Análise de vulnerabilidades de sistemas computacionais governamentais baseada em diretrizes OWASP

José Marcos Nogueira[1], Daniel F. Macedo[1], Guilherme A. S. Milanez[1], Henrique R. S. Ferreira[1], Lucas André Santos [1], Sabrina S. Leodoro, Luis F. C. Dias [2]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

[2]Diretoria de Análises e Tecnologia de Inteligência
Ministério Público de Minas Gerais (MPMG) – Belo Horizonte, MG – Brasil

{jmarcos, damacedo, guilhermeaguiarsilvamilanez, henriqueferreira}@dcc.ufmg.br

lucasandreufmg@gmail.com, {sleodoro, luisdias}@mpmg.mp.br

Resumo. *Com a digitalização de diversos processos, sistemas governamentais enfrentam desafios cada vez maiores em cibersegurança. Uma abordagem eficaz para aumentar a confiabilidade do software é a identificação e mitigação de vulnerabilidades ainda na fase de desenvolvimento. Este trabalho propõe uma metodologia prática para a detecção e correção de falhas no código, conforme diretrizes da comunidade OWASP e utilizando ferramentas tais como SonarQube e Trivy. Para validar a abordagem, duas aplicações governamentais em uso foram analisadas de forma automatizada e contínua. Os resultados demonstram a eficácia da metodologia na mitigação de vulnerabilidades, reforçando a importância de uma estratégia formal de segurança no desenvolvimento de software governamental.*

1. Introdução

Os sistemas computacionais de órgãos do estado, sejam federais, estaduais ou mesmo municipais, estão se tornando cada vez mais complexos em termos de tecnologias, de volume de dados armazenados e tratados, de capacidade de processamento e de quantidade e diversidade de potenciais usuários. A digitalização de processos e dados, bem como a oferta de serviços aos cidadãos acontece numa velocidade sem precedentes. Essa complexidade e escala, por sua vez, demanda infraestruturas computacionais de software e hardware bastante diversificadas, de alto poder de processamento, capacidade de análise de dados (*data analytics*) e de evolução constante. À parte das evoluções tecnológicas dos sistemas, e muito por causa da diversidade e alto grau de interação entre os diversos componentes dos sistemas, bem como da exposição ao mundo externo, em função da necessidade de prestação de serviços, vem a questão da segurança dos ambientes computacionais.

Como qualquer outro software, os sistemas internos ou externos do governo estão expostos a uma ampla gama de ataques, tanto conhecidos quanto desconhecidos. A robustez de um sistema frente a essas ameaças está diretamente relacionada à presença ou ausência de vulnerabilidades em seu código e infraestrutura. Identificar e mitigar essas falhas é essencial para garantir a segurança e integridade dos dados processados. Esse processo envolve a identificação, classificação e correção de falhas que possam ser exploradas por ameaças internas ou externas, reduzindo riscos de ataques e interrupções nos

serviços. Existem ferramentas na literatura para fazer análises de vulnerabilidades, as quais desempenham um papel importante no processo, possibilitando a detecção automatizada de vulnerabilidades em códigos-fonte, aplicações web, containers e infraestrutura.

Diversas pesquisas atuais propõem técnicas variadas para mitigar vulnerabilidades conhecidas em todas as fases do desenvolvimento de software [Liu et al. 2012]. Neste trabalho, é proposta e explorada uma metodologia de análise de segurança que integra ferramentas de detecção de vulnerabilidades durante a construção dos componentes de software. Essa abordagem visa garantir que um sistema não seja produzido com riscos conhecidos, levando a um desenvolvimento mais seguro e resiliente. Além da identificação de problemas, a análise de vulnerabilidades também contribui para a melhoria contínua dos sistemas, garantindo conformidade com padrões de segurança e boas práticas. Dessa forma, ao integrar a análise de vulnerabilidades no ciclo de vida do software, as organizações conseguem minimizar riscos, fortalecer suas defesas e garantir maior disponibilidade e confiabilidade dos sistemas em operação.

Essa abordagem está de acordo com os princípios da **Confiança Zero** (ou *Zero Trust*) [Rose et al. 2020], que é um modelo de segurança cibernética baseado no princípio de "nunca confie, sempre verifique". Diferentemente dos modelos tradicionais de segurança, que assumem que elementos internos da rede são confiáveis, a Confiança Zero parte do pressuposto de que todas as entidades — sejam usuários, dispositivos ou aplicativos — devem ser constantemente verificadas antes de receberem acesso a recursos sensíveis. O objetivo da metodologia proposta é auxiliar na observação de alguns dos princípios da Confiança Zero, no caso para ambientes computacionais de instituições governamentais.

A literatura atual explora diferentes abordagens para a detecção de vulnerabilidades, com destaque para metodologias baseadas no OWASP. Estudos prévios demonstram que a combinação de ferramentas de análise estática de código (SAST), como SonarQube e IntelliJ, aprimora a identificação de falhas em softwares governamentais [Setiawan et al. 2020]. Outras pesquisas propõem a integração de múltiplas ferramentas para melhorar a detecção de vulnerabilidades em aplicações web, especialmente na presença de módulos de terceiros [Aljabri et al. 2022]. Além disso, iniciativas como Programas de Recompensa por Vulnerabilidades (VRP) foram aplicadas no setor público para fortalecer a segurança cibernética [Chatfield and Reddick 2017].

Embora técnicas de análise de segurança sejam difundidas [Santos et al. 2020], a literatura carece de abordagens que detalhem a aplicação sistemática e formal de metodologias de detecção de vulnerabilidades especificamente no contexto de sistemas governamentais brasileiros, integrando ferramentas open-source de forma contínua. Este estudo visa preencher essa lacuna, propondo e validando uma metodologia prática em software de uma instituição pública. Sua contribuição reside na adaptação e demonstração de um ciclo estruturado de análise de segurança (focando na identificação e avaliação nesta fase do trabalho) para este ambiente crítico, integrando SonarQube e Trivy e promovendo uma verificação alinhada com as diretrizes OWASP, reforçando a necessidade de políticas proativas de segurança.

O restante deste trabalho está estruturado da seguinte forma: a Seção 2 apresenta a fundamentação técnica do trabalho, incluindo o ferramental utilizado; a Seção 3 apre-

senta a metodologia proposta; a Seção 4 descreve estudos de aplicação da metodologia, incluindo discussão de resultados; e, por fim, a Seção 5 apresenta as conclusões.

2. Fundamentação Técnica

Esta seção tem como objetivo apresentar brevemente os fundamentos teóricos e o ferramental que embasaram a elaboração da metodologia de testes de segurança de software proposta. São apresentadas diretrizes de segurança que orientam a realização de testes, a descrição das principais vulnerabilidades que podem comprometer a segurança do sistema e os diferentes tipos de testes utilizados, acompanhados de suas características específicas e propósitos dentro do contexto da metodologia adotada. As ferramentas de software utilizadas no trabalho são também apresentadas.

2.1. Diretrizes de segurança

A comunidade OWASP (Open Worldwide Application Security Project) é uma comunidade aberta dedicada a permitir que organizações projetem, desenvolvam, adquiram, operem e mantenham software para aplicativos seguros e confiáveis[1]. O padrão OWASP propõe um conjunto de regras de recomendações de segurança de aplicações a serem seguidas pelas equipes de desenvolvimento [Idris et al. 2021].

O principal artefato produzido pela comunidade do OWASP consiste em um guia direcionado para desenvolvedores, que identifica as principais vulnerabilidades, os tipos de testes a serem executados, elenca ferramentas para teste e análise, bem como a dinâmica para condução da execução dos respectivos testes.

A comunidade OWASP define um conjunto de diretrizes que são constantemente atualizadas, o que possibilita, quando aplicadas, uma maior garantia de segurança dos sistemas. Devido à alta dinamicidade e das constantes atualizações pela comunidade OWASP, a aplicação das diretrizes do OWASP busca a garantia da segurança de sistemas web, uma vez que as tecnologias e as técnicas empregadas nesses sistemas apresentam bastante dinamicidade, sendo atualizadas frequentemente. As diretrizes vão desde uma relação de vulnerabilidades dos sistemas, passando por recomendações de boas práticas, até um conjunto de projetos que servem como diretrizes e ferramentas para segurança.

2.2. Vulnerabilidades

Vulnerabilidade de software é entendida como uma falha ou condição que pode possibilitar, a um agente externo sem autorização, o acesso ao sistema [Alhazmi et al. 2005]. As vulnerabilidades em um sistema pode comprometer sua segurança, integridade, disponibilidade ou confidencialidade, tanto de suas aplicações quanto de seus dados. Elas podem surgir devido a erros de programação, configurações incorretas, bibliotecas desatualizadas ou práticas inseguras de desenvolvimento [Krsul 1998].

A comunidade OWASP disponibiliza uma lista das dez principais vulnerabilidades que acometem sistemas web, as quais são classificadas quanto à sua criticidade levando em conta fatores como prevalência de fraqueza no sistema, detecção, exploração, impacto técnico [Li 2020], [Ramadlan 2019].

[1] https://owasp.org

2.3. Análise de vulnerabilidades

A análise ou verificação de vulnerabilidades é um processo que identifica e avalia as falhas de segurança em sistemas, redes e aplicações. No desenvolvimento de softwares modernos, a integração contínua (CI) e a entrega contínua (CD) são práticas de amplo uso, possibilitando a automação de testes e implantação de novas versões de maneira rápida e eficiente. A CI foca na automação da integração do código, garantindo que novas alterações sejam testadas constantemente, reduzindo erros e conflitos. Já a CD automatiza a entrega do software, possibilitando implantações frequentes e seguras. Esses princípios trazem mais agilidade ao desenvolvimento, detecção precoce de falhas e maior confiabilidade das aplicações. A análise de vulnerabilidades tira proveito dessas práticas em busca de uma maior automatização dos seus processos.

Para garantir a segurança dentro desse fluxo automatizado, utilizam-se técnicas de teste como o **SAST** (Static Application Security Testing) e o **DAST** (Dynamic Application Security Testing). O **SAST** é uma técnica de análise estática de código que examina a base de código-fonte, bytecode ou binário em busca de vulnerabilidades antes da execução do software. Seus principais benefícios incluem a detecção precoce de falhas e a integração fácil em pipelines CI/CD. No entanto, pode gerar muitos falsos positivos e não identificar problemas que surgem apenas em tempo de execução. Já o **DAST** realiza testes dinâmicos, simulando ataques reais a uma aplicação em execução para identificar falhas de segurança. Sua vantagem é a detecção de vulnerabilidades reais, como XSS (Cross-site scripting, scripts maliciosos em sites web) e injeção de SQL. A desvantagem é a possibilidade de detectar falhas apenas em estágios mais avançados do desenvolvimento. A combinação de CI/CD com testes SAST e DAST fortalece a segurança do software, tornando os sistemas mais confiáveis e resilientes a ataques, além de garantir conformidade com boas práticas e normas regulatórias.

2.4. Ferramentas de apoio

O processo de teste e análise de vulnerabilidades hoje conta com um razoável conjunto de ferramentas de apoio, o que tem facilitado muito o trabalho dos desenvolvedores em busca da concepção de software seguro. Um aspecto importante a considerar na escolha é a cobertura, se para teste estático, dinâmico, de containers, etc. No caso deste trabalho, as principais ferramentas utilizadas estão brevemente descritas a seguir.

SonarQube[2]: é uma plataforma de análise estática de código que auxilia na identificação de vulnerabilidades, bugs e problemas de qualidade em aplicações. Ele suporta diversas linguagens de programação e fornece relatórios detalhados sobre a saúde do código, ajudando equipes de desenvolvimento a manterem padrões de qualidade e segurança ao longo do ciclo de vida do software [Marcilio et al. 2019].

Trivy[3]: é uma ferramenta de código aberto para análise de vulnerabilidades em contêiner, repositórios de código, dependências e infraestrutura como código (IaC). Rápido e eficiente, o Trivy permite identificar falhas de segurança em diferentes estágios do desenvolvimento, garantindo que imagens de contêiner e pacotes sejam monitorados continuamente contra ameaças conhecidas [Saxena 2023].

[2]https://www.sonarsource.com/products/sonarqube/

[3]https://trivy.dev/latest/

3. A Metodologia Proposta

Propõe-se aqui uma metodologia estruturada e sistemática para identificar, avaliar e corrigir vulnerabilidades em sistemas computacionais. Seu objetivo é aumentar a robustez e mitigar potenciais riscos de segurança [van Den Berghe 2015], [Mateo Tudela et al. 2020].

A metodologia pressupõe o trabalho integrado e cooperativo entre o desenvolvedor e o testador do software. Os agentes devem executar um conjunto de etapas que vão do desenvolvimento inicial até a revisão de código ou validação do sistema completo. O objetivo é a identificação de falhas e vulnerabilidades e a correspondente correção e mitigação das falhas e vulnerabilidades encontradas.

Figura 1. Metodologia de testes de vulnerabilidades de segurança

A Figura 1 ilustra as oito etapas da metodologia, cuja explicação e descrição das etapas que a compõem estão a seguir.

1. Planejamento de Análise de Segurança: Esta etapa define o escopo da análise de segurança, identificando os módulos, componentes ou sistemas que serão avaliados. Também estabelece os critérios de segurança com base na diretriz de segurança adotada, considerando vulnerabilidades como controle de acesso, injeção de código, exposição de dados sensíveis e uso de componentes desatualizados. **2. Inspeção Inicial de Segurança:** Nesta etapa, realiza-se uma revisão manual do código-fonte para identificar possíveis padrões inseguros e pontos críticos de segurança. Os módulos que apresentam características suspeitas são anotados para posterior correção, visando reduzir vulnerabilidades conhecidas e melhorar a segurança do software, ainda sem o uso de ferramentas automatizadas. **3. Seleção dos Tipos de Análise para Detecção de Vulnerabilidades:** Com base nas características do projeto, são selecionadas as técnicas de análise mais adequadas para identificar falhas intrínsecas, preservando a integridade do software. As abordagens adotadas incluem análise estática, análise dinâmica e análise de dependências. **4. Preparação do Ambiente de Testes:** Nesta etapa, configura-se um ambiente de testes seguro e controlado que simula o ambiente de produção. Ferramentas e recursos necessários para a execução dos testes são preparados e integrados ao ambiente para possibilitar uma análise eficiente e isolada. Caso o sistema a ser testado esteja em produção, esse isolamento evita eventuais impactos sobre ele. **5. Execução dos Testes:** A execução dos testes planejados ocorre nesta etapa, utilizando ferramentas automatizadas para identificar vulnerabilidades. Os testes incluem a coleta detalhada dos resultados, a documentação de

falhas e evidências, bem como a avaliação do impacto potencial das vulnerabilidades encontradas. Como exemplo de ferramentas a serem utilizadas nesta fase, tem-se o OWASP ZAP e SonarQube. **6. Análise de Resultados:** Os dados coletados durante a etapa de testes são analisados para classificar e priorizar as vulnerabilidades encontradas. Essa classificação é baseada na dificuldade de correção, urgência e impacto. O desenvolvedor deve utilizar esses dados para determinar quais vulnerabilidades devem ser corrigidas e em qual ordem. **7. Correção de Vulnerabilidades:** Nesta etapa, o desenvolvedor executa as correções necessárias de forma a eliminar as vulnerabilidades identificadas. As modificações realizadas são revisadas para garantir que novos problemas não sejam introduzidos e, se necessário, os testes são reexecutados para validar as correções antes da geração do relatório final. **8. Geração de Relatório Final:** A última etapa consiste na documentação completa dos resultados da análise de segurança, incluindo descrição das vulnerabilidades, nível de criticidade, impacto potencial e recomendações de correção.

As etapas mencionadas acima constituem o ciclo de vida da análise de vulnerabilidades do sistema, começando das atividades manuais iniciais com inspeções de código e de repositórios, e terminando com a geração de um relatório final. A principal ideia da automação da análise é garantir robustez, resiliência e confiabilidade no sistema por meio de uma checagem completa dos possíveis pontos fracos do sistema sob análise. A seguir, a título de ilustração, apresentamos dois casos de aplicação da metodologia.

4. Estudos de caso

Para validar na prática a metodologia proposta foram selecionadas duas aplicações de uso interno de um órgão público do estado de Minas Gerais para serem analisadas do ponto de vista da segurança durante o processo de integração contínua de software. Por se tratar de um trabalho contínuo e colaborativo entre as equipes de segurança e desenvolvimento, e considerando que este estudo se concentra exclusivamente nos aspectos de segurança, as etapas da metodologia que envolvem a refatoração de código não foram realizadas e, portanto, não serão abordadas no texto que se segue.

4.1. O ambiente computacional alvo

A título de exemplo, vamos considerar um ambiente computacional com as características descritas brevemente a seguir. Em alto nível, a infraestrutura computacional considerada tem como principais atores: uma rede de comunicação de dados que possibilita o acesso e o fluxo de dados no sistema; um sistema de *big data* que hospeda dados, ferramentas de software aplicações finalísticas; usuários que fazem acesso às aplicações. O acesso externo aos usuários é feito por meio de mecanismos tais como redes virtuais VPN point-to-site, VPN site-to-site e rede MPLS. O acesso público a aplicações de Internet é feito via mecanismos de proteção de acesso web (proxy reverso). O sistema de *big data* e as aplicações estão instaladas em servidores do *data center*, que utilizam a tecnologia da Nutanix e operam como uma nuvem privada.

A arquitetura de dados e serviços, isolada logicamente dos demais sistemas institucionais, configura um *data center* estruturado em seis camadas. Cada camada representa uma abstração de alto nível de etapas com ferramentas específicas, conforme ilustrado na Figura 2. São elas: (i) Fontes de dados: coleta inicial; (ii) Zona intermediária: primeiro armazenamento, acessível por ferramentas de *big data*; (iii) Data lake: repositório central

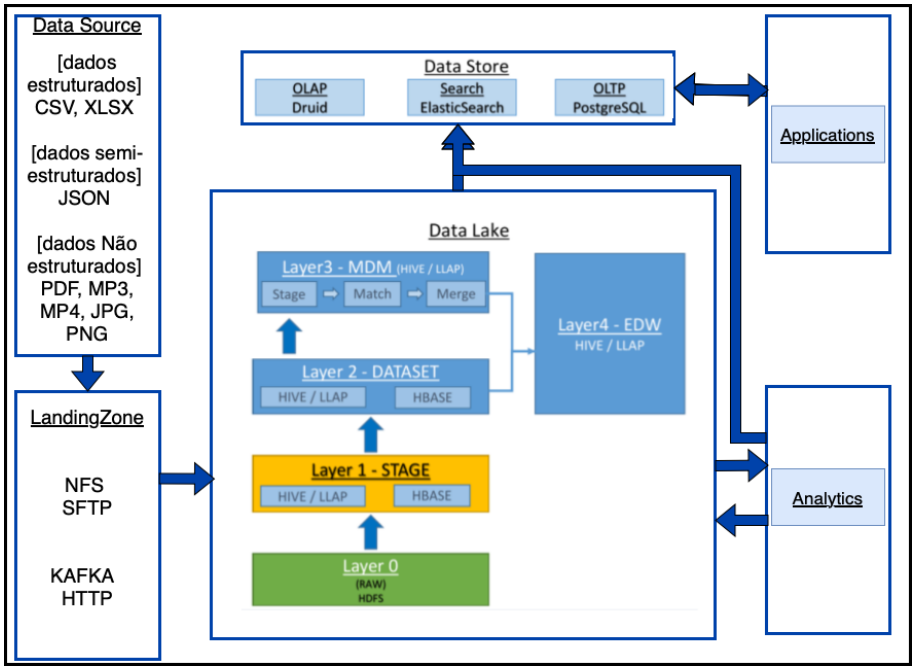

Figura 2. Arquitetura de dados/serviços do sistema exemplo.

de processamento; (iv) Data store: repositório final para uso por aplicações como Lins e Áduna; (v) Analytics: análise e exploração; e (vi) Aplicações: visualização dos dados processados.

As atividades da arquitetura tecnológica do ambiente dão-se a partir de uma nuvem privada onde os dados são coletados, armazenados e processados. A nuvem privada é constituída por um cluster da marca Nutanix, que oferece uma plataforma de software proprietária com recursos de virtualização de servidores.

O ambiente conta com uma diversidade de ferramentas e aplicações. Como exemplo de aplicações finalísticas, tem-se uma aplicação web de recuperação de dados de pessoas físicas e pessoas jurídicas e uma aplicação web para consulta a dados referentes a prestações de contas de municípios e gestão pública. Há ferramentas para *analytics*, para gerenciamento de recursos, armazenamento de grandes volumes de dados, entre outras.

4.2. Estudo de caso 1 - Aplicação web simples

A primeira aplicação finalística tem finalidade de exploração de dados governamentais de cidadãos brasileiros, utilizada apenas por usuários com autorização específica, como policiais militares e civis, agentes do ministério público e de demais órgãos governamentais que possuem de alguma forma acesso garantido.

A aplicação é composta por *front-end*, *back-end* e módulos de comunicação com o armazenamento de dados, todos hospedados em containers executados de forma isolada e orquestrados via Jenkins[4]. O sistema, baseado em plataforma web e majoritariamente escrito em TypeScript, configura-se como um sistema distribuído simples. Aqui, aplica-se apenas a análise estática (SAST), uma vez que a aplicação lida com dados sensíveis

[4]https://www.jenkins.io/

em ambiente restrito, e a análise no código-fonte evita riscos operacionais e exposição de informações durante os testes.

Etapa 1: - O planejamento de análise de segurança envolveu a verificação dos módulos back-end e front-end, utilizando o framework OWASP. A configuração do SonarQube foi realizada por meio de um arquivo de parâmetros que especifica os arquivos analisados, a linguagem e as exclusões necessárias.

Etapa 2: Foi realizada uma inspeção manual do código-fonte com foco em credenciais, variáveis de ambiente e configurações sensíveis, seguindo boas práticas do OWASP [Foundation 2021]. Foram verificados arquivos de configuração e trechos de código em busca de dados expostos. Nenhuma falha crítica foi identificada.

Etapa 3: A seleção de tipos de análise escolheu exclusivamente a análise estática utilizando o SonarQube, a que se mostrou adequada para identificar falhas estruturais e vulnerabilidades conhecidas no código-fonte, garantindo uma avaliação abrangente da segurança e qualidade do software.

Etapa 4: A preparação do ambiente de testes configurou o ambiente na plataforma Jenkins, cujos detalhes podem ser encontrados no documento referenciado[5], pela qual é possível criar pipelines exclusivos para cada módulo da aplicação e criar um ambiente idêntico ao ambiente oficial de produção. Ademais, com o isolamento necessário para garantir a não interrupção da montagem oficial do software.

Etapa 5: Os testes foram executados em um ambiente isolado; a configuração e execução do pipeline dos módulos componentes da aplicação foi feita por scripts compatíveis com o ambiente do Jenkins, quando foram definidas as configurações para a execução dos testes, a localização dos resultados e o estágio específico no qual os testes foram realizados. Os resultados estão mostrados na Tabela 1

As etapas **6, 7 e 8** são conduzidas não apenas pela equipe de segurança, mas também pelos desenvolvedores, que desempenham um papel fundamental na análise detalhada dos pontos levantados. Considerando, no caso, que não foram identificadas vulnerabilidades de alto risco no software, como pode ser observado na referida tabela de resultados, é necessária uma avaliação da viabilidade de refatoração.

Tabela 1. Resultados da análise de segurança

Métrica	**Front-end**	**Back-end**
Linhas de Código (LOC)	31.000	7.300
Cobertura de Testes	0%	0%
issues Detectadas	290	95
Vulnerabilidades	0	0
Security Hotstpots	23	3
Bugs	32	3
*code smell*s	235	89
Débito Técnico Estimado	4 dias e 2 horas	1 dia e 6 horas

Os resultados da tabela 1 indicam oportunidades significativas de melhoria na qua-

[5]https://docs.google.com/document/d/1YnlG0Bdx7Lokz0kUpp3MTNoEgrqZV2mJ6MQr0SNBP88/edit?usp=drivesdk

lidade do código, especialmente devido à ausência total de cobertura de testes de software padrão e ao alto número de *code smell*s e *issues* detectadas, principalmente no *front-end*. A ocorrência de *security hotspots* exige revisão para mitigar possíveis riscos de segurança, enquanto o débito técnico estimado sugere que correções demandarão um tempo considerável. Implementar testes automatizados, refatorar o código e priorizar correções críticas ajudará a reduzir riscos e melhorar a manutenção do sistema.

4.3. Estudo de caso 2 - Plataforma multi-módulos

O segundo estudo de caso consiste em uma plataforma visual chamada Lemonade[6], baseada em computação distribuída, que visa facilitar a implementação, experimentação, teste e implantação de aplicações de processamento de dados e aprendizado de máquina. A plataforma permite que usuários sem proficiência em linguagens de programação executem fluxos de trabalho de processamento paralelo e distribuído. Por meio de uma interface gráfica baseada em web, os usuários podem criar fluxos de trabalho que envolvem operações de extração, transformação e carregamento de dados.

O Lemonade é composto por diversos microsserviços implantados como containers gerenciados pela tecnologia Docker, que juntos oferecem funcionalidades para manipulação e análise de grandes volumes de dados. Os microsserviços são:

- **Citrus**:Fornece uma interface web, permitindo que os usuários interajam com a plataforma de maneira amigável e visualizem os resultados das análises.
- **Tahiti**: Responsável pelas operações com algoritmos de metadados.
- **Juicer**: Traduz os fluxos de trabalho definidos no Tahiti em código executável, como PySpark.
- **Limonero**: Armazena metadados sobre as fontes de dados disponíveis. Assim, facilita o acesso e a gestão dos dados utilizados nos fluxos de trabalho.
- **Caipirinha**: Prove uma interface de visualização, permitindo que os usuários interajam com os dados de forma gráfica.
- **Stand**: Executa o monitoramento de APIs do sistema.
- **Seed**: É responsável pelo gerenciamento de experimentos e pela persistência de seus resultados. Ele armazena configurações, dados e métricas geradas durante a execução de experimentos científicos.
- **Thorn**: É o módulo que implementa mecanismos de segurança e auditoria.

A aplicação da metodologia de análise de segurança do sistema Lemonade está brevemente descrita a seguir.

Etapa 1: Desenvolvimento da Etapa 1 Esta fase envolve a definição do escopo da análise, onde os módulos principais do sistema foram identificados. Arquivos como *pycache*, **.pyc*, */env*, *requirements.txt* foram excluídos da análise estática, enquanto *requirements.txt*, *Dockerfile* e *docker-compose.yml* foram incluídos para análise do contêiner.

Etapa 2: Inspeção Inicial de Segurança Seguindo as regras de boas práticas do OWASP, a inspeção manual do código revelou problemas no módulo Thorn, como a injeção de LDAP e o uso de conexões não criptografadas, bem como a falta de um sistema robusto de logging em todos os módulos. A análise de dependências no repositório Juicer identificou bibliotecas desatualizadas, como *autopep8*, *requests* e *numpy*.

[6]https://www.lemonade.org.br

Tabela 2. Métricas de Análise de Segurança Estática - Lemonade

Métrica	Caipirinha	Citrus	Juicer	Limonero	Seed	Stand	Tahiti
Linhas de Código	1.711	879	68.999	6.548	2.611	4.106	22.148
Cobertura de Testes	0%	0%	0%	0%	0%	0%	0%
issues Detectadas	39	209	1.694	114	54	98	582
Vulnerabilidades	0	0	0	0	0	0	0
Bugs	0	88	60	3	4	0	5
Security Hotspots	6	11	94	12	8	22	8
*code smell*s	33	110	1.540	99	42	76	569
Débito Técnico	3h 23min	1 dia	31 dias	2d 2h	4h	1d 2h	24 dias

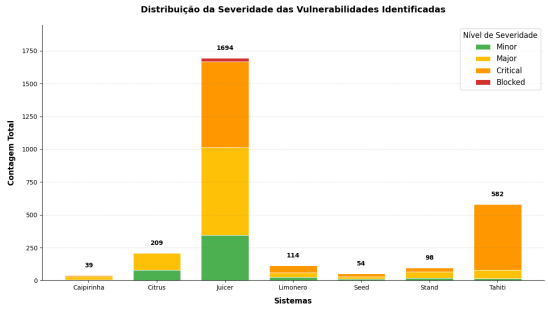

Figura 3. Severidades dos achados das análises do SonarQube - Lemonade

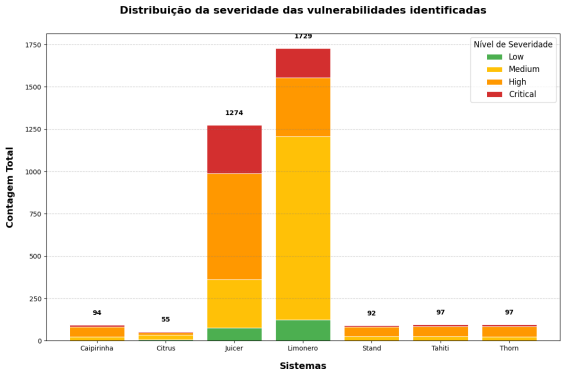

Figura 4. Severidades dos achados das análises do Trivy - Lemonade

Etapa 3: Seleção de Tipos de Análise Foi definido que as análises a serem realizadas seriam estática, dinâmica, de dependências e de contêiner. A análise estática será feita com o SonarQube. O Trivy é usado para análise de container.

Etapa 4: Preparação do Ambiente de Testes O ambiente de testes foi configurado de forma controlada e isolada, utilizando o Jenkins e o plugin SonarQube Scanner for Jenkins para integrar com o GitLab. O isolamento foi realizado para evitar interferência com o ambiente de produção. Para cada um dos microsserviços foram criados pipelines de montagem de código para execução automática, tanto do SonarQube quanto do Trivy.

Etapa 5: Execução dos Testes A execução dos testes consiste na aplicação dos cenários planejados para detectar falhas de segurança no sistema. Os resultados são mostrados na Tabela 2. Por conta das falhas identificadas na inspeção manual, bem como pelas análises automatizadas, como parte do processo é necessário comunicar a equipe de desenvolvimento a respeito dos potenciais riscos das falhas identificadas.

Assim como no caso de uso 1, ao observar a tabela 2 é notável a ausência de testes automatizados. O número elevado de *issues*, *code smell*s e security hotspots, especialmente no Juicer e Tahiti, indica a necessidade de revisão e refatoração para melhorar a qualidade e segurança do software. Mesmo sem a existência comprovada de vulnerabilidades, a presença de bugs e problemas estruturais reforça a importância de estratégias de mitigação de riscos e aprimoramento da qualidade do código.

A partir da análise de containers e arquivos de configuração, exposto na tabela 3,

Tabela 3. Análise de contêiner e arquivo de configuração - Lemonade

Vulnera-bilidades	Caipirinha	Citrus	Juicer	Limonero	Seed	Stand	Tahiti	Thorn
Pacotes do sistema	77	38	71	1061	-	77	77	77
Bibliotecas e linguagens	15	15	1200	666	15	13	17	18
Arquivos de configuração	2	2	3	2	2	2	3	2

é possível observar que os componentes **Juicer** e **Limonero** possuem um número significativamente maior de bibliotecas e pacotes do sistema em comparação com os demais, o que pode aumentar a superfície de ataque e a complexidade de manutenção. O indicado nesse caso consiste na análise sistemática do impacto das atualizações das bibliotecas apontadas, uma vez que não se trata de bibliotecas de uso direto do código e sim módulos que são utilizados em algum momento da montagem da imagem base. Os gráficos da Figura 3 e Figura 4 mostram severidades dos achados para as duas ferramentas, a título de exemplo apenas.

5. Conclusão

Este trabalho apresentou uma metodologia de análise de vulnerabilidade de códigos de ambientes computacionais governamentais com o objetivo de identificar problemas e aumentar a segurança de ambientes complexos, compostos por diversas aplicações. Essas aplicações, por sua vez, podem ser baseadas em arquiteturas de microsserviços, compostas de diversos componentes que interagem entre si.

As análises envolveram ferramentas de código aberto que geram relatórios a partir de análises estáticas e dinâmicas do código e da configuração do ambiente de virtualização. Essas análises detectaram diversos pontos com algum tipo de problema que não foram identificados anteriormente, quando do desenvolvimento. Os resultados, mesmo que parciais, pois até então não foram feitas análises dinâmicas, mostram a importância do uso sistemático de uma metodologia e correspondentes ferramentas automatizadas no processo de melhoria contínua do código de sistemas governamentais. Este trabalho está em andamento, agora na fase de análise dinâmica de código; em um futuro breve teremos novos resultados de testes. Pelo nosso aprendizado, podemos concluir que uma metodologia de teste de segurança deve ser um componente necessário e constante no apoio ao desenvolvimento de software com preocupação de segurança.

Agradecimento: Os autores agradecem às seguintes instituições brasileiras pelo apoio: MPMG - Ministério Público de Minas Gerais, CAPES, CNPq e FAPEMIG.

Referências

Alhazmi, O., Malaiya, Y., and Ray, I. (2005). Security vulnerabilities in software systems: A quantitative perspective. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 281–294. Springer.

Aljabri, M., Aldossary, M., Al-Homeed, N., Alhetelah, B., Althubiany, M., Alotaibi, O., and Alsaqer, S. (2022). Testing and exploiting tools to improve owasp top ten security

vulnerabilities detection. In *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 797–803. IEEE.

Chatfield, A. T. and Reddick, C. G. (2017). Cybersecurity innovation in government: A case study of u.s. pentagon's vulnerability reward program. In *Proceedings of the 18th Annual International Conference on Digital Government Research*, dg.o '17, page 64–73, New York, NY, USA. Association for Computing Machinery.

Foundation, O. (2021). OWASP Top Ten – 2021: The Ten Most Critical Web Application Security Risks. Accessed: 2025-03-22.

Idris, M., Syarif, I., and Winarno, I. (2021). Development of vulnerable web application based on owasp api security risks. In *2021 International Electronics Symposium (IES)*, pages 190–194. IEEE.

Krsul, I. V. (1998). *Software vulnerability analysis*. Purdue University.

Li, J. (2020). Vulnerabilities mapping based on owasp-sans: a survey for static application security testing (sast). *arXiv preprint arXiv:2004.03216*.

Liu, B., Shi, L., Cai, Z., and Li, M. (2012). Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219. IEEE.

Mateo Tudela, F., Bermejo Higuera, J.-R., Bermejo Higuera, J., Sicilia Montalvo, J.-A., and Argyros, M. I. (2020). On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Applied Sciences*, 10(24):9119.

Ramadlan, M. F. (2019). Introduction and implementation owasp risk rating management. *Open Web Application Security Project*.

Rose, S., Borchert, O., Mitchell, S., and Connell, S. (2020). Zero Trust Architecture, Draft (2nd) NIST Special Publication 800-207.

Santos, L. C. M., Prado, E. P. V., and Chaim, M. L. (2020). Técnicas e ferramentas para detecção de vulnerabilidades em ambientes de desenvolvimento ágil de software. *Brazilian Journal of Development*, 6(6):33921–33941.

Saxena, P. (2023). *Container image security with trivy and istio inter-service secure communication in kubernetes*. PhD thesis, Dublin, National College of Ireland.

Setiawan, H., Erlangga, L. E., and Baskoro, I. (2020). Vulnerability analysis using the interactive application security testing (iast) approach for government x website applications. In *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*, pages 471–475.

van Den Berghe, A. (2015). Towards a practical security analysis methodology. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 883–886. IEEE.