sejam auditados e reaplicados em diferentes contextos (Boettiger, 2015). Entretanto, em bioinformática, a diversidade de ferramentas, bibliotecas e versões de sistemas operacionais torna essa reprodutibilidade desafiadora (Grüning *et al.*, 2018). Cada aplicação possui requisitos específicos, e pequenas variações de ambiente podem gerar resultados diferentes, comprometendo a confiabilidade científica.

Para enfrentar esse desafio, surgiram plataformas como o Galaxy Project e o BioContainers, que oferecem interfaces web e imagens pré-configuradas para execução de análises de forma padronizada (Grüning *et al.*, 2018; Community, 2024). Essas iniciativas contribuem para a chamada ciência aberta, promovendo transparência e compartilhamento de métodos. No contexto local, o Grupo de Pesquisa em Bioinformática e Biologia Computacional da UNEB (G2BC) desenvolve aplicações como o LUMM, que demandam reprodutibilidade, rastreabilidade e segurança de dados, reforçando a importância de sistemas web confiáveis.

Apesar dos avanços, aplicações científicas ainda enfrentam desafios quanto à manutenção, escalabilidade e segurança. Muitas delas são desenvolvidas em ambientes acadêmicos com recursos limitados, o que dificulta a adoção de práticas sistemáticas de atualização

e monitoramento (Her *et al.*, 2024; Jarkas *et al.*, 2025). Nesse cenário, tecnologias como a conteinerização têm se destacado por viabilizar ambientes padronizados e controlados, favorecendo a reprodutibilidade e a manutenção contínua das aplicações científicas.

2.2 Reprodutibilidade e Engenharia de Software

A reprodutibilidade, entendida como a capacidade de repetir um experimento obtendo os mesmos resultados, é essencial para a credibilidade científica. No contexto computacional, ela envolve a preservação do código, das dependências e dos ambientes de execução (Boettiger, 2015; Grüning *et al.*, 2018). A engenharia de software, por sua vez, oferece o conjunto de métodos e práticas que permitem alcançar essa previsibilidade e estabilidade, ao padronizar processos de desenvolvimento, testes e manutenção (Sommerville, 2011; Pressman; Maxim, 2014).

O uso de ferramentas de controle de versão, como o Git, e CI/CD, tornou-se essencial tanto na indústria de software quanto em ambientes científicos. Essas práticas possibilitam o rastreamento de alterações, a execução automatizada de testes e a implantação segura de novas versões. Ao adotar esses mecanismos, grupos de pesquisa podem reduzir falhas e aumentar a confiabilidade dos resultados obtidos por meio de *pipelines* computacionais.

Em bioinformática, a aplicação de conceitos de engenharia de software promove ganhos significativos em rastreabilidade, documentação e automação de análises. Projetos com múltiplas dependências e linguagens passam a contar com ambientes controlados e versões bem definidas, o que facilita a manutenção e o reuso (Benevides, 2024; Boettiger, 2015; Haque; Babar, 2022). Essa integração entre engenharia e ciência estabelece um alicerce para a reprodutibilidade sustentável, conceito que alia estabilidade técnica e segurança operacional, assegurando que resultados possam ser reproduzidos ao longo do tempo, mesmo em cenários de atualização contínua.

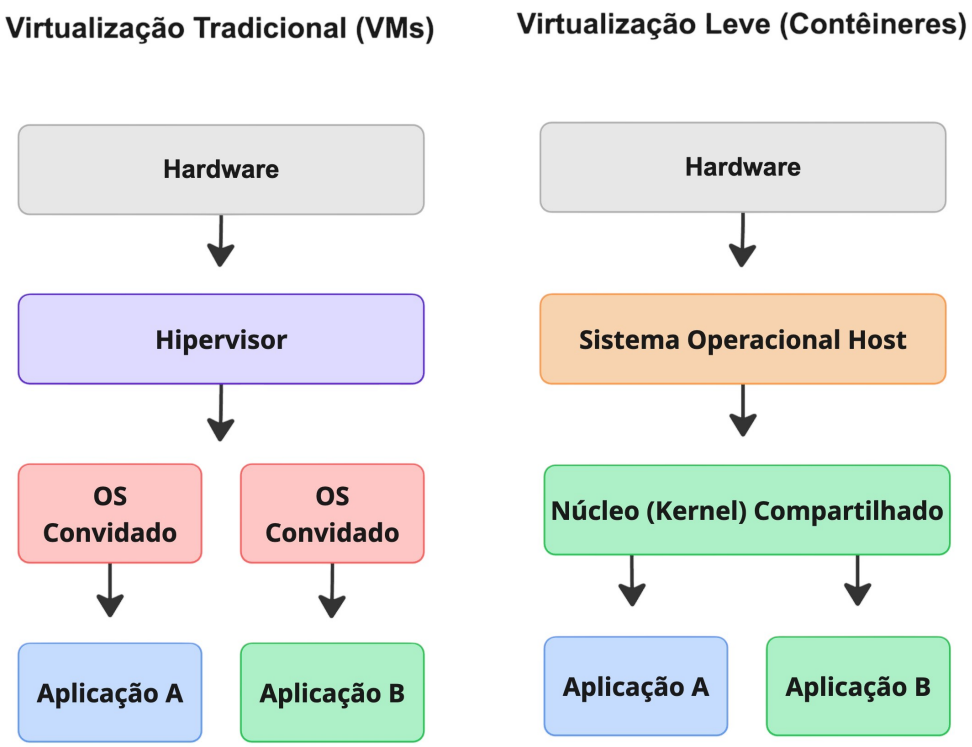
Assim, a reprodutibilidade computacional e a engenharia de software se complementam: enquanto a primeira assegura a validade científica, a segunda provê a metodologia necessária para manter essa validade em ambientes dinâmicos. Essa relação é central para o desenvolvimento de sistemas que conciliem qualidade, transparência e segurança.

2.3 Virtualização Leve e Conteinerização

Com o avanço da conteinerização, sobretudo impulsionado pelo Docker, houve uma transformação nas formas de empacotar, distribuir e executar aplicações (Docker Inc., 2016; Baresi *et al.*, 2023). Esse paradigma, no entanto, introduz novos desafios no campo da segurança da informação (Haque; Babar, 2022; Alghawli; Radivilova, 2024). Para compreender esses desafios, é importante entender os fundamentos técnicos da virtualização leve, base da conteinerização moderna.

A virtualização leve, também chamada de virtualização em nível de sistema operacional, é uma técnica que permite a execução de múltiplas instâncias de aplicações isoladas utilizando o mesmo núcleo do sistema operacional (Soltesz *et al.*, 2007; Felter *et al.*, 2015). Esse modelo é suportado por tecnologias como *namespaces* e Grupos de controle (*Control groups*) (cgroups) no *kernel* Linux, que garantem o isolamento e controle de recursos entre processos (Rosen, 2015). Ao contrário da virtualização tradicional baseada em hipervisores, que emula hardware completo para executar múltiplos sistemas operacionais independentes, a virtualização leve compartilha diretamente o núcleo (*kernel*) do sistema hospedeiro, tornando-a mais eficiente (Nimmagadda *et al.*, 2018). A Figura 1 ilustra essa diferença fundamental entre a virtualização tradicional e o modelo de conteinerização, evidenciando o compartilhamento do *kernel* no último.

Figura 1 – Comparação entre máquinas virtuais e contêineres



Fonte: Elaborado pelo autor (2025), com base na documentação do Docker. (Docker Inc., 2016).

2.4 Arquitetura e Funcionamento do Docker

O Docker é uma plataforma de conteinerização que automatiza o processo de empacotamento, distribuição e execução de aplicações em contêineres (Docker Inc., 2016). Lançado em 2013, ele popularizou o conceito de virtualização leve ao oferecer uma interface padronizada para o gerenciamento de contêineres baseados em Linux (Docker Inc., 2016; Foundation, 2023).

A arquitetura do Docker segue um modelo cliente-servidor e é composta por componentes que juntos gerenciam os contêineres (Docker Inc., 2016). São eles:

1. **O Daemon Docker (`dockerd`):** É o serviço persistente que escuta as solicitações da API Docker e gerencia os objetos Docker, como imagens, contêineres, redes e volumes (Docker Inc., 2016). O daemon quem realiza o trabalho de construir, executar e monitorar os contêineres.

2. **A API REST:** Define a interface programática que os clientes utilizam para interagir com o daemon. As requisições podem ser feitas via `curl`, bibliotecas específicas ou pelo cliente oficial.

3. **O Cliente Docker (`docker`):** É a Interface de linha de comando (*Command Line Interface*) (CLI) e a principal forma de interação do usuário com o Docker (Docker Inc., 2016). Quando comandos como `docker run` ou `docker build` são executados, o cliente envia essas instruções para o `dockerd` através da API REST (Docker Inc., 2016). O cliente pode se comunicar com um ou mais daemons.

4. **Docker Registries:** São repositórios para armazenar e distribuir imagens Docker. O Docker Hub é o registro público padrão, mas registros privados também são amplamente utilizados. O comando `docker pull` baixa imagens de um registro, enquanto `docker push` envia imagens para um registro (Docker Inc., 2016).

Um dos conceitos fundamentais no ecossistema Docker é o de imagem. Uma imagem Docker é uma estrutura imutável que contém tudo o que uma aplicação precisa para ser executada: código-fonte, bibliotecas, dependências, arquivos de configuração e variáveis de ambiente (Docker Inc., 2024). Essas imagens são criadas a partir de um arquivo de instruções chamado Dockerfile (ver Apêndice A), que define, de forma declarativa, todas as etapas necessárias para montar o ambiente de execução (Docker Inc., 2016).

O Dockerfile é composto por uma sequência de instruções, como a definição da imagem base (`FROM`), instalação de pacotes (`RUN`), cópia de arquivos (`COPY`) e configuração de comandos de inicialização (`CMD` ou `ENTRYPOINT`) (Docker Inc., 2024). Cada instrução no Dockerfile resulta em uma camada (layer) na imagem final (Docker Inc., 2024). Esse modelo de camadas traz importantes benefícios, como a reutilização de conteúdo entre *builds* e a eficiência no uso de cache, permitindo compilações mais rápidas e modulares (Merkel, 2014; Nimmagadda *et al.*, 2018).

Na prática, contêineres são instâncias em execução dessas imagens (Docker Inc., 2024). O Docker Engine cria uma instância isolada da imagem, que roda como um processo no sistema operacional hospedeiro, mantendo seu próprio sistema de arquivos, variáveis de ambiente e configurações de rede (Docker Inc., 2016). Essa abordagem garante que a

aplicação conteinerizada se comporte de maneira idêntica, independentemente do ambiente subjacente (Alghawli; Radivilova, 2024).

2.5 Riscos e o Paradigma DevSecOps

A adoção do Docker, apesar de solucionar o problema da reprodutibilidade, introduz novos desafios de segurança (Baresi *et al.*, 2023). Diferentemente das máquinas virtuais que oferecem isolamento completo, os contêineres compartilham o mesmo núcleo (*kernel*) do sistema operacional hospedeiro. Isso aumenta a superfície de ataque e torna viável a ocorrência de falhas como o *container escape*, onde um processo malicioso "escapa"do contêiner e obtém acesso ao *host* (Jarkas *et al.*, 2025; Her *et al.*, 2024).

As vulnerabilidades em ambientes Docker são frequentemente categorizadas em quatro domínios principais: infraestrutura, aplicação, rede e operações (Jarkas *et al.*, 2025).

No nível da infraestrutura, riscos incluem o uso de contêineres privilegiados (`--privileged`), montagem insegura de volumes (como `/var/run/docker.sock`) e falhas de isolamento entre contêineres (Jarkas *et al.*, 2025).

Na camada de aplicação, o principal risco é o uso de imagens base (ex: Ubuntu, Alpine) que contêm bibliotecas desatualizadas com vulnerabilidades conhecidas (CVEs) (Haque; Babar, 2022). Como aponta Haque e Babar (2022), é comum que imagens Docker públicas contenham bibliotecas vulneráveis, mesmo quando são aparentemente seguras.

No contexto da bioinformática, a origem de muitas vulnerabilidades não está apenas na complexidade técnica, mas em fatores organizacionais e culturais. Equipes de desenvolvimento em projetos de bioinformática frequentemente são compostas por pesquisadores com formação em biologia ou estatística, e não por especialistas em engenharia de software (Boettiger, 2015). Essa realidade contribui para a adoção de práticas de codificação pouco rigorosas e a falta de atualização contínua de bibliotecas (Alghawli; Radivilova, 2024; Haque; Babar, 2022). Além disso, há uma forte pressão por reprodutibilidade, o que leva ao uso de ambientes *congelados* e imagens antigas, muitas vezes com pacotes vulneráveis (Boettiger, 2015; Haque; Babar, 2022).

Para enfrentar esses riscos, a integração da segurança em todas as etapas do desenvolvimento de software deu origem ao paradigma DevSecOps (OWASP Foundation, 2023). Essa abordagem enfatiza o conceito de *shift-left security*, que consiste em antecipar práticas de segurança para as fases iniciais do ciclo de vida do software, como o *build* da imagem Docker, reduzindo o custo e o impacto de correções posteriores (Pressman; Maxim, 2014).

2.5.1 Ferramentas e Estratégias de Escaneamento

A segurança em ambientes conteinerizados exige a adoção de mecanismos preventivos que atuem antes mesmo da execução das aplicações (Haque; Babar, 2022). Nesse cenário, o escaneamento automatizado de imagens Docker tem se destacado como uma das abordagens eficazes para a identificação e mitigação de vulnerabilidades conhecidas, especialmente aquelas catalogadas em bases como o CVE (Haque; Babar, 2022).

O processo de escaneamento consiste na análise das camadas que compõem uma imagem Docker, rastreando assinaturas de vulnerabilidades em repositórios públicos como o Banco de Dados Nacional de Vulnerabilidade (*National Vulnerability Database*) (NVD) e Vulnerabilidades de código aberto (*Open Source Vulnerabilities*) (OSV) (Haque; Babar, 2022). Ferramentas como Trivy, Clair e Grype realizam essa tarefa de forma automatizada, verificando a presença de falhas conhecidas em sistemas operacionais base, bibliotecas de terceiros e pacotes instalados (Security, 2025; Quay, 2025; Anchore, 2025). A vantagem desse modelo é a possibilidade de integração direta ao ciclo de desenvolvimento via *pipelines* CI/CD, promovendo o alinhamento com a filosofia DevSecOps (Foundation, 2023).

A Tabela 1 apresenta uma comparação entre as ferramentas Trivy, Clair e Grype com base em seus aspectos técnicos e funcionais. A análise foi elaborada com base nas documentações oficiais das ferramentas (Security, 2025; Quay, 2025; Anchore, 2025).

Tabela 1 – Comparação entre ferramentas de escaneamento de imagens Docker

Ferramenta	**Licença**	**CVEs**	**CI/CD**	**Linguagens Suportadas**	**Tipo de Detecção**	**Destaques**
Trivy	Código aberto	Diária	Alta	Python, R, Bash, Node.js	CVEs, configurações inseguras, segredos	Leve, rápido e com escaneamento profundo de imagens, arquivos e pacotes
Clair	Código aberto	Regular	Alta	Foco em sistemas operacionais	CVEs apenas	Arquitetura escalável, integração via API, mas análise limitada a pacotes do sistema
Grype	Código aberto	Diária	Alta	Python, Java, Node.js	CVEs, algumas configurações	Alta precisão e suporte a múltiplos formatos, integração com Syft para inspeção detalhada

Fonte: Elaborado pelo autor (2025), com base em: Aqua Security (Security, 2025); Quay (Quay, 2025); Anchore (Anchore, 2025).

2.6 Abordagens de Análise de Segurança: SAST e DAST

Dentro do paradigma DevSecOps, a detecção de vulnerabilidades ocorre principalmente por meio de duas abordagens complementares: o SAST e o DAST (OWASP Foundation, 2023).

O SAST, examina o código-fonte, binários ou, no contexto de contêineres, as camadas da imagem e arquivos de configuração em repouso, sem a necessidade de executar a aplicação (Chess; West, 2007). Segundo Pressman e Maxim (2014), essa abordagem

permite a identificação precoce de CVEs, segredos expostos e más configurações ainda na fase de construção (*build*), reduzindo o custo de correção.

Por outro lado, o DAST, avalia a aplicação enquanto ela está em execução (Maniraj; Ranganathan; Sekar, 2024). Ele simula ataques externos para identificar falhas que só se manifestam em tempo de execução (*runtime*), como problemas de configuração de cabeçalhos Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*) (HTTP), falhas de autenticação e injeção de código que dependem da interação com o servidor.

A combinação dessas duas técnicas é essencial para uma cobertura de segurança abrangente, mitigando riscos tanto na estrutura estática do artefato quanto no comportamento dinâmico da aplicação implantada (Haq *et al.*, 2024).

2.7 Trabalhos Correlatos

A revisão de estudos relacionados demonstra que a combinação entre reprodutibilidade, manutenção e segurança em ambientes conteinerizados tem sido abordada sob diferentes perspectivas, ainda que de forma fragmentada.

O BioDockFlow, proposto por Benevides (2024), representa um avanço positivo na sistematização de processos de manutenção de aplicações web de bioinformática. Atualmente em sua 3ª versão, o processo estabelece fases, atividades e um fluxo de trabalho estruturado para guiar a manutenção e a conteinerização, já tendo sido aplicado na sustentação de ferramentas desenvolvidas pelo G2BC, como o próprio sistema LUMM. A versão atual do processo, bem como sua documentação e guias de uso, encontram-se disponíveis publicamente no repositório oficial do grupo no GitHub (https://github.com/G2BC/BioDockFlow).

Contudo, embora o BioDockFlow tenha estruturado de maneira eficiente a manutenção, a estabilidade e a reprodutibilidade dos ambientes, a segurança da informação não foi abordada de forma central em seu escopo original.

No domínio específico da segurança em contêineres, diversos estudos fornecem a base técnica e empírica para a identificação e mitigação de riscos:

- O estudo de Haque e Babar (2022) oferece uma visão empírica da paisagem de vulnerabilidades em imagens base, demonstrando que a simples presença de bibliotecas desatualizadas pode comprometer a segurança da aplicação. Os autores reforçam que a ausência de validação contínua das imagens agrava esses riscos, dificultando a correção preventiva de falhas.

- Jarkas *et al.* (2025) fornecem um panorama abrangente, categorizando mais de 200 vulnerabilidades em contêineres e analisando 47 tipos de exploração. No entanto, os

autores reconhecem que a maioria dos resultados ainda carece de validação prática em ambientes de produção.

- Her *et al.* (2024) apresentam o *KUBEROSY*, um sistema proativo de mitigação baseado em filtragem dinâmica de chamadas de sistema, atuando na prevenção de ataques como o *container escape*. Contudo, os testes indicam que a sobrecarga introduzida pode impactar significativamente a performance de aplicações sensíveis, o que é um fator crítico em fluxos de trabalho de bioinformática.

A análise dos trabalhos correlatos evidencia, portanto, uma lacuna central: embora existam avanços na padronização da manutenção (*BioDockFlow*) e na análise técnica de vulnerabilidades e defesa, a literatura carece de um protocolo prático que unifique esses pilares. O desafio identificado no estado da arte não é tratar reprodutibilidade, manutenção e segurança de forma isolada, mas sim conciliá-los em um fluxo unificado e operacionalmente leve, especialmente no contexto de aplicações de bioinformática. Identifica-se, assim, a necessidade de uma proposta que estenda processos de manutenção já existentes, como o *BioDockFlow*, para incorporar de forma sistemática a verificação de vulnerabilidades e a mitigação contínua de riscos, equilibrando a profundidade da análise com a viabilidade técnica e a ausência de *overhead* excessivo.

3 METODOLOGIA DE PESQUISA

É fundamental distinguir o escopo deste trabalho em relação à proposta original do BioDockFlow desenvolvida por Benevides (2024). O BioDockFlow foi concebido como um processo de manutenção de software focado na padronização de ambientes e na agilidade de correções funcionais, priorizando a disponibilidade e a manutenibilidade dos contêineres.

A proposta desse trabalho, evolui o processo original ao integrar explicitamente práticas de segurança. A principal alteração consiste na inserção de um *Security Gate* automatizado entre as etapas de implementação e implantação. Enquanto a versão original permitia a publicação direta de contêineres funcionais, a versão atualizada condiciona a implantação à aprovação no protocolo de segurança, transformando um fluxo de manutenção puramente corretiva em um processo de manutenção segura.

Este trabalho adotou uma abordagem de pesquisa aplicada, utilizando o método DSR por ser adequado a estudos que envolvem a criação e avaliação de soluções práticas (Goecks *et al.*, 2021; Pimentel; Filippo; Santos, 2020). O objetivo foi propor e validar um protocolo de identificação e mitigação de vulnerabilidades em ambientes Docker, por meio da integração de uma ferramenta existente de escaneamento automatizado de imagens ao *pipeline* de desenvolvimento de um sistema web de bioinformática, bem como permitir sua aplicação em ambientes de produção.

3.1 Design Science Research (DSR)

O processo metodológico foi estruturado em seis etapas principais, conforme proposto por Peffers *et al.* (2007):

3.1.1 Identificação do Problema

A adoção de contêineres Docker em projetos de bioinformática tem crescido significativamente, impulsionada por sua capacidade de garantir reprodutibilidade e portabilidade (Boettiger, 2015; Haque; Babar, 2022). No entanto, essa adoção revelou uma lacuna: a ausência de soluções acessíveis e bem documentadas para integração contínua de segurança nesses ambientes. Estudos recentes mostram que mais de 60% das imagens utilizadas em aplicações científicas públicas contêm vulnerabilidades conhecidas (Haque; Babar, 2022; Alghawli; Radivilova, 2024), muitas delas relacionadas a bibliotecas desatualizadas ou configurações inseguras que passam despercebidas por falta de processos automatizados.

Além disso, análises de ferramentas amplamente utilizadas no setor indicam que, embora existam soluções robustas, sua implementação frequentemente exige conhecimento

avançado em segurança da informação ou infraestrutura, realidades muitas vezes distantes da formação de pesquisadores em bioinformática (Alghawli; Radivilova, 2024; Grüning *et al.*, 2018). Essa dificuldade prática é corroborada por relatos em discussões acadêmicas e em fóruns especializados na área.

Diante dessa evidência, o problema que este trabalho buscou resolver foi a ausência de uma sistematização para o escaneamento automatizado e contínuo de imagens Docker em ambientes de bioinformática, utilizando ferramentas de código aberto. Em vez de propor uma solução teórica genérica ou adaptar modelos corporativos complexos, este estudo propôs uma abordagem aplicada, construída com base na realidade operacional de projetos científicos, integrando verificações de segurança ao pipeline de desenvolvimento por meio de tecnologias padronizadas e de livre acesso.

3.1.2 Definição do Artefato

O artefato desenvolvido neste estudo consiste na extensão de segurança do processo BioDockFlow. Enquanto a versão original proposta por Benevides (2024) focava na padronização da manutenção e reprodutibilidade, este artefato evolui o modelo para incorporar o ciclo de vida seguro (DevSecOps).

A materialização do artefato ocorre através da redefinição das fases de *Implementar e Testar* (Fase 3) e *Implantar e Monitorar* (Fase 4) do processo original. Essas fases foram instrumentadas tecnicamente por meio de *pipelines* de CI/CD que executam automaticamente as ferramentas Trivy (para SAST) e OWASP ZAP (para DAST), atuando como guardiões de segurança antes da integração do código. Dessa forma, o artefato compreende tanto a formalização da nova versão do processo quanto sua implementação técnica nos *scripts* de automação.

3.1.3 Desenvolvimento do Artefato

Foi implementada uma estratégia de escaneamento automatizado de vulnerabilidades com base em um protocolo específico que integra a ferramenta de código aberto Trivy ao ciclo de desenvolvimento de aplicações conteinerizadas. A escolha do Trivy se justificou por sua ampla cobertura de bancos de dados de vulnerabilidades, escaneamento profundo de imagens, suporte a linguagens como Python e R, comuns na bioinformática, além de sua facilidade de integração com *pipelines* de CI/CD e relatórios detalhados.

O protocolo foi aplicado em duas etapas principais:

- **Durante o desenvolvimento**: integração direta ao *pipeline* CI/CD, permitindo que as imagens Docker fossem escaneadas no momento do *build*.

- **Em produção**: escaneamento após o lançamento de uma nova versão, além de um escaneamento periódico por meio de tarefas agendadas sem impacto direto sobre contêineres ativos.

Além da identificação de vulnerabilidades, o protocolo incorporou mecanismos de mitigação. Quando possível, o Trivy ofereceu sugestões automatizadas, como a substituição de bibliotecas vulneráveis. Para falhas críticas sem correção automática, os relatórios foram utilizados para ações manuais priorizadas com base na gravidade das vulnerabilidades (ex.: via Sistema Comum de Pontuação de Vulnerabilidades (*Common Vulnerability Scoring System*) (CVSS)). O protocolo também definiu critérios objetivos para interromper ou seguir com o processo de *deploy*, conforme políticas de segurança previamente estabelecidas. Essa abordagem híbrida visou manter o equilíbrio entre segurança e continuidade operacional, promovendo um ciclo de melhoria contínua alinhado aos princípios de DevSecOps.

3.1.4 Demonstração do Artefato

A demonstração prática do protocolo foi realizada no contexto do sistema web LUMM, uma aplicação real desenvolvida em Python no *backend* e React no *frontend*. Este ambiente representou um caso típico de sistemas científicos com uso intensivo de contêineres Docker e múltiplas dependências externas, incluindo bibliotecas científicas e ferramentas de visualização.

Como parte da demonstração, o protocolo desenvolvido foi aplicado com base na metodologia do processo de manutenção BioDockFlow, conforme proposto por Benevides (2024). Por se tratar de um processo estruturado para a manutenção de aplicações web de bioinformática baseadas em Docker, o BioDockFlow foi utilizado como referência para incorporar práticas de verificação de imagens ao fluxo de desenvolvimento do LUMM, contribuindo para a validação da solução em um contexto realista e alinhado às práticas atuais de conteinerização.

3.1.5 Avaliação

A avaliação da efetividade prática da solução foi realizada com base em dois eixos complementares: (i) a gravidade das vulnerabilidades detectadas, mensurada quantitativamente por meio de uma métrica composta, e (ii) a viabilidade de integração da ferramenta ao fluxo de trabalho do projeto.

- **Gravidade das vulnerabilidades identificadas**: para quantificar a gravidade agregada das falhas detectadas em imagens Docker, será utilizada uma métrica original, o EPV, desenvolvida especificamente para este trabalho. O EPV atribui

pesos proporcionais à criticidade das vulnerabilidades com base na classificação CVSS, sendo calculado da seguinte forma:

$$EPV = (1{,}0 \times C) + (0{,}75 \times H) + (0{,}5 \times M) + (0{,}25 \times L) \tag{3.1}$$

onde C, H, M e L representam o número de vulnerabilidades classificadas como *critical*, *high*, *medium* e *low*, respectivamente.

A atribuição dos pesos (0,25; 0,50; 0,75; 1,00) não é arbitrária, mas fundamentada em uma estratégia de linearização da escala qualitativa de severidade do CVSS.

Para operacionalizar essa mensuração, as quatro categorias de severidade do padrão foram mapeadas em uma escala numérica normalizada de 0 a 1, dividida em intervalos equidistantes:

- ***Low* (Baixa):** Primeiro quartil da escala (Peso 0,25).
- ***Medium* (Média):** Ponto mediano de risco (Peso 0,50).
- ***High* (Alta):** Terceiro quartil, risco severo (Peso 0,75).
- ***Critical* (Crítica):** Risco máximo/total (Peso 1,00).

Esta abordagem permite converter a classificação qualitativa em um indicador quantitativo de densidade de falhas, onde o acúmulo linear de vulnerabilidades de menor impacto compõe o escore de risco final do projeto.

- **Facilidade de integração e impacto operacional**: foi avaliado o tempo total de integração da ferramenta ao pipeline CI/CD do sistema LUMM, o número de ajustes manuais exigidos e a compatibilidade com as práticas preexistentes. Também foram monitorados o tempo no *build*, a ocorrência de falhas ou interrupções e a clareza dos relatórios.

A formulação da métrica EPV fundamenta-se na necessidade prática de automação determinística para o *pipeline*.

Embora existam padrões consolidados para classificação de vulnerabilidades, eles focam na avaliação individual de cada falha. Para que um sistema de CI/CD possa tomar uma decisão autônoma de bloqueio, é necessário transformar essa lista dispersa de vulnerabilidades em um indicador único.

Portanto, a criação do EPV não visa substituir metodologias de gestão de risco complexas, mas sim preencher uma lacuna de orquestração: criar uma camada de abstração matemática que permita ao *Security Gate* processar o volume de falhas de forma objetiva, instantânea e sem a necessidade de intervenção humana para análise subjetiva de contexto a cada execução.

A avaliação foi conduzida de forma observacional e empírica, com foco na mensuração objetiva da efetividade da abordagem proposta, garantindo a reprodutibilidade dos dados e a comparação com práticas correntes de mercado em segurança para ambientes conteinerizados.

3.1.6 Comunicação dos Resultados

Os resultados são comunicados nos capítulos subsequentes por meio de tabelas e gráficos que apresentam as vulnerabilidades identificadas, o tempo de execução dos escaneamentos, o nível de criticidade dos alertas e observações sobre a facilidade de adoção da solução proposta. Além disso, são discutidas as limitações encontradas durante a aplicação prática, bem como sugestões para melhorias e trabalhos futuros.

4 DESENVOLVIMENTO E DEMONSTRAÇÃO

Este capítulo detalha as etapas de desenvolvimento e demonstração do artefato proposto, seguindo a metodologia DSR. O artefato consiste em um protocolo para identificação e mitigação de vulnerabilidades em aplicações web de bioinformática conteinerizadas em Docker.

O desenvolvimento foca na estruturação do protocolo, na seleção das ferramentas de análise (estática e dinâmica) e na sua integração automatizada em *pipelines* de CI/CD. A demonstração, por sua vez, apresenta a aplicação prática deste protocolo sobre o sistema web LUMM, um objeto de estudo real que se beneficia da conteinerização.

4.1 O Protocolo de Segurança

O artefato central deste trabalho é um protocolo que integra práticas de segurança ao ciclo de vida de aplicações conteinerizadas, alinhando-se aos princípios DevSecOps. O objetivo não é criar uma nova ferramenta, mas sim formalizar um processo acessível e reprodutível que blinda (*hardening*) processos de manutenção existentes, como o BioDockFlow, com verificações de segurança automatizadas (Benevides, 2024).

4.1.1 Seleção de Ferramentas de Análise

Para uma cobertura de segurança, o protocolo atua em duas frentes: análise estática do artefato (a imagem Docker) e análise dinâmica da aplicação em execução.

4.1.1.1 Análise de Vulnerabilidades de Imagem (SAST)

A ferramenta selecionada foi o Trivy. Sua escolha se baseia no fato de ser *open source*, possuir ampla cobertura de CVEs (Security, 2025), facilidade de integração em *pipelines* CI/CD e suporte a linguagens como Python e R, que são predominantes na bioinformática.

4.1.1.2 Análise Dinâmica de Segurança da Aplicação (DAST)

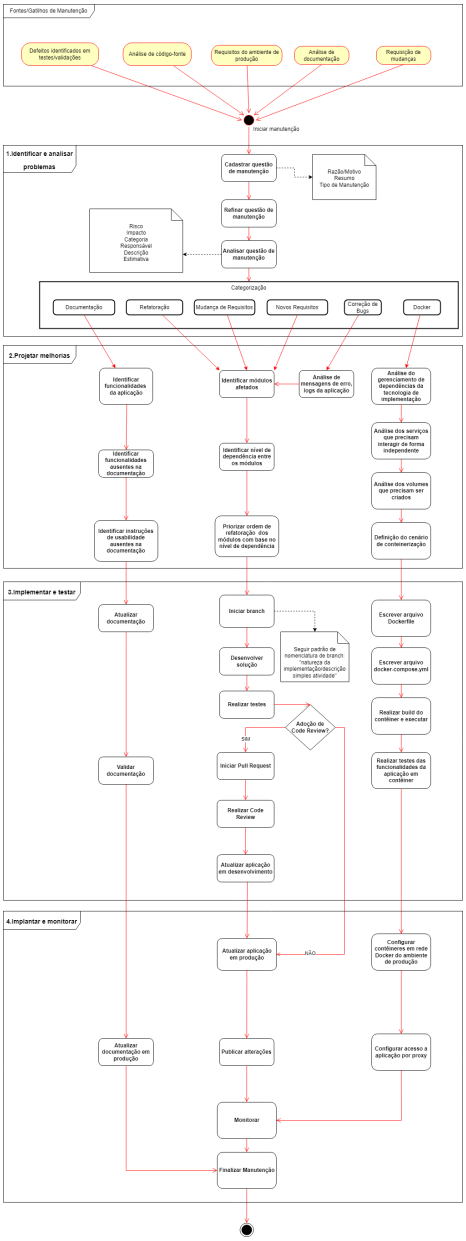
Para a análise dinâmica, foi selecionado o Projeto Aberto Mundial para Segurança de Aplicações (*Open Worldwide Application Security Project*) (OWASP) ZAP (Zed Attack Proxy). O ZAP é o padrão *de facto* da indústria para testes de segurança dinâmicos *open source*. Ele é capaz de simular ataques a uma aplicação web em execução, identificando vulnerabilidades como *Cross-Site Scripting* (XSS), injeção de SQL e configurações inseguras de *headers* HTTP, que não são detectáveis pela análise estática da imagem.

4.1.2 Estrutura do Protocolo e Integração ao BioDockFlow

O protocolo foi projetado para se integrar organicamente ao processo de manutenção BioDockFlow desenvolvido por Benevides (2024). As Fases 1 (Identificar e analisar problemas) e 2 (Projetar melhorias) do processo original permanecem inalteradas, servindo como fluxo de entrada padrão para o desenvolvimento seguro.

Para evidenciar a evolução processual e o impacto da camada de segurança proposta, apresenta-se a seguir uma comparação direta entre os fluxos completos. A Figura 2 ilustra o BioDockFlow, conforme concebido por Benevides (2024), focado primariamente na manutenção funcional e padronização.

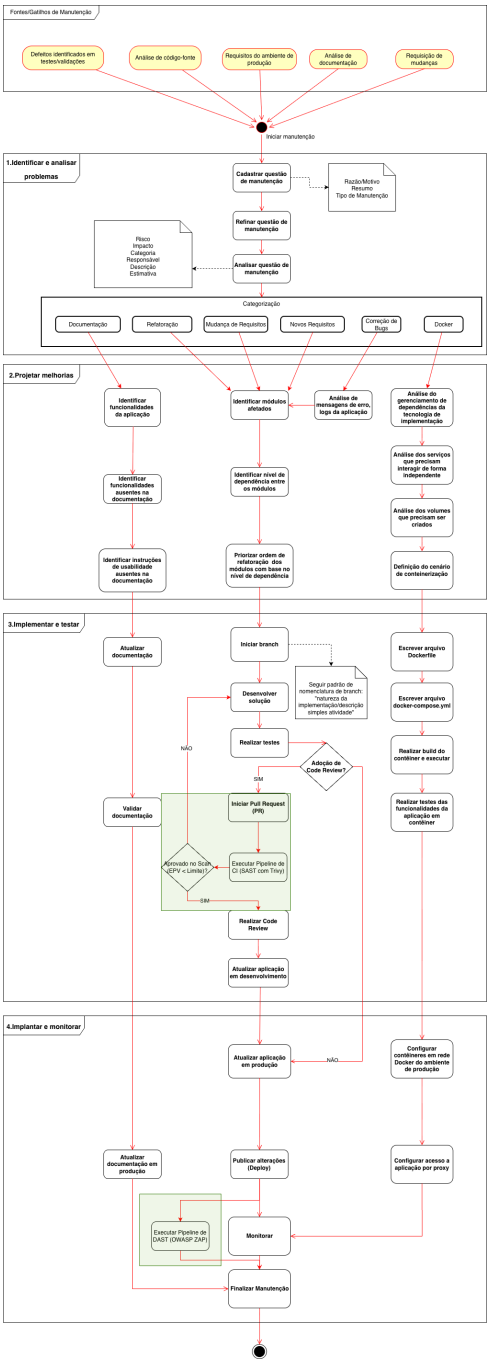
Figura 2 – BioDockFlow (v3) - Foco em Manutenção Funcional



Fonte: Benevides (2024).

Em contrapartida, a Figura 3 demonstra o BioDockFlow atualizado, evidenciando a inserção do ponto de verificação de segurança (*Security Gate*), a integração do Trivy SAST e as etapas de bloqueio/aprovação que transformam o fluxo em um ciclo de manutenção segura.

Figura 3 – BioDockFlow (v4) - Com Protocolo de Segurança Integrado



Fonte: Adaptado de Benevides (2024).

A operacionalização dessas intervenções ocorre por meio de duas estratégias de integração principais:

1. **Integração Preventiva (CI):** Aplicada na Fase 3, esta etapa é acionada automaticamente na abertura de uma *pull request*. O *pipeline* de Integração contínua (*Continuous integration*) (CI) executa a construção da imagem e a varredura estática (*scan*) com a ferramenta Trivy. Esta etapa atua como um portão de segurança (*security gate*): se, após a análise, o EPV ultrapassar o limite máximo configurado, o pipeline falha intencionalmente, impedindo que o artefato inseguro avance para a revisão ou fusão (*merge*) com o código principal.

2. **Integração de Vigilância (CD/DAST):** Esta estratégia ocorre na Fase 4. Após a aprovação na etapa anterior e a publicação da nova versão (*release*) pelo *pipeline* de Entrega contínua (*Continuous delivery*) (CD), é acionado o fluxo de DAST utilizando a ferramenta OWASP ZAP. O objetivo é monitorar a aplicação em execução no ambiente de produção, identificando vulnerabilidades de configuração e tempo de execução que a análise estática não seria capaz de detectar.

4.1.3 Pipelines (CI/CD Workflows)

A automatização do protocolo é materializada através de três *pipelines* (*workflows*) distintos no GitHub Actions, cada um servindo a um propósito específico alinhado às fases do BioDockFlow (Benevides, 2024):

1. **CI (Integração Contínua):** Focado na *Integração Preventiva* (Fase 3 do BioDockFlow).

2. **CD (Entrega Contínua):** Focado na publicação do artefato para a atividade de *Implantar* (Fase 4 do BioDockFlow).

3. **DAST (Monitoramento):** Focado na atividade de *Monitorar* (Fase 4 do BioDockFlow).

O *workflow* de CI (*Build & Scan*) representa a etapa de integração preventiva. Conforme definido na metodologia, ele é acionado em toda `pull_request` para a *branch* `main`. Seu propósito é oferecer *feedback* automatizado sobre a segurança do artefato (imagem Docker) antes da integração, prevenindo que falhas sejam incorporadas ao código principal.

O *job* `scan` é o núcleo do protocolo: executa múltiplas varreduras com o Trivy (sistema de arquivos, imagem e configurações), calcula o EPV e atua como um *security gate*, falhando o *build* (`exit 1`) caso o EPV ultrapasse o limiar configurado. A implementação completa deste *workflow* está detalhada nos Apêndices C e D.

O *workflow* de CD (*Deploy*) materializa a publicação do artefato para a atividade *Implantar* da Fase 4 do BioDockFlow (Benevides, 2024). Ele é acionado apenas em `push`

para a *branch* `main` (ou seja, após um *Pull Request* ser aprovado e mesclado). Sua única responsabilidade é construir a imagem Docker final, marcá-la com a *tag* `latest` e publicá-la no Registo de Contêineres do GitHub (*GitHub Container Registry*) (GHCR) que é o repositório adotado para armazenar as imagens Docker do LUMM. O *workflow* completo pode ser consultado nos Apêndices E e F.

Já o *workflow* de DAST (Monitoramento) representa a integração de vigilância e implementa a atividade *Monitorar* (Fase 4 do BioDockFlow) (Benevides, 2024), com foco em segurança. De forma desacoplada, ele é acionado em três cenários:

- **Após a nova versão:** `on: workflow_run` garante que uma análise DAST seja executada logo após o *workflow Nova release* ser concluído com sucesso.
- **Agendamento:** `on: schedule` executa a análise periodicamente (a cada 14 dias), garantindo o monitoramento contínuo.
- **Manual:** `on: workflow_dispatch` permite que a equipe acione a análise sob demanda.

O *job* `dast` utiliza o OWASP ZAP para realizar uma análise dinâmica na Interface de Programação de Aplicações (*Application Programming Interface*) (API) e interface web em produção, identificada via *secret* ('secrets.DAST_TARGET'), buscando vulnerabilidades de tempo de execução que o Trivy (SAST) não poderia detectar. A implementação deste *workflow* está disponível nos Apêndices G e H.

4.2 Aplicação no LUMM

Para validar a eficácia do protocolo, ele foi aplicado no sistema web LUMM, conforme definido na metodologia.

4.2.1 O Sistema LUMM

O LUMM é um banco de dados *web* desenvolvido no âmbito do G2BC, contando com a participação técnica do autor deste trabalho em seu desenvolvimento. O sistema tem como objetivo centralizar informações taxonômicas, dados de emissão de luz (partes luminescentes), distribuição geográfica e referências bibliográficas sobre as espécies catalogadas. A plataforma oferece uma interface pública de busca para pesquisadores e entusiastas, além de um módulo administrativo para curadoria dos dados (em desenvolvimento), permitindo o cadastro de novas espécies e o gerenciamento de mídias associadas.

Do ponto de vista arquitetural, o LUMM é uma aplicação distribuída composta por uma API desenvolvida em Python (utilizando o *framework* Flask) e uma interface de usuário (*frontend*) construída em ReactJS. Ambos os componentes são conteinerizados e

orquestrados via Docker, garantindo a reprodutibilidade do ambiente de execução. Esta arquitetura, que possui múltiplas dependências externas (bibliotecas Python e pacotes NPM), representa um caso de uso ideal, pois é suscetível a vulnerabilidades herdadas.

Para a definição do ambiente de execução (*runtime*) dos contêineres do estudo de caso, optou-se pela utilização da imagem base Alpine Linux (versão 3.21). Esta escolha constitui a primeira camada de defesa passiva do protocolo proposto, alinhando-se ao princípio de *Hardening* e redução de superfície de ataque.

Diferentemente de distribuições convencionais (como *Ubuntu* ou *Debian*), que trazem nativamente um amplo conjunto de utilitários e bibliotecas, o Alpine Linux é construído sobre a biblioteca *musl libc* e o conjunto de ferramentas *BusyBox*. Esta arquitetura minimalista resulta em imagens finais significativamente menores e, consequentemente, com um número reduzido de componentes passíveis de conter CVEs.

Essa estratégia foi fundamental para estabelecer um ponto de partida controlado para a validação do *Security Gate*, garantindo que as vulnerabilidades detectadas fossem predominantemente oriundas das dependências da aplicação, e não do sistema operacional base.

O código-fonte do projeto é aberto e está versionado em dois repositórios no GitHub, facilitando a contribuição da comunidade e a manutenção evolutiva:

- *Frontend:* https://github.com/G2BC/LUMM-web (G2BC, 2025b)
- *Backend:* https://github.com/G2BC/LUMM-server (G2BC, 2025a)

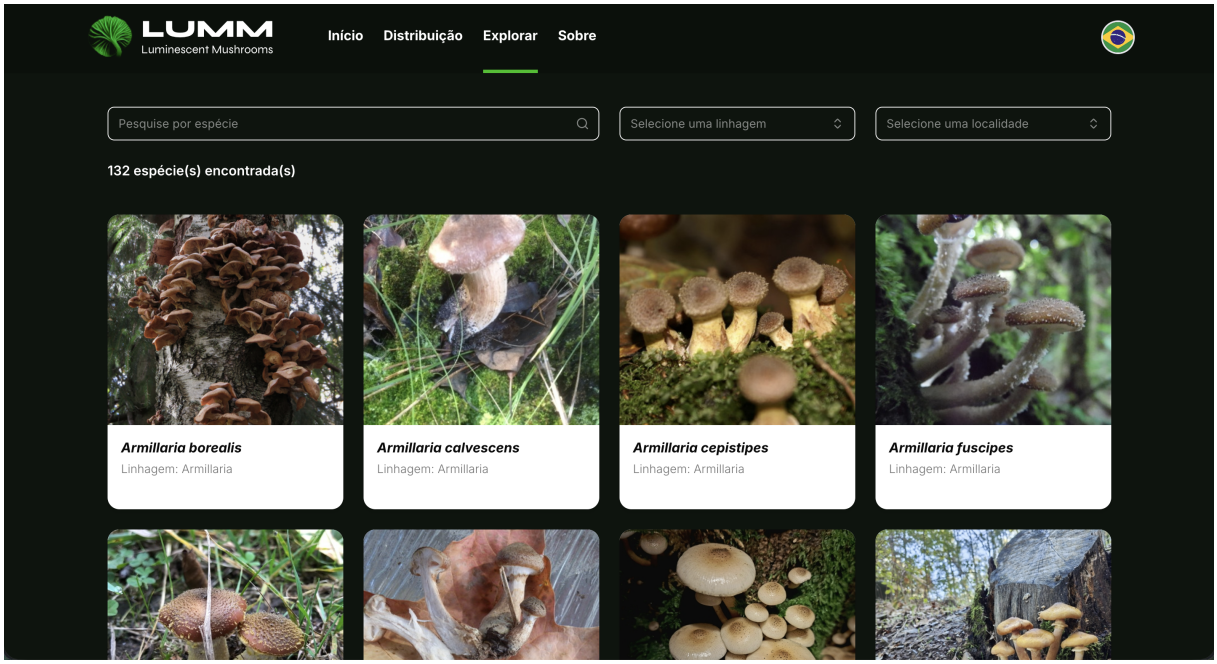
A aplicação está implantada e acessível publicamente através do endereço eletrônico do grupo de pesquisa:

https://lumm.uneb.br [1]

A Figura 4 apresenta a interface principal do sistema, destacando a listagem de espécies e as ferramentas de filtro disponíveis para o usuário final.

[1] Acesso realizado em outubro de 2025.

Figura 4 – Interface Principal do Sistema LUMM



Fonte: Elaborada pelo autor (2025).

4.2.2 Execução do Protocolo e Mitigação

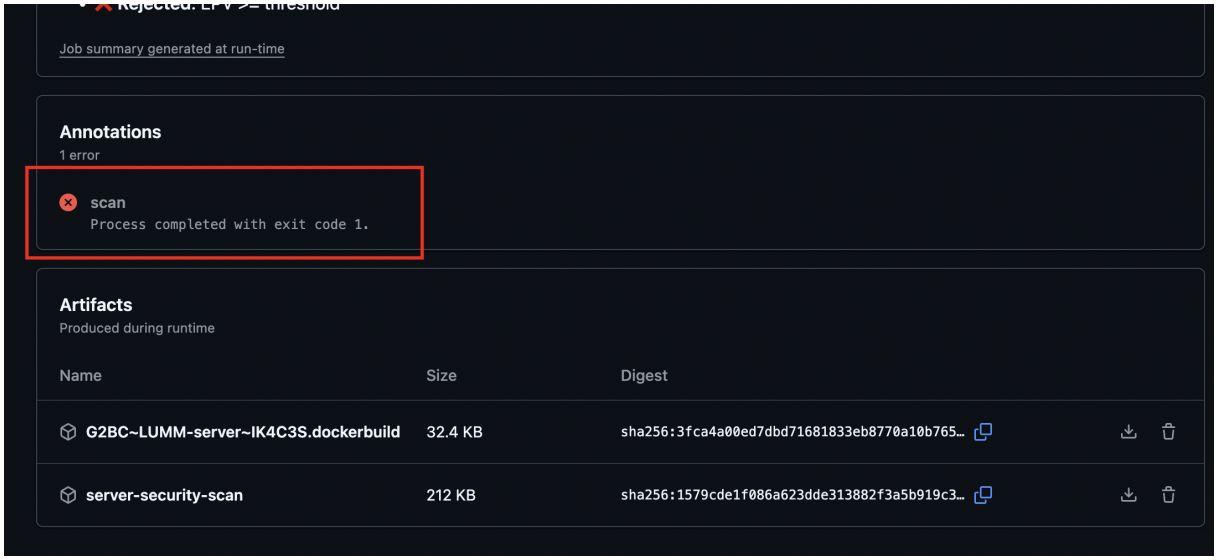
4.2.2.1 Execução Inicial (Detecção)

O *pipeline* de CI, atuando como camada preventiva, foi executado sobre o código-fonte original do LUMM. O *job* de varredura (*scan*) utilizou a ferramenta Trivy para analisar as dependências do *backend* e do *frontend*.

A execução detectou vulnerabilidades em bibliotecas de terceiros e na imagem base, gerando relatórios de artefatos que foram processados para o cálculo do EPV. Como os valores calculados ultrapassaram o limiar de segurança (*threshold*) configurado no *workflow*, o mecanismo de *Security Gate* foi acionado, resultando na falha intencional do *pipeline* e bloqueio da integração.

A Figura 5 ilustra a interrupção do fluxo de trabalho no GitHub Actions, confirmando o funcionamento do bloqueio preventivo.

Figura 5 – Interrupção do *Pipeline* de CI pelo *Security Gate* (Trivy)



Fonte: Elaborada pelo autor (2025).

4.2.2.2 Aplicação das Correções (Mitigação)

Com base nos relatórios gerados na etapa de detecção, foram aplicadas as seguintes ações corretivas no código-fonte:

Backend (Python):

- **Atualização da Imagem Base:** A imagem foi alterada no `Dockerfile` de `python:3.8-slim` para `python:3.10-slim-bullseye` para mitigar vulnerabilidades críticas no sistema operacional (ex: CVE-2025-6965).
- **Atualização de Dependências:** A biblioteca `Werkzeug` foi atualizada da versão 2.3.8 para 3.0.6 no arquivo `requirements.txt` para corrigir a falha CVE-2024-34069.

Frontend (React):

- **Atualização do Servidor Web:** A imagem do Nginx utilizada no estágio final do *multi-stage build* foi atualizada da versão 1.27-alpine para 1.29.3-alpine-slim, mitigando vulnerabilidades críticas na biblioteca `libxml2`.

4.2.2.3 Reexecução e Implantação

Após o envio (*push*) das correções ao repositório, o *pipeline* de CI foi acionado automaticamente. Nesta nova execução, o cálculo do EPV manteve-se dentro do limite tolerável, aprovando o *Security Gate* e permitindo a conclusão do *build*.

Com a aprovação na CI, o *pipeline* de CD foi iniciado, publicando as imagens seguras no GHCR e disparando o fluxo de monitoramento DAST (OWASP ZAP) no ambiente de produção.

A análise detalhada dos dados quantitativos gerados nestas execuções, bem como a avaliação da eficácia da mitigação através do EPV, são apresentadas no capítulo a seguir.

5 AVALIAÇÃO E DISCUSSÃO DOS RESULTADOS

5.1 Avaliação Quantitativa (Eficácia do Artefato)

A eficácia da extensão de segurança proposta foi avaliada através da métrica EPV, calculada pela Equação 3.1. Onde as variáveis representam a contagem de vulnerabilidades por severidade (Crítica, Alta, Média e Baixa). O protocolo estabeleceu um limiar de tolerância (*threshold*) de 5,0 para o *Security Gate*.

O limiar de tolerância (*threshold*) foi estabelecido em 5,0 para o contexto deste estudo. Embora normas como o OWASP preconizem a correção total de falhas críticas, a gestão de vulnerabilidades residuais (médias e baixas) exige a definição de limites aceitáveis de risco.

Assim, o valor 5,0 foi estabelecido como um critério de corte operacional para este protocolo. Ele representa uma direção de segurança que tolera um nível residual de vulnerabilidades de baixo impacto, necessário para a continuidade do desenvolvimento em equipes enxutas, mas bloqueia a integração de código que acumule um volume de falhas capaz de comprometer a manutenibilidade ou facilitar vetores de ataque complexos.

A parametrização deste valor é flexível no protocolo, permitindo que diferentes equipes ajustem o rigor do bloqueio conforme a maturidade de segurança e a criticidade de seus projetos.

5.1.1 Análise Pré-Mitigação

A primeira execução do protocolo no sistema LUMM revelou um cenário de risco. No *Backend*, foram identificadas 74 vulnerabilidades (1 Crítica, 16 Altas, 48 Médias e 9 Baixas). Aplicando a métrica:

$$EPV_{backend\_inicial} = (1 \times 1) + (0,75 \times 16) + (0,5 \times 48) + (0,25 \times 9) = \mathbf{39,25}$$

No *Frontend*, foram detectadas 28 vulnerabilidades (2 Críticas, 5 Altas, 16 Médias e 5 Baixas):

$$EPV_{frontend\_inicial} = (1 \times 2) + (0,75 \times 5) + (0,5 \times 16) + (0,25 \times 5) = \mathbf{15,00}$$

Ambos os valores ultrapassaram significativamente o *threshold* de 5,0, justificando a interrupção automática do *pipeline* demonstrada no capítulo anterior.

5.1.2 Análise Pós-Mitigação

Após a aplicação das correções, a reexecução do protocolo gerou os dados comparativos apresentados na Tabela 2.

Tabela 2 – Comparativo de Eficácia da Mitigação e Decisão de Bloqueio

Componente	**EPV Inicial**	**EPV Final**	**Status do *gate***	**Decisão Final**
Backend	39,25	14,00	Bloqueado	Aprov. Manual*
Frontend	15,00	0,00	Aprovado	Aprov. Automática

*Nota: O EPV final (14,00) excedeu o limiar (5,0), acionando o bloqueio automático. A aprovação seguiu via intervenção humana baseada em critérios de gestão de risco.

A análise da Tabela 2 evidencia que o mecanismo de *security gate* atuou corretamente ao impor um ponto de controle objetivo. O resultado de 14,00 no *backend*, superior ao limiar estabelecido, resultou na interrupção automática do *pipeline*.

Diante do bloqueio, a equipe optou pela estratégia de risco aceito (*Risk Acceptance*). Os critérios técnicos para esta aprovação manual foram:

1. **Mitigação de Críticos:** Todas as vulnerabilidades de nível *Critical* foram sanadas.
2. **Contexto Operacional:** A necessidade de disponibilização do sistema LUMM justificou a aceitação temporária de vulnerabilidades de severidade menor.
3. **Registro de Dívida Técnica:** Os itens restantes foram documentados para tratamento futuro.

É fundamental distinguir a contribuição das ferramentas de varredura da contribuição do protocolo proposto. A redução quantitativa do risco (100% no *frontend* e 64% no *backend*) deriva diretamente da capacidade de detecção do motor do Trivy e da ação corretiva dos desenvolvedores. No entanto, a eficácia do protocolo reside na automação da governança.

Sem a integração proposta (fases e *workflows*), a execução das ferramentas dependeria da disciplina individual do desenvolvedor, e os resultados poderiam ser ignorados em prol da agilidade. O protocolo garantiu que:

- A verificação de segurança fosse um passo obrigatório e não opcional (*Enforcement*);
- O critério de aceitação fosse baseado em métricas objetivas (EPV) e não na subjetividade do operador;
- A decisão de implantar código vulnerável (Status 14,00) exigisse uma ação explícita de *override*, gerando rastro de auditoria.

Quanto à análise dinâmica (DAST), embora tenha identificado um volume menor de falhas em comparação ao SAST, sua eficácia marginal reside na cobertura de escopo. As falhas detectadas pelo OWASP ZAP são invisíveis à análise estática do código ou do sistema de arquivos, demonstrando que a complementaridade das técnicas é necessária para cobrir vetores de ataque que só se manifestam em tempo de execução.

5.1.3 Resultados da Análise Dinâmica (DAST)

A etapa de vigilância (DAST) executada pelo OWASP ZAP no ambiente de produção identificou falhas de configuração que não poderiam ser detectadas na análise da imagem.

Tabela 3 – Resultados do Monitoramento Dinâmico (OWASP ZAP)

Severidade	**Qtd.**	**Exemplo de Falha Detectada**
Médio	3	Ausência de cabeçalho Anti-clickjacking
Baixo	5	Configuração de isolamento de site incorreta

A escolha pelo perfil de varredura *Baseline Scan* do OWASP ZAP, em detrimento dos modos *Active* ou *Full Scan*, foi uma decisão de projeto deliberada para adequar o protocolo à dinâmica de um fluxo de CI.

Enquanto um *Full Scan* realiza ataques ativos (*fuzzing*, injeção de payloads) que podem levar horas para serem concluídos e oferecem risco de integridade aos dados, o *Baseline Scan* opera de forma passiva e não-intrusiva, completando a verificação em poucos minutos. Este *trade-off* prioriza o tempo de feedback, requisito essencial para que a verificação de segurança não se torne um gargalo que desestimule a frequência de *deploys* no BioDockFlow.

É importante ressaltar que, embora os resultados quantitativos do DAST (Tabela 3) sejam numericamente inferiores aos do SAST, o componente DAST mantém sua indispensabilidade no protocolo devido à complementaridade de escopo.

O SAST (Trivy) analisa o sistema de arquivos e o código estático, mas é cego para vulnerabilidades que só se manifestam em tempo de execução (*runtime*). As falhas detectadas pelo ZAP no estudo de caso (ex: ausência de cabeçalhos *Anti-Clickjacking* e *Content Security Policy*) referem-se a configurações do servidor web e da resposta HTTP, vetores que nenhuma ferramenta de análise estática seria capaz de identificar. Portanto, a remoção do DAST criaria um ponto cego de segurança (falhas de configuração de *runtime*), invalidando a proposta de cobertura em múltiplas camadas.

5.2 Avaliação Qualitativa (Viabilidade e Impacto Operacional)

O segundo pilar da avaliação analisou a "facilidade de integração e impacto operacional"do protocolo.

5.2.1 Viabilidade de Integração ao BioDockFlow

A integração do protocolo ao processo BioDockFlow (Benevides, 2024) se mostrou altamente viável. O artefato (os três *pipelines*) não altera o fluxo de manutenção existente; ele o estende e o *blinda*, como proposto nas Figura 3.

A Fase 3 (Implementar e Testar) do BioDockFlow (Benevides, 2024) agora é protegida pelo *pipeline* de CI, garantindo que artefatos inseguros não avancem. A Fase 4 (Implantar e Monitorar) (Benevides, 2024) é automatizada e reforçada pelos *pipelines* de CD (publicação do artefato) e DAST (monitoramento de segurança). A adoção do protocolo exigiu apenas a inclusão dos arquivos ‘.yml‘ no repositório, caracterizando baixo atrito de implementação.

5.2.2 Impacto Operacional e Clareza dos Relatórios

Conforme definido na etapa de avaliação, foram analisados os critérios de *tempo no build* e da *clareza dos relatórios*. A execução do *job* `scan`, que envolve múltiplas análises do Trivy e o cálculo do EPV, adicionou aproximadamente 1 minuto e 11 segundos ao tempo total do *build* de CI. Este incremento representa um custo operacional aceitável frente à camada adicional de segurança preventiva que o protocolo proporciona.

A clareza dos relatórios do Trivy (gerados como artefatos `.txt` e `.json`) foi um fator decisivo para a mitigação. Os relatórios indicaram com precisão as bibliotecas vulneráveis (ex: `libsqlite3-0`, `Werkzeug`, `libxml2`) e suas versões corrigidas. Isso permitiu que a mitigação fosse cirúrgica e rápida (ex: atualizar a imagem base para `python:3.10-slim-bullseye` e executar `npm audit fix`), demonstrando a utilidade prática do artefato.

5.2.3 Eficácia da Abordagem Híbrida (SAST + DAST)

A decisão de adotar uma abordagem híbrida (Trivy para SAST e OWASP ZAP para DAST) demonstrou-se justificada pelos resultados obtidos. A análise SAST do Trivy foi eficaz na redução substancial das vulnerabilidades do artefato (a imagem), conforme detalhado na Tabela 2.

Contudo, a análise estática isolada se mostrou insuficiente. Os resultados do DAST (Tabela 3), executados na aplicação em produção, revelaram falhas de configuração de

runtime, como "Missing Anti-clickjacking Header". Estas são vulnerabilidades que o Trivy, por analisar apenas o código e a imagem, é incapaz de detectar.

Isso valida a necessidade de ambas as abordagens para uma cobertura de segurança completa, cumprindo o critério estabelecido de analisar a aplicação *durante o desenvolvimento* (CI/SAST) e *em produção* (CD/DAST).

A avaliação realizada no contexto do sistema LUMM apresenta evidências de que o protocolo proposto cumpre os objetivos definidos na metodologia DSR para o cenário analisado. Os resultados obtidos indicam que o protocolo apresentou as seguintes características:

- **Eficaz:** Para o caso avaliado, o protocolo evidenciou uma redução de risco mensurável entre 64,33% no *backend* e 100% para o *frontend* no EPV após as etapas de mitigação.
- **Viável:** A integração ao processo *BioDockFlow* mostrou-se tecnicamente exequível no ambiente do estudo, exigindo baixo esforço de configuração e introduzindo um incremento de tempo considerado aceitável no fluxo de construção.
- **Robusto:** A aplicação da abordagem híbrida SAST e DAST no objeto de estudo permitiu a identificação de vulnerabilidades de configuração em tempo de execução que não foram detectadas pela ferramenta de análise estática isoladamente.

6 CONSIDERAÇÕES FINAIS

Este trabalho abordou um desafio central na engenharia de software científica: conciliar a necessidade de reprodutibilidade, alcançada via conteinerização, com a exigência de segurança da informação, frequentemente negligenciada em processos de manutenção acadêmicos. A pesquisa partiu da identificação de uma lacuna específica: o processo BioDockFlow, embora eficaz para a manutenção e padronização de aplicações de bioinformática (Benevides, 2024), não contemplava mecanismos integrados de segurança.

O objetivo geral deste trabalho foi desenvolver e validar a extensão de segurança do processo de manutenção BioDockFlow, transformando-o em um ciclo DevSecOps completo. Utilizando a metodologia DSR, o artefato foi materializado como um conjunto de três pipelines (CI, CD e DAST) que introduzem a verificação automatizada de segurança e impõem critérios objetivos de bloqueio em fases importantes do ciclo de manutenção.

A etapa de avaliação (Capítulo 5), realizada por meio da aplicação do protocolo no sistema LUMM, permitiu validar a eficácia da solução proposta em múltiplos eixos para o caso analisado.

Primeiramente, a aplicação do protocolo apresentou evidências de eficácia em termos quantitativos. A métrica EPV (Equação 3.1), proposta neste trabalho, foi fundamental para traduzir 102 vulnerabilidades iniciais em escores de risco objetivos (39,25 no *backend* e 15,00 no *frontend*). Adicionalmente, para o caso analisado, a métrica permitiu mensurar o impacto das correções, apresentando uma redução de risco entre 64% e 100% no EPV. Observou-se também a atuação efetiva do *security gate* na interrupção de *builds* que não atendiam aos critérios de segurança estabelecidos.

Em segundo lugar, o protocolo demonstrou ser viável. A sua integração ao BioDockFlow (Benevides, 2024) apresentou baixo atrito, estendendo as Fases 3 e 4 do processo existente sem exigir sua modificação estrutural. O impacto no tempo de *build* foi considerado um *trade-off* operacional aceitável pela garantia de segurança preventiva obtida.

Por fim, os resultados obtidos indicam a robustez do protocolo no cenário avaliado. A abordagem híbrida, combinando SAST (com Trivy) e DAST (com OWASP ZAP), mostrou-se fundamental para a eficácia da detecção. A análise DAST identificou vulnerabilidades de configuração em *runtime* (ex: "Missing Anti-clickjacking Header") que a análise estática (SAST) não poderia detectar, evidenciando a importância da complementaridade entre as técnicas para uma cobertura de segurança mais abrangente.

Conclui-se, assim, que os objetivos deste trabalho foram atingidos dentro do escopo definido. O artefato desenvolvido validou a extensão de segurança para o processo Bio-

DockFlow, apresentando-se como uma solução funcional para a introdução de práticas de DevSecOps em projetos de bioinformática conteinerizados. Embora não se possa generalizar imediatamente para todos os paradigmas de software científico, os resultados demonstram a viabilidade de integrar controles de segurança ao fluxo de manutenção sem degradar a estabilidade operacional do ambiente conteinerizado, preservando a infraestrutura necessária para a reprodutibilidade.

6.1 Limitações do Trabalho

Uma limitação deste estudo reside na condução da execução e validação do protocolo pelo próprio autor da pesquisa. Reconhece-se que a ausência de uma validação externa independente, realizada por uma equipe de desenvolvimento alheia ao trabalho, constitui uma ameaça à validade interna, introduzindo potenciais vieses na seleção das estratégias de correção. No entanto, esta limitação foi parcialmente mitigada pela natureza determinística das ferramentas empregadas, garantindo que o critério de sucesso fosse a conformidade objetiva com o limiar numérico, e não a avaliação subjetiva do pesquisador.

Adicionalmente, a validação do protocolo foi restrita ao sistema LUMM, o que impõe limites à generalização dos resultados. Reconhece-se que a eficácia observada pode ter sido influenciada por características específicas deste objeto de estudo, tais como:

- **Tecnologia da Aplicação:** O projeto utiliza Python e *pip* para gerenciamento de dependências. A aplicação do mesmo protocolo em projetos desenvolvidos em outras linguagens (como Java ou .NET) ou com gerenciadores de pacotes distintos poderia revelar desafios diferentes na detecção e correção de vulnerabilidades, que não puderam ser observados neste estudo.
- **Imagem Base do Contêiner:** O uso da distribuição *Alpine Linux* no LUMM, caracterizada pelo minimalismo, contribuiu para um cenário inicial de superfície de ataque reduzida. A replicação do processo em sistemas que necessitem de imagens base mais robustas (como *Ubuntu* ou *Debian*) provavelmente resultaria em um volume maior de vulnerabilidades iniciais, o que demandaria recalibragem dos pesos do EPV.
- **Arquitetura do Sistema:** As métricas de tempo de execução do *pipeline* foram obtidas no contexto da arquitetura específica do LUMM. Em sistemas com arquiteturas de microsserviços mais complexas ou altamente distribuídas, a inserção dos *Security Gates* poderia impactar o tempo total de integração de forma diferente da registrada nesta pesquisa.
- **Acoplamento Tecnológico (CI/CD e Docker):** O protocolo possui uma dependência arquitetural estrita de tecnologias de conteinerização (Docker) e de plataformas de orquestração de *pipelines*. A solução não é aplicável a processos de manutenção

manuais ou a softwares científicos monolíticos tradicionais que não utilizem contêineres. Além disso, a implementação atual é acoplada à sintaxe do GitHub Actions; a adoção por grupos que utilizem outras plataformas, como GitLab CI ou Jenkins, exigiria a refatoração dos arquivos de configuração do *workflow*, embora a lógica do protocolo (fases e métricas) permaneça agnóstica.

- **Delimitação de Ferramentas e Estado da Arte:** A seleção tecnológica deste trabalho priorizou estritamente soluções de código aberto (*Open Source*) e gratuitas (como Trivy e OWASP ZAP) para garantir a reprodutibilidade e a acessibilidade financeira por grupos de pesquisa com recursos limitados. Consequentemente, soluções comerciais de mercado (como Snyk, Aqua Security ou GitHub Advanced Security) e frameworks complexos de *Policy-as-Code* (como Open Policy Agent - OPA) não foram incluídos no escopo de implementação ou comparação. Embora reconheça-se a robustez dessas tecnologias no estado da arte industrial, sua exclusão foi deliberada para reduzir a barreira de entrada e a complexidade de configuração, focando na validação do processo de integração ao BioDockFlow e não em um *benchmark* exaustivo de ferramentas proprietárias.

- **Abrangência da Detecção e Dependência de CVEs:** A eficácia do protocolo está intrinsecamente ligada às bases de dados públicas de vulnerabilidades (como NVD e OSV). O método atua na detecção de falhas catalogadas CVEs, estabelecendo uma linha de base de segurança. Contudo, o protocolo possui uma limitação fundamental na detecção de vulnerabilidades do tipo *zero-day* (não catalogadas) ou falhas lógicas específicas de negócio. Portanto, a aprovação no *security gate* não deve ser interpretada como garantia absoluta de invulnerabilidade, sendo recomendável a complementação com testes manuais (*pentesting*) para sistemas críticos.

- **Recorte Técnico do conceito DevSecOps:** O trabalho focou estritamente na vertente de automação de segurança dentro do paradigma DevSecOps. Reconhece-se que a implementação de um ciclo DevSecOps *completo* ou *holístico* exigiria a abordagem de dimensões sociotécnicas não contempladas nesta pesquisa, tais como: transformação cultural da equipe, implementação de Modelagem de Ameaças (*Threat Modeling*) na fase de *design*, estabelecimento de programas de *Security Champions* e feedback loops humanos. Portanto, o termo DevSecOps é utilizado neste texto referindo-se especificamente à integração técnica de ferramentas SAST/DAST em esteiras de CI/CD, e não à maturidade organizacional plena.

Portanto, embora os resultados comprovem a viabilidade do protocolo para o cenário testado, a sua aplicabilidade universal depende de novas validações (triangulação) em projetos com *stacks* tecnológicas diversas, conforme indicado para trabalhos futuros.

6.2 Trabalhos Futuros

Com base nos resultados e limitações identificados, sugerem-se as seguintes direções para trabalhos futuros:

- **Expansão da Validação:** Aplicar o protocolo de segurança aos demais sistemas web do G2BC (como Cogumelos Comestíveis Brasileiros (*Brazilian Edible Mushrooms*) (BEM), FunRegulation, FunExpression, etc.) para validar sua eficácia em diferentes arquiteturas.
- **Evolução da Métrica (EPV):** Aprimorar o EPV, métrica original desenvolvida nesta pesquisa, para que o escore possa ser incrementado pelos resultados do DAST, unificando o risco de *build* (SAST) com o risco de *runtime* (DAST) em uma única métrica.
- **Automação da Mitigação:** Evoluir o protocolo para além da identificação, explorando a criação automática de *issues*, no GitHub ou Jira, a partir das vulnerabilidades críticas detectadas pelo *pipeline*, automatizando o início do ciclo de correção.
- **Aprofundamento do DAST:** Implementar *scans* DAST autenticados e completos (*full scan*) do OWASP ZAP, que exigem maior configuração, mas oferecem uma detecção de falhas significativamente mais profunda.
- **Inclusão de Teste de Intrusão Manual (Pentest):** Complementar as varreduras automatizadas (SAST e DAST) com testes de intrusão manuais periódicos. Esta abordagem permitiria a identificação de vulnerabilidades complexas, como falhas de lógica de negócio, que ferramentas automatizadas frequentemente não conseguem detectar.

REFERÊNCIAS

ALGHAWLI, A. S. A.; RADIVILOVA, T. Resilient cloud cluster with devsecops security model, automates a data analysis, vulnerability search and risk calculation. **Alexandria Engineering Journal**, v. 107, p. 136–149, 2024. Disponível em: https://www.sciencedirect.com/science/article/pii/S1110016824007567. Citado nas páginas 13, 17, 20, 24 e 25.

ANCHORE. **Grype: A Vulnerability Scanner for Container Images and Filesystems**. 2025. Disponível em: https://github.com/anchore/grype. Acesso em: 14 jul. 2025. Citado nas páginas 13 e 21.

BARESI, L. *et al.* A qualitative and quantitative analysis of container engines. **arXiv preprint arXiv:2303.04080**, 2023. Disponível em: https://arxiv.org/abs/2303.04080. Acesso em: 15 jul. 2025. Citado nas páginas 13, 17 e 20.

BENEVIDES, P. V. S. **BioDockFlow: Um Processo de Manutenção Baseado em Contêineres Docker para Aplicações Web Heterogêneas de Bioinformática**. 2024. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Universidade do Estado da Bahia. Disponível em: https://saberaberto.uneb.br/items/806ac4b7-4f47-478f-8a1d-1a0add21337d. Repositório: https://github.com/G2BC/BioDockFlow. Acesso em: 11 jun. 2025. Citado nas páginas 14, 17, 22, 24, 25, 26, 29, 30, 31, 32, 33, 41 e 43.

BOETTIGER, C. An introduction to docker for reproducible research. **SIGOPS Operating Systems Review**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 1, p. 71–79, 2015. Disponível em: https://doi.org/10.1145/2723872.2723882. Acesso em: 15 jul. 2025. Disponível em: https://doi.org/10.1145/2723872.2723882. Citado nas páginas 13, 16, 17, 20 e 24.

CHESS, B.; WEST, J. **Secure programming with static analysis**. [S.l.]: Addison-Wesley Professional, 2007. Citado na página 21.

COMMUNITY, T. G. The Galaxy platform for accessible, reproducible, and collaborative data analyses: 2024 update. **Nucleic Acids Res.**, Oxford University Press, v. 52, n. W1, p. W83–W94, 2024. Disponível em: https://doi.org/10.1093/nar/gkae410. Citado na página 16.

Docker Inc. **Docker Overview Whitepaper**. 2016. Disponível em: https://www.docker.com/resources/what-container. Acesso em: 15 jul. 2025. Citado nas páginas 17, 18 e 19.

Docker Inc. **What is a container image?** 2024. Disponível em: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Acesso em: 14 jul. 2025. Citado na página 19.

FELTER, W. *et al.* An updated performance comparison of virtual machines and linux containers. In: **2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.]: IEEE, 2015. p. 171–172. Citado na página 18.

FOUNDATION, C. N. C. **Cloud Native Computing Foundation Annual Survey 2023**. 2023. Relatório licenciado sob a Licença Creative Commons Atribuição-SemDerivações 4.0 Internacional. Disponível em: https://www.cncf.io/reports/cncf-annual-survey-2023/. Citado nas páginas 13, 18 e 21.

G2BC. **LUMM-server: API Backend do Sistema Cogumelos Luminescentes**. 2025. Repositório de código-fonte. Disponível em: https://github.com/G2BC/LUMM-server. Citado na página 34.

G2BC. **LUMM-web: Interface Frontend do Sistema Cogumelos Luminescentes**. 2025. Repositório de código-fonte. Disponível em: https://github.com/G2BC/LUMM-web. Citado na página 34.

GOECKS, L. S. *et al.* Design science research in practice: review of applications in industrial engineering. **Gestão & Produção**, Universidade Federal de São Carlos, v. 28, n. 4, p. e5811, 2021. ISSN 0104-530X. Disponível em: https://doi.org/10.1590/1806-9649-2021v28e5811. Citado na página 24.

GRÜNING, B. *et al.* Biocontainers: an open-source and community-driven framework for software standardization. **Bioinformatics**, Oxford University Press, v. 34, n. 13, p. 2190–2192, 2018. Citado nas páginas 13, 16, 17 e 25.

HAQ, M. S. *et al.* SoK: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In: **Anais do 2024 IEEE Symposium on Security and Privacy (SP)**. IEEE, 2024. p. 4573–4590. Disponível em: https://ieeexplore.ieee.org/abstract/document/10646668/?casa_token=83N9S0-BKWEAAAAA:-EfM5e2zGveWhbBgI4GNDfZPT19IU6brwLJOS0E0nKLFW0YsR9xTUlYUoms27Flj1qyxG3R-hKw. Citado na página 22.

HAQUE, M. U.; BABAR, M. A. Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities. In: **Anais do 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. IEEE, 2022. p. 1066–1077. Disponível em: https://ieeexplore.ieee.org/abstract/document/9825857/?casa_token=yd6oVeKDrdsAAAAA:w1ezUPXKSHTqbjiH8yZ-hs3o1D_oEJf_fbJj-otdJx4Kb1DYiOSXbUTrj2qVgdMaKY50rep7QiU. Citado nas páginas 13, 17, 20, 21, 22 e 24.

HER, J. *et al.* Kuberosy: A dynamic system call filtering framework for containers. **IEEE Access**, 2024. Disponível em: IEEE Xplore. Disponível em: https://ieeexplore.ieee.org/abstract/document/10736597/. Citado nas páginas 17, 20 e 23.

JARKAS, O. *et al.* A container security survey: exploits, attacks, and defenses. **ACM Computing Surveys**, v. 57, n. 7, p. 1–36, 2025. Disponível em: ACM Digital Library. Disponível em: https://dl.acm.org/doi/10.1145/3715001. Citado nas páginas 13, 17, 20 e 22.

MANIRAJ, S. P.; RANGANATHAN, C. S.; SEKAR, S. Securing web applications with owasp zap for comprehensive security testing. **International Journal of Advances in Signal and Image Sciences**, XLESCIENCE, v. 10, n. 2, p. 27–36, 2024. Citado na página 22.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, Linux New Media USA, LLC, v. 2014, n. 239, 2014. Disponível em: https://linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment. Acesso em: 15 jul. 2025. Citado nas páginas 13 e 19.

NIMMAGADDA, S. *et al.* Linux namespaces and cgroups as os primitives for lightweight virtualization: Architecture, isolation mechanisms, and performance evaluation. **arXiv preprint arXiv:1802.01164**, 2018. Disponível em: https://arxiv.org/abs/1802.01164. Acesso em: 15 jul. 2025. Citado nas páginas 18 e 19.

OWASP Foundation. **DevSecOps Maturity Model**. 2023. Disponível em: https://owasp.org/www-project-devsecops-maturity-model/. Acesso em: 14 jul. 2025. Citado nas páginas 20 e 21.

PEFFERS, K. *et al.* A design science research methodology for information systems research. **Journal of Management Information Systems**, Taylor & Francis, v. 24, n. 3, p. 45–77, 2007. Citado na página 24.

PIMENTEL, M.; FILIPPO, D.; SANTOS, T. M. d. Design science research: pesquisa científica atrelada ao design de artefatos. **Revista de Educação a Distância e Elearning**, v. 3, n. 1, 2020. Disponível em: https://revistas.rcaap.pt/lead_read/article/view/21898. Citado na página 24.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de Software: Uma Abordagem Profissional**. 8. ed. [S.l.]: AMGH Editora, 2014. Citado nas páginas 17, 20 e 21.

QUAY. **Clair: Vulnerability Static Analysis for Containers**. 2025. Disponível em: https://github.com/quay/clair. Acesso em: 14 jul. 2025. Citado nas páginas 13 e 21.

ROSEN, R. Namespaces and cgroups—the basis of linux containers. In: **NetDev Conference**. [S.l.: s.n.], 2015. Disponível em: https://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf. Acesso em: 15 jul. 2025. Citado na página 18.

SECURITY, A. **Trivy: Vulnerability Scanner for Containers and Other Artifacts**. 2025. Disponível em: https://github.com/aquasecurity/trivy. Acesso em: 14 jul. 2025. Citado nas páginas 13, 21 e 29.

SOLTESZ, S. *et al.* Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: **ACM SIGOPS Operating Systems Review**. [S.l.]: ACM, 2007. v. 41, n. 3, p. 275–287. Citado na página 18.

SOMMERVILLE, I. **Software Engineering**. 9. ed. [S.l.]: Addison-Wesley, 2011. Citado na página 17.

WILSON, G. *et al.* **Ten Essential Guidelines for Building High-Quality Research Software**. 2025. Citado na página 13.

Apêndices

APÊNDICE A – DOCKERFILE DA APLICAÇÃO (BACKEND) LUMM

Este apêndice apresenta o código-fonte do arquivo `Dockerfile` utilizado para a construção da imagem do *backend* (Python) da aplicação LUMM. O arquivo define a imagem base, as dependências do sistema operacional e as configurações de execução do servidor.

Listagem A.1 – Dockerfile da Aplicação (Backend) LUMM.

```
FROM python:3.10-slim-bullseye AS base

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends
    curl \
  && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

RUN sed -i "s/\r$//" entrypoint.sh && chmod +x entrypoint.sh

HEALTHCHECK --interval=10s --timeout=3s --retries=5 \
  CMD curl -fsS http://localhost:4000/health || exit 1

ENTRYPOINT ["./entrypoint.sh"]
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE B – DOCKERFILE DA APLICAÇÃO (FRONTEND) LUMM

Este apêndice apresenta o código-fonte do arquivo `Dockerfile` utilizado para a construção da imagem do *frontend* (React) da aplicação LUMM. O arquivo utiliza a estratégia de *multi-stage build* para otimizar o tamanho final da imagem, servindo os arquivos estáticos através do servidor Nginx.

Listagem B.1 – Dockerfile da Aplicação (Frontend) LUMM.

```
# BASE
FROM node:22-alpine AS base

WORKDIR /app

COPY package*.json .

RUN npm ci

COPY . .

# DEV
FROM base AS dev

ENV HOST=0.0.0.0
ENV PORT=5173
EXPOSE 5173

CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0", "--port",
   "5173"]

# BUILD
FROM base AS build
ENV NODE_ENV=production

ARG VITE_API_URL
ARG VITE_API_KEY
ARG VITE_ROBOTS
ARG VITE_SITE_URL

ENV VITE_API_URL=${VITE_API_URL}
```

```
ENV VITE_API_KEY=${VITE_API_KEY}
ENV VITE_ROBOTS=${VITE_ROBOTS}
ENV VITE_SITE_URL=${VITE_SITE_URL}

RUN npm run build

# PROD
FROM nginx:1.29.3-alpine-slim AS prod

COPY --from=build /app/dist /usr/share/nginx/html

COPY ./nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

RUN apk add --no-cache curl=8.14.1-r2

HEALTHCHECK --interval=30s --timeout=5s --start-period=10s
   --retries=3 \
  CMD curl -fsS http://localhost:80/pt || exit 1

CMD ["nginx", "-g", "daemon off;"]
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE C – WORKFLOW DE CI (BACKEND) DO LUMM

Este apêndice detalha a configuração do *workflow* de CI para o *backend*. O código define os passos automatizados para a construção da imagem, a execução da varredura de vulnerabilidades (*scan*) com a ferramenta Trivy, o cálculo do EPV e a verificação do limiar de segurança (*security gate*).

Listagem C.1 – Workflow de CI (backend) do LUMM.

```yaml
name: CI

on:
  pull_request:
    branches: [main]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Buildx
        uses: docker/setup-buildx-action@v3
      - name: Login GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - name: Build image only (no push)
        uses: docker/build-push-action@v6
        with:
          context: .
          file: ./Dockerfile
          push: false
          tags: ghcr.io/${{ secrets.REPOSITORY_OWNER
             }}/lumm-server:ci

```

```
31  scan:
32    runs-on: ubuntu-latest
33    needs: build
34    steps:
35      - uses: actions/checkout@v4
36      - name: Ensure dirs
37        run: mkdir -p artifacts
38      - name: Login GHCR
39        uses: docker/login-action@v3
40        with:
41          registry: ghcr.io
42          username: ${{ github.actor }}
43          password: ${{ secrets.GITHUB_TOKEN }}
44      - name: Install Trivy CLI
45        run: |
46          curl -sfL
               https://raw.githubusercontent.com/aquasecurity/ \
47            trivy/main/contrib/install.sh | sh -s -- -b
                 /usr/local/bin
48      - name: Trivy fs
49        run: |
50          trivy fs . \
51            --scanners vuln,secret \
52            --severity LOW,MEDIUM,HIGH,CRITICAL \
53            --ignore-unfixed \
54            -f table -o artifacts/trivy-fs.txt || true
55          trivy fs . \
56            --scanners vuln,secret \
57            --severity LOW,MEDIUM,HIGH,CRITICAL \
58            --ignore-unfixed \
59            -f json -o artifacts/trivy-fs.json || echo '{}' >
                 artifacts/trivy-fs.json
60      - name: Trivy image
61        run: |
62          docker build -t ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-server:ci .
63          trivy image ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-server:ci \
64            --severity LOW,MEDIUM,HIGH,CRITICAL \
65            --ignore-unfixed \
66            -f table -o artifacts/trivy-image.txt || true
67          trivy image ghcr.io/${{ secrets.REPOSITORY_OWNER
```

```
          }}/lumm-server:ci \
68          --severity LOW,MEDIUM,HIGH,CRITICAL \
69          --ignore-unfixed \
70          -f json -o artifacts/trivy-image.json || echo '{}' >
              artifacts/trivy-image.json
71      - name: Trivy config
72        run: |
73          trivy config . \
74            --severity LOW,MEDIUM,HIGH,CRITICAL \
75            -f table -o artifacts/trivy-config.txt || true
76          trivy config . \
77            --severity LOW,MEDIUM,HIGH,CRITICAL \
78            -f json -o artifacts/trivy-config.json || echo '{}'
                > artifacts/trivy-config.json
79      - name: Calculate EPV
80        run: |
81          mkdir -p artifacts
82          C=$(jq '[.Results[].Vulnerabilities[]? |
                select(.Severity=="CRITICAL")] | length'
                artifacts/trivy-image.json)
83          H=$(jq '[.Results[].Vulnerabilities[]? |
                select(.Severity=="HIGH")] | length'
                artifacts/trivy-image.json)
84          M=$(jq '[.Results[].Vulnerabilities[]? |
                select(.Severity=="MEDIUM")] | length'
                artifacts/trivy-image.json)
85          L=$(jq '[.Results[].Vulnerabilities[]? |
                select(.Severity=="LOW")] | length'
                artifacts/trivy-image.json)
86          EPV=$(echo "scale=2;
                (1.0*$C)+(0.75*$H)+(0.5*$M)+(0.25*$L)" | bc)
87          echo "$EPV" > artifacts/epv.txt
88          if [ ! -f artifacts/epv.csv ]; then echo
                'date,commit,epv' > artifacts/epv.csv; fi
89          printf '%s,%s,%s\n' "$(date +%F)" "${GITHUB_SHA::7}"
                "$EPV" >> artifacts/epv.csv
90      - name: Generate EPV graph
91        run: |
92          pip install matplotlib pandas seaborn
93          python3 scripts/plot_epv.py
94      - name: Check EPV threshold
95        run: |
```

```
96          EPV=$(< artifacts/epv.txt)
97          echo "EPV detected: $EPV (threshold: $EPV_THRESHOLD)"
98          RESULT=$(echo "$EPV >= $EPV_THRESHOLD" | bc)
99          C=$(jq '[.Results[].Vulnerabilities[]? |
               select(.Severity=="CRITICAL")] | length'
               artifacts/trivy-image.json)
100         H=$(jq '[.Results[].Vulnerabilities[]? |
               select(.Severity=="HIGH")] | length'
               artifacts/trivy-image.json)
101         M=$(jq '[.Results[].Vulnerabilities[]? |
               select(.Severity=="MEDIUM")] | length'
               artifacts/trivy-image.json)
102         L=$(jq '[.Results[].Vulnerabilities[]? |
               select(.Severity=="LOW")] | length'
               artifacts/trivy-image.json)
103         echo "### Security Scan Summary" >>
               $GITHUB_STEP_SUMMARY
104         echo "" >> $GITHUB_STEP_SUMMARY
105         echo "| Severity | Count |" >> $GITHUB_STEP_SUMMARY
106         echo "|---|---|" >> $GITHUB_STEP_SUMMARY
107         echo "| CRITICAL | $C |" >> $GITHUB_STEP_SUMMARY
108         echo "| HIGH | $H |" >> $GITHUB_STEP_SUMMARY
109         echo "| MEDIUM | $M |" >> $GITHUB_STEP_SUMMARY
110         echo "| LOW | $L |" >> $GITHUB_STEP_SUMMARY
111         echo "" >> $GITHUB_STEP_SUMMARY
112         echo "- **EPV**: \`$EPV\`" >> $GITHUB_STEP_SUMMARY
113         echo "- **Threshold**: \`$EPV_THRESHOLD\`" >>
               $GITHUB_STEP_SUMMARY
114         if [ "$RESULT" -eq 1 ]; then
115           echo "- **Rejected**: EPV >= threshold" >>
                 $GITHUB_STEP_SUMMARY
116           exit 1
117         else
118           echo "- **Approved**: EPV below threshold" >>
                 $GITHUB_STEP_SUMMARY
119         fi
120       env:
121         EPV_THRESHOLD: ${{ vars.EPV_THRESHOLD || 5 }}
122     - name: Upload artifacts
123       if: always()
124       uses: actions/upload-artifact@v4
125       with:
```

```
126           name: server-security-scan
127           path: artifacts/*
128       - name: Clean workspace
129         if: always()
130         run: |
131           docker container prune -f
132           docker volume prune -f
133           rm -rf artifacts/
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE D – WORKFLOW DE CI (FRONTEND) DO LUMM

Este apêndice detalha a configuração do *workflow* de CI para o *frontend*. O arquivo contém as instruções para o *build* da aplicação web e a execução das análises de segurança estática (SAST) na imagem e nos arquivos de configuração.

Listagem D.1 – Workflow de CI (frontend) do LUMM.

```
name: CI

on:
  pull_request:
    branches: [main]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build image only (no push)
        uses: docker/build-push-action@v6
        with:
          context: .
          file: ./Dockerfile
          push: false
          tags: ghcr.io/${{ secrets.REPOSITORY_OWNER
```

```
            }}/lumm-web:ci
          build-args: |
            VITE_API_URL=${{ secrets.VITE_API_URL }}
            VITE_API_KEY=${{ secrets.VITE_API_KEY }}
            VITE_ROBOTS=${{ secrets.VITE_ROBOTS }}
            VITE_SITE_URL=${{ secrets.VITE_SITE_URL }}

  scan:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - uses: actions/checkout@v4
      - name: Ensure dir
        run: mkdir -p artifacts

      - name: Login GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Install Trivy CLI
        run: |
          curl -sfL
             https://raw.githubusercontent.com/aquasecurity/ \
          trivy/main/contrib/install.sh | sh -s -- -b
             /usr/local/bin

      - name: Trivy fs
        run: |
          mkdir -p artifacts
          trivy fs . \
            --scanners vuln,secret \
            --severity LOW,MEDIUM,HIGH,CRITICAL \
            --ignore-unfixed \
            -f table -o artifacts/trivy-fs.txt || true

          trivy fs . \
            --scanners vuln,secret \
            --severity LOW,MEDIUM,HIGH,CRITICAL \
            --ignore-unfixed \
```

```
72            -f json -o artifacts/trivy-fs.json || echo '{}' >
                 artifacts/trivy-fs.json
73
74      - name: Trivy image
75        run: |
76          if ! docker pull ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-web:latest; then
77          docker build -t ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-web:latest .
78          fi
79
80          # Table format
81          trivy image ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-web:latest \
82            --severity LOW,MEDIUM,HIGH,CRITICAL \
83            --ignore-unfixed \
84            -f table -o artifacts/trivy-image.txt || true
85
86          # JSON format
87          trivy image ghcr.io/${{ secrets.REPOSITORY_OWNER
               }}/lumm-web:latest \
88            --severity LOW,MEDIUM,HIGH,CRITICAL \
89            --ignore-unfixed \
90            -f json -o artifacts/trivy-image.json || echo '{}' >
                 artifacts/trivy-image.json
91
92      - name: Trivy config
93        run: |
94          mkdir -p artifacts
95          trivy config . \
96            --severity LOW,MEDIUM,HIGH,CRITICAL \
97            -f table -o artifacts/trivy-config.txt || true
98
99          trivy config . \
100           --severity LOW,MEDIUM,HIGH,CRITICAL \
101           -f json -o artifacts/trivy-config.json || echo '{}'
                 > artifacts/trivy-config.json
102
103     - name: Calculate EPV
104       run: |
105         mkdir -p artifacts
106
```

```
107           C=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="CRITICAL")] | length'
                  artifacts/trivy-image.json)
108           H=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="HIGH")] | length'
                  artifacts/trivy-image.json)
109           M=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="MEDIUM")] | length'
                  artifacts/trivy-image.json)
110           L=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="LOW")] | length'
                  artifacts/trivy-image.json)
111
112           echo "Detected failures - CRITICAL: $C, HIGH: $H,
                  MEDIUM: $M, LOW: $L"
113
114           EPV=$(echo "scale=2; (1.0*$C) + (0.75*$H) + (0.5*$M) +
                  (0.25*$L)" | bc)
115           echo "Calculated EPV: $EPV"
116           echo "$EPV" > artifacts/epv.txt
117
118           if [ ! -f artifacts/epv.csv ]; then
119             echo 'date,commit,epv' > artifacts/epv.csv
120           fi
121
122           printf '%s,%s,%s\n' "$(date +%F)" "${GITHUB_SHA::7}"
                  "$EPV" >> artifacts/epv.csv
123
124       - name: Generate EPV graph
125         run: |
126           pip install matplotlib pandas seaborn
127           python3 scripts/plot_epv.py
128
129       - name: Check EPV threshold
130         run: |
131           EPV=$(< artifacts/epv.txt)
132           echo "EPV detected: $EPV (threshold: $EPV_THRESHOLD)"
133           RESULT=$(echo "$EPV >= $EPV_THRESHOLD" | bc)
134
135           C=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="CRITICAL")] | length'
                  artifacts/trivy-image.json)
```

```
136           H=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="HIGH")] | length'
                  artifacts/trivy-image.json)
137           M=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="MEDIUM")] | length'
                  artifacts/trivy-image.json)
138           L=$(jq '[.Results[].Vulnerabilities[]? |
                  select(.Severity=="LOW")] | length'
                  artifacts/trivy-image.json)
139
140           echo "### Security Scan Summary" >>
                  $GITHUB_STEP_SUMMARY
141           echo "" >> $GITHUB_STEP_SUMMARY
142           echo "| Severity  | Count |" >> $GITHUB_STEP_SUMMARY
143           echo "|-----------|-------|" >> $GITHUB_STEP_SUMMARY
144           echo "| CRITICAL  |   $C   |" >> $GITHUB_STEP_SUMMARY
145           echo "| HIGH      |   $H   |" >> $GITHUB_STEP_SUMMARY
146           echo "| MEDIUM    |   $M   |" >> $GITHUB_STEP_SUMMARY
147           echo "| LOW       |   $L   |" >> $GITHUB_STEP_SUMMARY
148           echo "" >> $GITHUB_STEP_SUMMARY
149           echo "- **EPV**: \`$EPV\`" >> $GITHUB_STEP_SUMMARY
150           echo "- **Threshold**: \`$EPV_THRESHOLD\`" >>
                  $GITHUB_STEP_SUMMARY
151
152           if [ "$RESULT" -eq 1 ]; then
153             echo "- **Rejected**: EPV >= threshold" >>
                    $GITHUB_STEP_SUMMARY
154             exit 1
155           else
156             echo "- **Approved**: EPV below threshold" >>
                    $GITHUB_STEP_SUMMARY
157           fi
158         env:
159           EPV_THRESHOLD: ${{ vars.EPV_THRESHOLD || 5 }}
160
161       - name: Upload artifacts
162         if: always()
163         uses: actions/upload-artifact@v4
164         with:
165           name: web-security-scan
166           path: artifacts/*
167
```

```
168       - name: Clean workspace
169         if: always()
170         run: |
171           echo "Cleaning up..."
172           docker container prune -f
173           docker volume prune -f
174           rm -rf artifacts/
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE E – WORKFLOW DE CD (BACKEND) DO LUMM

Este apêndice apresenta o código do *workflow* de CD para o *backend*. O arquivo descreve a automação responsável por construir a imagem final aprovada e publicá-la no GHCR para implantação.

Listagem E.1 – Workflow de CD (backend) do LUMM.

```yaml
name: CD

on:
  push:
    branches:
      - main

permissions:
  contents: read
  packages: write

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Docker Buildx
        uses: docker/setup-buildx-action@v3
      - name: Login GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - name: Build and Push Image
        uses: docker/build-push-action@v6
        with:
          context: .
          file: ./Dockerfile
          push: true
          tags: |
            ghcr.io/${{ secrets.REPOSITORY_OWNER
```

```
            }}/lumm-server:${{ github.sha }}
33          ghcr.io/${{ secrets.REPOSITORY_OWNF
            }}/lumm-server:latest
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE F – WORKFLOW DE CD (FRONTEND) DO LUMM

Este apêndice apresenta o código do *workflow* de CD para o *frontend*. Ele define os passos para a publicação automática da imagem da interface web no registro de contêineres após a aprovação no ciclo de CI.

Listagem F.1 – Workflow de CD (frontend) do LUMM.

```yaml
name: CD

on:
  push:
    branches:
      - main

permissions:
  contents: read
  packages: write

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and Push Image
        uses: docker/build-push-action@v6
        with:
          context: .
          file: ./Dockerfile
```

```
33            push: true
34            tags: |
35              ghcr.io/${{ secrets.REPOSITORY_OWNER }}/lumm-web:${{
                   github.sha }}
36              ghcr.io/${{ secrets.REPOSITORY_OWNER
                   }}/lumm-web:latest
37            build-args: |
38              VITE_API_URL=${{ secrets.VITE_API_URL }}
39              VITE_API_KEY=${{ secrets.VITE_API_KEY }}
40              VITE_ROBOTS=${{ secrets.VITE_ROBOTS }}
41              VITE_SITE_URL=${{ secrets.VITE_SITE_URL }}
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE G – WORKFLOW DE DAST (BACKEND) DO LUMM

Este apêndice documenta o *workflow* de DAST para o *backend*. O código configura a execução agendada ou acionada por eventos da ferramenta OWASP ZAP contra a API em produção, visando identificar vulnerabilidades em tempo de execução.

Listagem G.1 – Workflow de DAST (backend) do LUMM.

```yaml
name: DAST

on:
  workflow_run:
    workflows: [ "Nova release"]
    types: [ completed]
  schedule:
    - cron: "0 3 */14 * *"
  workflow_dispatch:

permissions:
  contents: read
  packages: read

jobs:
  zap_scan:
    runs-on: ubuntu-latest
    if: >
      github.event_name != 'workflow_run' ||
      (github.event.workflow_run.conclusion == 'success')
    steps:
      - uses: actions/checkout@v4
      - name: OWASP ZAP API Scan (prod)
        uses: zaproxy/action-api-scan@v0.9.0
        env:
          ZAP_AUTH_HEADER: X-API-Key
          ZAP_AUTH_HEADER_VALUE: ${{ secrets.API_KEY }}
        with:
          target: ${{ secrets.DAST_TARGET }}
          cmd_options: "-I"
          allow_issue_writing: false
          artifact_name: "dast"
```

```
33      - name: Collect DAST reports
34        run: |
35          mkdir -p artifacts
36          mv zap_report.* artifacts/ || true
37      - name: Upload DAST Artifacts
38        uses: actions/upload-artifact@v4
39        with:
40          name: server-dast
41          path: artifacts/*
42      - name: Cleanup
43        if: always()
44        run: |
45          docker rm -f api || true
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE H – WORKFLOW DE DAST (FRONTEND) DO LUMM

Este apêndice documenta o *workflow* de DAST para o *frontend*. O arquivo define a automação para varredura da aplicação web em produção utilizando o OWASP ZAP, focando na detecção de falhas de configuração de segurança e cabeçalhos HTTP.

Listagem H.1 – Workflow de DAST (frontend) do LUMM.

```yaml
name: DAST

on:
  workflow_run:
    workflows: ["Nova release"]
    types: [completed]
  schedule:
    - cron: "0 3 */14 * *"
  workflow_dispatch:

permissions:
  contents: read
  packages: read

jobs:
  dast:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: OWASP ZAP Baseline
        uses: zaproxy/action-baseline@v0.14.0
        with:
          target: ${{ secrets.DAST_TARGET }}
          cmd_options: "-I"
          allow_issue_writing: false
          artifact_name: "dast"
        continue-on-error: true

      - name: Collect DAST reports
        run: |
          mkdir -p artifacts
```

```
33          mv zap_report.* artifacts/ || true
34
35      - name: Upload DAST Artifacts
36        uses: actions/upload-artifact@v4
37        with:
38          name: web-dast
39          path: artifacts/*
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE I – RESULTADOS CONSOLIDADOS DO ESCANEAMENTO (BACKEND - ANTES DA MITIGAÇÃO)

Este apêndice contém o relatório consolidado gerado pela ferramenta Trivy durante a análise inicial da imagem do *backend*, antes da aplicação das mitigações. Os dados evidenciam as vulnerabilidades e más configurações detectadas, servindo de base para o cálculo do EPV inicial.

Listagem I.1 – Relatório de Má Configuração do Dockerfile (trivy config).

```
Report Summary

+------------+------------+--------------------+
|   Target   |    Type    | Misconfigurations  |
+------------+------------+--------------------+
| Dockerfile | dockerfile |         1          |
+------------+------------+--------------------+
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)


Dockerfile (dockerfile)
=======================
Tests: 27 (SUCCESSES: 26, FAILURES: 1)
Failures: 1 (LOW: 0, MEDIUM: 0, HIGH: 1, CRITICAL: 0)

AVD-DS-0002 (HIGH): Specify at least 1 USER command in
   Dockerfile with non-root user as argument
========================================
Running containers with 'root' user can lead to a container
   escape situation.
 It is a best practice to run containers as non-root users,
    which can be done by adding a 'USER' statement to the
    Dockerfile.
 See https://avd.aquasec.com/misconfig/ds002
----------------------------------------
```

Fonte: Elaborado pelo autor (2025).

Listagem I.2 – Relatório de Vulnerabilidades do Filesystem (trivy fs).

```
Report Summary

+------------------+------+-----------------+---------+
|      Target      | Type | Vulnerabilities | Secrets |
+------------------+------+-----------------+---------+
| requirements.txt | pip  |        8        |    -    |
+------------------+------+-----------------+---------+
...
requirements.txt (pip)
======================
Total: 8 (LOW: 0, MEDIUM: 7, HIGH: 1, CRITICAL: 0)

+------------+----------------+----------+--------+--------------+
|  Library   | Vulnerability  | Severity | Status | Installed
   Version | Fixed Version |                              Title
                                |
+------------+----------------+----------+--------+--------------+
| Flask-Cors | CVE-2024-6839  | MEDIUM   | fixed  | 5.0.0
               | 6.0.0         | corydolphin/flask-cors version
   4.0.1 contains an improper... |
| Flask-Cors | CVE-2024-6844  | MEDIUM   | fixed  | 5.0.0
               | 6.0.0         | A vulnerability in
   corydolphin/flask-cors version 4.0.1...   |
| Flask-Cors | CVE-2024-6866  | MEDIUM   | fixed  | 5.0.0
               | 6.0.0         | corydolphin/flask-cors version
   4.01 contains a...            |
| urllib3    | CVE-2025-50181 | MEDIUM   | fixed  | 2.2.3
               | 2.5.0         | urllib3: urllib3 redirects are
   not disabled when retries are...|
| urllib3    | CVE-2025-50182 | MEDIUM   | fixed  | 2.2.3
               | 2.5.0         | urllib3: urllib3 does not
   control redirects in browsers and... |
| werkzeug   | CVE-2024-34069 | HIGH     | fixed  | 2.3.8
               | 3.0.3         | python-werkzeug: user may
   execute code on a developer's...   |
| werkzeug   | CVE-2024-49766 | MEDIUM   | fixed  | 2.3.8
               | 3.0.6         | werkzeug: python-werkzeug:
   Werkzeug safe_join not safe on... |
| werkzeug   | CVE-2024-49767 | MEDIUM   | fixed  | 2.3.8
               | 3.0.6         | werkzeug: python-werkzeug:
   Werkzeug possible resource...     |
```

```
24 +------------+-----------------+----------+--------+--------------+
```

Fonte: Elaborado pelo autor (2025).

Listagem I.3 – Resumo do Relatório de Vulnerabilidades da Imagem (trivy image).

```
Report Summary

+------------+----------------+----------+--------+--------------+
|                            Target                                |
      Type    | Vulnerabilities | Secrets |
+------------+----------------+----------+--------+--------------+
| ghcr.io/g2bc/lumm-server:ci (debian 12.7)
   |   debian   |       53        |    -    |
| ... (pacotes python omitidos para brevidade) ...                 |
   python-pkg |       12        |    -    |
+------------+----------------+----------+--------+--------------+

ghcr.io/g2bc/lumm-server:ci (debian 12.7)
=========================================
Total: 53 (LOW: 9, MEDIUM: 33, HIGH: 10, CRITICAL: 1)

+------------+----------------+----------+--------+--------------+
|   Library    | Vulnerability  | Severity | Status | Installed
   Version |  Fixed Version   |                              Title
                               |
+------------+----------------+----------+--------+--------------+
| libc-bin     | CVE-2025-4802  | HIGH     | fixed  |
   2.36-9+deb12u8    | 2.36-9+deb12u11  | glibc: static setuid
   binary dlopen may incorrectly search... |
| libc6        | CVE-2025-4802  | HIGH     | fixed  |
   2.36-9+deb12u8    | 2.36-9+deb12u11  | glibc: static setuid
   binary dlopen may incorrectly search... |
| libexpat1    | CVE-2023-52425 | HIGH     | fixed  |
   2.5.0-1+deb12u1   | 2.5.0-1+deb12u2  | expat: parsing large
   tokens can trigger a denial of service  |
| libgnutls30  | CVE-2025-32988 | HIGH     | fixed  |
   3.7.9-2+deb12u3   | 3.7.9-2+deb12u5  | gnutls: Vulnerability
   in GnuTLS otherName SAN export          |
| libgnutls30  | CVE-2025-32990 | HIGH     | fixed  |
   3.7.9-2+deb12u3   | 3.7.9-2+deb12u5  | gnutls: Vulnerability
   in GnuTLS certtool template parsing     |
| liblzma5     | CVE-2025-31115 | HIGH     | fixed  | 5.4.1-0.2
           | 5.4.1-1          | xz: XZ has a heap-use-after-free
   bug in threaded .xz decoder |
| libsqlite3-0 | CVE-2025-6965  | CRITICAL | fixed  | 3.40.1-2
            | 3.40.1-2+deb12u2 | sqlite: Integer Truncation in
```

```
   SQLite                              |
| libsqlite3-0 | CVE-2023-7104   | HIGH     | fixed  | 3.40.1-2
           | 3.40.1-2+deb12u1 | sqlite: heap-buffer-overflow at
   sessionfuzz                    |
| perl-base    | CVE-2023-31484 | HIGH     | fixed  |
   5.36.0-7+deb12u1  | 5.36.0-7+deb12u3 | perl: CPAN.pm does not
   verify TLS certificates when...        |
| perl-base    | CVE-2024-56406 | HIGH     | fixed  |
   5.36.0-7+deb12u1  | 5.36.0-7+deb12u2 | perl: Perl 5.34, 5.36,
   5.38 and 5.40 are vulnerable to a...   |
| ... (42 vulnerabilidades LOW/MEDIUM omitidas para brevidade)
   ...      |
+------------+----------------+----------+--------+--------------+

Python (python-pkg)
===================
Total: 12 (LOW: 0, MEDIUM: 8, HIGH: 4, CRITICAL: 0)

+------------+----------------+----------+--------+--------------+
|        Library        | Vulnerability  | Severity | Status |
   Installed Version | Fixed Version |
   Title                              |
+------------+----------------+----------+--------+--------------+
| Werkzeug (METADATA)   | CVE-2024-34069 | HIGH     | fixed  |
   2.3.8             | 3.0.3          | python-werkzeug: user may
   execute code on a developer's...    |
| setuptools (METADATA) | CVE-2022-40897 | HIGH     | fixed  |
   57.5.0            | 65.5.1         | ppa-setuptools: Regular
   Expression Denial of Service...       |
| setuptools (METADATA) | CVE-2024-6345  | HIGH     | fixed  |
   57.5.0            | 70.0.0         | pypa/setuptools: Remote
   code execution via download...         |
| setuptools (METADATA) | CVE-2025-47273 | HIGH     | fixed  |
   57.5.0            | 78.1.1         | setuptools: Path Traversal
   Vulnerability in setuptools...     |
| ... (8 vulnerabilidades MEDIUM omitidas para brevidade) ...
           |
+------------+----------------+----------+--------+--------------+
```

Fonte: Elaborado pelo autor (2025).

APÊNDICE J – RESULTADOS CONSOLIDADOS DO ESCANEAMENTO (FRONTEND - ANTES DA MITIGAÇÃO)

Este apêndice contém o relatório consolidado gerado pela ferramenta Trivy durante a análise inicial da imagem do *frontend*, antes da aplicação das mitigações. O documento lista as falhas de segurança identificadas nas dependências e no sistema de arquivos, fundamentando a necessidade de intervenção.

Listagem J.1 – Relatório de Má Configuração do Dockerfile (trivy config).

```
Report Summary

+------------+------------+--------------------+
|   Target   |    Type    | Misconfigurations  |
+------------+------------+--------------------+
| Dockerfile | dockerfile |         1          | [cite: 1]
+------------+------------+--------------------+
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)


Dockerfile (dockerfile)
=======================
Tests: 27 (SUCCESSES: 26, FAILURES: 1)
Failures: 1 (LOW: 0, MEDIUM: 0, HIGH: 1, CRITICAL: 0) [cite: 1]

AVD-DS-0002 (HIGH): Specify at least 1 USER command in
   Dockerfile with non-root user as argument
========================================
Running containers with 'root' user can lead to a container
   escape situation.
[cite: 2] It is a best practice to run containers as non-root
   users, which can be done by adding a 'USER' statement to the
   Dockerfile.
[cite: 3] See https://avd.aquasec.com/misconfig/ds002
----------------------------------------
```

Fonte: Elaborado pelo autor (2025).

Listagem J.2 – Relatório de Vulnerabilidades do Filesystem (trivy fs).

```
Report Summary

+--------------------+------+-----------------+---------+
|       Target       | Type | Vulnerabilities | Secrets |
+--------------------+------+-----------------+---------+
| package-lock.json | npm  |        1        |    -    | [cite:
   2745]
+--------------------+------+-----------------+---------+
...
package-lock.json (npm)
=======================
Total: 1 (LOW: 0, MEDIUM: 1, HIGH: 0, CRITICAL: 0) [cite: 2748]

+------------+----------------+----------+--------+--------------+
| Library | Vulnerability  | Severity | Status | Installed
   Version |        Fixed Version          |
                            Title                              |
+------------+----------------+----------+--------+--------------+
| vite    | CVE-2025-62522 | MEDIUM   | fixed  | 6.3.6
              | 7.1.11, 7.0.8, 6.4.1, 5.4.21 | vite: vite allows
   server.fs.deny bypass via backslash on | [cite: 2749]
|         |                |          |        |
                      |                                  | Windows
                                                         | [cite: 2750]
+------------+----------------+----------+--------+--------------+
```

Fonte: Elaborado pelo autor (2025).

Listagem J.3 – Resumo do Relatório de Vulnerabilidades da Imagem (trivy image).

```
Report Summary

+------------+----------------+----------+--------+--------------+
|                  Target                   |  Type  |
   Vulnerabilities | Secrets |
+------------+----------------+----------+--------+--------------+
| ghcr.io/g2bc/lumm-web:ci (alpine 3.21.3) | alpine |       26
          |    -    | [cite: 4]
+------------+----------------+----------+--------+--------------+
...
ghcr.io/g2bc/lumm-web:ci (alpine 3.21.3)
========================================
Total: 26 (LOW: 5, MEDIUM: 15, HIGH: 4, CRITICAL: 2)

+------------+----------------+----------+--------+--------------+
| Library | Vulnerability  | Severity | Status | Installed
   Version | Fixed Version |                              Title
                                  |
+------------+----------------+----------+--------+--------------+
| libxml2 | CVE-2025-49794 | CRITICAL | fixed  | 2.13.4-r5
           | 2.13.9-r0      | libxml: Heap use after free (UAF)
   leads to Denial of service | [cite: 47]
| libxml2 | CVE-2025-49796 | CRITICAL | fixed  | 2.13.4-r5
           | 2.13.9-r0      | libxml: Type confusion leads to
   Denial of service (DoS)       | [cite: 51]
| libxml2 | CVE-2025-32414 | HIGH     | fixed  | 2.13.4-r5
           | 2.13.4-r6      | libxml2: Out-of-Bounds Read in
   libxml2                          | [cite: 53]
| libxml2 | CVE-2025-32415 | HIGH     | fixed  | 2.13.4-r5
           | 2.13.4-r6      | libxml2: Out-of-bounds Read in
   xmlSchemaIDCFillNodeTables     | [cite: 55]
| libxml2 | CVE-2025-49795 | HIGH     | fixed  | 2.13.4-r5
           | 2.13.9-r0      | libxml: Null pointer dereference
   leads to Denial of service   | [cite: 58]
| libxml2 | CVE-2025-6021  | HIGH     | fixed  | 2.13.4-r5
           | 2.13.9-r0      | libxml2: Integer Overflow in
   xmlBuildQName() Leads to Stack   | [cite: 61]
| ... (20 vulnerabilidades LOW/MEDIUM omitidas para brevidade)
   ...      |
+------------+----------------+----------+--------+--------------+
```

Fonte: Elaborado pelo autor (2025).