DevSecOps - integração da segurança contínua em pipelines DevOps: um estudo de caso

Rafael Pio de França[1], **Vinicius Barcelos da Silva**[1]

[1]Instituto Federal Fluminense (IFF)
Rua Dr. Siqueira, 273 - Parque Dom Bosco, Campos dos Goytacazes – RJ – Brazil

rafaelpfranca73@gmail.com, viniciusbs@iff.edu.br

Abstract. *DevOps seeks to integrate the development and operations worlds, through strong automation in the software development and delivery phases. While DevOps does not exclude security, its adoption is seen as a bottleneck for DevOps. The present work presents itself to demonstrate, through a case study, the development of a DevOps pipeline and its integration with continuous security, thus achieving a DevSecOps pipeline. Through the comparison between the pipelines, their automation roles, however, the DevSecOps pipeline managed to prevent the delivery of a vulnerable application to production, if as an adequate solution for the use of DevOps in a safe and agile way.*

Resumo. *O DevOps busca integrar os mundos de desenvolvimento e operações, por meio de forte automatização nas fases de desenvolvimento e entrega o software. Apesar do DevOps não excluir a segurança, sua adoção é vista como um gargalo para o fluxo DevOps. O presente trabalho se propõe a demonstrar, por meio de um estudo de caso, o desenvolvimento de um pipeline DevOps e sua integração com a segurança contínua, alcançando assim um pipeline DevSecOps. Através da comparação entre os pipelines, contatou-se que ambos cumprem seus papéis de automatização, porém, o pipeline DevSecOps conseguiu impedir à entrega de uma aplicação vulnerável à produção, se mostrando como solução pertinente para o uso do DevOps de maneira segura e ágil.*

1. Introdução

O desenvolvimento ágil permitiu que os desenvolvedores atendessem às necessidades do mercado de maneira mais dinâmica, onde pequenas funcionalidades são constantemente integradas ao software em intervalos de tempo mais curtos, permitindo atender novas demandas do negócio na medida em que estas surgem [Koskinen 2019].

No entanto, conforme o ritmo de desenvolvimento acelerava, a equipe de operações, responsável pelo gerenciamento das modificações do software na produção, se tornou incapaz de acompanhar o novo fluxo de mudanças proposto pelos desenvolvedores. Na prática, isso resultou na equipe de operações se tornando um gargalo, ocasionando longos atrasos entre as atualizações e a implantação do código [Leite et al. 2019].

Como solução, surge o movimento DevOps, que visa aproximar as equipes de desenvolvimento e operações, automatizando os processos necessários para levar um software do desenvolvimento à implantação em um ambiente de produção. Em um ambiente DevOps, recursos de software são adicionados e corrigidos por meio de ciclos rápidos. O DevOps não exclui a segurança, porém, o que ocorreu na realidade foi que apesar das

equipes de desenvolvimento e operações se aproximarem, a equipe de segurança continuou isolada [Jetbrains 2021].

A segurança, quando implementada de maneira tradicional, ocorre no estágio de teste e com forte intervenção manual, provocando um atraso na implantação do software desenvolvido, e por consequência, prejudicando o fluxo DevOps. Essa desaceleração tem levado organizações a deixar a segurança de lado, colocando em risco tanto o software quanto os dados de seus clientes [Ahmed 2019].

A literatura indica o surgimento de diversos problemas de segurança ocasionados pelo uso do DevOps sem a adequada implementação da segurança, tais como configurações incorretas de container, exposição de credenciais, vulnerabilidades de imagens, violações de dados, uso de bibliotecas vulneráveis e falta de política de privilégios mínimos para acesso de usuários [Koskinen 2019].

Em uma pesquisa conduzida pela Osterman Research, apenas 42% dos entrevistados estão confiantes de que suas equipes de DevOps e segurança estão gerenciando adequadamente suas responsabilidades e eliminando pontos cegos na proteção de aplicações [Radware 2020].

Em um esforço para solucionar esse problema, criou-se o termo DevSecOps, que visa abordar, de maneira explícita, a segurança nos processos DevOps. Essa nova abordagem busca integrar a equipe de segurança às equipes de DevOps, além de adotar práticas e ferramentas automatizadas de segurança a fim de evitar que o fluxo de trabalho fique lento. O DevSecOps faz uso extensivo da automatização na segurança, permitindo que os controles de segurança sejam rápidos, escalonáveis e eficazes, de maneira que vulnerabilidades sejam detectadas, alertadas e corrigidas em um ritmo acelerado [Myrbakken and Colomo-Palacios 2017].

No entanto, embora o DevSecOps se mostre fundamental para organizações que almejam a adoção do DevOps, por se tratar de uma área nova, ainda há carência de trabalhos na área, dificultando a compreensão de sua importância assim como sua correta implementação dentro das organizações [Koskinen 2019].

Este estudo tem como objetivo geral demonstrar a integração da segurança contínua através de um procedimento de migração de um *pipeline* DevOps para um *pipeline* DevSecOps e compará-los, fazendo uso de um estudo de caso fictício. De forma mais específica, busca-se implementar um estudo de caso de uma API vulnerável em um *pipeline* DevOps e, posteriormente, realizar um procedimento de migração para um *pipeline* DevSecOps, a fim de comparar a segurança fornecida pelo *pipeline* antes e após o uso de práticas e ferramentas DevSecOps.

Sucintamente, este artigo está organizado da seguinte forma: a seção 2 apresenta uma revisão bibliográfica sobre DevSecOps, destacando a importância da adição da segurança contínua nos fluxos DevOps; a seção 3 apresenta a metodologia adotada nesse trabalho; a seção 4 descreve o estudo de caso proposto, destacando as principais etapas e os resultados obtidos; e, finalmente são apresentadas as considerações finais.

2. DevSecOps

O DevOps busca integrar os mundos de desenvolvimento e operações, por meio de forte automatização nas fases de desenvolvimento, implantação e monitoramento de infraestru-

tura. É uma mudança organizacional, na qual equipes independentes que desempenhavam suas funções separadamente agora trabalham em conjunto visando entregas contínuas de recursos operacionais [Ebert et al. 2016].

Devido à popularidade do DevOps, o número de organizações implementando suas práticas aumentou nos últimos anos. No entanto, um *survey* realizado pela Hewlett Packard Enterprise indica que a segurança não é algo que as organizações estejam implementando em seus processos DevOps, embora acreditem ser necessário [Enterprise 2016].

Nesse contexto, DevSecOps surge como um aprimoramento do DevOps. Assim como o DevOps lida com a lacuna existente entre a equipe de desenvolvimento e a de operações, o DevSecOps surgiu para aliar a equipe de segurança à equipe de DevOps, inserindo práticas de segurança em todas as fases do projeto [Carter 2017].

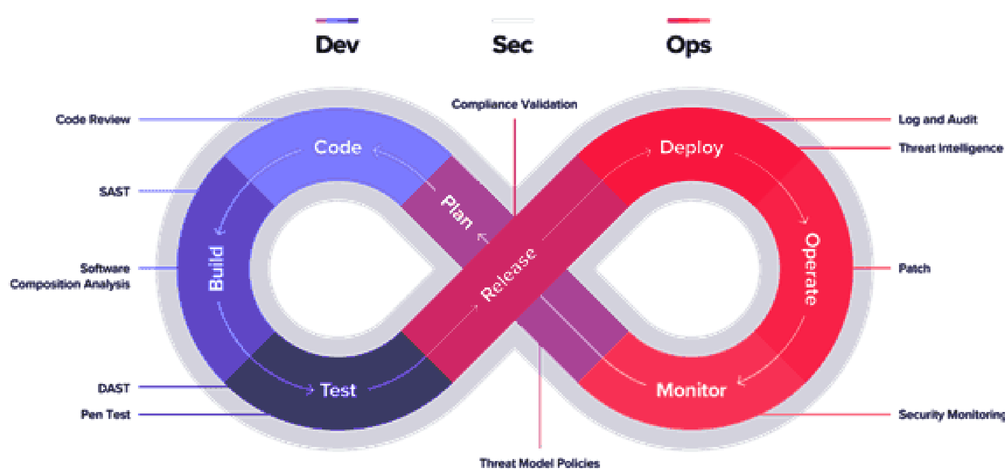

Figura 1. Ciclo DevSecOps [Microfocus 2022]

O DevSecOps promove a ideia de que a segurança é responsabilidade de todos e todos devem realizar suas funções prezando pela segurança. Os desenvolvedores devem escrever seus códigos cientes de como lidar com as possíveis ameaças de segurança. O mesmo se aplica aos profissionais de operações, que devem provisionar e configurar os ambientes seguindo as boas práticas de segurança [CeArley et al. 2016]. Como pode ser visto na Figura 1, a segurança é implementada em cada fase do ciclo DevOps. Portanto, o DevSecOps visa manter os processos já existentes dentro de um ambiente DevOps, apenas adequando práticas de segurança.

Em um *pipeline* DevSecOps, conceitos básicos de segurança como autenticação, autorização, disponibilidade, confidencialidade, identificação, integridade e não repúdio são avaliados por diferentes tipos de testes de segurança. Segundo [Danielecki 2019], a abordagem mais geral é dividi-los nas categorias de SAST, DAST e IAST.

Static Application Security Testing (SAST), também conhecido como teste de caixa branca, é executado durante o desenvolvimento sem ter a aplicação em execução, a fim de encontrar previamente vulnerabilidades de segurança diretamente no código-fonte. Segundo [Freitas 2020], SAST realiza: Análise do código-fonte em relação a um conjunto de regras que verificam os dados e o fluxo de controle do código-fonte em busca de vulnerabilidades de segurança; Verificação através de regex de segredos no código-fonte da aplicação, como senhas, chaves de API, etc; Verificação de dependências, através da

comparação das bibliotecas importadas no projeto com um banco de dados de vulnerabilidades conhecidas; Scan de Container, analisando a imagem do container produzida em relação a um banco de dados de pacotes vulneráveis conhecidos.

Dynamic Application Security Testing (DAST), também conhecido como teste de caixa preta, é um teste de segurança que tenta simular o comportamento de um invasor contra uma versão em execução da aplicação, introduzindo intencionalmente injeções de falha para encontrar vulnerabilidades, sem realizar acesso direto ao código fonte da aplicação. Através do DAST, pode-se realizar: Validação de entrada, realizando verificações na aplicação a fim de se detectar vulnerabilidades como *SQL Injection*, *Cross-site scripting (XSS)*, *Buffer Overflows*, etc; Problemas de configuração do servidor, tais como falta de cabeçalhos de segurança, comunicações não criptografadas, etc. As descobertas no DAST são altamente confiáveis (baixa taxa de falsos positivos) [Simpson 2018].

Interactive Application Security Testing (IAST) combina o DAST, testando em aplicações em execução, e SAST, testando o código em um estado de não execução e é facilmente integrado e automatizado em um pipeline. É capaz de analisar tanto do código fonte quanto do fluxo de tráfego para dentro e para fora da aplicação [Handova 2020].

Especificamente, o presente trabalho fará uso de duas ferramentas SAST e uma ferramenta DAST no estudo de caso que será apresentado nas próximas seções.

3. Metodologia

O presente trabalho se propôs a desenvolver um estudo de caso fictício visando servir, como uma referência, para organizações que fazem uso do DevOps tornarem seus *pipelines* mais seguros. O estudo de caso consistiu na migração de um *pipeline* DevOps para um *pipeline* DevSecOps, onde posteriormente foi feita uma comparação entre ambos, demonstrando como o uso de práticas e ferramentas *open source* de segurança contribuem para a mitigação de vulnerabilidades em softwares.

O estudo de caso foi dividido em cinco etapas:

1. Desenvolvimento de uma API contendo diversos problemas de segurança. Essa API será usada como o software alvo do *deploy* pelo *pipeline*. A API vulnerável desenvolvida seguiu a arquitetura REST e foi construída fazendo uso do JavaScript *runtime environment* Node.js e do *framework* Express.js, além do banco de dados MongoDB, tecnologias comumente utilizadas pela indústria de software para o desenvolvimento de APIs. O repositório remoto utilizado para a hospedagem do código fonte da API está no GitHub;
2. Implementação do *pipeline* DevOps responsável por automatizar o *deploy* da API para um ambiente de produção. O *pipeline* DevOps implementado para este estudo de caso possui como finalidade o *deploy* contínuo de novas versões da API a cada *commit/push* realizado. Para compor o *pipeline*, foram utilizadas tecnologias que se destacam em ambientes DevOps na atualidade, como GitHub Actions e *containers* Docker. Com o propósito de validação, deverá ser observado que o *pipeline* DevOps realiza o *deploy* da API apesar das vulnerabilidades existentes;
3. Implementação do *pipeline* DevSecOps, no qual incorporou no *pipeline* DevOps uma série de práticas e ferramentas de segurança que serão executadas de maneira automatizada a cada novo *commit/push* do código da API, com a finalidade de

que as vulnerabilidades predeterminadas sejam detectadas em tempo real e de que o processo de *deploy* seja interrompido. Com o propósito de validação, deverá ser observado que o *pipeline* DevSecOps não permita o *deploy* enquanto houver vulnerabilidades na API;

4. Correção das vulnerabilidades no código fonte da API. Para tanto, foram adicionadas bibliotecas de segurança na aplicação, atualização de dependências, além da adoção de boas práticas de segurança de código;
5. Execução do pipeline DevSecOps com a API segura. Após as correções, espera-se que o *deploy* ocorra com sucesso e testes manuais de segurança comprovam que o software em produção se encontra seguro, evidenciando assim, a importância de *pipelines* funcionarem de maneira combinada com a segurança contínua.

As vulnerabilidades utilizadas no estudo de caso foram escolhidas com base em sua recorrência em ambientes reais, criticidade e na viabilidade de sua implementação. Como forma de lidar com as vulnerabilidades, foram pesquisadas práticas e ferramentas de segurança utilizadas ao se elaborar um *pipeline* DevSecOps para APIs. As ferramentas foram selecionadas com base nas vulnerabilidades escolhidas, documentação disponível, popularidade entre a comunidade e por serem *open source*. Por meio do presente estudo de caso, foi feita uma comparação final entre os resultados obtidos pelos dois *pipelines*, onde foram analisados aspectos da segurança fornecida por ambos.

4. Estudo de Caso, Resultados e Discussões

Esta seção tem como objetivo apresentar o desenvolvimento da API vulnerável utilizada para o presente estudo de caso, assim como a migração para a sua versão segura. Também será apresentada a construção do *pipeline* DevOps, responsável pela automatização da execução dos testes de software e pelo *deploy* da nova versão da API à produção, tal qual sua migração para o DevSecOps, onde serão adicionados testes de segurança ao *pipeline*.

Para fins didáticos, optou-se pela combinação da seção de Resultados e Discussões na presente seção. Essa decisão justifica-se pela natureza desse estudo de caso, onde torna-se intuitivo para o leitor o entendimento de todo o processo a partir da compreensão do resultado de cada tentativa de *deploy* que será discorrida ao longo dessa seção.

4.1. Desenvolvimento da API Vulnerável

Para o desenvolvimento do estudo de caso proposto, foi desenvolvida uma API minimalista para um contexto fictício, tendo como finalidade permitir que administradores cadastrados em um banco de dados possam fazer *login* e consultar palestrantes e espectadores cadastrados para um evento.

A API foi desenvolvida fazendo uso da popular pilha de tecnologias backend composta por Node.js, Express.js e MongoDB. Duas bibliotecas adicionais também foram utilizadas no projeto. A primeira delas é o mongoose, responsável por fornecer uma solução baseada em esquemas para modelar os dados do MongoDB. A outra biblioteca adicionada ao projeto foi o jsonwebtoken, uma implementação de JSON Web Token, sendo este um padrão definido pela RFC7519, com o propósito de transmitir ou armazenar de maneira compacta e segura objetos JSON entre diferentes aplicações.

A API desenvolvida fornece três *endpoints*:

- POST /login: *endpoint* utilizado para login de administradores. Recebe via *body* um objeto json contendo e-mail e senha, e responde um *token* JWT para o administrador provar nas próximas requisições que efetuou o login com sucesso.
- GET /speakers: *endpoint* utilizado para obter os palestrantes cadastrados no banco de dados. Recebe via *header* um *token* JWT, transmitido ao administrador durante o processo de login.
- GET /participants: *endpoint* utilizado para obter os participantes cadastrados no banco de dados. Também recebe via *header* um *token* JWT, transmitido ao administrador durante o processo de login.

A fim de constatar a importância de um *pipeline* integrado com a segurança, a API foi inicialmente desenvolvida contendo uma série de problemas de segurança:

- *NoSQL Injection: Login Bypass (Query Selector Injection Attack)*
- *Sensitive Data Exposure: Username and password exposure*
- *Broken Authentication: JWT algorithm confusion*
- *Security Misconfiguration: Lack of security headers or directives*

A primeira vulnerabilidade inserida propositalmente na API é a *NoSQL injection*. Esse tipo de vulnerabilidade permite que invasores injetem comandos em bancos de dados que não usam consultas SQL, como o MongoDB. O *NoSQL injection* é listado na terceira posição do OWASP 2021 [Owasp 2021]. Um dos ataques realizados por meio do *NoSQL injection* é o *Login Bypass*, onde um invasor consegue, sem credencias válidas, um *login* de sucesso. Na API desenvolvida, essa vulnerabilidade ocorre na seguinte linha de código: *const user = await Admin.findOne('email':req.body.email , 'password':req.body.password);*. Esse código é vulnerável a *NoSQL Injection*, pois não é realizado nenhum tipo de tratamento nos dados recebidos do usuário, sendo esses dados usados diretamente para execução da *query*.

A segunda vulnerabilidade inserida propositalmente na API é a *Broken Authentication*. Ao utilizar componentes de *software* de terceiros na aplicação, estes podem apresentar vulnerabilidades de segurança além da possibilidade de conter códigos maliciosos. Especificamente, a vulnerabilidade *Broken Authentication* é um termo para vulnerabilidades que os invasores exploram para se passar por usuários legítimos online. O OWASP o classifica especificamente esse tipo de vulnerabilidade na sétima posição em seu *ranking* de 2021 [Owasp 2021].

Para o presente projeto, essa vulnerabilidade se encontram na implementação do JSON Web Token (JWT). Até a versão 4.2.2, a biblioteca jsonwebtoken apresentou uma vulnerabilidade de *Broken Authentication*, chamada *Algorithm Confusion*. JSON Web Token aceita criptografia simétrica e assimétrica. A vulnerabilidade surge quando uma aplicação não verifica se o algoritmo do *token* recebido corresponde ao algoritmo esperado, como ocorre com as versões anteriores a 4.2.2 no jsonwebtoken. Um invasor é capaz de gerar um token com base em uma chave pública recebida e configura seu algoritmo para HS256. Desta forma, a API fará uma verificação de chave simétrica utilizando a chave pública como segredo. Como o *token* criado pelo invasor foi assinado com a chave pública, a API considerará o invasor como autenticado.

A terceira vulnerabilidade inserida propositalmente na API é a *Sensitive Data Exposure*. Devido ao controle de acesso inadequado ao código-fonte da aplicação, desenvolvedores dos quais não se espera que vejam dados confidenciais podem, na verdade,

ser capazes de acessá-los. Infelizmente, uma prática comum entre os desenvolvedores é inserir logins e senhas de bancos de dados, microserviços, entre outros, diretamente no código fonte da aplicação. A Figura 2 exibe dados sensíveis expostos diretamente no código fonte da API do presente projeto. *Sensitive Data Exposure* é listado na segunda posição do *ranking* do OWASP 2021 [Owasp 2021].

```
const username = 'root'
const password = 'password123'
```

Figura 2. Credenciais de acesso ao Banco de Dados sendo expostos diretamente no código fonte da API.

Por fim, a última vulnerabilidade inserida propositalmente na API é a *Security Misconfiguration*. Os cabeçalhos de segurança HTTP fornecem uma camada extra de segurança, restringindo os comportamentos que o navegador e o servidor permitem quando uma aplicação está em execução. O OWASP lista a falta de cabeçalhos apropriados de segurança em seu *ranking* de 2021 na quinta posição [Owasp 2021].

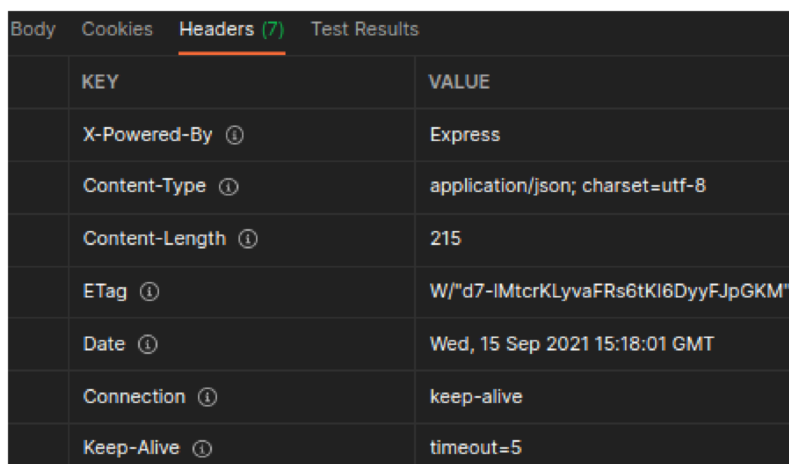

Figura 3. Cabeçalho recebido nas respostas da API.

A Figura 3 mostra o cabeçalho padrão recebido das respostas da API. Como se pode observar, o cabeçalho carece de parâmetros de segurança, tais como o *X-Content-Type-Options: nosniff*, que serve para evitar que navegadores interpretem o conteúdo recebido, evitando a execução de possíveis códigos maliciosos. A má configuração de cabeçalhos também resulta no vazamento de informações, como a presença do cabeçalho *X-Powered-By* que exibe informações sobre a tecnologia sendo usada pela API, no caso do presente estudo de caso, o *framework* Express.

4.2. Desenvolvimento do *pipeline* DevOps

A ferramenta escolhida para a construção do *pipeline* DevOps foi o Github Actions devido à sua simplicidade e natureza não opnativa, propiciando a construção de um *pipeline* genérico, proposto para este trabalho. No GitHub Actions, um *workflow* DevOps é um processo automatizado composto de *jobs*. Deve-se criar um arquivo YAML para definir a configuração do *workflow*. Outra configuração importante é a dos *secrets* de repositórios,

tornando dados sensíveis armazenados de maneira segura e tornado-os acessíveis durante a execução do *workflow*.

```
name: Node.js CI
on:
  push:
    branches: [ main ]
jobs:
  tests_unit:
    runs-on: ubuntu-latest
    steps:
    ...
    ...
  deploy-images:
    needs: tests-unit
    runs-on: ubuntu-latest
    steps:
    ...
    ...
  deploy-production:
    needs: deploy-images
    runs-on: ubuntu-latest
    steps:
    ...
    ...
```

Figura 4. *Workflow* DevOps simplificado

Como o arquivo YAML gerado para definir a configuração do workflow DevOps é relativamente grande, a Figura 4 apresenta uma versão resumida de sua estrutura, ocultando os códigos que são executados em cada job. Conforme Figura 4, a primeira linha informa o nome a ser dado ao *workflow*, nesse caso, *Node.js CI*. A palavra-chave *on* define os eventos do Github que acionam esse *workflow*. Nesse caso, o *workflow* será acionado para a operação de *push* na *branch* main.

A próxima fase será a definição dos *jobs*. *Jobs* são executados de forma independente (padrão) ou sequencial se o *job* atual depender do *job* anterior para ser bem-sucedido. *Jobs* contém a definição dos *steps*, que são sempre executados em um mesmo *runner*, sendo os responsáveis por definir as tarefas/comandos que serão executadas.

Em todos os *jobs* descritos na Figura 4 é definido que o *runner* executará a última versão do Ubuntu. O *job* deploy-images exibido na Figura 4 inicia com a palavra chave *needs*, indicando que este *job* só será iniciado caso o *job* de teste seja concluído com sucesso. De forma similar, o *job* deploy-production somente será iniciado caso o *job* anterior, deploy-images, seja concluído com sucesso.

Após a criação do *workflow* DevOps, foi realizada a execução do *pipeline* DevOps da aplicação contendo vulnerabilidades. A Figura 5 exibe um gráfico fornecido pelo GitHub Actions, no qual é possível visualizar as etapas do *workflow* DevOps, também chamado de *pipeline*. Pode-se verificar que as três etapas do *pipeline* foram executadas com sucesso. Desta forma, uma nova versão da API se torna disponível de forma automatizada em ambiente de produção.

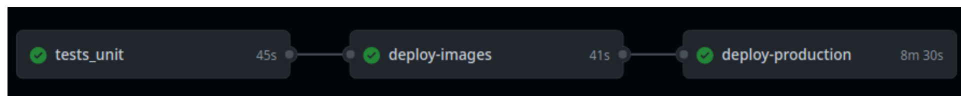

Figura 5. Resultado da execução do pipeline DevOps

Para fins de automatização da entrega de software, esse *workflow* DevOps cumpre seu objetivo. Entretanto, a aplicação que foi para o ambiente de produção possui inúmeras vulnerabilidades que não foram detectadas pelo *workflow* DevOps. Nesse instante, portanto, temos o *software* em execução com diversas brechas de segurança. Desta forma, o objetivo do passo a seguir é criar um *workflow* DevSecOps que seja capaz de detectar as vulnerabilidades da API e impedir o *deploy* da aplicação.

4.3. Desenvolvimento do *pipeline* DevSecOps

Essa etapa do estudo de caso se concentra na elaboração de um pipeline *DevSecOps* através da integração de ferramentas de segurança no *pipeline* construído na etapa anterior, que serão executados após a fase de testes unitários. Aqui também será analisado como o *pipeline* se comportará mediante tentativa de *deploy* de um *software* inseguro.

Foram realizadas buscas por ferramentas SAST e ferramentas DAST. As ferramentas SAST escolhidas foram Snyk CLI e Njsscan, pelas suas popularidades entre a comunidade e por funcionarem por linha de comando (CLI), tornando o o estudo de caso mais abrangente e independente do orquestrador do *pipeline*. A ferramenta DAST escolhida foi a Stackhawk, que é baseada na ferramenta OWASP ZAP, porém com mais recursos de integração em *pipelines*. Por não haver uma solução Stackhawk em CLI, optou-se pelo uso de uma *action* oficial fornecida pelos desenvolvedores da ferramenta.

Para o *pipeline* DevSecOps foram adicionados novos *jobs* de segurança ao código do *pipeline* DevOps explicado anteriormente. Também foi realizada uma mudança no *job* deploy-images (Figura 6), que agora dependerá dos *jobs* sast-njsscan, dast-stackhawk e sast-check_dependencies, que serão explicados a seguir. Ou seja, a construção da imagem da aplicação só ocorrerá se os *jobs* de segurança forem realizados com sucesso.

```
deploy-images:
  needs: [tests_unit, sast-njsscan, dast-stackhawk, sast-check_dependencies]
  runs-on: ubuntu-latest
```

Figura 6. Novas dependências para o *job* deploy-images

Para a integração da ferramenta Snyk CLI, o job sast-checkdependencies exibido na Figura 7 foi criado. Para execução em projetos Node.js, o Snyk CLI necessita de acesso ao diretório node_modules, onde são armazenados os pacotes baixados para o projeto. Dessa forma, optou-se por realizar a instalação do utilitário npm através da *action* setup-node, e com acesso ao arquivo package.json com a definição de todas as dependências do projeto, foi possível gerar o diretório node_modules por meio do comando ”npm install”, conforme Figura 7. O último *step* define a execução do Snyk CLI.

Para a integração da ferramenta njsscan, o *job* sast-njsscan exibido na Figura 7 foi criado. Pode-se perceber que é feito uso da *action* setup-python para a instalação da linguagem python na versão 3, pois a linguagem é uma dependência da ferramenta

```
sast-check_dependencies:
  needs: tests_unit
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v2
    - uses: actions/setup-node@v2
      with:
        node-version: '14'
    - run: npm install

    - name: Install and configure SNYK
      run: sudo npm install -g snyk && snyk auth ${{ secrets.SNYK_TOKEN }}

    - name: snyk scan for lib vulnerabilities (SAST)
      run: sudo snyk test .

sast-njsscan:
  needs: tests_unit
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v2

    - uses: actions/setup-python@v2
      with:
        python-version: '3.x'

    - name: njsscan (install dependencies)
      run: |
        python -m pip install --upgrade pip
        pip install requests_html njsscan

    - name: Execute njsscan
      run: njsscan .

dast-stackhawk:
  needs: tests_unit
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v2

    - name: create env file
      run: |
        touch .env
        echo APP_PRIVATE_KEY="${{ secrets.APP_PRIVATE_KEY }}" >> .env
        echo MONGO_INITDB_USER="${{ secrets.MONGO_INITDB_USER }}" >> .env
        echo MONGO_INITDB_PWD="${{ secrets.MONGO_INITDB_PWD }}" >> .env

    - name: Start containers
      run: docker-compose -p projeto -f "docker-compose.yml" up -d

    - name: Scan my app
      uses: stackhawk/hawkscan-action@v1.3.0
      with:
        apiKey: ${{ secrets.HAWK_API_KEY }}
        environmentVariables: HAWK_SCAN_USERNAME HAWK_APPLICATION_ID
            HAWK_SCAN_PASSWORD
      env:
        HAWK_SCAN_USERNAME: ${{ secrets.HAWK_SCAN_USERNAME }}
        HAWK_APPLICATION_ID: ${{ secrets.HAWK_APPLICATION_ID }}
        HAWK_SCAN_PASSWORD: ${{ secrets.HAWK_SCAN_PASSWORD }}
```

Figura 7. Jobs para integração das ferramentas de segurança Snyk CLI, njsscan e Stackhawk

njsscan. O *step* seguinte realiza a instalação da ferramenta njsscan, através do gerenciador de pacotes do python pip, assim como a instalação de sua dependência requests_html.

A fim de integrar a ferramenta DAST StackHawk, o *job* dast-stackhawk mostrado na Figura 7 foi inserido no *workflow*. Diferentemente de ferramentas SAST, no DAST é necessário ter aplicação em execução. Para tanto, são executados os *containers* da aplicação. Em seguida, é utilizada a *action* ”hawkscan-action”para instalação, configuração e execução do Stackhawk. A configuração do Stackhawk ocorreu por meio de dois arquivos: swagger.yml e stackhawk.yml. O arquivo swagger.yml contém a documentação da API no formato OpenAPI, permitindo ao HawkScan realizar uma varredura mais rápida e completa. O segundo arquivo necessário foi o stackhawk.yml, uma dependência do Stackhawk de onde são obtidas configurações para os *scans*.

Após a adição dos *jobs* de segurança ao *workflow* DevOps, transformando-o assim em um *workflow* DevSecOps, foi realizada a execução do *pipeline* DevSecOps da aplicação contendo vulnerabilidades. Conforme Figura 8 gerada pelo GitHub Actions, o *deploy* da API contendo vulnerabilidades não ocorre, sendo bloqueado pelas três ferramentas de segurança. Desta forma, a aplicação que está em produção não será substituída por essa nova versão até que as vulnerabilidades de segurança sejam corrigidas. Pode-se observar a seguir as saídas geradas por cada uma das ferramentas de segurança.

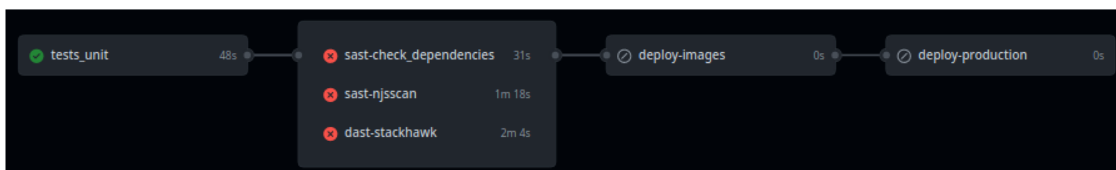

Figura 8. Resultado do *pipeline* DevSecOps

Conforme Figura 9, a ferramenta SAST Snyk CLI, dedicada à identificação de vulnerabilidades em dependências, identifica a vulnerabilidade *Algorithm Confusion* na biblioteca jsonwebtoken desatualizada. É interessante observar que, além de exibir a falha de segurança, a ferramenta também informa a possível solução para corrigir a vulnerabilidade, neste caso, atualizar o jsonwebtoken da versão 4.0.0 para a versão 5.0.0.

```
Tested 101 dependencies for known issues, found 3 issues, 4 vulnerable paths.


Issues to fix by upgrading:

  Upgrade jsonwebtoken@4.0.0 to jsonwebtoken@5.0.0 to fix
  ✗ Uninitialized Memory Exposure [High Severity][https://snyk.io/vuln/npm:base64url:20180511] in base64url@1.0.6
    introduced by jsonwebtoken@4.0.0 > jws@2.0.0 > base64url@1.0.6 and 1 other path(s)
  ✗ Forgeable Public/Private Tokens [High Severity][https://snyk.io/vuln/npm:jws:20160726] in jws@2.0.0
    introduced by jsonwebtoken@4.0.0 > jws@2.0.0
  ✗ Authentication Bypass [High Severity][https://snyk.io/vuln/npm:jsonwebtoken:20150331] in jsonwebtoken@4.0.0
    introduced by jsonwebtoken@4.0.0
```

Figura 9. Resultado do SNYK CLI para identificação de Algorithm Confusion

A ferramenta SAST njsscan foi capaz de identificar as vulnerabilidades *Sensitive Data Exposure* e *NoSQL Injection*. A Figura 10 exibe especificamente o log da vulnerabilidade *NoSQL Injection*. Pode-se observar que a ferramenta apresenta: (1) qual foi a vulnerabilidade encontrada; (2) em qual arquivo a vulnerabilidade foi encontrada (index.js); (3) qual o comando que gera a vulnerabilidade de segurança. Esses relatórios

detalhados pela ferramenta njsscan é de suma importância para a equipe de desenvolvimento na resolução das vulnerabilidades.

```
| RULE ID     | node_nosqli_injection
| OWASP       | A1: Injection
| CWE         | CWE-943: Improper Neutralization of Special Elements in Data Query Logic
| DESCRIPTION | Untrusted user input in findOne() function can result in NoSQL Injection.
| SEVERITY    | ERROR
| FILES       | | File            | index.js
|             | | Match Position  | 22 - 92
|             | | Line Number(s)  | 35
|             | | Match String    | const user = await Admin.findOne({'email': req.body.email,'password': req.body.password});
```

Figura 10. Resultado do njsscan para identificação de *NoSQL Injection*

A ferramenta DAST StackHawk foi capaz de identificar uma série de problemas de segurança causadas devido à carência de cabeçalhos de segurança (*Wildcard Directive*, *Policy Header Not Set*, *X-Content-Type-Options Header Missing* e *Server Leaks Information via "X-Powered-By"HTTP Response Header Field(s)*) nas respostas da API e uso de HTTP ao invés de HTTPS. A Figura 11 apresenta especificamente a parte da saída do StackHawk referente a presença do campo *"X-Powered-By"*, informando o nível de risco da vulnerabilidade e em quais *endpoints* ela ocorre.

```
6) Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)
   Risk: Low
   Cheatsheet:
   Paths (9):
     [New] GET /
     [New] POST /login?redirect_url=redirect_url
     [New] GET /participants
     [New] GET /speakers
     ... 4 more in details
View on StackHawk platform: https://app.stackhawk.com/scans/045f3955-82bf-492f-a0ef-a001294250a3
```

Figura 11. Parte do resultado do StackHawk para identificação de problemas no cabeçalho HTTP

4.4. Correção das vulnerabilidades da API

Esta etapa se concentra na correção das vulnerabilidades explanadas anteriormente.

A vulnerabilidade NoSQL *Injection* ocorre devido à falta de tratamento na entrada do usuário. Uma maneira simples e segura de realizar a higienização em entradas do usuário é por meio do uso de bibliotecas reconhecidas entre organizações e pela comunidade. A biblioteca escolhida para resolver a vulnerabilidade é a mongo-sanitize, voltada para a sanitização de entradas contra *query selector injection attacks*.

A vulnerabilidade *Broken Authentication* ocorre devido a versão defasada da biblioteca jsonwebtoken. A resolução da vulnerabilidade é realizada através da atualização da biblioteca para versão 8.5.1, última versão estável no momento da escrita desse artigo.

Como forma de impedir que dados sensíveis como nomes de usuário e senha sejam escritos diretamente em código, propiciando a vazamentos de credenciais a terceiros não

autorizados (*Sensitive Data Exposure*), a solução foi a adoção de variáveis de ambiente gerenciadas pelos *secrets* do GitHub Actions, onde durante a execução do *workflow* os valores serão obtidos e configurados no ambiente do *container* em tempo real.

Por fim, uma maneira simples de corrigir o problema de *Security Misconfiguration* é através da biblioteca helmet [Helmet 2022], uma solução popular para configurar automaticamente uma série de cabeçalhos de segurança nas respostas enviadas ao cliente, além de retirar cabeçalhos que podem expor informações valiosas sobre a aplicação.

4.5. Execução do *pipeline* DevSecOps com API sem vulnerabilidades

Esta etapa busca exibir o comportamento do *pipeline* DevSecOps com a nova versão da API. Como apresentado pela Figura 12, o código da API é validado pelas ferramentas de segurança e o pipeline DevSecOps, dessa vez, é executado com sucesso. Desta forma, uma versão nova e segura da API é colocada em produção de forma automatizada, substituindo a versão anterior.

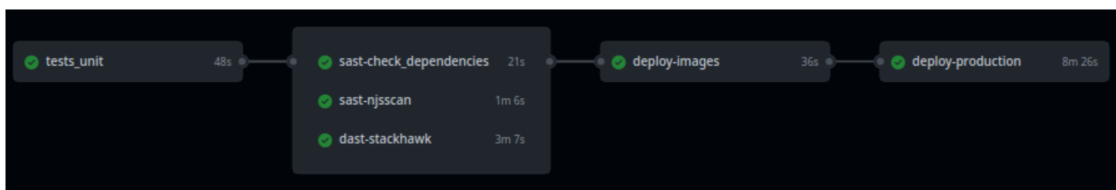

Figura 12. Resultado da execução do *pipeline* DevSecOps com API corrigida

Após o *deploy*, foram realizadas tentativas manuais de ataque visando a exploração das vulnerabilidades selecionadas, e os resultados mostraram que não é mais possível explorar as vulnerabilidades existentes no código anterior da API.

5. Considerações Finais

O presente trabalho teve como proposta demonstrar a integração da segurança contínua em *pipelines* DevOps. Para tanto, foi proposto um estudo de caso que contou com a construção de um *pipeline DevOps* em duas versões: a primeira voltada apenas para a automatização de testes de software e de *deploys*, e outra versão onde adicionalmente foram integradas ferramentas de segurança. Posteriormente, foi analisado o comportamento desses *pipelines* mediante a tentativa de *deploy* de uma API insegura.

Apesar do *pipeline* DevOps construído atender sua função de automatizar os processos necessários para levar uma nova versão da API à produção, ressalta-se que não houve nenhuma verificação quanto à segurança dessa nova versão. Dessa maneira, o presente estudo de caso demonstrou como um *pipeline* sem a devida integração com a segurança torna o ambiente de produção suscetível ao *deploy* de softwares inseguros.

De modo a contornar essa adversidade, foi construída uma nova versão do *pipeline*, um *pipeline* DevSecOps, onde foram integradas ferramentas de análise SAST e DAST, a fim de encontrar de maneira estática e dinâmica, respectivamente, problemas de segurança. Duas ferramentas SAST foram integradas ao *pipeline*, Snyk CLI e Njsscan, além da ferramenta DAST Stackhawk. Foi demonstrado nessa etapa do estudo de caso que com a devida integração das ferramentas de segurança o *pipeline* DevSecOps impossibilita o *deploy* para a produção, até que os problemas de segurança sejam corrigidos.

Por meio desse estudo de caso pôde comprovar como ferramentas de segurança integradas no *pipeline* de maneira contínua e automatizada foram capazes de identificar brechas de segurança de maneira ágil, mantendo o ritmo DevOps de entrega ágil e automatizada do software.

Referências

Ahmed, A. (2019). DevSecOps: Enabling security by design in rapid software development. Master's thesis.

Carter, K. (2017). Francois raynaud on DevSecOps. *IEEE Software*, 34(5):93–96.

CeArley, D., Burke, B., Searle, S., and Walker, M. J. (2016). Top 10 strategic technology trends for 2018. *The Top*, 10:1–246.

Danielecki, D. M. (2019). Security first approach in development of single-page application based on angular. Master's thesis, University of Twente.

Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). DevOps. *Ieee Software*, 33(3):94–100.

Enterprise, H. P. (2016). Application security and DevOps. Technical report, Technical report, Hewlett Packard Enterprise.

Freitas, F. D. S. (2020). Application security in continuous delivery.

Handova, D. (2020). How does IAST fit into DevSecOps? https://www.synopsys.com/blogs/software-security/iast-devsecops-appsec-program/. (Accessed on 31/10/2021).

Helmet (2022). helmet - npm. https://www.npmjs.com/package/helmet. (Accessed on 05/28/2022).

Jetbrains (2021). O que é DevSecOps e qual seu papel no CD? — guia de ci/cd do teamcity. https://www.jetbrains.com/pt-br/teamcity/ci-cd-guide/what-is-devsecops/. (Accessed on 04/23/2021).

Koskinen, A. (2019). DevSecOps: building security into the core of devops.

Leite, L., Rocha, C., Kon, F., Milojicic, D., and Meirelles, P. (2019). A survey of DevOps concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35.

Microfocus (2022). What is DevSecOps? https://www.microfocus.com/en-us/what-is/devsecops. (Accessed on 05/28/2022).

Myrbakken, H. and Colomo-Palacios, R. (2017). DevSecOps: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination*, pages 17–29. Springer.

Owasp (2021). Introdução ao OWASP Top 10 2021. https://owasp.org/Top10/pt_BR/. (Accessed on 10/13/2021).

Radware (2020). Radware research: The state of WEB application and API protection. Technical report.

Simpson, G. B. (2018). CI/CD software security automation. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).