UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL MARECO BATISTA DE SOUTO

UMA ABORDAGEM PARA DETECÇÃO E CORREÇÃO DE VULNERABILIDADES EM ÁRVORES DE DEPENDÊNCIAS DE SOFTWARE

CAMPINA GRANDE - PB

2024

GABRIEL MARECO BATISTA DE SOUTO

UMA ABORDAGEM PARA DETECÇÃO E CORREÇÃO DE VULNERABILIDADES EM ÁRVORES DE DEPENDÊNCIAS DE SOFTWARE

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Cláudio de Souza Baptista

CAMPINA GRANDE - PB

2024

GABRIEL MARECO BATISTA DE SOUTO

UMA ABORDAGEM PARA DETECÇÃO E CORREÇÃO DE VULNERABILIDADES EM ÁRVORES DE DEPENDÊNCIAS DE SOFTWARE

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Cláudio de Souza Baptista

Orientador – UASC/CEEI/UFCG

Professora Dra. Patrícia Duarte de Lima Machado

Examinador – UASC/CEEI/UFCG

Professor Dr. Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 15 de maio de 2024.

CAMPINA GRANDE - PB

RESUMO

Vulnerabilidades em dependências de software, transitivas e indiretas, são uma realidade frequente devido ao uso intensivo de bibliotecas e frameworks. Esse cenário aumenta os riscos de falhas de segurança e compromete a integridade dos sistemas. Diante dessa problemática, neste artigo propõe-se o Safer, uma ferramenta automatizada projetada para detectar e corrigir vulnerabilidades nas árvores de dependências de software. O Safer não só identifica versões seguras das dependências, mas também verifica sua compatibilidade através de testes exploratórios. A metodologia adotada envolve uma análise comparativa de ferramentas existentes e a aplicação do Open Source Insights para diagnóstico de vulnerabilidades, complementada pelo uso do Randoop para os testes de compatibilidade. Os resultados obtidos com o Safer demonstram sua eficácia em reduzir significativamente as vulnerabilidades em todos os níveis de severidade, com uma redução geral aproximada de 90,46%. Destaca-se a capacidade da ferramenta de mitigar ameaças e indicar a viabilidade de sua expansão para outras linguagens e gerenciadores de dependências, fortalecendo assim a segurança e a confiabilidade dos sistemas de software.

AN APPROACH TO DETECT AND CORRECT VULNERABILITIES IN SOFTWARE DEPENDENCY TREES

ABSTRACT

Vulnerabilities in software dependencies, both transitive and indirect, are a common reality due to the extensive use of libraries and frameworks. This scenario increases the risks of security breaches and compromises the integrity of systems. Faced with this issue, this article proposes Safer, an automated tool designed to detect and address vulnerabilities in software dependency trees. Safer not only identifies secure versions of dependencies but also verifies their compatibility through exploratory testing. The methodology adopted involves a comparative analysis of existing tools and the application of Open Source Insights for vulnerability diagnosis, complemented by the use of Randoop for compatibility testing. The results obtained with Safer demonstrate its effectiveness in significantly reducing vulnerabilities at all severity levels, with an overall reduction of approximately 90.46%. The tool's ability to mitigate threats and indicate the feasibility of its expansion to other languages and dependency managers is noteworthy, thereby strengthening the security and reliability of software systems.

Uma Abordagem para Detecção e Correção de Vulnerabilidades em Árvores de Dependências de Software

Gabriel Mareco Batista de Souto
Universidade Federal de Campina Grande
Campina Grande, Brasil
gabriel.souto@ccc.ufcg.edu.br

Cláudio de Souza Baptista
Universidade Federal de Campina Grande
Campina Grande, Brasil
baptista@computacao.ufcg.edu.br

RESUMO

Vulnerabilidades em dependências de software, transitivas e indiretas, são uma realidade frequente devido ao uso intensivo de bibliotecas e frameworks. Esse cenário aumenta os riscos de falhas de segurança e compromete a integridade dos sistemas. Diante dessa problemática, neste artigo propõe-se o Safer, uma ferramenta automatizada projetada para detectar e corrigir vulnerabilidades nas árvores de dependências de software. O Safer não só identifica versões seguras das dependências, mas também verifica sua compatibilidade através de testes exploratórios. A metodologia adotada envolve uma análise comparativa de ferramentas existentes e a aplicação do *Open Source Insights* para diagnóstico de vulnerabilidades, complementada pelo uso do *Randoop* para os testes de compatibilidade. Os resultados obtidos com o Safer demonstram sua eficácia em reduzir significativamente as vulnerabilidades em todos os níveis de severidade, com uma redução geral aproximada de 90,46%. Destaca-se a capacidade da ferramenta de mitigar ameaças e indicar a viabilidade de sua expansão para outras linguagens e gerenciadores de dependências, fortalecendo assim a segurança e a confiabilidade dos sistemas de software.

PALAVRAS-CHAVE

Dependências de software, vulnerabilidade, testes de regressão, automação, compatibilidade.

REPOSITÓRIO

https://gitlab.com/lsi-ufcg/vulnerabilidades/safer

1 INTRODUÇÃO

A aceleração do desenvolvimento de software é uma necessidade impulsionada pelo mercado global e a procura constante por inovação e atualização [3]. Uma estratégia amplamente adotada para acelerar esse processo é o uso de bibliotecas e frameworks de prateleira, permitindo que os desenvolvedores não reinventem a roda, mas aproveitem soluções pré-existentes e de eficácia comprovada. Essa prática, entretanto, apesar de seus benefícios inegáveis, traz consigo riscos de segurança e desafios de manutenção [23]. Ao adotar uma biblioteca ou framework em um projeto, é essencial estar ciente de que isso pode incluir a adoção de vulnerabilidades preexistentes e a exposição a riscos relacionados ao versionamento das dependências. O versionamento semântico é um sistema que tenta mitigar esses riscos ao estabelecer regras claras para a mudança de versões, baseando-se em três números principais: *major*, *minor* e *patch*. Cada atualização numérica reflete o tipo e a gravidade das alterações feitas:

- ***Major***: introduz alterações que podem ser incompatíveis com versões anteriores, conhecidas como '*breaking changes*'.
- ***Minor***: adiciona funcionalidades de maneira compatível com versões anteriores.
- ***Patch***: corrige bugs sem afetar funcionalidades.

Entretanto, quando mantenedores de bibliotecas não aderem rigorosamente ao versionamento semântico, podem surgir mudanças significativas e inesperadas que resultam em incompatibilidades. Essas alterações, também conhecidas como '*breaking changes*', podem comprometer a estabilidade do sistema, introduzindo bugs ou quebrando a compatibilidade com versões anteriores. Isso força os desenvolvedores a gastar tempo considerável ajustando ou substituindo dependências críticas para manter a funcionalidade do projeto [11]. Estes componentes, quando integrados, criam uma árvore de dependências, que se não gerida de forma eficaz, poderão ser uma porta de entrada para ataques maliciosos e originar custos ocultos de manutenção e atualização [8].

O cenário torna-se ainda mais complexo quando consideramos a natureza interconectada dessas dependências. A vulnerabilidade em uma única biblioteca pode se propagar e comprometer outras partes do sistema [8]. Imagine um edifício, onde cada tijolo representa uma dependência. Se um tijolo estiver comprometido, não apenas aquele local, mas todo o edifício pode estar em risco. No mundo do software, um exemplo prático seria um projeto que utiliza uma biblioteca para autenticação, que por sua vez, depende de outra para criptografia. Se a biblioteca de criptografia possuir uma vulnerabilidade, a autenticação, e consequentemente, todo o sistema, pode ser comprometido do ponto de vista de segurança.

Para ilustrar a magnitude dessa problemática em um caso real, podemos tomar como exemplo o incidente relacionado ao *CVE-2021-44228*[1], associado à biblioteca *Log4j*[2]. Uma vulnerabilidade crítica foi detectada neste popular componente de *logging*, amplamente utilizado em inúmeros projetos em todo o mundo, que permitia que atacantes tivessem controle total por meio de execução de scripts maliciosos. Considerando o impacto potencial dessa vulnerabilidade, organizações como a Amazon e a Cisco, empreenderam esforços urgentes no sentido de minimizar os impactos [13], evidenciando os riscos associados às dependências. Esse incidente ressalta uma realidade alarmante: a vulnerabilidade em uma única dependência pode desencadear um efeito dominó, comprometendo sistemas inteiros e, consequentemente, dados sensíveis e infraestruturas críticas.

[1] https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[2] https://logging.apache.org/log4j/2.x/

O relatório *State of DevSecOps 2024* da Datadog destaca a importância da gestão de árvores de dependências para mitigar vulnerabilidades em bibliotecas de terceiros. Revela que 90% dos serviços *Java*[3] são vulneráveis a falhas críticas ou de alta gravidade devido às dependências indiretas, ou seja, bibliotecas não intencionalmente incluídas pelos desenvolvedores [4]. Isso sublinha a necessidade de ferramentas como a Safer, aqui proposta, que automatiza a detecção e correção de vulnerabilidades nas dependências.

Além de eventos pontuais, organizações como a OWASP[4] (Open Web Application Security Project) têm desempenhado um papel crucial na conscientização e educação sobre riscos de segurança em aplicações web. Seu documento, o "*OWASP Top 10*", é uma lista anualmente atualizada que destaca as dez principais vulnerabilidades de aplicações web, servindo como um guia para desenvolvedores e organizações sobre onde focar seus esforços de mitigação de tais vulnerabilidades [15]. A existência e a necessidade de tais diretrizes sublinham a complexidade e a relevância do tema.

A gestão segura de dependências torna-se ainda mais complexa devido à dificuldade intrínseca na detecção de vulnerabilidades em software. Como destacado na pesquisa de Luca Allodi et al. [2], a avaliação das vulnerabilidades de software é um processo chave no desenvolvimento de software e na gestão de segurança, que exige a consideração de múltiplos fatores técnicos e humanos. Assim, mesmo que haja consciência dos riscos associados, a identificação e a correção eficazes permanecem como desafios significativos.

A questão emergente, portanto, não é apenas identificar a existência dessas vulnerabilidades, mas também tomar ações corretivas eficazes. Ferramentas tradicionais de segurança, como o *OWASP Dependency-Check*[5], muitas vezes limitam-se a diagnosticar o problema, alertando sobre dependências vulneráveis, mas não fornecendo soluções automatizadas para a correção. Ademais, em um ambiente no qual os projetos podem ter dezenas ou até centenas de dependências, a tarefa de atualizar manualmente cada componente vulnerável é crítica, onerosa e propensa a erros [16].

Portanto, este trabalho enquadra-se no âmbito da identificação e correção de vulnerabilidades em dependências, ao mesmo tempo que verifica a compatibilidade das alterações por meio de testes de regressão. Busca-se compreender como as ferramentas podem desempenhar um papel fundamental na identificação de falhas em árvores de dependências. Em seguida, propomos uma solução automatizada que se concentra no contexto do *backend* de aplicações, com atenção especial para projetos que utilizam o gerenciador de dependências *Maven*[6] e *Java*. Foi realizado um estudo de caso em projetos com uma empresa parceira, visando identificar vulnerabilidades em dependências e, ao contrário das ferramentas tradicionais que apenas identificam a dependência vulnerável, esta abordagem também toma a ação corretiva. A solução proposta identifica a versão mais segura da dependência, categorizando-as conforme a pontuação do *Common Vulnerability Scoring System (CVSS)*[7], e a instala em substituição à versão vulnerável. Tal ação, garante assim, através do *build* e testes de regressão subsequentes, a integridade e segurança do sistema após a intervenção. O objetivo é diminuir o número de dependências vulneráveis, contribuindo assim para a segurança e estabilidade dos sistemas.

Este trabalho apresenta contribuições na área de segurança de software, especificamente na detecção e correção de dependências com vulnerabilidades. As principais contribuições são:

- o desenvolvimento de uma ferramenta automatizada, Safer, que integra detecção de vulnerabilidades e atualização de dependências, visando a segurança e a compatibilidade em projetos de software.
- a implementação de testes exploratórios que complementam os testes unitários, para assegurar que as atualizações de dependências não introduzam novos problemas no software.
- um estudo de caso com aplicação prática da ferramenta em um ambiente empresarial real, demonstrando sua eficácia e aplicabilidade.

A estrutura deste documento é organizada da seguinte forma: A Seção 2 descreve a metodologia empregada no desenvolvimento e implementação da ferramenta Safer. Na Seção 3, é realizada uma análise de algumas ferramentas disponíveis no mercado. Os detalhes sobre o Safer são apresentados na Seção 4. As Seções 5 e 6 apresentam os resultados e concluem o documento com um resumo das principais contribuições, além de discutir os trabalhos futuros planejados para a expansão e aprimoramento da ferramenta Safer.

2 METODOLOGIA

Este trabalho foi delineado para explorar e avaliar ferramentas de verificação de dependências de software, visando identificar uma solução eficaz para a gestão automatizada de atualizações de dependências. Inicialmente, conduzimos uma análise comparativa de mercado focada em duas ferramentas amplamente reconhecidas: o *Dependency-Check (DC)* e o *Snyk*[8]. Essa análise envolveu a avaliação de características como capacidade de detecção de vulnerabilidades, usabilidade, necessidade de intervenções manuais e confiabilidade nas atualizações de dependências.

Após identificar limitações significativas no uso do DC, principalmente relacionadas à exigência de intervenções manuais e à baixa confiabilidade nas atualizações das dependências, a qual era necessária uma atualização manual, propõe-se o desenvolvimento de uma nova ferramenta, denominada Safer. O Safer foi projetado para superar essas lacunas, implementando funcionalidades de atualizações automáticas e a execução de testes exploratórios para aumentar a confiabilidade das atualizações.

Para validar o Safer, foi realizado um estudo de caso em colaboração com uma empresa parceira, que implementa um sistema composto por diversos módulos. Essa empresa é uma das principais fornecedoras de serviços de conformidade fiscal e tributária no país, atendendo uma ampla gama de conglomerados econômicos. Isso reflete a grande escala do sistema utilizado pela organização, onde aplicamos a ferramenta em vários módulos de suas soluções de software. A metodologia empregada para validação do Safer consistiu em três fases principais:

(1) **Análise Prévia dos Módulos:** Executamos o *Dependency-Check* nos módulos selecionados para estabelecer uma linha

[3] https://www.java.com/en/
[4] https://owasp.org/
[5] https://owasp.org/www-project-dependency-check/
[6] https://maven.apache.org/
[7] https://www.first.org/cvss/v3.1/specification-document
[8] https://snyk.io/

de base do estado das dependências e das vulnerabilidades existentes.

(2) **Execução do Safer:** Aplicamos o Safer para atualizar automaticamente as dependências e executar testes exploratórios, com o objetivo de verificar a integridade e funcionalidade do software após as atualizações.

(3) **Análise Posterior dos Módulos:** Realizamos uma nova análise utilizando o *Dependency-Check* para avaliar a eficácia do Safer em reduzir vulnerabilidades e em gerenciar as dependências de forma mais eficiente e confiável.

Os resultados dessas fases foram documentados, permitindo uma análise comparativa entre o estado inicial e final das dependências e das vulnerabilidades, além de fornecer *insights* sobre a performance e a confiabilidade do Safer como uma solução no gerenciamento de dependências de software.

3 ANÁLISE DE FERRAMENTAS DE MERCADO: SONAR DEPENDENCY-CHECK VS SNYK

Nesta seção, descreve-se a abordagem e os resultados de nossa análise comparativa entre duas ferramentas líderes de segurança em desenvolvimento de software: *Sonar Dependency-Check (DC)* e *Snyk*. A análise teve como objetivo identificar as funcionalidades, vantagens e limitações de cada ferramenta, fornecendo assim bases para a seleção da ferramenta mais adequada para a integração no ambiente de desenvolvimento da empresa parceira.

3.1 Sonar Dependency-Check

O *Sonar Dependency-Check* é uma extensão do *SonarQube*[9], projetada para identificar vulnerabilidades em dependências de bibliotecas e componentes de terceiros utilizados nos projetos de software. A ferramenta realiza comparações das versões das bibliotecas usadas com um banco de dados de vulnerabilidades conhecidas, alertando sobre potenciais riscos de segurança.

3.1.1 *Funcionalidades Principais.*

- **Análise em Tempo Real:** Permite a integração com sistemas de Integração Contínua/Entrega Contínua (CI/CD) para análises automáticas.
- **Relatórios Detalhados:** Oferece visões detalhadas das vulnerabilidades detectadas, facilitando o processo de mitigação.

3.1.2 *Vantagens.*

- **Integração Facilitada:** Devido à sua integração com o *SonarQube*, apresenta facilidade de adoção em ambientes já utilizando esta plataforma.
- **Amplo Suporte a Linguagens:** Compatível com múltiplas linguagens de programação, o que o torna versátil para diversos projetos.

3.1.3 *Limitações.*

- **Falsos Positivos:** Como muitas ferramentas de análise estática, pode gerar alertas de falsos positivos que requerem revisão manual.
- **Usabilidade:** Sua interface pode ser menos intuitiva, especialmente para novos usuários.

3.2 Snyk

O *Snyk* é uma plataforma avançada de segurança de código e gestão de vulnerabilidades que facilita a identificação e a correção de vulnerabilidades em software. É especialmente projetada para ser integrada ao ciclo de desenvolvimento, oferecendo análises contínuas e recomendações de correção.

3.2.1 *Funcionalidades Principais.*

- **Monitoramento Contínuo:** Acompanha os repositórios de código para identificar novas vulnerabilidades assim que são introduzidas.
- **Integração com Repositórios de Código:** Suporte para integração com *GitHub*, *Bitbucket*, e *GitLab*.

3.2.2 *Vantagens.*

- **Interface Amigável:** Possui uma interface clara e fácil de usar, o que ajuda na rápida adoção por parte das equipes de desenvolvimento.
- **Recomendações Práticas:** Fornece sugestões detalhadas para a resolução de problemas identificados.

3.2.3 *Limitações.*

- **Acesso Limitado em Versões Gratuitas:** Recursos mais avançados são limitados à versão paga.
- **Falsos Positivos:** Como muitas ferramentas de análise estática, pode gerar alertas de falsos positivos que requerem revisão manual.
- **Configuração Inicial:** Requer uma configuração inicial mais complexa comparada ao DC na empresa parceira.

3.3 Comparação e Escolha

Durante a execução dos testes com as ferramentas *Snyk* e *Dependency-Check* em um ambiente controlado na empresa parceira, ambas as aplicações detectaram uma série de vulnerabilidades e *CVEs (Common Vulnerabilities and Exposures)* nas dependências utilizadas nos projetos de software analisados. Os resultados obtidos foram significativos e indicativos das capacidades e limitações de cada ferramenta no reconhecimento de potenciais riscos de segurança.

A aplicação *Snyk* identificou um total de 19 problemas associados às dependências do software, revelando 20 *CVEs* relacionados. Esta detecção evidencia a eficiência do *Snyk* em identificar vulnerabilidades críticas, ainda que o número total de problemas detectados tenha sido menor em comparação com o *Dependency-Check*.

Por outro lado, o *Dependency-Check* apresentou resultados mais abrangentes, identificando 29 questões relacionadas às dependências e um total de 43 *CVEs*. A maior quantidade de vulnerabilidades detectadas por esta ferramenta pode ser atribuída à sua extensa base de dados e capacidade de análise aprofundada, embora isso também possa resultar em um aumento de falsos positivos, como observado nas limitações discutidas anteriormente.

Esses resultados reforçam as características distintas das ferramentas analisadas. Enquanto o *Snyk* mostrou ser uma ferramenta robusta com uma interface amigável e eficaz na identificação de problemas, o *Dependency-Check* destacou-se por sua capacidade de

[9] https://docs.sonarsource.com/sonarqube/latest/

realizar uma varredura mais extensa, embora possa necessitar de uma filtragem adicional para distinguir efetivamente as verdadeiras vulnerabilidades dos falsos positivos.

Portanto, a comparação detalhada revelou que, enquanto o *Snyk* oferece uma abordagem eficiente e uma interface mais amigável, o *Sonar Dependency-Check* destaca-se pela sua integração com o *SonarQube* e pela amplitude de sua varredura, o que permite uma análise mais abrangente, apesar de um maior volume de falsos positivos. Levando em consideração o ambiente e as necessidades da empresa parceira, optou-se pela implementação do *Sonar Dependency-Check*. A escolha foi reforçada pela necessidade de uma ferramenta que se integre de forma mais fluída aos sistemas já utilizados pela empresa do nosso estudo de caso.

4 SOLUÇÃO PROPOSTA

Nesta seção, é apresentada a solução proposta, o Safer, uma aplicação desenvolvida para identificar e mitigar vulnerabilidades presentes nas dependências de software, avaliando a compatibilidade de suas versões com o projeto em questão. Este trabalho concentra-se, primariamente, em projetos que adotam o *Maven* como gestor de dependências no backend, com ênfase nos que são baseados na linguagem *Java*, com planos de expansão para dar suporte a outros gerenciadores no futuro. A solução proposta materializa-se por meio de uma interface de linha de comando (CLI), que utiliza a ferramenta open source *deps.dev*, criada pelo Google, para analisar a árvore de dependências do projeto visando identificar vulnerabilidades. Posteriormente, foram classificadas as versões das dependências utilizando o sistema de pontuação *CVSS* para, em seguida, tentar implementar as versões consideradas mais seguras. Este processo inclui testes de compatibilidade através dos procedimentos de construção (*build*) e testes unitários existentes no projeto-alvo. Adicionalmente, foi oferecida a opção de realizar testes exploratórios com o auxílio da ferramenta open source *Randoop*, que elabora testes ad-hoc empregados como testes de regressão [1]. Ao término da intervenção, é produzido um relatório detalhado, destacando as dependências que apresentaram vulnerabilidades, as versões atualizadas e as modificações realizadas.

4.1 Arquitetura do Safer

A arquitetura do Safer, descrita na Figura 1, é uma estrutura concebida para identificar, analisar e corrigir vulnerabilidades em dependências de software. Seu design modular e extensível é ideal para integração nos mais diversos sistemas de gerenciamento de dependências. Foi decidido começar com *Maven*, com projeções para expansão para outros gerenciadores no futuro. A arquitetura é projetada de acordo com os princípios de arquitetura de camadas, cada uma com responsabilidades bem definidas, promovendo a separação de preocupações e permitindo uma maior flexibilidade e facilidade de manutenção. [5, 7]

- **Camada de Apresentação:** essa camada é responsável por coletar as informações necessárias do usuário, como configurações do projeto-alvo e preferências de execução. Embora possa não haver uma GUI visual tradicional, considera-se a estrutura de configuração e os scripts de inicialização como parte desta camada, já que é a camada em que o usuário primeiro interage com o sistema.

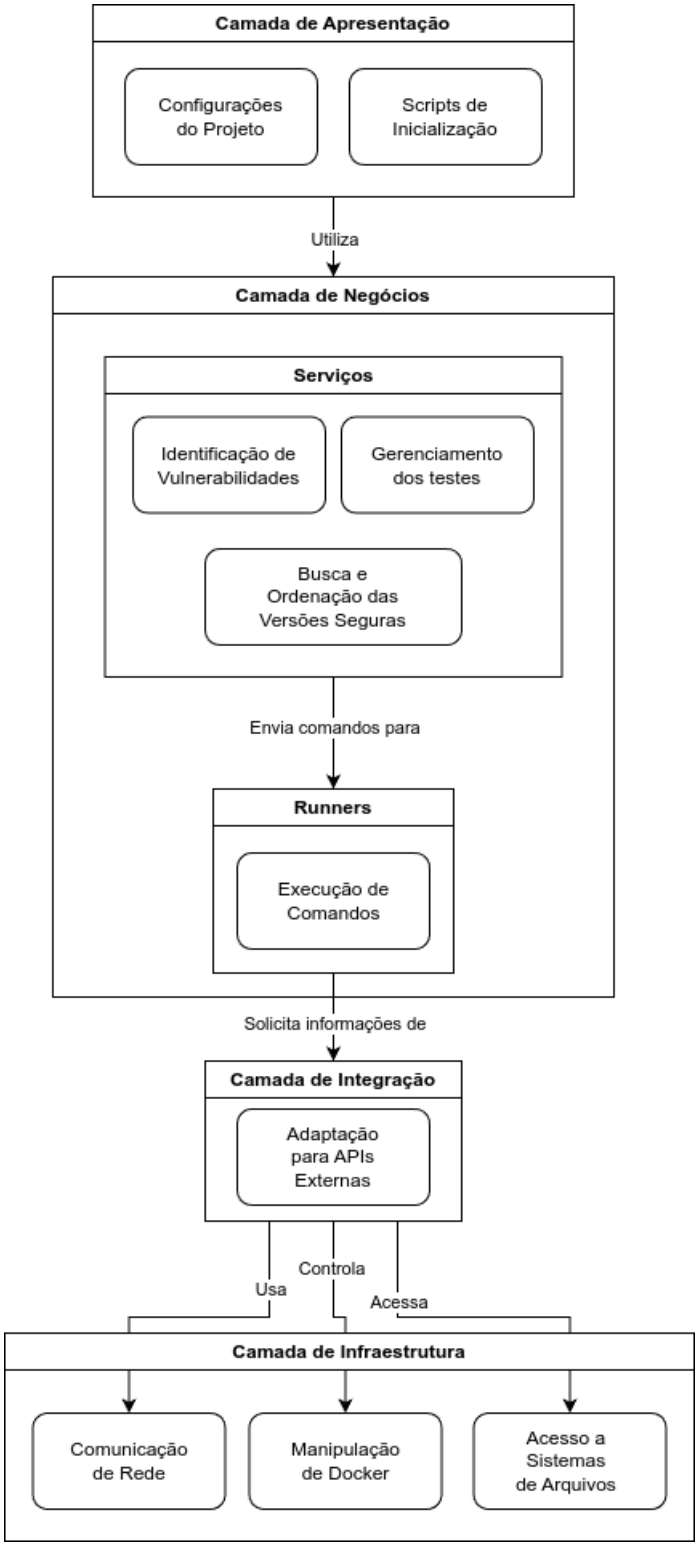

Figura 1: Visão Geral da Arquitetura em Camadas do Safer.

- **Camada de Negócios:** o núcleo lógico da solução, que executa as estratégias de detecção de vulnerabilidades e implementa os algoritmos de correção. Esta camada é subdividida em:
  - **Serviços:** contém a lógica para identificar vulnerabilidades nas dependências e buscar por versões seguras e compatíveis. Utilizam o padrão *Strategy* para permitir diferentes estratégias de busca e análise, dependendo do tipo de vulnerabilidade e do repositório de dependências. [20]
  - **Runners:** responsáveis por executar comandos específicos, como *builds* e testes, em ambientes isolados. Implementam o padrão *Command*, encapsulando a solicitação de execução como um objeto que pode ser armazenado, parametrizado e executado. [18]
- **Camada de Integração:** conecta a camada de negócios aos sistemas externos, como repositórios de dependências e APIs de análise de vulnerabilidades. O padrão *Adapter* é empregado no módulo de integração, conectando a solução aos sistemas externos, para garantir compatibilidade e facilitar a integração de serviços adicionais. [17]
- **Camada de Infraestrutura:** contém aspectos técnicos, como comunicação de rede, acesso a sistemas de arquivos e manipulação de *Docker*. Utiliza o padrão *Singleton* para instâncias de configuração global única, garantindo que exista uma única configuração e conexões, garantindo consistência em toda a aplicação. [19]

Esta arquitetura foi concebida com o objetivo de maximizar a reutilização de componentes, facilitar a manutenção do sistema e permitir a expansão futura para dar suporte a uma gama mais ampla de gerenciadores de dependências.

4.2 Questões de Implementação

No desenvolvimento da solução proposta, optou-se pela utilização do ambiente de execução *Node.js*[10], juntamente com a linguagem *TypeScript*[11] e o *Axios*[12] para as requisições HTTP, escolhidos pela ampla aceitação e facilidade de uso dentro do ecossistema *Node.js*. Para assegurar a compatibilidade com diversas versões das linguagens de programação e seus respectivos gerenciadores de dependências, destacou-se a utilização de contêineres *Docker*[13] como estratégia para possibilitar a execução da solução em diversos ecossistemas. Adicionalmente, para manter uma versão da dependência compatível com o projeto-alvo após atualização, é realizado o *Build* e testes no projeto, garantindo que as atualizações não comprometam a integridade e o funcionamento do sistema.

A análise de vulnerabilidades nas dependências dos projetos foi conduzida através da ferramenta *Open Source Insights*[14], desenvolvida e hospedada pelo Google. Esta ferramenta oferece uma visão detalhada da árvore de dependências e realiza consultas no banco de dados *Open Source Vulnerabilities (OSV)*[15] para identificar avisos de segurança [14]. Nas seções 4.2.1 e 4.2.2, será aprofundado o uso

[10]https://nodejs.org/docs/latest/api/documentation.html
[11]https://www.typescriptlang.org/docs
[12]https://axios-http.com/docs/intro
[13]https://docs.docker.com
[14]https://deps.dev/about
[15]https://google.github.io/osv.dev

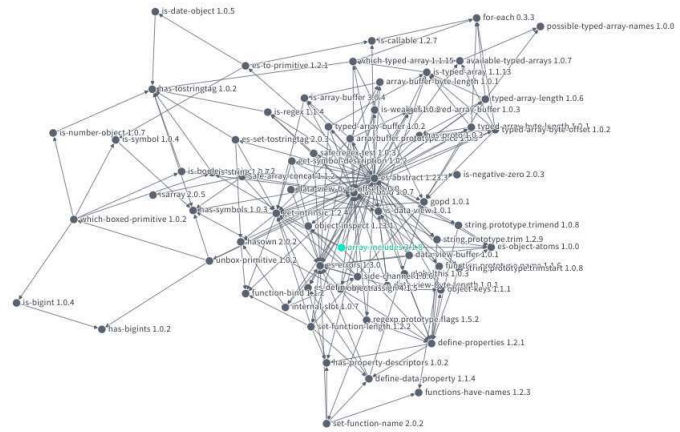

Figura 2: Grafo das dependências do pacote *array-includes* do *npm*.

do *Open Source Insights*: a seção 4.2.1 focará na análise da árvore de dependências, enquanto a seção 4.2.2 discutirá as vulnerabilidades identificadas. Na seção 4.2.3, será detalhada a estratégia para verificar a compatibilidade de versão das dependências, descrevendo as práticas adotadas para garantir que atualizações necessárias não introduzam erros de funcionamento.

4.2.1 Entendendo a Árvore de Dependências.

Uma dependência em desenvolvimento de software é qualquer biblioteca, módulo ou pacote externo necessário para o funcionamento de um projeto. As dependências são classificadas como diretas, quando adicionadas explicitamente pelo desenvolvedor no arquivo de configuração do projeto (por exemplo, `package.json` em projetos *Node.js*), e indiretas (ou transitivas), que são requisitadas pelas dependências diretas e não são explicitamente mencionadas pelo projeto. A estrutura que organiza e representa todas essas dependências é conhecida como árvore de dependências. Ela exibe não apenas as dependências diretas e indiretas de um projeto de forma hierárquica, mas também ilustra a relação recursiva entre elas, onde dependências indiretas podem possuir suas próprias dependências, diretas ou indiretas.

Para facilitar o entendimento, é possível a visualização da árvore de dependências por meio de uma representação gráfica, normalmente estruturada como um grafo. Nessa representação, os nós simbolizam as dependências, enquanto as arestas representam as relações entre elas. Um exemplo dessa visualização é fornecido na Figura 2, destacando a interconexão complexa entre dependências diretas e indiretas e a natureza recursiva da árvore.

Para analisar a árvore de dependências de um projeto, recorremos à ferramenta *Open Source Insights*, acessando um endpoint que nos fornece uma visão detalhada das dependências diretas e indiretas de uma dependência do projeto-alvo. Este endpoint, disponível em *deps.dev*[16], traz informações importantes como a distância de cada dependência em relação à dependência de origem do projeto-alvo e potenciais vulnerabilidades associadas a cada dependência,

[16]https://deps.dev/_/{system}/maven/p/{package.name}/v/{package.version}/dependencies

sejam elas diretas ou indiretas, e essas informações são utilizadas no relatório gerado.

4.2.2 *Entendendo as Vulnerabilidades.*

As vulnerabilidades em dependências de software representam falhas ou defeitos nas bibliotecas e pacotes que os sistemas de informação utilizam. Estas falhas podem comprometer significativamente a segurança e a integridade dos sistemas, tornando-os susceptíveis a ataques cibernéticos, perda de dados, e outras ameaças digitais. A detecção e correção dessas vulnerabilidades são essenciais para preservar a segurança cibernética, demandando um esforço contínuo e meticuloso por parte das equipes de desenvolvimento e manutenção de sistemas.

A complexidade da gestão de vulnerabilidades é amplificada pela natureza das dependências transitivas, nas quais uma vulnerabilidade em uma dependência pode ser indiretamente herdada por outros pacotes que dependem dela, aumentando assim o vetor de ataque e o risco associado [10]. Essa interconectividade e a possibilidade de propagação de vulnerabilidades destacam a importância de uma gestão eficaz e proativa das dependências de software.

Neste cenário, ferramentas como o *Open Source Insights* tornam-se fundamentais. Esta ferramenta consulta o *Open Source Vulnerabilities (OSV)*, um agregador de registros de vulnerabilidades que compila informações de diversas fontes, tais como, *curl-CVEs*, *National Vulnerability Database (NVD), GitHub Advisory Database (GHSA)*, entre outros. O *OSV.dev* oferece uma visão abrangente das vulnerabilidades conhecidas em software de código aberto, facilitando a identificação e resolução de fragilidades em dependências. [6]

Vários exemplos destacam a importância de uma abordagem proativa na gestão de vulnerabilidades. Um desses casos é a vulnerabilidade *Heartbleed*, que foi descoberta em 2014 na biblioteca de criptografia *OpenSSL*. Esta falha de segurança permitia a leitura não autorizada da memória de servidores, comprometendo a confidencialidade de dados sensíveis [22]. A agilidade e a cooperação da comunidade, juntamente com a utilização eficaz de ferramentas de detecção, como o *Dependency-Check*, foram fundamentais para minimizar os danos causados por essa vulnerabilidade. Tais incidentes enfatizam a importância de recursos como o Safer para a identificação e gestão proativa de vulnerabilidades em dependências de software.

4.2.3 *Estratégias para avaliar compatibilidade de versões.*

Na gestão de dependências de software, a manutenção da compatibilidade de versões emerge como um desafio essencial, especialmente em contextos de atualizações frequentes, onde mudanças incompatíveis podem afetar negativamente os pacotes dependentes [24]. Tradicionalmente, ferramentas de gestão de dependências recomendam versões seguras ou mais recentes das bibliotecas utilizadas. Contudo, esta abordagem muitas vezes desconsidera a compatibilidade dessas versões com o projeto em desenvolvimento, podendo resultar na introdução de erros ou incompatibilidades que comprometem a funcionalidade do software.

Exemplos notórios de ferramentas que priorizam a segurança em detrimento da compatibilidade incluem o *Dependabot*[17] e o *Snyk*. Ambas são eficazes na identificação de vulnerabilidades e na sugestão de atualizações para mitigá-las. No entanto, tais ferramentas não garantem a compatibilidade dessas atualizações com o código existente, um aspecto fundamental para a integridade do software. Um estudo exploratório sobre o *Dependabot* revela que, apesar de sua popularidade, as pontuações de compatibilidade oferecidas são insuficientes para reduzir a suspeita de atualizações, destacando uma lacuna significativa na garantia de que novas versões de dependências não introduzam conflitos no projeto existente [9].

O Safer distingue-se por abordar essa lacuna. Além de identificar versões seguras, o Safer realiza uma série de procedimentos destinados a verificar a compatibilidade dessas versões com o projeto-alvo. Isso inclui a execução de builds do projeto, a realização de testes unitários (quando aplicáveis) e a implementação de testes exploratórios. Essa abordagem assegura que as atualizações não apenas corrijam vulnerabilidades conhecidas mas também se integrem harmoniosamente ao software em desenvolvimento, evitando a introdução de novos problemas.

Para os ambientes *Java* com *Maven*, optou-se pelo uso de *Randoop* para a geração de testes exploratórios. *Randoop* é capaz de criar testes de regressão automáticos que ajudam a identificar incompatibilidades introduzidas por atualizações de dependências [1]. Embora estudos como o realizado por Souza e Machado [21] recomendem que testes gerados manualmente podem ser mais eficazes em termos de cobertura de linhas e mutação, a geração automática de testes, como os produzidos pelo *Randoop*, oferece benefícios significativos em termos de eficiência e rapidez, especialmente útil em ambientes dinâmicos e em evolução. A integração de ferramentas como o *Randoop* pode complementar as suítes de testes manuais, oferecendo um balanço entre cobertura e eficiência. A flexibilidade do Safer permite adaptar o que será executado para analisar a compatibilidade às necessidades específicas do projeto-alvo, uma característica que será discutida com mais profundidade na seção 4.3.1, dedicada à configuração da nossa solução.

A capacidade do Safer para avaliar a compatibilidade das dependências atualizadas, além da segurança, é o que o distingue das soluções existentes. Ao integrar procedimentos de verificação de compatibilidade, o Safer não apenas eleva o padrão de qualidade das atualizações de dependências, mas também minimiza o risco de regressões, garantindo que novas versões de bibliotecas possam ser adotadas com confiança.

4.3 Ferramenta em Uso

Nesta subseção, apresenta-se o Safer sendo utilizado. O Safer é projetado para apoiar desenvolvedores, cujas rotinas intensas frequentemente excluem a possibilidade de pausas para a análise meticulosa de vulnerabilidades nas dependências de seus projetos. Por meio de uma configuração ágil, requerendo apenas a instalação prévia de *Node.js*, *npm* e *Docker*, o Safer possibilita que os usuários atualizem suas dependências para versões mais seguras e potencialmente compatíveis.

4.3.1 *Configuração Inicial.*

```
{
  "$schema": "../../src/types/schema_config.json",
  "projectType": "maven",
```

[17] https://docs.github.com/en/code-security/dependabot

```
  "projectAbsolutePath": "/home/tcc/projeto",
  "weightVulnerability": {
    "low": 1,
    "medium": 2,
    "high": 3,
    "critical": 5
  },
  "executions": {
    "buildProject": true,
    "projectTests": true,
    "exploratoryTesting": true,
    "clearLastLogs": true,
    "clearLastReport": true
  },
  "exploratoryTestingParams": {
    "timeLimitGeneratingTests": 30
  }
}
```

Código 1: Arquivo de configuração JSON do Safer.

O arquivo de configuração apresentado, conforme ilustrado na Código 1, é estruturado em formato JSON e constitui a porta de entrada para a nossa solução. Este arquivo é projetado para facilitar a configuração personalizada de várias fases e parâmetros do Safer, assegurando um controle aprimorado sobre a execução e análise do projeto-alvo. A seguir, são detalhados os componentes chave desta configuração:

- *schema*: este campo atua como uma referência ao esquema de configuração e é designado como somente leitura, indicando que não deve ser alterado pelos usuários. Ele estabelece um padrão de estrutura que deve ser seguido para garantir a compatibilidade e funcionamento adequado do sistema.
- *projectType*: especifica o tipo de ferramenta de gerenciamento e construção utilizada no projeto, como *maven* ou *gradle*. Essa versatilidade permite que o sistema se adapte ao ambiente de desenvolvimento escolhido pelo usuário.
- *projectAbsolutePath*: estabelece o caminho absoluto do diretório do projeto, que deve começar com uma barra (`'/'`) para indicar um caminho absoluto. Esse detalhe é importante para que o sistema localize precisamente a raiz do projeto.
- *weightVulnerability*: este objeto aninhado atribui pesos às diferentes severidades de vulnerabilidades detectadas no projeto. Com categorias que variam de baixa a crítica, esses pesos são utilizados para avaliar a gravidade das vulnerabilidades encontradas, permitindo ao usuário priorizar correções baseadas no risco que cada vulnerabilidade representa.
- *executions*: um objeto que detalha as bandeiras de execução para diferentes fases de teste e construção do projeto-alvo. Isso inclui controle sobre se os testes automatizados do sistema-alvo devem ser realizados, se o projeto deve ser construído, e se testes exploratórios devem ocorrer, entre outros. Esse controle granular facilita a gestão dos recursos e do tempo durante o ciclo de vida da execução.
- *exploratoryTestingParams*: contém parâmetros específicos à execução de testes exploratórios, incluindo um limite de tempo para a geração de testes. Isso permite que os usuários definam quanto tempo o sistema deve dedicar à criação de testes durante a fase de teste exploratório.

Em termos práticos, essa configuração permite uma adaptação detalhada do processo de análise e correção de vulnerabilidades. Por exemplo, se um usuário definir pesos diferentes para os níveis de severidade das vulnerabilidades, o sistema pode priorizar a aplicação de correções de forma mais alinhada com a política de segurança da organização. Isso é particularmente útil em cenários onde múltiplas versões de uma dependência apresentam diferentes níveis de risco. O sistema poderá então recomendar a versão que, de acordo com os pesos configurados, apresenta o menor risco acumulado. Por exemplo, considere a situação em que a versão 1.2 de uma dependência X possui uma vulnerabilidade crítica com pontuação 9, enquanto uma versão anterior, 1.1, tem três vulnerabilidades médias, cada uma com pontuação 4. Se o usuário configurar o sistema para atribuir um peso de 2 para vulnerabilidades médias e um peso de 3 para vulnerabilidades críticas, o cálculo das pontuações ponderadas seria o seguinte: para a versão 1.1, as três vulnerabilidades médias resultariam em uma pontuação total de `3 x (4 x 2) = 24`, enquanto para a versão 1.2, a vulnerabilidade crítica resultaria em uma pontuação de `9 x 3 = 27`. Neste caso, o Safer consideraria a dependência com três vulnerabilidades médias como mais segura, uma vez que sua pontuação ponderada é menor. Assim, o sistema recomendaria aplicar primeiro a versão 1.1, que, de acordo com os pesos configurados, apresenta um menor risco acumulado.

Vale ressaltar que, quando um usuário define as opções *buildProject*, *projectTests* e *exploratoryTesting* como falsas, o sistema seleciona as versões consideradas mais seguras, mesmo que elas não sejam compatíveis. Esta abordagem é adotada como uma opção para maximizar a segurança do projeto-alvo. Contudo, para garantir que a segurança não comprometa a funcionalidade do sistema, recomenda-se a realização de intervenções manuais. Tais intervenções têm como objetivo verificar e ajustar a compatibilidade, assegurando assim a implementação efetiva da versão mais segura disponível. Esta prática, embora possa exigir um esforço adicional por parte dos desenvolvedores, é importante para segurança do sistema.

```
1  def processar_projeto(tipo, config):
2    if tipo == "Maven":
3      maven_scripts(config)
4    elif tipo == "Gradle":
5      # Codigo para Gradle
6      pass
7    else:
8      raise Exception("Tipo nao suportado.")
9
10 def maven_scripts(config):
11   dados = coletar_dados(config["caminho"])
12   versoes = buscar_versoes(dados["deps"])
13   ordenadas = ordenar_cvss(versoes)
14   criar_docker(dados["java"])
15   executar_build()
16   if config.get("testesExploratorios"):
17     criar_testes_exploratorios()
18
19   for dep in ordenadas:
```

```
20      for versao in dep:
21        try:
22          aplicar_dep(versao)
23          if config.get("executarBuild"):
24            if executar_build() == "falha":
25              continue  # Proxima versao
26          if config.get("executarTestes"):
27            if executar_testes() == "falha":
28              continue  # Proxima versao
29          if config.get("testesExploratorios"):
30            if exec_testes_expl() == "falha":
31              continue  # Proxima versao
32        except Exception:
33          continue  # Proxima versao
34        # Sucesso, ir para proxima dep
35        break
36      # Terminou todas as versoes de uma dep
37    gerar_relatorio(dados_atuais)
38
39  def safer():
40    config = ler_validar("config.json")
41    inicializar_ambiente()
42    processar_projeto(config["tipo"], config)
```

Código 2: Pseudocódigo em Python do fluxo geral de execução do Safer.

Para ilustrar o fluxo de execução do Safer, pode-se observar o Código 2. O processo inicia com a leitura e validação do arquivo de configurações através da função ‘safer()‘ (linhas 39-42). Dependendo do tipo do projeto, como o *Maven*, diferentes scripts são executados, conforme definido na função ‘processar_projeto()‘ (linhas 1-8). Para projetos *Maven*, a função ‘maven_scripts()‘ é chamada (linha 3), onde dados são coletados e versões das dependências são buscadas e ordenadas por critérios de segurança (linhas 10-13). Se configurado, criamos testes de regressão que serão executados para verificar a compatibilidade (linhas 16-17). O processo continua tentando aplicar cada versão das dependências, verificando o sucesso dos *builds* e testes iterativamente (linhas 19-35). Falhas em qualquer estágio resultam na tentativa da próxima versão, até que todas as dependências sejam testadas. Finalmente, um relatório é gerado (linha 37), completando o fluxo geral de execução.

4.4 Relatório gerado após execução

O relatório gerado pela ferramenta Safer oferece uma visão detalhada sobre as vulnerabilidades de segurança presentes nas dependências de um projeto de software. O objetivo principal do relatório é fornecer *insights* acionáveis que auxiliem os desenvolvedores e gestores de projeto a mitigar riscos associados às bibliotecas e pacotes utilizados. Ele é construído a partir da análise das informações coletadas pelo *Open Source Insights*, que por sua vez, agrega dados de vulnerabilidades a partir do *OSV*. Essa análise inclui a avaliação de diferentes versões de cada dependência utilizada no projeto, destacando aquelas que possuem vulnerabilidades conhecidas e classificando-as por severidade — de baixa a crítica.

O Safer direciona o usuário ao apresentar um resumo das vulnerabilidades antes e depois da execução das análises. Especificamente, ele informa o número de dependências afetadas e o total de vulnerabilidades, além de detalhar as características de cada vulnerabilidade, como ID, severidade, pontuação e se são transitivas, conforme o exibido na Figura 3. Isso permite que os usuários visualizem o impacto das mudanças realizadas durante o ciclo de desenvolvimento e entendam como as atualizações de dependências podem influenciar a segurança do projeto.

```
Vulnerabilities Summary:
The information was obtained from Open Source Insights, which gathers security
 advisories information from OSV.

Number of dependencies with vulnerabilities:
 Before: 14 After: 4
Number of vulnerabilities:
 Before: 110 After: 12
Before execution, total vulnerabilities were:
 Low: 2, Medium: 38, High: 66, Critical: 4
After execution, total vulnerabilities are:
 Low: 1, Medium: 4, High: 5, Critical: 2
Details of vulnerabilities before execution:
Dependency org.springframework.boot:spring-boot-starter-actuator: 3.0.4:
- ID: GHSA-g5h3-w546-pj7f, Severity: critical, Score: 9.8, Transitive: Yes
```

Figura 3: Resumo do relatório gerado.

Os resultados são organizados em seções que facilitam a interpretação e o acompanhamento das mudanças de segurança entre as versões das dependências. Cada dependência é listada com suas respectivas vulnerabilidades, fornecendo um panorama claro da situação atual e histórica do projeto em relação às questões de segurança. O relatório é apresentado tanto no terminal como em um arquivo de log, localizado no diretório ‘safer/.temp/report‘, facilitando o acesso e a análise posterior dos dados.

No final do relatório, são fornecidas recomendações estratégicas para a melhoria contínua da segurança do projeto. Essas incluem a atualização prioritária das dependências para suas versões mais recentes e seguras, a realização regular de auditorias de segurança e a adoção de práticas de codificação segura. Além disso, sugere-se a implementação de testes automatizados e a utilização de pipelines de integração e entrega contínuas (CI/CD), para garantir que todas as modificações, incluindo atualizações de dependências, sejam extensivamente testadas antes de serem implementadas em ambientes de produção.

5 AVALIAÇÃO

Para avaliar a eficácia da ferramenta Safer, realizou-se um estudo de caso em cinco projetos, todos desenvolvidos em *Maven* e *Java* 17 por equipes distintas de uma empresa parceira. Devido às exigências de confidencialidade, os projetos foram anonimizados e identificados como P1 a P5. A Tabela 1 apresenta o tamanho de cada projeto em *KLOC* (milhares de linhas de código) e o número de dependências diretas. Como referência inicial, foram utilizados os resultados produzidos pelo *Dependency-Check (DC)* para compreensão das vulnerabilidades presentes antes e depois da aplicação do Safer.

Os resultados obtidos com o Safer são apresentados em duas tabelas subsequentes. A primeira delas, Tabela 2, compara as mudanças identificadas pelo DC antes e após o uso do Safer, enquanto a segunda, Tabela 3, mostra os resultados diretamente obtidos pelo Safer. As vulnerabilidades são classificadas em níveis de baixa a crítica, conforme a severidade do risco que representam. Foi avaliada a eficácia da ferramenta não apenas pelo número de vulnerabilidades detectadas e corrigidas, mas também pela sua capacidade de identificar e mitigar aquelas de maior severidade

Projeto	KLOC	Quant. Dependências Diretas
P1	83	26
P2	4.6	24
P3	1.5	11
P4	43.2	21
P5	20.3	25

Tabela 1: Tamanho dos projetos analisados

Projeto	Severidade	DC Antes	DC Depois	Diferença
P1	Baixa	0	0	0
	Média	11	11	0
	Alta	37	33	-4
	Crítica	11	11	0
P2	Baixa	0	0	0
	Média	7	7	0
	Alta	20	20	0
	Crítica	13	10	-3
P3	Baixa	0	0	0
	Média	5	6	+1
	Alta	29	29	0
	Crítica	14	11	-3
P4	Baixa	0	0	0
	Média	13	11	-2
	Alta	46	41	-5
	Crítica	15	15	0
P5	Baixa	0	0	0
	Média	11	13	+2
	Alta	46	45	-1
	Crítica	21	15	-6

Tabela 2: Comparativo de severidade das vulnerabilidades - *Dependency-Check*

A análise das tabelas indica que, embora o *Dependency-Check* apresente uma redução modesta no número total de vulnerabilidades, principalmente mantendo ou reduzindo ligeiramente as de alta criticidade, o Safer exibiu uma capacidade notável de reduzir significativamente as vulnerabilidades em todos os níveis de severidade. Sugere-se que a diferença nos resultados entre o DC e o Safer pode ser atribuída ao uso de diferentes bases de dados de vulnerabilidades (*NVD* pelo DC e *OSV* pelo Safer), sendo que o *OSV* oferece informações mais atualizadas e detalhadas, fundamentais para a identificação e mitigação de vulnerabilidades emergentes e mais graves.

5.1 Cálculo da Eficácia Geral do Safer na Redução de Vulnerabilidades

Para avaliar a eficácia do Safer na redução do número de vulnerabilidades nos projetos analisados, foram realizadas somas das vulnerabilidades identificadas antes e depois da implementação da ferramenta.

Projeto	Severidade	Safer Antes	Safer Depois	Diferença
P1	Baixa	1	0	-1
	Média	58	2	-56
	Alta	80	6	-74
	Crítica	1	0	-1
P2	Baixa	0	0	0
	Média	32	2	-30
	Alta	56	6	-50
	Crítica	1	0	-1
P3	Baixa	0	0	0
	Média	36	3	-33
	Alta	55	5	-50
	Crítica	1	0	-1
P4	Baixa	2	1	-1
	Média	38	4	-34
	Alta	66	5	-61
	Crítica	4	2	-2
P5	Baixa	1	1	0
	Média	43	5	-38
	Alta	66	8	-58
	Crítica	4	2	-2

Tabela 3: Comparativo de severidade das vulnerabilidades - Safer

5.1.1 *Soma das Vulnerabilidades Antes da Implementação do Safer:*

- Projeto P1: 1 + 58 + 80 + 1 = 140
- Projeto P2: 0 + 32 + 56 + 1 = 89
- Projeto P3: 0 + 36 + 55 + 1 = 92
- Projeto P4: 2 + 38 + 66 + 4 = 110
- Projeto P5: 1 + 43 + 66 + 4 = 114

Total antes: 140 + 89 + 92 + 110 + 114 = 545

5.1.2 *Soma das Vulnerabilidades Depois da Implementação do Safer:*

- Projeto P1: 0 + 2 + 6 + 0 = 8
- Projeto P2: 0 + 2 + 6 + 0 = 8
- Projeto P3: 0 + 3 + 5 + 0 = 8
- Projeto P4: 1 + 4 + 5 + 2 = 12
- Projeto P5: 1 + 5 + 8 + 2 = 16

Total depois: 8 + 8 + 8 + 12 + 16 = 52

5.1.3 *Diferença Total e Porcentagem de Redução:*

Diferença total = Total antes - Total depois = 545 − 52 = 493

Porcentagem de Redução:

$$\begin{aligned}\text{Porcentagem de Redução} &= \left(\frac{\text{Diferença Total}}{\text{Total Antes}}\right) \times 100 \\ &= \left(\frac{493}{545}\right) \times 100 \\ &\approx 90.46\%\end{aligned}$$

Portanto, o Safer reduziu as vulnerabilidades em aproximadamente 90,46% nos projetos utilizados na avaliação. O Safer não só diminuiu drasticamente o número total de vulnerabilidades, como

também mostrou uma eficácia significativa na redução das vulnerabilidades de alta e crítica severidade. Por exemplo, no projeto P1, as vulnerabilidades de alta severidade foram reduzidas de 80 para 6, e as críticas foram completamente eliminadas após a aplicação do Safer, demonstrando a eficácia da ferramenta em priorizar e resolver as ameaças mais sérias aos sistemas. Um aspecto importante do processo de atualização das dependências é o emprego do *Randoop*, uma ferramenta para testes exploratórios, que verifica a compatibilidade das versões atualizadas das dependências. O *Randoop*, através de testes de regressão automáticos, facilita a identificação de incompatibilidades que possam surgir após as atualizações, aumentando a confiança de que as correções aplicadas não afetem adversamente as funcionalidades do sistema.

É fundamental que a execução e análise dos resultados do Safer sejam conduzidas por indivíduos com conhecimento dos projetos aplicados, garantindo uma avaliação precisa e uma implementação efetiva das correções sugeridas. Estes resultados sugerem que o Safer é eficaz na identificação e correção de vulnerabilidades em dependências de software, demonstrando uma alta taxa de acerto na mitigação das vulnerabilidades nos projetos testados e, consequentemente, ampliando a segurança dos sistemas analisados.

6 CONCLUSÃO E TRABALHOS FUTUROS

A crescente demanda por sistemas de software mais seguros é um reflexo direto dos desafios enfrentados pelas empresas, onde a segurança das informações tornou-se um componente crítico para a continuidade dos negócios. Neste contexto, ferramentas como o Safer emergem como soluções indispensáveis para fortalecer a segurança das aplicações, ao oferecer métodos de detecção e correção de vulnerabilidades que complementam as ferramentas existentes, como o *Dependency-Check*. A utilização do Safer proporciona uma camada adicional de segurança, melhorando significativamente a gestão de dependências de software e reduzindo potenciais riscos.

O estudo de caso realizado demonstrou a capacidade do Safer em identificar e mitigar vulnerabilidades de forma eficaz, mesmo em um curto período de desenvolvimento. Apesar das dificuldades encontradas, como a escolha inicial de focar apenas em projetos *Maven*, os resultados obtidos confirmam que ferramentas como o Safer são úteis para uma gestão de segurança eficaz. Elas automatizam trabalhos como a análise das dependências. Isso aumenta a confiabilidade dos sistemas desenvolvidos e proporciona maior tranquilidade para os desenvolvedores e as organizações.

No entanto, é necessário reconhecer que a eficácia do Safer, como de qualquer outra ferramenta de segurança, pode ser influenciada pela qualidade e atualização dos bancos de dados de vulnerabilidades que utiliza. A confiabilidade dessas bases de dados, como o *OSV* utilizado pelo Safer, é crucial para a precisão das detecções e correções propostas. Estudos adicionais envolvendo um número maior de projetos e diferentes configurações de software são recomendados para validar ainda mais a eficácia do Safer e explorar sua aplicabilidade em diferentes contextos e tecnologias.

Como trabalho futuro, planeja-se expandir a funcionalidade do Safer para incluir o suporte a outros tipos de projetos, como os que utilizam *NPM*, e a integração de outras ferramentas de testes exploratórios adaptadas para projetos com frontend, como o *Cytestion* [12]. Além disso, a parametrização do tipo de banco de dados usado para coletar informações sobre vulnerabilidades oferecerá mais flexibilidade aos usuários, permitindo escolher as fontes mais confiáveis e atualizadas. A implementação de técnicas de Inteligência Artificial para auxiliar na detecção e correção de vulnerabilidades também está entre os planos futuros, prometendo elevar ainda mais a capacidade de resposta e eficiência das análises de segurança realizadas pelo Safer.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me conceder saúde e sabedoria, e dedico a Ele todos os esforços realizados para alcançar esta etapa. Sou grato ao meu orientador e professor, Dr. Cláudio de Souza Baptista, pela chance de estar no laboratório e pelos valiosos ensinamentos. Também agradeço a Hugo Feitosa de Figueirêdo, pelos conhecimentos compartilhados e pela confiança depositada ao longo desses anos, elementos essenciais na formação do profissional e pessoa que sou hoje. Expresso minha gratidão aos colegas do LSI, especialmente a Cristóvão e Thiago, cuja ajuda foi fundamental para a viabilidade deste trabalho. Agradeço ao corpo docente do curso de Ciência da Computação da UFCG pelo apoio constante e pelo conhecimento transmitido.

Um agradecimento especial à minha noiva, Layla, que sempre me apoiou e incentivou a ser melhor, por todo amor, suporte e paciência, especialmente nesta reta final. Agradeço aos amigos que fiz durante o curso - Eric, Ruan, Victor e Wellisson - cuja companhia tornou esta jornada mais fácil e memorável. Por fim, sou eternamente grato à minha família, aos meus pais e irmãos, pelo amor e suporte incondicional durante todo o meu percurso acadêmico.

REFERÊNCIAS

[1] [n. d.]. *Randoop Manual.* Acesso em: 12/04/2024.
[2] Luca Allodi, Marco Cremonini, Fabio Massacci, et al. 2020. Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empirical Software Engineering* 25 (2020), 1063–1094. https://doi.org/10.1007/s10664-019-09797-4
[3] R. G. Cooper. 2021. Accelerating innovation: some lessons from the pandemic. *Journal of Product Innovation Management* 38, 2 (10 Feb. 2021).
[4] Datadog. 2024. State of DevSecOps. Online. https://www.datadoghq.com/state-of-devsecops/ Acessado em: 30 de abril de 2024.
[5] E. Evans. 2014. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley, Boston, Mass.; Munich.
[6] FAQ. [n. d.]. FAQ. https://google.github.io/osv.dev/faq/. Acesso em: 11/03/2024.
[7] M. Fowler. 2015. *Patterns of enterprise application architecture.* Addison-Wesley, Boston, Mass.; Munich.
[8] C.-P. Georg and A. Mele. 2023. A Strategic Model of Software Dependency Networks. *SSRN Electronic Journal* (2023).
[9] R. He et al. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Transactions on Software Engineering* (1 Jan. 2023), 1–18.
[10] A. J. Jafari et al. 2023. Dependency Practices for Vulnerability Mitigation. arXiv preprint arXiv:2310.00000.
[11] R. G. Kula et al. 2017. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (11 May 2017), 384–417.
[12] Thiago Santos de Moura, Everton L. G. Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated GUI Testing for Web Applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES '23).* Association for Computing Machinery, New York, NY, USA, 388–397. https://doi.org/10.1145/3613372.3613408
[13] Michael Novinson. 2021. 10 Technology Vendors Affected by the Log4j Vulnerability. Online. https://www.crn.com/slide-shows/security/10-technology-vendors-affected-by-the-log4j-vulnerability?page=2 Acessado em: 29 de abril de 2024.
[14] Open Source Insights. [n. d.]. Open Source Insights. https://deps.dev/about. Acesso em: 16 abr. 2024.
[15] OWASP. [n. d.]. OWASP Top Ten. https://owasp.org/www-project-top-ten/. Último acesso em 10/03/2024.

[16] S. E. Ponta, H. Plate, and A. Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (30 June 2020), 3175–3215.
[17] REFACTORING GURU. [n. d.]. Adapter. https://refactoring.guru/design-patterns/adapter.
[18] REFACTORING GURU. [n. d.]. Command. https://refactoring.guru/design-patterns/command.
[19] REFACTORING GURU. [n. d.]. Singleton. https://refactoring.guru/design-patterns/singleton.
[20] REFACTORING GURU. [n. d.]. Strategy. https://refactoring.guru/design-patterns/strategy.
[21] B. Souza and P. Machado. 2020. A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*.
[22] SYNOPSYS. [n. d.]. Heartbleed Bug. https://heartbleed.com/.
[23] W. Tang et al. 2022. Towards Understanding Third-party Library Dependency in C/C++ Ecosystem. arXiv preprint arXiv:2210.00000.
[24] D. Venturini et al. 2023. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. *ACM Transactions on Software Engineering and Methodology* 32, 4 (26 May 2023), 1–26.