**FACULDADE DE TECNOLOGIA DE AMERICANA "Ministro Ralph Biasi"**
**Curso Superior de Tecnologia em Segurança da Informação**

Felipe Antonio Santos da Silva

# ANÁLISE DE RESULTADOS DE FERRAMENTAS SAST EM PIPELINES CI

**Americana, SP**

**2025**

**FACULDADE DE TECNOLOGIA DE AMERICANA "Ministro Ralph Biasi"**
**Curso Superior de Tecnologia em Segurança da Informação**

Felipe Antonio Santos da Silva

# ANÁLISE DE RESULTADOS DE FERRAMENTAS SAST EM PIPELINES CI

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação sob a orientação do Prof. Esp. José William Pinto Gomes

Área de concentração: Segurança da Informação.

**Americana, SP**

**2025**

**FICHA CATALOGRÁFICA – Biblioteca Fatec Americana Ministro Ralph Biasi- CEETEPS Dados Internacionais de Catalogação-na-fonte**

SILVA, Felipe Antonio Santos da

Analise de resultados de ferramenta SAST em pipeline CI. / Felipe Antonio Santos da Silva – Americana, 2025.

54f.

Monografia (Curso Superior de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana Ministro Ralph Biasi – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Esp. José William Pinto Gomes

1. Algoritmos 2. Python – linguagem de programação 3. Segurança em sistemas de informação. I. SILVA, Felipe Antonio Santos da II. GOMES, José William Pinto III. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana Ministro Ralph Biasi

CDU: 501.5
681.3.061Python
681.518.5

Elaborada pelo autor por meio de sistema automático gerador de ficha catalográfica da Fatec de Americana Ministro Ralph Biasi.

**Felipe Antonio Santos da Silva**

**Análise de resultados de ferramentas sast em pipelines ci**

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Curso Superior de Tecnologia em Segurança da Informação pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana Ministro Ralph Biasi.
Área de concentração: Segurança da informação.

Americana, 02 de dezembro de 2025.

**Banca Examinadora:**

José William Pinto Gomes
Especialista
Fatec Americana "Ministro Ralph Biasi"

Lidia Regina de Carvalho Freitas Barban
Especialista
Fatec Americana "Ministro Ralph Biasi"

Thiago da Silva Vieira
Mestre
Fatec Americana "Ministro Ralph Biasi"

## RESUMO

Este trabalho propõe uma análise comparativa da eficiência, cobertura e precisão de três ferramentas de Análise Estática de Código (SAST): Semgrep, Bandit e CodeQL. As ferramentas foram integradas a pipelines de Integração Contínua (CI) utilizando a plataforma GitHub *Actions*, com o objetivo de detectar vulnerabilidades em um repositório Python propositalmente vulnerável. A metodologia empregou um ambiente controlado com *workflow* automatizado, executando as ferramentas sobre o mesmo código-fonte. A análise comparativa considerou a severidade e tipo de falha detectada, além da taxa de falsos positivos. Os resultados evidenciaram que o Bandit se mostrou eficaz para verificações rápidas e diretas; o Semgrep proporcionou ampla cobertura contextual; e o CodeQL alcançou a maior profundidade analítica ao rastrear fluxos de dados, resultando em menor incidência de falsos positivos. Conclui-se que a combinação dessas ferramentas amplia significativamente a eficácia das análises de segurança, reforçando o conceito de DevSecOps e promovendo a integração contínua da segurança ao ciclo de desenvolvimento de *software*.



**Palavras-chave:** Teste de segurança, DevSecOps, Integração Contínua.

## ABSTRACT

*This study presents a comparative analysis of three Static Application Security Testing (SAST) tools Semgrep, Bandit, and CodeQL integrated into Continuous Integration (CI) pipelines using GitHub Actions. The goal was to evaluate their efficiency, coverage, and accuracy in detecting vulnerabilities within a purposely vulnerable Python source code. An experimental approach was adopted, executing each tool on the same repository and comparing results regarding severity, types of detections, and false-positive rates. The results demonstrated that Bandit excels in fast and direct detections, Semgrep offers broader contextual coverage, and CodeQL provides deeper semantic analysis through data flow tracking and reduced false positives. The study concludes that combining multiple SAST tools enhances detection precision and reinforces DevSecOps practices by embedding security consistently throughout the software development lifecycle.*



***Keywords:*** *Security Testing; DevSecOps; Continuous Integration.*

## LISTA DE ILUSTRAÇÕES

## LISTA DE QUADROS

**SUMÁRIO**

# INTRODUÇÃO

Conforme o tempo passa, cada vez mais o ser humano se torna dependente da tecnologia e, com ela, vêm também os seus riscos. A todo momento, as pessoas estão expostas, seja navegando na internet ou utilizando um aplicativo, por exemplo, vivendo à mercê de que as empresas cuidem devidamente da segurança dos dados de suas aplicações.

Uma área que vem ganhando cada vez mais espaço no âmbito tecnológico e de desenvolvimento de sistemas é *Development Security and Operations* (DevSecOps), setor que enfatiza o desenvolvimento seguro e a importância de olhar para a infraestrutura e segurança de uma aplicação desde o início de sua construção até o momento em que ela é publicada, podendo usufruir de metodologias como *Continuous Integration* (CI).

A segurança cibernética tem se mostrado uma necessidade central para organizações ao redor do mundo, o quão complexas e escaláveis as aplicações estão se tornando. Consequentemente, técnicas de ataques cada vez mais sofisticadas surgem com o objetivo de explorar vulnerabilidades, podendo causar enormes impactos financeiros, de reputação e influenciar diretamente na privacidade de dados. No cenário atual de desenvolvimento, em que práticas de CI são amplamente utilizadas pelas equipes para agilizar o processo de integração de soluções, é fundamental garantir a segurança durante toda a pipeline.

Este trabalho, portanto, se justifica pela necessidade constante e crescente de avaliar as ferramentas de segurança dentro dessas pipelines. Mostrar as principais diferenças quanto aos resultados obtidos por ferramentas *Static Application Security Testing* (SAST), por exemplo, é essencial para compreender a efetividade da segurança no desenvolvimento de sistemas, contribuindo para a melhoria dos processos e da efetividade das operações, visto que a utilização desses meios de testes de segurança é fortemente recomendada por padrões de segurança como o *Open Web Application Security Project* (OWASP).

No contexto da cibersegurança, é fundamental analisar o desenvolvimento das aplicações e garantir que elas sejam devidamente testadas antes de serem disponibilizadas para clientes. Seguindo essa ideia, tem-se como problema de

pesquisa: quais seriam as possíveis diferenças nos resultados obtidos por ferramentas de teste de software estático?

Tem-se como objetivo geral, neste trabalho, analisar os resultados obtidos através da integração de ferramentas SAST em *pipelines* CI, com o intuito de melhorar a segurança no setor de desenvolvimento de sistemas, otimizando a detecção antecipada de vulnerabilidades e problemas nas aplicações, a fim de promover práticas de desenvolvimento seguro, bem como a integração contínua.

Dito isso, pode-se levantar algumas hipóteses relacionadas ao uso das ferramentas SAST. Elas influenciam positivamente o desenvolvimento de sistemas, pois, com elas, é possível manter um controle melhor sobre vulnerabilidades e erros dentro do ambiente de desenvolvimento, a incidência desses riscos e podendo melhorar significativamente a qualidade do código desenvolvido, garantindo assim o desenvolvimento eficaz da aplicação, com ênfase na segurança e na agilidade da metodologia CI.

O percurso metodológico deste trabalho foi realizar um estudo de caso que permitiu a análise prática das ferramentas SAST em um ambiente de pipeline CI. Os dados foram coletados por meio da execução e configuração destas ferramentas no pipeline que simularam também o processo de integração contínua. A partir dos resultados observados nas simulações, foi realizado uma análise comparativa, onde pôde ser levado em consideração a precisão e capacidade de identificação de vulnerabilidades. A escolha desse método justifica-se pela necessidade de um ambiente controlado que permita observar diretamente o desempenho das ferramentas no contexto de automação CI.

# 1 FUNDAMENTAÇÃO TEÓRICA

## 1.1 *Continuous Integration*

Segundo Shahin, Babar e Zhu (2017), a Integração Contínua (CI) é uma prática do desenvolvimento de software na qual os membros de uma equipe integram frequentemente seus códigos em um mesmo repositório, podendo realizar múltiplas integrações ao longo de um único dia. Essa metodologia contribui para um desenvolvimento mais rápido, com maior qualidade, aumentando a produtividade das equipes, uma vez que está fortemente associada à execução de testes automatizados.

## 1.2 Benefícios da utilização de CI

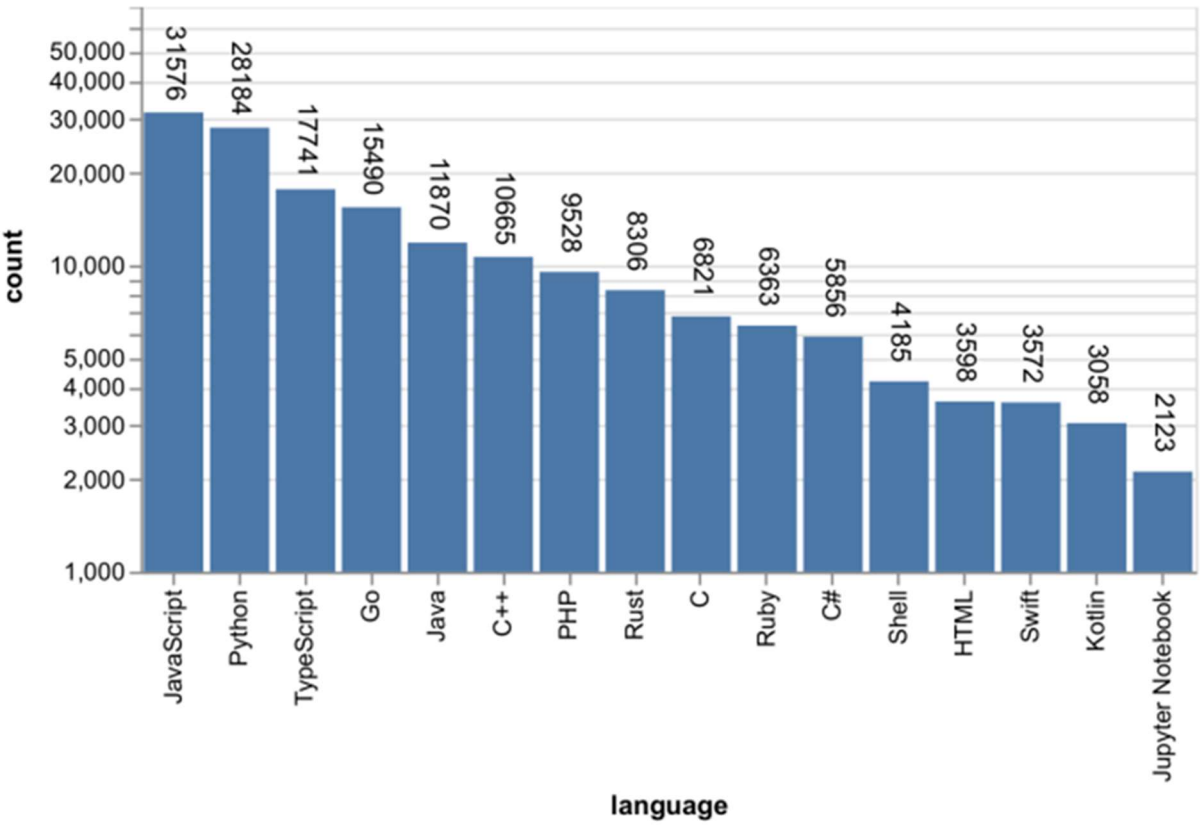
Jani (2023), traz uma lista de benefícios que devem ser considerados ao decidir entre utilizar ou não a metodologia CI.

1. **Tempo de lançamento no mercado:** Realizar a automatização de tarefas como *Build*, testes, processos de implantação e otimização do desenvolvimento de novas soluções e atualizações, consequentemente, permite que as empresas atendam as rápidas mudanças do mercado.
2. **Qualidade Melhorada:** Tarefas automatizadas e integração contínua de novos códigos no ambiente de desenvolvimento, permitem a detecção e resolução precoce de problemas, garantindo uma melhor qualidade e confiança no sistema.
3. **Redução de riscos:** A implantação contínua permite mudanças mais ágeis, com isso, consegue-se reduzir os riscos em grandes lançamentos, onde nestes casos, a complexidade da aplicação se torna muito maior.

Conforme a pesquisa feita por Cunha, Gião, Pereira e Flores (2024), onde foram analisados dados de criação de repositórios no GitHub entre 2012 e 2023, realiza-se a análise de 612.557 repositórios, destes, 200.023, ou seja, 32.7% possuem a metodologia CI integrada. Os autores também complementam dizendo sobre a

usabilidade desta metodologia integrado a diferentes linguagens de programação, onde mostra um cenário vasto de oportunidades de uso, sendo que Python e JavaScript são as linguagens que aparecem no topo do gráfico.

**Figura 1** - Contagem de repositórios por linguagem de programação.




**Fonte:** Gião et al. (2024).

## 1.3 *Static Application Security Testing* (SAST)

> "O teste de segurança é conhecido como um processo destinado a revelar falhas nos mecanismos de segurança de um sistema de informação que protegem os dados e mantêm a funcionalidade conforme o esperado. Existem dois tipos principais de teste de segurança: o teste estático e o teste dinâmico. O SAST utiliza uma ferramenta de análise de código estático para examinar o código-fonte e identificar possíveis vulnerabilidades ou falhas de software" (Nguyen-Duc et al, 2021, p. 3).

A respeito de técnicas comuns utilizadas por ferramentas SAST, os autores complementam dizendo:

1- Análise sintática, como chamadas a funções de API inseguras ou uso de opções de configuração inseguras. Um exemplo dessa categoria seria a

análise de programas em Java que chamam java.util.random (que não fornece um gerador de números aleatórios criptográficos seguro).

2- Análise semântica, que exige o entendimento das semânticas do programa, como o fluxo de dados ou o controle de fluxo de um programa. Essa análise começa representando o código-fonte por meio de um modelo abstrato (por exemplo, grafo de chamadas, grafo de controle de fluxo ou diagrama de classe/sequência UML). Um exemplo dessa categoria seria uma verificação de fluxos diretos de dados de uma entrada de formulário de usuário para uma instrução SQL (indicando uma vulnerabilidade potencial de injeção de SQL).

## 1.4 Benefícios da implantação de ferramentas SAST

Nutalapati (2023), traz uma série de benefícios a respeito da implantação desse tipo de ferramentas, a seguir:

1- Eficiência e velocidade
   a. Como base dos benefícios estão a eficiência e velocidade da SAST, que possui a capacidade de executar diversos testes de forma rápida, reduzindo significativamente o tempo de descoberta de falhas no código, prevenindo vulnerabilidades e aumentando, de modo geral, a segurança no ambiente da aplicação.

2- Cobertura abrangente
   a. As ferramentas de teste de segurança automatizado cobrem uma ampla gama de cenários e vulnerabilidades de segurança, permitindo testes sistemáticos contra problemas como XSS, injeção SQL, armazenamento inseguro de dados e controles de acesso inadequados. Isso garante uma avaliação completa da segurança do aplicativo, reduzindo a chance de falhas críticas passarem despercebidas.

3- Consistência e confiabilidade
   a. Automatizados, esses testes oferecem resultados padronizados e reproduzíveis, evitando os erros humanos dos testes manuais e garantindo avaliações de vulnerabilidades confiáveis ao longo do desenvolvimento.

4- Escalabilidade
    a. Essas ferramentas são escaláveis, suportando testes extensivos e repetitivos conforme a complexidade e o tamanho das aplicações crescem, particularmente em ambientes de CI.
5- Detecção precoce de vulnerabilidade
    a. Ao integrar testes de segurança desde o início do desenvolvimento, é possível identificar e corrigir vulnerabilidades antes do lançamento, reduzindo riscos de exploração e facilitando a correção.
6- Custo-benefício
    a. Embora requeira um investimento inicial, o teste automatizado reduz a necessidade de extensos testes manuais e os custos de incidentes de segurança, como esforços de remediação e danos reputacionais.
7- Integração com processos de desenvolvimento
    a. Esses testes podem ser integrados aos fluxos de trabalho de desenvolvimento, especialmente em *pipelines* de CI, assegurando uma avaliação contínua da segurança em tempo real.

## 1.5 Semgrep

Com base na documentação da ferramenta, Semgrep (2025), Semgrep Code é um mecanismo de análise estática de códigos com uma vasta gama de linguagens de programação possíveis de se utilizar e tem a capacidade de detecção tanto de falhas de segurança quanto a aplicação de regras customizadas de estilo e vulnerabilidade, caso necessário. O site da ferramenta também deixa de forma muito explícita dizendo "*You can use* Semgrep Code *to scan local repositories or integrate it into your* CI/CD *pipeline to automate the continuous scanning of your code*" (SEMGREP, 2025), ou seja, ressaltando a capacidade e um ponto forte, que é justamente a integração com *pipelines* CI para a melhora de processos de segurança durante o desenvolvimento de software.

Para a linguagem Python, a documentação de Semgrep (2025), traz uma vasta capacidade de detecções, como por exemplo, a Figura 2 mostra os *frameworks* disponíveis para que Semgrep Code consiga realizar *scans.*

**Figura 2** - Contagem de repositórios por linguagem de programação.

| Framework / library | Category |
|---|---|
| Django | Web framework |
| Flask | Web framework |
| FastAPI | Web framework |

**Fonte:** Documentação Semgrep. (2025).

A ferramenta também apresenta constante evolução no contexto de melhorias para segurança quando diz: “Semgrep's *benchmarking process involves scanning open source repositories, triaging the findings, and making iterative rule updates. This process was developed and is used internally by the* Semgrep *security research team to monitor and improve rule performance*.” (Semgrep, 2025) ou seja, as equipes de desenvolvimento de Semgrep buscam realizar *scans* de forma periódica em repositórios *open source* a fim de melhorar os padrões de regras do Semgrep. Na Figura 3 é apresentada uma tabela disponibilizada pela desenvolvedora do Semgrep referente à última execução dos testes de melhoria da ferramenta.

**Figura 3** - Contagem de repositórios por linguagem de programação.

| | |
|---|---|
| Benchmark true positive rate (before AI processing) for latest ruleset | **84%** |
| Lines of code scanned | **~20 million** |
| Repositories scanned | **192** |
| Findings triaged to date | **~1000** |

**Fonte:** Documentação Semgrep. (2025).

## 1.6 Bandit

De acordo com a documentação oficial do Bandit (2024), diz que é uma ferramenta desenvolvida com o intuito de identificar problemas e falhas em códigos

Python por meio da análise da árvore sintática abstrata (AST) utilizando plugins que analisam o código com base em padrão conhecidos de segurança da informação e geram relatórios detalhados após as verificações. A documentação mostra que, a ferramenta é capaz de ser configurada via arquivos de configuração como YAML, e integrado em ferramentas de *pipelines* CI, como o Github Actions.

O Bandit também suporta a configuração de *plugins* de teste de forma individual, nos quais o usuário pode ajustar parâmetros internos de cada verificação. Essa abordagem justamente implementada nestes arquivos de configuração em formato YAML, possibilita alterar o comportamento de regras específicas, como o tratamento de chamadas a comandos do sistema operacional (os.system, entre outros) permitindo ajustar o nível de rigor da análise conforme a necessidade de verificação por parte do usuário.

A documentação da ferramenta Bandit (2024) também traz explicações sobre a capacidade de integrações possíveis em ambientes de *pipelines* CI. Onde também é explicitado com um exemplo de código, a possibilidade do uso integrado ao Github Actions, como mostra a Figura 4:

**Figura 4** - Exemplo de configuração Bandit no GitHub Actions.

## Example YAML Code for GitHub Actions Pipeline

Below is an example configuration for the GitHub Actions pipeline:

```yaml
name: Bandit

on:
    workflow_dispatch:

jobs:
    analyze:
        runs-on: ubuntu-latest
        permissions:
            # Required for all workflows
            security-events: write
            # Only required for workflows in private repositories
            actions: read
            contents: read
        steps:
            - name: Perform Bandit Analysis
              uses: PyCQA/bandit-action@v1
```

**Fonte: D**ocumentação oficial do Bandit (2025).

Para os resultados, a documentação Bandit (2024) traz uma ampla gama de formatos possíveis para se exportar relatórios do Bandit após análises. Nesta lista estão presentes os formatos csv, html, json, sarif, screen, text, xml, yaml.

## 1.7 CodeQL

A documentação CodeQL (2025), traz informações dizendo a respeito o foco da ferramenta é ajudar desenvolvedores a automatização checagens de segurança e integrar isto a seus respectivos *workflows* de desenvolvimento. É também uma ferramenta altamente vasta quando se diz a quais linguagens ela pode abranger, por exemplo JavaScript, C, C++, C#, Java e claro, Python. Sob cada linguagem, é bem importante também notar a capacidade da ferramenta de ser aplicada *frameworks*, que no caso do python nota-se uma lista com mais de 30 bibliotecas, mas por exemplo dentre os mais conhecidos destacam-se Django, FastAPI, Flask, Pycurl, requests etc.

CodeQL funciona com base em *queries* que são utilizadas para encontrar problemas em códigos fontes, problemas estes que podem estar associados a vulnerabilidades e falhas de segurança. De acordo com a documentação oficial da CodeQL (2025), cada *query* é responsável por identificar padrões de vulnerabilidades, falhas lógicas ou comportamentos suspeitos em um determinado trecho de código. As consultas são divididas em dois tipos principais: *alert queries* e *path queries*. As primeiras servem para destacar trechos de código que apresentam problemas pontuais, enquanto as segundas descrevem o fluxo de dados entre uma origem (*source*) e um destino (*sink*), permitindo detectar vulnerabilidades de fluxo, como SQL Injection e Cross-Site Scripting (XSS).

Para as linguagens, como mostra a Figura 5, CodeQl possui várias das mais famosas linguagens disponíveis para uso.

**Figura 5** -Linguagens disponiveis CodeQl.


**Fonte:** Documentação oficial do CodeQL (2025).

# 2 DESENVOLVIMENTO

Nesta etapa, será contextualizado o formato do desenvolvimento prático deste trabalho. Será utilizada a ferramenta GitHub para realizar a configuração de um repositório e através da ferramenta nativa do GitHub, chamada GitHub Actions será construído e configurado o *workflow* CI.

Utilizou-se um único repositório Git com *branch* principal *main,* assegurando que Bandit, Semgrep e CodeQL analisassem sempre o mesmo *snapshot* de código.

A partir deste repositório será realizado *push requests* para ativar os *workflows.* Importante destacar que, propositalmente a fim de testes, estas *push requests* terá códigos altamente vulneráveis que foram desenvolvidos na linguagem python.

## 2.1 Estrutura do *workflow*

Para a condução dos experimentos, foi definido um workflow dentro do diretório .github/workflows/ do repositório, utilizando a ferramenta GitHub Actions para automação das execuções.

Denominado sast_all.yml, o workflow foi estruturado com o propósito de simular um ambiente real de integração contínua (CI) em um contexto DevSecOps, no qual múltiplas ferramentas de análise de segurança são executadas de forma automatizada a cada alteração no código-fonte. Esse *workflow* é acionado automaticamente por eventos de *push request* na *branch* principal, e executa em sequência as ferramentas Bandit, Semgrep e CodeQL. Ao término da execução, os resultados são consolidados em artefatos (arquivos JSON e TXT). Tal abordagem representa o cenário de uma pipeline corporativa, onde diferentes testes de segurança operam simultaneamente, permitindo avaliar a integração prática dessas ferramentas no ciclo de desenvolvimento.

## 2.2 Configuração Inicial do Workflow.

1. **Nome e Triggers**: O *workflow* é denominado "SAST - CodeQL + Semgrep + Bandit" e é acionado automaticamente em três situações: quando há um *push*

para a *branch main*, quando uma *pull request* é aberta para a *branch main*, ou manualmente através do workflow_dispatch.

2. **Permissões**: O *workflow* define permissões específicas de leitura para conteúdo e ações do repositório, além de permissão de escrita para eventos de segurança, essencial para o registro de vulnerabilidades identificadas.
3. **Controle de Concorrência**: Implementa um mecanismo que garante que apenas uma execução do *workflow* ocorra por vez para cada referência (*branch*), cancelando execuções anteriores ainda em andamento quando uma nova é iniciada.

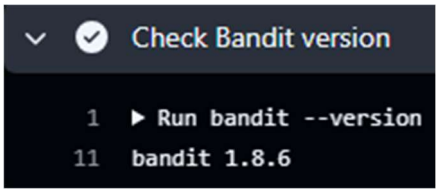
## 2.3 Configuração do Ambiente de Execução

4. **Ambiente do Job**: O *job* é executado em um ambiente Ubuntu na versão mais recente, com um tempo limite de 30 minutos para evitar execuções indefinidas que possam consumir recursos desnecessariamente.
5. **Checkout do Código**: Utiliza a *action* checkout@v4 para clonar o repositório e disponibilizar o código-fonte para análise pelas ferramentas de segurança.
6. **Configuração do Python**: Instala o Python na versão 3.11, estabelecendo o ambiente necessário para execução das ferramentas de análise estática que serão utilizadas.
7. **Criação do Diretório de Relatórios**: Cria uma pasta dedicada chamada "*reports*" onde todos os relatórios gerados pelas diferentes ferramentas serão armazenados de forma organizada.

## 2.4 Análise com Bandit.

8. **Instalação e Verificação**: O Bandit, ferramenta especializada em identificar problemas de segurança comuns em código Python, é instalado via pip e sua versão é verificada para garantir a instalação correta.

**Figura 6** - Versão utilizada do Bandit.



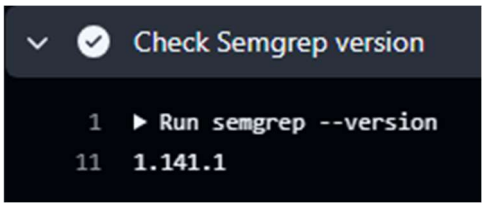

**Fonte:** GitHub Action (2025).

9. **Execução da Análise**: Executa o Bandit recursivamente em todo o diretório do projeto, gerando dois formatos de relatório: JSON para processamento automatizado e TXT para leitura humana. O comando utiliza "|| true" para garantir que falhas não interrompam o workflow.

## 2.5 Análise com Semgrep

10. **Instalação e Verificação**: Instala o Semgrep junto com a ferramenta já para processamento de JSON. O Semgrep é uma ferramenta de análise estática que utiliza padrões customizáveis para identificar vulnerabilidades.

**Figura 7** - Versão utilizada do Semgrep.




**Fonte:** GitHub Action (2025).

11. **Execução com Múltiplas Configurações**: Executa o Semgrep utilizando três conjuntos de regras: regras específicas para Python (p/python), auditoria de segurança (p/security-audit) e as dez principais vulnerabilidades da OWASP (p/owasp-top-ten).
12. **Processamento dos Resultados**: Gera um relatório inicial em formato JSON e, em seguida, utiliza o jq para extrair e formatar as informações relevantes em um arquivo TXT legível, incluindo ID da verificação, severidade, localização e mensagem.

## 2.6 Análise com CodeQL

13. **Inicialização**: Utiliza a *action* oficial do GitHub para inicializar o CodeQL, especificando Python como linguagem alvo. O CodeQL é uma ferramenta avançada de análise semântica de código desenvolvida pelo GitHub.

14. **Verificação de Metadados**: Executa um comando para exibir a versão da *action* do CodeQL utilizada, auxiliando na rastreabilidade e resolução de possíveis problemas.
15. **Execução da Análise**: Realiza a análise completa do código utilizando o CodeQL, configurado para não fazer *upload* automático dos resultados e armazenar a saída localmente no diretório de relatórios.
16. **Conversão de Formatos**: Converte o arquivo SARIF (formato padrão de saída do CodeQL) para JSON e extrai informações em formato TXT, processando o ID da regra violada e a mensagem correspondente para cada vulnerabilidade identificada.

## 2.7 Armazenamento dos Artefatos

17. **Upload dos Relatórios**: Utiliza a *action* upload-artifact@v4 para armazenar todos os relatórios gerados no workflow. A condição "if: always()" garante que os artefatos sejam salvos mesmo se etapas anteriores falharem.
18. **Configuração de Retenção**: Define que os artefatos serão mantidos por 7 dias, permitindo análise posterior dos resultados enquanto gerencia o espaço de armazenamento de forma eficiente. O artefato é nomeado "sast-reports" e inclui todo o conteúdo do diretório *reports*.

## 2.8 Geração do código vulnerável e metodologia de teste.

Para a execução prática deste trabalho, foi desenvolvido um conjunto de códigos propositalmente vulneráveis, com o objetivo de testar a capacidade de detecção das ferramentas SAST integradas à *pipeline*. A linguagem escolhida foi Python, por ser amplamente suportada por todas as ferramentas utilizadas (Bandit, Semgrep e CodeQL) e possuir um ecossistema consolidado de bibliotecas e práticas de segurança documentadas.

O código vulnerável foi criado de forma controlada, buscando representar vulnerabilidades reais encontradas no contexto de desenvolvimento de aplicações. A tabela 1 mostrará todas as vulnerabilidades selecionadas e as respectivas identificadores de acordo com OWASP e CWE.

**Quadro 1** - Mapeamento das vulnerabilidades segundo OWASP 2021 e CWE.

| Vulnerabilidades | OWASP 2021 | CWE |
|---|---|---|
| Hardcoded Credentials | A02:2021 Sensitive Data Exposure | CWE-798 |
| SQL Injection (Query por concatenação) | A03:2021 Injection | CWE-89 |
| Command Injection | A03:2021 Injection | CWE-78 |
| Path traversal | A05:2021 Security Misconfiguration | CWE-22 |
| Insecure Deserialization | A05:2021 Security Misconfiguration | CWE-502 |
| Uso de eval() - RCE | A03:2021 Injection | CWE-94 |
| Criptografia fraca | A02:2021 Sensitive Data Exposure | CWE-327 |
| Sem validação de input | A01:2021 Input Validation | CWE-20 |

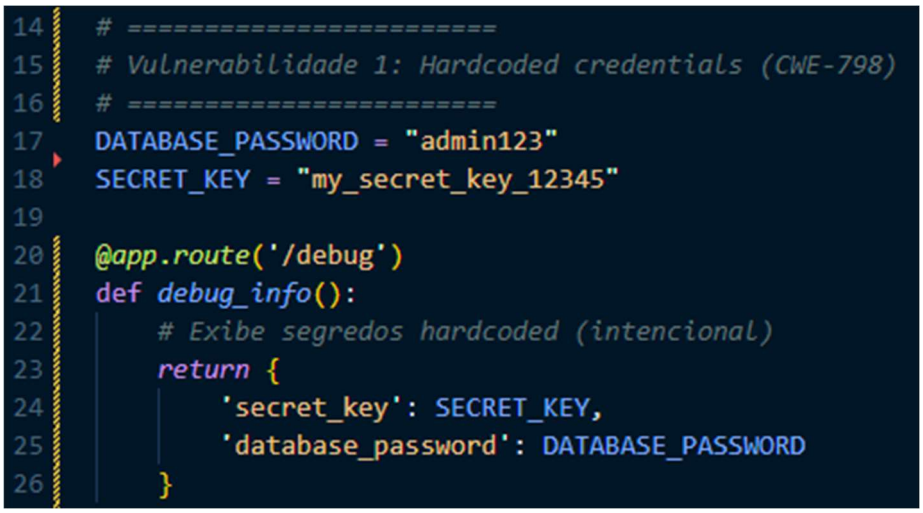
**Fonte:** Adaptado de OWASP (2021) e MITRE CWE (2024)

### 2.8.1 Armazenamento de senhas em texto plano.

Manter credenciais diretamente no código-fonte expõe informações sensíveis e facilita o vazamento de segredos em repositórios públicos, *logs* ou *pipelines*. A OWASP (2025), explica que essa vulnerabilidade pode comprometer ambientes de produção caso o código seja compartilhado ou comprometido.

De acordo com a OWASP (2025), credenciais nunca devem ser armazenadas em código-fonte. Recomenda-se utilizar *secret managers* (como AWS Secrets Manager, Hashicorp Vault ou GitHub Actions Secrets) e aplicar rotação periódica de segredos.

**Figura 8** - Credenciais Hardcoded.



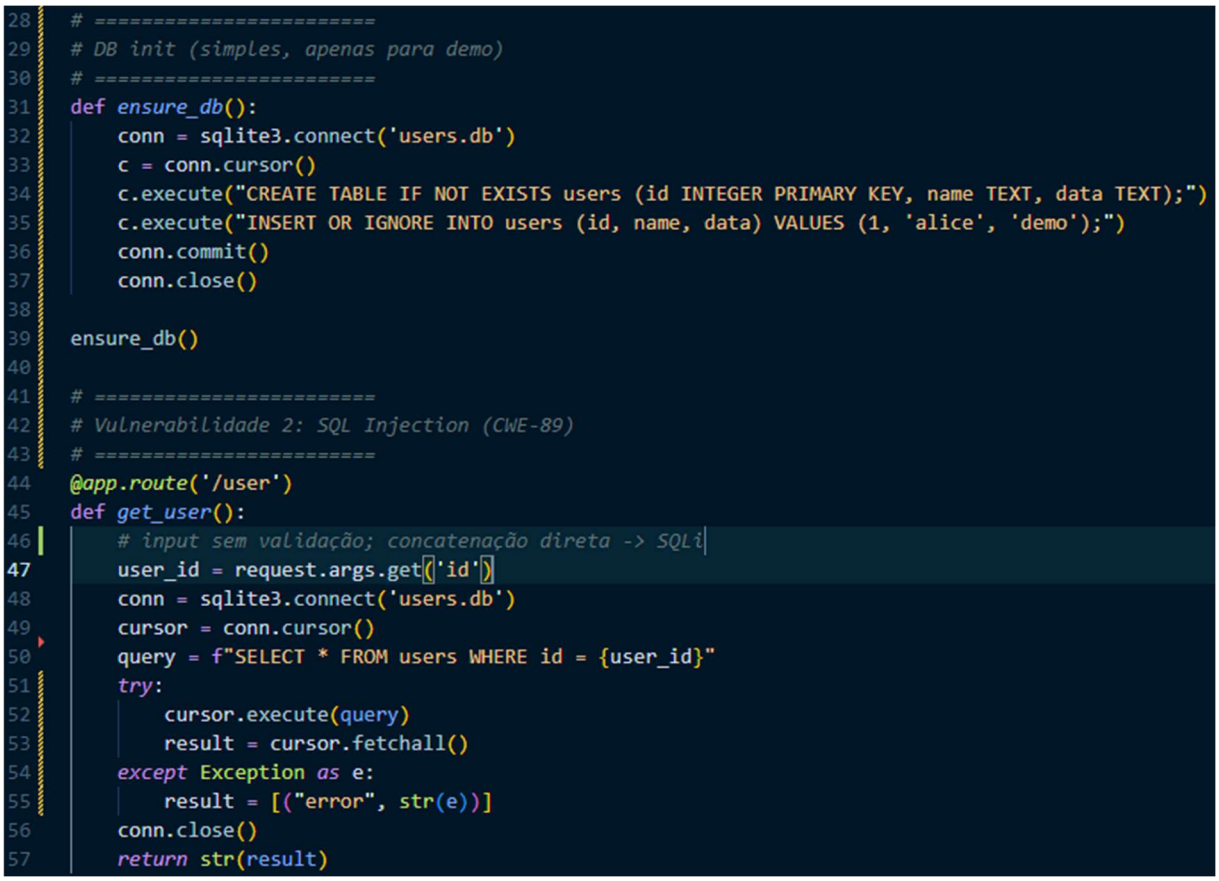

**Fonte:** Desenvolvido pelo autor (2025).

### 2.8.2 Injeção SQL

A OWASP (2025) explica que a vulnerabilidade de SQL *Injection* ocorre quando comandos SQL são construídos dinamicamente a partir de entradas do usuário sem qualquer validação ou parametrização, permitindo que um atacante modifique a consulta original e execute comandos arbitrários no banco de dados. No código utilizado neste estudo, o *endpoint* /user constrói a *query* por concatenação direta ("SELECT * FROM users WHERE id = {user_id}") e o *endpoint* /update atualiza registros concatenando user_input, demonstrando cenários clássicos de injeção (OWASP A03:2021 — *Injection*; CWE-89). O impacto inclui vazamento massivo de dados, alteração ou exclusão de registros, elevação de privilégios e possível tomada completa do servidor de banco de dados.

Como mitigação, a OWASP (2025) recomenda o uso de *prepared statements*/*queries* parametrizadas (por exemplo, cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))), validação estrita de tipos/formatos de entrada (ex.: aceitar somente inteiros para id) e políticas de mínimos privilégios no usuário do banco. Essas medidas reduzem drasticamente a superfície de ataque e permitem atribuir com precisão qualquer achado da ferramenta SAST ao trecho vulnerável.

**Figura 9** - Injeção SQL.



```python
# ======================
# DB init (simples, apenas para demo)
# ======================
def ensure_db():
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, data TEXT);")
    c.execute("INSERT OR IGNORE INTO users (id, name, data) VALUES (1, 'alice', 'demo');")
    conn.commit()
    conn.close()

ensure_db()

# ======================
# Vulnerabilidade 2: SQL Injection (CWE-89)
# ======================
@app.route('/user')
def get_user():
    # input sem validação; concatenação direta -> SQLi
    user_id = request.args.get('id')
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE id = {user_id}"
    try:
        cursor.execute(query)
        result = cursor.fetchall()
    except Exception as e:
        result = [("error", str(e))]
    conn.close()
    return str(result)
```

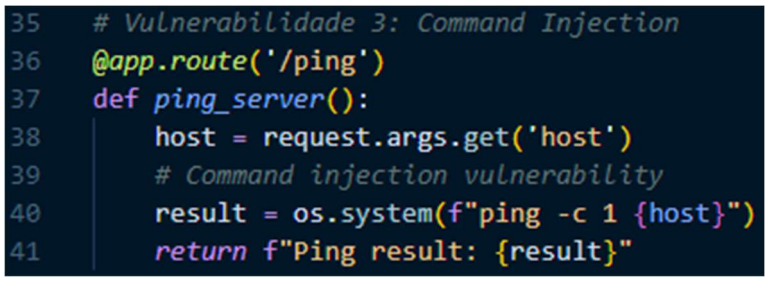
**Fonte:** Desenvolvido pelo autor (2025).

### 2.8.3 Execução de comandos de sistema sem sanitização

Determinadas funções, de acordo com a OWASP, permitem a execução de comandos do sistema operacional. Quando combinadas com entradas externas não validadas, tornam-se vetores para *Command Injection*, permitindo que o atacante execute comandos arbitrários no servidor. Essa vulnerabilidade é classificada pela OWASP como *Injection* (A03:2021).

A OWASP recomenda evitar a execução direta de comandos, preferindo APIs de alto nível. Se for inevitável, deve-se usar lista de argumentos totalmente controladas, sanitização rigorosa, e nunca concatenar *strings* vindas do usuário.

**Figura 10** - Vulnerabilidade *Command Injection*.



```
35  # Vulnerabilidade 3: Command Injection
36  @app.route('/ping')
37  def ping_server():
38      host = request.args.get('host')
39      # Command injection vulnerability
40      result = os.system(f"ping -c 1 {host}")
41      return f"Ping result: {result}"
```

**Fonte: Desenvolvido pelo autor** (2025).

### 2.8.4 Path traversal

É a falha que permite que um invasor acesse arquivos arbitrários do sistema ao manipular entradas que representam caminhos de ficheiros (por exemplo, ../../etc/passwd). No exemplo prático do repositório, o *endpoint* /file abre diretamente o caminho informado por filename sem normalização ou confinamento, expondo assim qualquer arquivo legível pelo processo da aplicação (OWASP A05:2021 — *Security Misconfiguration*; CWE-22). As consequências incluem divulgação de arquivos sensíveis (configurações, chaves, credenciais), informação que facilita ataques subsequentes e, em casos extremos, modificação de arquivos se houver escrita.

A mitigação recomendada envolve restringir a leitura/escrita a um diretório específico (chroot-like ou verificar os.path.commonpath), normalizar e validar o caminho (remover .. e caracteres inesperados) e, quando possível, mapear nomes lógicos (IDs) para arquivos reais em vez de aceitar caminhos arbitrários. Implementar essas proteções também facilita a detecção de falsos positivos nas ferramentas SAST, pois o padrão inseguro fica mais simples de identificar.

**Figura 11** - Vulnerabilidade path traversal.

```
69  # ========================
70  # Vulnerabilidade 4: Path Traversal (CWE-22)
71  # ========================
72  @app.route('/file')
73  def read_file():
74      filename = request.args.get('name')
75      # abre caminho direto vindo do usuário (path traversal)
76      with open(filename, 'r') as f:
77          content = f.read()
78      return content
```

**Fonte:** Desenvolvido pelo autor (2025).

### 2.8.5 Desserialização insegura.

Ocorre quando um objeto serializado recebido de fonte externa é desserializado sem validação, possibilitando a execução de código arbitrário ou instância de classes maliciosas no contexto da aplicação. No código analisado, o *endpoint* /load chama pickle.loads(data.encode()) sobre dados recebidos externamente — uso que é notoriamente perigoso em Python, já que *pickle* pode executar funções arbitrárias durante a desserialização (OWASP A05:2021 — *Security Misconfiguration*; CWE-502). O impacto típico inclui execução remota de código (RCE), escalonamento de privilégios e comprometimento total do servidor da aplicação.

As principais mitigações de acordo com a OWASP, consistem em não utilizar *pickle* para dados não confiáveis, optar por formatos seguros (JSON, por exemplo), aplicar *whitelist* de tipos esperados ao desserializar, ou utilizar mecanismos de desserialização com validação e *sandboxing*. Quando a aplicação exige serialização rica, é recomendado empregar bibliotecas que implementem mecanismos explícitos de segurança e exigir assinatura/assinatura HMAC dos *blobs* serializados para garantir integridade e origem.

**Figura 12** - Código desserialização insegura.

```
80  # ========================
81  # Vulnerabilidade 5: Insecure Deserialization (CWE-502)
82  # ========================
83  @app.route('/load')
84  def load_data():
85      data = request.args.get('data')  # espera string representando bytes serializados
86      # desserialização insegura com pickle
87      obj = pickle.loads(data.encode())
88      return str(obj)
```

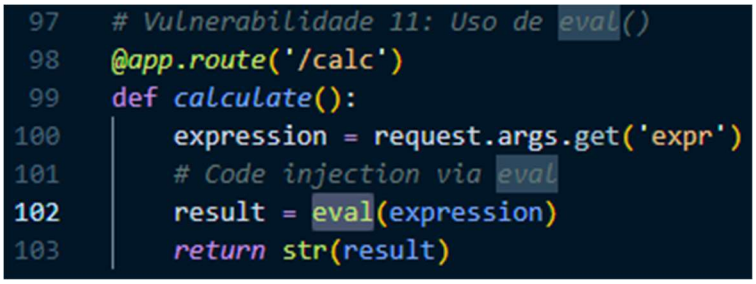
**Fonte:** Desenvolvido pelo autor (2025).

### 2.8.6 Uso inseguro da função eval.

A vulnerabilidade de *Code Injection* ocorre quando código malicioso é injetado e executado pela aplicação, explorando o tratamento inadequado de dados não confiáveis. Segundo a OWASP, este tipo de ataque é possível devido à falta de validação adequada de entrada e saída de dados, incluindo verificação de caracteres permitidos, formato de dados e quantidade esperada de informações.

A função eval é particularmente perigosa, pois executa dinamicamente uma *string* como código na linguagem de programação utilizada. Quando essa função recebe dados controlados pelo usuário sem validação apropriada, abre-se caminho para RCE, permitindo que atacantes executem comandos arbitrários no sistema.

**Figura 13** - Código vulnerabilidade eval.



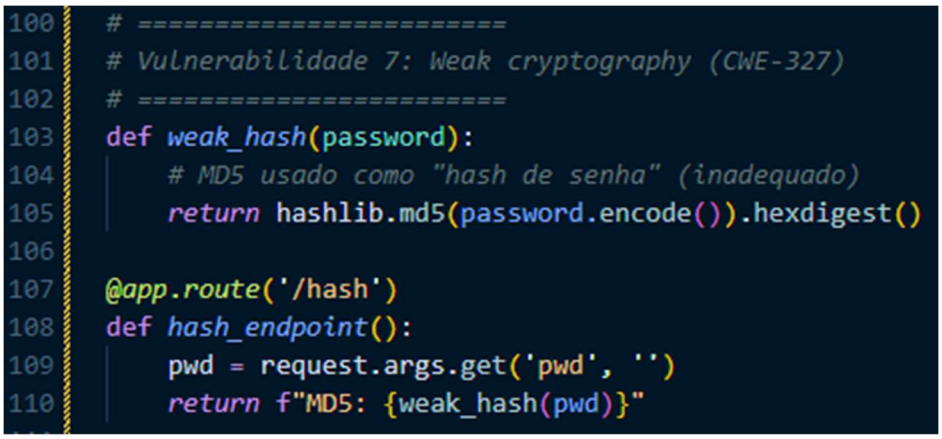

**Fonte:** Desenvolvido pelo autor (2025).

### 2.8.7 Criptografia fraca.

A OWASP (2025), explica que o uso de algoritmos criptográficos fracos para *hashing* de senhas ou proteção de dados sensíveis compromete a resistência a ataques de força bruta e a tabelas arco-íris. No código disponibilizado, a função weak_hash utiliza hashlib.md5 para derivar um "*hash*" de senha, prática inadequada para armazenamento de credenciais (OWASP A02:2021 — *Sensitive Data Exposure*; CWE-327). MD5 é considerado criptograficamente quebrável e não provê resistência suficiente contra ataques modernos; senhas *hashed* com MD5 são rapidamente recuperáveis.

A mitigação adequada, de acordo com a OWASP é empregar algoritmos e derivações de chave projetados para senhas: bcrypt, scrypt, argon2 ou, quando necessário, pbkdf2_hmac com *salt* único por senha e parâmetros de iteração elevados. Além disso, nunca se deve armazenar segredos *hardcoded* (ver parágrafo já existente) e é importante combinar hashing seguro com políticas de *salting, throttling* de tentativas de login e armazenamento em repositórios protegidos.

**Figura 14** - Criptografia fraca.



```
100  # ========================
101  # Vulnerabilidade 7: Weak cryptography (CWE-327)
102  # ========================
103  def weak_hash(password):
104      # MD5 usado como "hash de senha" (inadequado)
105      return hashlib.md5(password.encode()).hexdigest()
106
107  @app.route('/hash')
108  def hash_endpoint():
109      pwd = request.args.get('pwd', '')
110      return f"MD5: {weak_hash(pwd)}"
```

**Fonte:** Desenvolvido pelo autor (2025).

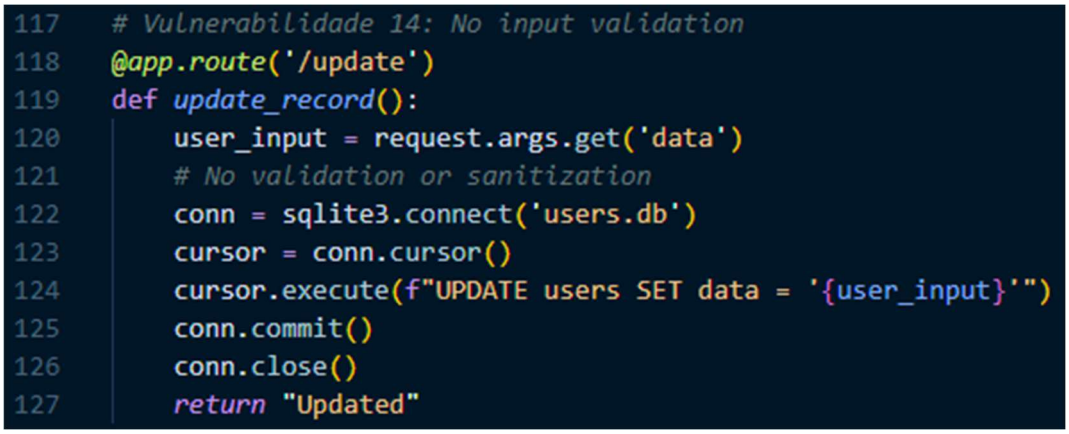
### 2.8.8 Ausência de tratamento de exceções e validação de entrada

A falta de validação dos dados fornecidos pelo usuário pode resultar em *crashes*, vazamento de informações e comportamentos inesperados. Além disso, a ausência de tratamento de exceções (*try*/*except*) facilita a exposição de erros internos ao usuário, o que pode ser explorado para engenharia reversa ou ataques de enumeração. Essa categoria está relacionada ao *Security Misconfiguration* (A05:2021) e à *Input Validation* (A01:2021 – *Broken Access Control*).

Para mitigação, a OWASP recomenda:

- validação positiva (“*allowlist*”),
- verificação de tipos, tamanhos e formatos,
- tratamento adequado de exceções,
- mensagens de erro genéricas para o usuário.

**Figura 15** - Input de dados no banco sem validação.



```
117  # Vulnerabilidade 14: No input validation
118  @app.route('/update')
119  def update_record():
120      user_input = request.args.get('data')
121      # No validation or sanitization
122      conn = sqlite3.connect('users.db')
123      cursor = conn.cursor()
124      cursor.execute(f"UPDATE users SET data = '{user_input}'")
125      conn.commit()
126      conn.close()
127      return "Updated"
```

**Fonte:** Desenvolvido pelo autor (2025).

# 3 Resultados

Neste capítulo, foi abordada a execução das ferramentas mencionadas no capítulo anterior, sobre o código malicioso ~~que foi~~ desenvolvido para teste. O objetivo principal é adicionar os resultados obtidos por cada ferramenta em cada uma das vulnerabilidades propostas.

A abordagem tomada neste capítulo será na mesma sequência que foi desenvolvido o capítulo anterior, será passado por cada vulnerabilidade de forma única e dissertado sobre cada resultado obtido por cada ferramenta.

## 3.1 *Hardcoded Credentials*

O uso de credenciais escritas diretamente no código-fonte representa um alto risco à segurança. De acordo com os testes feitos, apenas a ferramenta Bandit foi capaz de detectá-lo, como é possível observar na Figura 16.

**Figura 16** - Detecção hardcoded credentials Bandit.

```
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'admin123'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/plugins/b105_hardcoded_password_string.html
   Location: ./app.py:17:20
16 # =======================
17 DATABASE_PASSWORD = "admin123"
18 SECRET_KEY = "my_secret_key_12345"
```

**Fonte:** Desenvolvido pelo autor (2025).

O Bandit possui regras nativas *hardcoded_password_string* para detectar *strings* sensíveis. Semgrep e CodeQL não sinalizaram, possivelmente por ausência de regras específicas de *secrets scanning* no *ruleset* usado. A partir da análise dos arquivos de resultados, pode-se concluir que o Bandit apresentou uma classificação *true positive* (TP), enquanto Semgrep e CodeQl resultaram em *false* negative (FN).

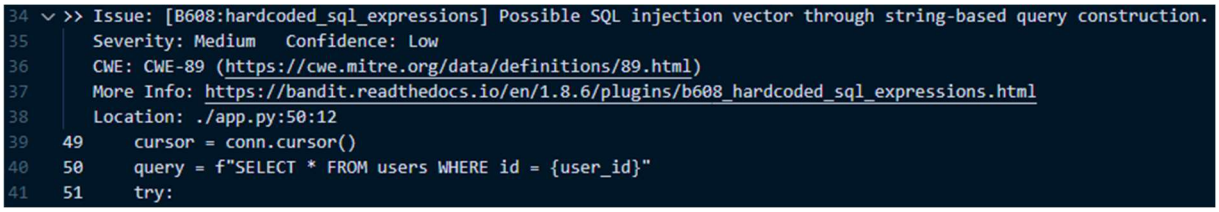
## 3.2 SQL Injection

Dentro das análises para SQL *Injection*, todas as três ferramentas obtiveram sucesso na detecção. Podendo-se confirmar com base nas Figuras 17, 18 e 19 sendo que, são os resultados de Bandit, Semgrep e CodeQl, respectivamente.

### 3.2.1 SQL *Injection* Bandit

De acordo com a Figura 17, o Bandit detectou com o identificador interno da ferramenta B608, identificador este que está diretamente ligado à possibilidade do vetor de ataque de SQL *Injection*. Severidade média e confiança baixa, conclui-se que, apesar da detecção, a ferramenta atribuiu um certo grau de incerteza quanto ao contexto deste teste executado.

**Figura 17** - Detecção SQL *Injection* Bandit.




**Fonte:** Desenvolvido pelo autor (2025).

### 3.2.2 SQL *Injection* Semgrep

Semgrep por sua vez gerou duas detecções para esta vulnerabilidade, como mostra a figura 18.

A primeira foi denominada python.django.security.injection.sql.sql-injection-using-db-cursor-execute.sql-injection-db-cursor-execute e foi detectada na linha 47 do código, justamente o trecho que foi mostrado no capítulo 3 e foi classificada como *Warning* e corresponde a um alerta genérico para situações em que dados controlados pelo usuário são passados diretamente ao método execute.

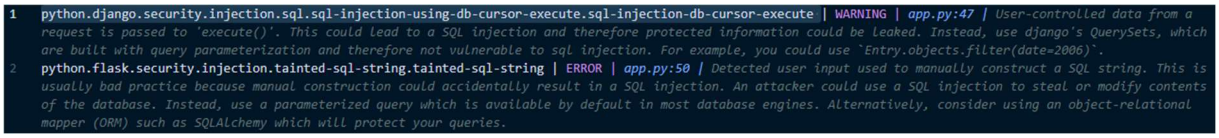
Essa regra tem como foco aplicações Django, porém, por falar sobre um padrão comum em consultas SQL quando são construídas manualmente, também acabou se aplicando no caso de teste do código analisado. O aviso enfatiza o risco de exposição de informações sensíveis devido à ausência de parametrização e recomenda o uso

do método *QuerySet* ou mecanismos ORM com *query parameterization*, que eliminam a vulnerabilidade.

Já a segunda detecção foi denominada python.flask.security.injection.tainted-sql-string.tainted-sql-string, apresentou nível de severidade *Error*, que é uma característica de um risco mais alto. Essa detecção refere-se ao uso explícito de interpolação de *strings* (via f-string) na construção de uma query SQL no contexto de uma aplicação Flask, prática considerada insegura por permitir que dados não sanitizados sejam incorporados diretamente ao comando SQL.

O relatório sugere como mitigação o uso de consultas parametrizadas — disponíveis por padrão em diversos motores de banco de dados — ou a adoção de bibliotecas ORM, como o SQLAlchemy, que abstraem e previnem esse tipo de falha.

**Figura 18** - Detecção SQL Injection Semgrep.




**Fonte:** Desenvolvido pelo autor (2025).

### 3.2.3 SQL Injection CodeQl

O CodeQL também identificou a vulnerabilidade de injeção de SQL no código de teste, classificando-a sob o identificador de regra py/sql-injection.

Essa regra pertence ao conjunto de consultas de segurança para Python e tem como objetivo detectar situações em que consultas SQL são construídas a partir de dados controlados pelo usuário, sem o devido processo de sanitização ou uso de parâmetros preparados.

Na prática, o CodeQL analisou o fluxo de dados desde a origem (função que recebe a requisição do usuário) até o ponto em que o valor é interpolado na *string* SQL. Esse mecanismo de rastreamento de *tainted data flow* permite identificar vulnerabilidades que não dependem apenas de *pattern matching*, mas da propagação real de variáveis inseguras dentro da aplicação.

O relatório gerado aponta que a consulta SQL é construída diretamente com dados não tratados, utilizando interpolação de *string* (f-string) e o método execute, o que torna possível a injeção de comandos arbitrários por um atacante.

Além de apresentar a vulnerabilidade, a própria consulta py/sql-injection fornece uma explicação detalhada e recomendações de mitigação, destacando a importância de utilizar parâmetros de consulta (*prepared statements*) ou bibliotecas ORM, como o SQLAlchemy, para evitar que o dado do usuário seja concatenado diretamente ao comando SQL.

**Figura 19** - Detecção SQL *Injection* CodeQl.


```
{
  "id": "py/sql-injection",
  "name": "py/sql-injection",
  "shortDescription": {
    "text": "SQL query built from user-controlled sources"
  },
  "fullDescription": {
    "text": "Building a SQL query from user-controlled sources is vulnerable to insertion of malicious SQL code by the user."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "error"
  },
  "help": {
    "text": "# SQL query built from user-controlled sources\nIf a database query (such as a SQL or NoSQL query) is built from user-provided data without sufficient sanitization, a user may be able to run malicious database queries.\n\nThis also includes using the `TextClause` class in the `[SQLAlchemy](https://pypi.org/project/SQLAlchemy/)` PyPI package, which is used to represent a literal SQL fragment and is inserted directly into the final SQL when used in a query built using the ORM.\n\n\n## Recommendation\nMost database connector libraries offer a way of safely embedding untrusted data into a query by means of query parameters or prepared statements.\n\n\n## Example\nIn the following snippet, a user is fetched from the database using three different queries.\n\nIn the first case, the query string is built by directly using string formatting from a user-supplied request parameter. The parameter may include quote characters, so this code is vulnerable to a SQL injection attack.\n\nIn the second case, the user-supplied request attribute is passed to the database using query parameters. The database connector library will take care of escaping and inserting quotes as needed.\n\nIn the third case, the placeholder in the SQL string has been manually quoted. Since most databaseconnector libraries will insert their own quotes, doing so yourself will make the code vulnerable to SQL injection attacks. In this example, if `username` was `; DROP ALL TABLES -- `, the final SQL query would be `SELECT * FROM users WHERE username = ''; DROP ALL TABLES -- '`\n\n\n```python\nfrom django.conf.urls import url\nfrom django.db import connection\n\n\ndef show_user(request, username):\n    with connection.cursor() as cursor:\n        # BAD -- Using string formatting\n        cursor.execute(\"SELECT * FROM users WHERE username = '%s'\" % username)\n        user = cursor.fetchone()\n\n        # GOOD -- Using parameters\n        cursor.execute(\"SELECT * FROM users WHERE username = %s\", username)\n        user = cursor.fetchone()\n\n        # BAD -- Manually quoting placeholder\n        cursor.execute(\"SELECT * FROM users WHERE username = '%s'\", username)\n        user = cursor.fetchone()\n\nurlpatterns = [url(r'^users/(?P<username>[^/]+)$', show_user)]\n```\n\n## References\n* Wikipedia: [SQL injection](https://en.wikipedia.org/wiki/SQL_injection).\n* OWASP: [SQL Injection Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html).\n* [SQLAlchemy documentation for TextClause](https://docs.sqlalchemy.org/en/14/core/sqlelement.html#sqlalchemy.sql.expression.text.params.text).\n* Common Weakness Enumeration: [CWE-89](https://cwe.mitre.org/data/definitions/89.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

### 3.2.4 Conclusão SQL *Injection*

Com estas informações, é possível classificar os resultados obtidos como TP, dado que, todas as 3 ferramentas foram devidamente capazes de detectar a vulnerabilidade, e devidamente explicadas cada um dentro dos critérios da ferramenta.

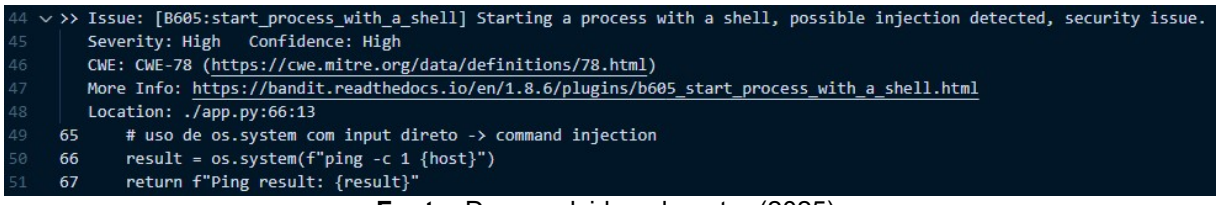
## 3.3 *Command Injection*

Para esta vulnerabilidade, as três ferramentas obtiveram sucesso no teste e foram capazes de detectá-la. Neste parágrafo será destrinchado e dissertado de forma individual cada resultado.

### 3.3.1 *Command Injection* Bandit

O relatório do Bandit identificou um problema de *Command Injection* na chamada a um processo do sistema operacional a partir de dados controlados pelo usuário. No *log* do Bandit o achado aparece da seguinte forma como mostra a Figura 20:

**Figura 20** - Detecção *Command Injection* Bandit.



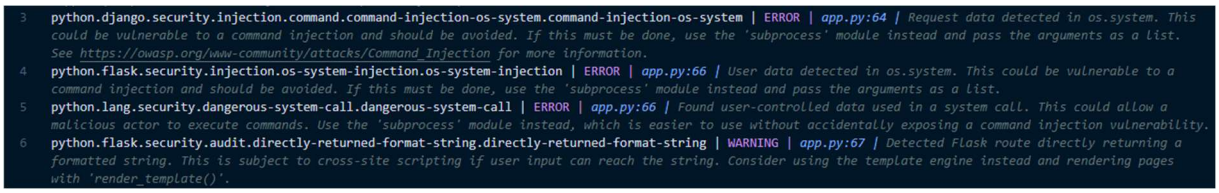

**Fonte:** Desenvolvido pelo autor (2025).

O Bandit classifica esse tipo de detecção como alta severidade por se tratar de um vetor clássico de *command injection* (CWE-78). A regra B605 detecta chamadas que executam comandos em shell (por exemplo os.system, subprocess.call com shell=True, os.popen) em que o comando contém valores interpolados diretamente vindos do usuário. Nesse caso, a variável *host* (proveniente de uma requisição) é concatenada/interpolada numa f-string e passada ao os.system, permitindo que um atacante injete argumentos ou comandos adicionais (por exemplo rm -rf / ou && curl http://malicious), com potencial execução arbitrária no servidor. O alto nível de confiança informado pelo Bandit indica que o padrão detectado é claro (uso direto de os.system com entrada dinâmica) e dificilmente se trata de um falso positivo no contexto apresentado.

### 3.3.2 *Command Injection* Semgrep

O Semgrep produziu quatro alertas relacionados ao trecho que executa comandos do sistema operacional com dados controlados pelo usuário (linhas 64–67 de app.py). As regras cobrem tanto contextos específicos de *framework* (Django/Flask) quanto verificações genéricas da linguagem. A figura 14 mostra exatamente como foi estas detecções:

**Figura 21** - Detecção *Command Injection* Semgrep.




**Fonte: Desenvolvido pelo autor** (2025).

#### 3.3.2.1 Detecções

A detecção um de Semgrep para esta vulnerabilidade verifica o uso da biblioteca “os” utilizando o comando os.system e recebendo dados da requisição. Esta detecção ocorre justamente por conta da variável proveniente do *request* fluir diretamente para o os.system, o que caracteriza a possibilidade de *command injection*, o que foi caracterizada como alto risco de acordo com a CWE-78. Semgrep deixa como recomendação caso isso realmente precise ser feito, utilizar o módulo “subprocess” e passar os argumentos em formato de lista.

A detecção dois observou o mesmo padrão inseguro para a vulnerabilidade CWE-78, porém agora com o *ruleset* específico para o *framework* flask, novamente deixando como recomendação caso isso seja necessário, utilizar o módulo subprocess e os dados serem passados em formato de lista.

A terceira detecção segue o mesmo padrão, contudo, esta foi feita com base no *ruleset* do Semgrep feita para a linguagem python, diferentemente das outras que são *rulesets* para *frameworks* especificos (django e flask respectivamente). E repetindo as recomendações de mitigação com uso de subprocess.

Já a quarta detecção diz que, se o conteúdo formatado vier (ou puder vir) de usuário, pode abrir margem para XSS; aqui aparece como *audit* (aviso) porque depende do contexto. Deixa-se como recomendação renderizar a resposta via *template engine* ao invés de *render_template*.

O Semgrep cobriu o caso com redundância saudável (regras Django, Flask e linguagem) e classificou corretamente com *ERROR* os pontos que expõem execução de comando. Essa multiplicidade de regras aumenta a cobertura e a confiabilidade da detecção, apontando tanto o ponto crítico (os.system) quanto a boa prática de saída (evitar formatar e retornar *strings* diretamente).

### 3.3.3 *Command Injection* CodeQl

O CodeQL sinalizou o uso de cadeias de comando controladas externamente, classificado como *error* (nível alto). A regra py/command-line-injection detecta pontos do código onde dados de usuário fluem para funções que executam comandos ou interpretam código (por exemplo, os.system, subprocess.* com entrada dinâmica, exec/eval), permitindo que o atacante altere o significado do comando. A vulnerabilidade se relaciona diretamente às classificações CWE-78 (OS *Command Injection*) e CWE-88 (*Argument Injection*).

**Figura 22** - Detecção *Command Injection* CodeQl.


```
{
  "id": "py/command-line-injection",
  "name": "py/command-line-injection",
  "shortDescription": {
    "text": "Uncontrolled command line"
  },
  "fullDescription": {
    "text": "Using externally controlled strings in a command line may allow a malicious user to change the meaning of the command."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "error"
  },
  "help": {
    "text": "# Uncontrolled command line\nCode that passes user input directly to `exec`, `eval`, or some other library routine that executes a
    command, allows the user to execute malicious code.\n\n\n## Recommendation\nIf possible, use hard-coded string literals to specify the
    command to run or the library to load. Instead of passing the user input directly to the process or library function, examine the user input
    and then choose among hard-coded string literals.\n\nIf the applicable libraries or commands cannot be determined at compile time, then add
    code to verify that the user input string is safe before using it.\n\n\n## Example\nThe following example shows two functions. The first is
    unsafe as it takes a shell script that can be changed by a user, and passes it straight to `subprocess.call()` without examining it first.
    The second is safe as it selects the command from a predefined allowlist.\n\n\n```python\n\nurlpatterns = [\n    # Route to
    command_execution\n    url(r'^command-ex1$', command_execution_unsafe, name='command-execution-unsafe'),\n    url(r'^command-ex2$',
    command_execution_safe, name='command-execution-safe')\n]\n\nCOMMANDS = {\n    \"list\" :\"ls\",\n    \"stat\" : \"stat\"\n}\n\ndef
    command_execution_unsafe(request):\n    if request.method == 'POST':\n        action = request.POST.get('action', '')\n        #BAD -- No
    sanitizing of input\n        subprocess.call([\"application\", action])\n\ndef command_execution_safe(request):\n    if request.method ==
    'POST':\n        action = request.POST.get('action', '')\n        #GOOD -- Use an allowlist\n        subprocess.call([\"application\",
    COMMANDS[action]])\n\n```\n\n## References\n* OWASP: [Command Injection](https://www.owasp.org/index.php/Command_Injection).\n* Common
    Weakness Enumeration: [CWE-78](https://cwe.mitre.org/data/definitions/78.html).\n* Common Weakness Enumeration: [CWE-88](https://cwe.mitre.
    org/data/definitions/88.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

A query py/command-line-injection do CodeQL demonstrou elevada precisão por rastrear o dado contaminado até o ponto de execução do comando, oferecendo recomendações prescritivas (*allowlist*, subprocess sem shell e validação). Isso reduz falsos positivos típicos de regras puramente sintáticas e reforça o CodeQL como ferramenta muito eficaz para detectar *Command Injection* em aplicações Python.

### 3.3.4 Conclusão *Command Injection*

Pode-se conclui que a vulnerabilidade *Command Injection*, correlacionada pela CWE-78, foi plenamente identificada pelas três soluções de análise estática, com o

CodeQL destacando-se pela profundidade da inspeção e rastreamento de fluxo, o Semgrep pela cobertura de regras contextualizadas por *framework*, e o Bandit pela simplicidade e precisão na identificação de padrões diretos.

Esse resultado evidencia que a combinação das ferramentas potencializa a detecção e validação cruzada de falhas críticas de segurança em pipelines de CI/CD voltados a aplicações Python.

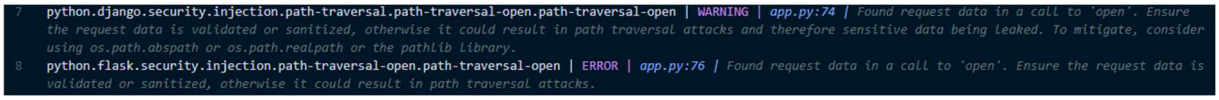
## *3.4 Path traversal*

Neste parágrafo será dissertado sobre os resultados das três soluções de análise estática para a vulnerabilidade *path traversal*. Pôde-se observar que apenas Semgrep e CodeQl obtiveram algum tipo de resultado quanto a presença desta falha de segurança, enquanto nos testes com Bandit, ela passou de forma despercebida pela ferramenta.

### 3.4.1 *Path traversal* Semgrep

O Semgrep gerou dois alertas distintos referentes à vulnerabilidade de *Path Traversal*, ambos relacionados ao uso da função open com dados provenientes diretamente da requisição do usuário. Os avisos estão localizados nas linhas 74 e 76 do arquivo app.py, e foram classificados respectivamente como *Warning* e *Error*.

**Figura 23** - Detecção Path Traversal Semgrep.




**Fonte:** Desenvolvido pelo autor (2025).

#### 3.4.1.1 Detecções

A primeira detecção registrada, tem foco em aplicações Django e foi categorizada como *Warning*. Ela mostra que o código realiza a abertura de um arquivo (open) utilizando valores controlados externamente, sem qualquer mecanismo de validação ou sanitização. Esse padrão expõe a aplicação ao risco de leitura de

arquivos arbitrários no sistema, especialmente quando o usuário consegue manipular o caminho do arquivo (por exemplo, utilizando o padrão ../../etc/passwd).

A segunda detecção foi classificada como *Error*, corresponde ao mesmo problema, mas no contexto de aplicações Flask. Essa redundância de regras é proposital e reflete a capacidade do Semgrep de aplicar políticas específicas para diferentes *frameworks*, aumentando a cobertura e a precisão de detecção.

Ambas as regras trazem recomendações diretas de mitigação, sugerindo o uso de funções como os.path.abspath(), os.path.realpath() ou da biblioteca pathlib para normalizar e restringir os caminhos acessíveis, além de validar rigorosamente os nomes de arquivos permitidos antes de realizar a leitura.

### 3.4.2 Path Traversal CodeQl

O CodeQL identificou a vulnerabilidade de *Path Traversal* no código de teste através da regra py/path-injection, classificada com nível *Error*.

**Figura 24** - Detecção Path Traversal CodeQl.


```
{
  "id": "py/path-injection",
  "name": "py/path-injection",
  "shortDescription": {
    "text": "Uncontrolled data used in path expression"
  },
  "fullDescription": {
    "text": "Accessing paths influenced by users can allow an attacker to access unexpected resources."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "error"
  },
  "help": {
    "text": "# Uncontrolled data used in path expression\nAccessing files using paths constructed from user-controlled data can allow an attacker
    to access unexpected resources. This can result in sensitive information being revealed or deleted, or an attacker being able to influence
    behavior by modifying unexpected files.\n\n\n## Recommendation\nValidate user input before using it to construct a file path, either using an
    off-the-shelf library function like `werkzeug.utils.secure_filename`, or by performing custom validation.\n\nIdeally, follow these
    rules:\n\n* Do not allow more than a single \".\" character.\n* Do not allow directory separators such as \"/\" or \"\\\\\" (depending on the
    file system).\n* Do not rely on simply replacing problematic sequences such as \"../\". For example, after applying this filter to \".../...//
    \", the resulting string would still be \"../\".\n* Use an allowlist of known good patterns.\n\n## Example\nIn the first example, a file name
    is read from an HTTP request and then used to access a file. However, a malicious user could enter a file name that is an absolute path, such
    as `\"/etc/passwd\"`.\n\nIn the second example, it appears that the user is restricted to opening a file within the `\"user\"` home
    directory. However, a malicious user could enter a file name containing special characters. For example, the string `\"../../../etc/passwd\"`
    will result in the code reading the file located at `\"/server/static/images/../../../etc/passwd\"`, which is the system's password file.
    This file would then be sent back to the user, giving them access to all the system's passwords. Note that a user could also use an absolute
    path here, since the result of `os.path.join(\"/server/static/images/\", \"/etc/passwd\")` is `\"/etc/passwd\"`.\n\nIn the third example, the
    path used to access the file system is normalized *before* being checked against a known prefix. This ensures that regardless of the user
    input, the resulting path is safe.\n\n\n```python\nimport os.path\nfrom flask import Flask, request, abort\n\napp = Flask(__name__)\n\n@app.
    route(\"/user_picture1\")\ndef user_picture1():\n    filename = request.args.get('p')\n    # BAD: This could read any file on the file
    system\n    data = open(filename, 'rb').read()\n    return data\n\n@app.route(\"/user_picture2\")\ndef user_picture2():\n    base_path = '/
    server/static/images'\n    filename = request.args.get('p')\n    # BAD: This could still read any file on the file system\n    data = open(os.
    path.join(base_path, filename), 'rb').read()\n    return data\n\n@app.route(\"/user_picture3\")\ndef user_picture3():\n    base_path = '/
    server/static/images'\n    filename = request.args.get('p')\n    #GOOD -- Verify with normalised version of path\n    fullpath = os.path.
    normpath(os.path.join(base_path, filename))\n    if not fullpath.startswith(base_path):\n        raise Exception(\"not allowed\")\n    data =
    open(fullpath, 'rb').read()\n    return data\n\n```\n\n## References\n* OWASP: [Path Traversal](https://owasp.org/www-community/attacks/
    Path_Traversal).\n* npm: [werkzeug.utils.secure_filename](http://werkzeug.pocoo.org/docs/utils/#werkzeug.utils.secure_filename).\n* Common
    Weakness Enumeration: [CWE-22](https://cwe.mitre.org/data/definitions/22.html).\n* Common Weakness Enumeration: [CWE-23](https://cwe.mitre.
    org/data/definitions/23.html).\n* Common Weakness Enumeration: [CWE-36](https://cwe.mitre.org/data/definitions/36.html).\n* Common Weakness
    Enumeration: [CWE-73](https://cwe.mitre.org/data/definitions/73.html).\n* Common Weakness Enumeration: [CWE-99](https://cwe.mitre.org/data/
    definitions/99.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

Essa query faz parte do conjunto de análises voltadas à integridade de acesso ao sistema de arquivos em aplicações Python e tem como objetivo detectar o uso de dados não controlados na construção de caminhos de arquivo.

De acordo com a descrição do arquivo gerado pela ferramenta, o alerta é emitido quando informações fornecidas por usuários são utilizadas diretamente na formação de um caminho de arquivo, sem validação, sanitização ou normalização adequadas. Essa prática permite que um atacante acesse, modifique ou exponha recursos inesperados do servidor, como diretórios fora da área permitida da aplicação.

O comportamento foi corretamente identificado nas linhas 74–76 do arquivo app.py, onde o parâmetro recebido da requisição é utilizado na função open() sem qualquer tipo de restrição.

Esse cenário representa o risco descrito pelo CWE-22 (*Improper Limitation of a Pathname to a Restricted Directory*) e pelo CWE-23 (*Relative Path Traversal*), ambos relacionados à manipulação indevida de caminhos de arquivos.

A query py/path-injection recomenda explicitamente validar o *input* do usuário antes de utilizá-lo na construção do caminho. Entre as práticas sugeridas estão:

1. Usar funções de validação como werkzeug.utils.secure_filename, amplamente empregada em aplicações Flask;
2. Restringir caracteres e símbolos proibidos, como "/", "\", ".." e múltiplos pontos;
3. Evitar depender apenas de substituição de sequências (../), pois ainda podem permitir travessias relativas;
4. Implementar *allowlists* de nomes de arquivos ou extensões válidas;
5. Normalizar o caminho antes de validar, utilizando os.path.normpath() ou pathlib.Path.resolve().

### 3.4.3 Conclusão path traversal

A vulnerabilidade de *Path Traversal* foi corretamente identificada por duas das três ferramentas analisadas Semgrep e CodeQL, enquanto o Bandit não apresentou qualquer detecção relacionada a esse tipo de falha. De modo geral, a combinação dos resultados obtidos indica que, para vulnerabilidades do tipo *Path Traversal*, ferramentas baseadas em análise semântica e contextual, como o Semgrep e o CodeQL, oferecem desempenho superior e maior profundidade de análise. Já o

Bandit, apesar de eficiente para casos mais diretos, não apresentou cobertura suficiente para esse tipo de falha.

Conclui-se, portanto, que as ferramentas Semgrep e CodeQL apresentaram TPs consistentes para o caso de *Path Traversal*, enquanto o Bandit apresentou FN, não reconhecendo a vulnerabilidade existente.

## *3.5 Insecure Deserialization*

Para esta vulnerabilidade, as três ferramentas obtiveram sucesso na detecção e neste parágrafo será mostrado de forma unitária o que cada uma retornou.

### 3.5.1 *Insecure Deserialization* Bandit

O Bandit identificou a vulnerabilidade de desserialização insegura por meio da regra B301:*blacklist*, classificada com severidade média e alta confiança, conforme trecho localizado na linha 87 do arquivo app.py.

**Figura 25** - Detecção Insecure Deserialization Bandit.

```
>> Issue: [B301:blacklist] Pickle and modules that wrap it can be unsafe when used to deserialize untrusted data, possible security issue.
   Severity: Medium   Confidence: High
   CWE: CWE-502 (https://cwe.mitre.org/data/definitions/502.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/blacklists/blacklist_calls.html#b301-pickle
   Location: ./app.py:87:10
86      # desserialização insegura com pickle
87      obj = pickle.loads(data.encode())
88      return str(obj)
```

Fonte: Desenvolvido pelo autor (2025).

O alerta faz referência direta ao CWE-502 (*Deserialization of Untrusted Data*), indicando que o uso da função pickle.loads() com dados provenientes de fontes externas representa um risco elevado de execução arbitrária de código.

A regra B301 pertence à categoria de *blacklists* de chamadas inseguras, que identificam o uso de funções e bibliotecas conhecidas por introduzir vulnerabilidades críticas, mesmo sem análise contextual do fluxo de dados.

O Bandit destaca que módulos como *pickle* permitem que objetos arbitrários sejam reconstruídos a partir de dados serializados, o que possibilita que um atacante injete instruções maliciosas executadas durante o processo de desserialização.

### 3.5.2 *Insecure Deserialization* Semgrep

O Semgrep também detectou a vulnerabilidade em duas regras distintas, ambas na linha 87 de app.py.

A primeira regra foi classificada como *Error*, identifica a presença de uma biblioteca de desserialização insegura utilizada em uma rota Flask. Ela alerta que bibliotecas como *pickle* podem permitir RCE se dados de usuário forem passados diretamente à função loads().

Já a segunda regra foi categorizada como *Warning*, reforça a recomendação de evitar o uso de *pickle* e sugere o emprego de alternativas seguras como json.loads() ou yaml.safe_load().

A existência de duas regras sobre o mesmo ponto demonstra a abordagem redundante e detalhada do Semgrep, que visa cobrir tanto contextos de *frameworks* específicos quanto práticas inseguras da linguagem em geral.

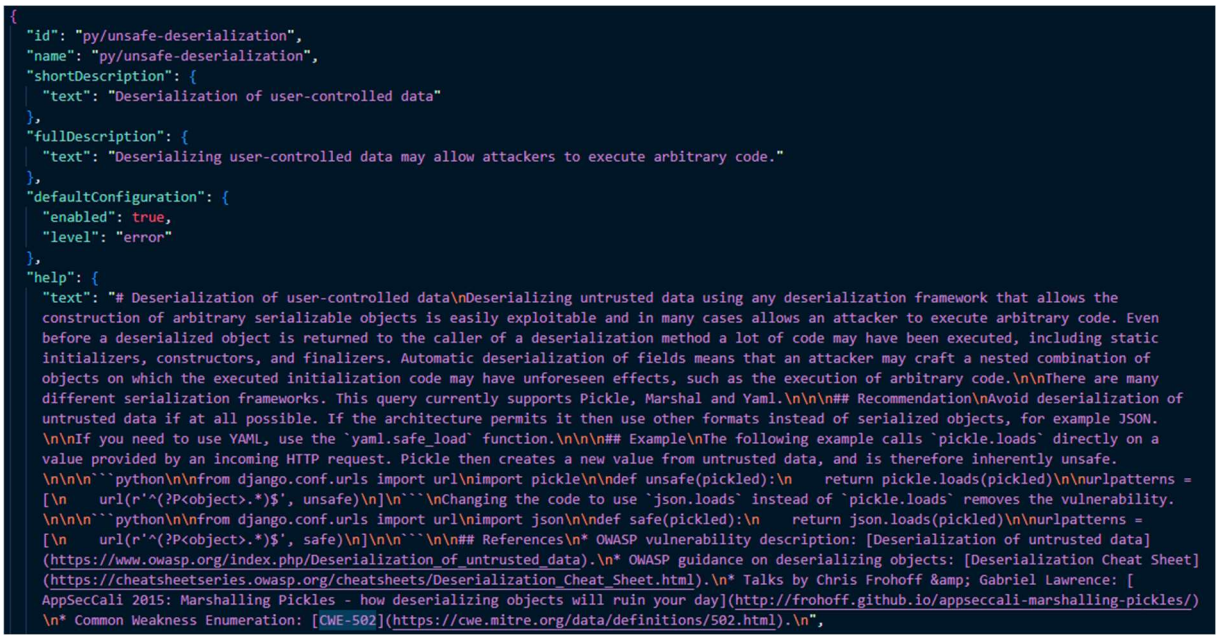
Em ambas as detecções, o Semgrep também forneceu orientações claras de mitigação: substituir a biblioteca de desserialização por alternativas seguras, restringir a entrada de dados e, se necessário, validar rigorosamente o conteúdo recebido antes do processamento.

### 3.5.3 CodeQl

O CodeQL detectou a mesma vulnerabilidade através da query py/unsafe-deserialization, classificada com nível *Error* e mapeada para o CWE-502.

A regra identifica cenários em que dados controlados por usuário são desserializados diretamente usando *frameworks* que permitem reconstruir objetos arbitrários, como *Pickle*, Marshal e YAML, resultando em alto risco de execução de código arbitrário. Diferente das outras ferramentas, o CodeQL realiza uma análise semântica de fluxo de dados, rastreando o valor recebido de fontes externas até o ponto de desserialização. O relatório do CodeQL inclui uma explicação detalhada e um exemplo prático, mostrando o caso inseguro pickle.loads e a alternativa segura json.loads(request_data).

**Figura 26** - Detecção *Insecure Deserialization* CodeQl.



```
{
  "id": "py/unsafe-deserialization",
  "name": "py/unsafe-deserialization",
  "shortDescription": {
    "text": "Deserialization of user-controlled data"
  },
  "fullDescription": {
    "text": "Deserializing user-controlled data may allow attackers to execute arbitrary code."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "error"
  },
  "help": {
    "text": "# Deserialization of user-controlled data\nDeserializing untrusted data using any deserialization framework that allows the
    construction of arbitrary serializable objects is easily exploitable and in many cases allows an attacker to execute arbitrary code. Even
    before a deserialized object is returned to the caller of a deserialization method a lot of code may have been executed, including static
    initializers, constructors, and finalizers. Automatic deserialization of fields means that an attacker may craft a nested combination of
    objects on which the executed initialization code may have unforeseen effects, such as the execution of arbitrary code.\n\nThere are many
    different serialization frameworks. This query currently supports Pickle, Marshal and Yaml.\n\n\n## Recommendation\nAvoid deserialization of
    untrusted data if at all possible. If the architecture permits it then use other formats instead of serialized objects, for example JSON.
    \n\nIf you need to use YAML, use the `yaml.safe_load` function.\n\n\n## Example\nThe following example calls `pickle.loads` directly on a
    value provided by an incoming HTTP request. Pickle then creates a new value from untrusted data, and is therefore inherently unsafe.
    \n\n\n```python\n\nfrom django.conf.urls import url\nimport pickle\n\ndef unsafe(pickled):\n    return pickle.loads(pickled)\n\nurlpatterns =
    [\n    url(r'^(?P<object>.*)$', unsafe)\n]\n```\nChanging the code to use `json.loads` instead of `pickle.loads` removes the vulnerability.
    \n\n\n```python\n\nfrom django.conf.urls import url\nimport json\n\ndef safe(pickled):\n    return json.loads(pickled)\n\nurlpatterns =
    [\n    url(r'^(?P<object>.*)$', safe)\n]\n\n```\n\n## References\n* OWASP vulnerability description: [Deserialization of untrusted data]
    (https://www.owasp.org/index.php/Deserialization_of_untrusted_data).\n* OWASP guidance on deserializing objects: [Deserialization Cheat Sheet]
    (https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html).\n* Talks by Chris Frohoff &amp;amp; Gabriel Lawrence: [
    AppSecCali 2015: Marshalling Pickles - how deserializing objects will ruin your day](http://frohoff.github.io/appseccali-marshalling-pickles/)
    \n* Common Weakness Enumeration: [CWE-502](https://cwe.mitre.org/data/definitions/502.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

A ferramenta recomenda, sempre que possível, evitar completamente a desserialização de dados não confiáveis. Caso o uso de bibliotecas seja indispensável, deve-se preferir funções seguras como yaml.safe_load() e aplicar validações rígidas.

### 3.5.4 Conclusão *Insecure Desserialization*

As três ferramentas analisada, Bandit, Semgrep e CodeQL identificaram corretamente a vulnerabilidade de desserialização insegura, presente na linha 87 do arquivo app.py, onde a função pickle.loads() é utilizada para processar dados recebidos de forma direta e sem validação. demonstrando maturidade e eficácia tanto das regras baseadas em padrões de Bandit e Semgrep quanto da análise semântica aprofundada oferecida pelo CodeQL, que se destaca na interpretação do fluxo de dados e na precisão da análise contextual.

## 3.6 Uso de Eval

Para a vulnerabilidade apontada como CWE-94, foi obtido resultado de todas as três ferramentas, e neste parágrafo será explicado qual resultado foi obtido por estas ferramentas.

### 3.6.1 Bandit

O Bandit identificou a vulnerabilidade associada ao uso da função eval() na linha 97 do arquivo app.py, classificando-a pela regra B307:*blacklist*, com severidade média e alta confiança.

**Figura 27** - Detecção Uso de eval Bandit.

```
>> Issue: [B307:blacklist] Use of possibly insecure function - consider using safer ast.literal_eval.
   Severity: Medium   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/blacklists/blacklist_calls.html#b307-eval
   Location: ./app.py:97:13
96     # execução direta de expressão -> RCE
97     result = eval(expression)
98     return str(result)
```

**Fonte:** Desenvolvido pelo autor (2025).

Essa regra alerta para o uso de funções potencialmente inseguras como eval() e exec(), estas falhas podem permitem a execução dinâmica de expressões controladas externamente. O Bandit associa esse comportamento ao CWE-78, pois, embora o eval() execute código Python e não comandos do sistema diretamente, o impacto final é equivalente a uma injeção de comando, já que o invasor pode executar instruções arbitrárias que comprometem a integridade do sistema. No trecho analisado, o conteúdo da variável *expression*, proveniente de entrada de usuário, é avaliado diretamente pela função eval(), sem qualquer sanitização ou validação prévia. Isso abre a possibilidade de RCE, uma das falhas mais críticas em aplicações Python.

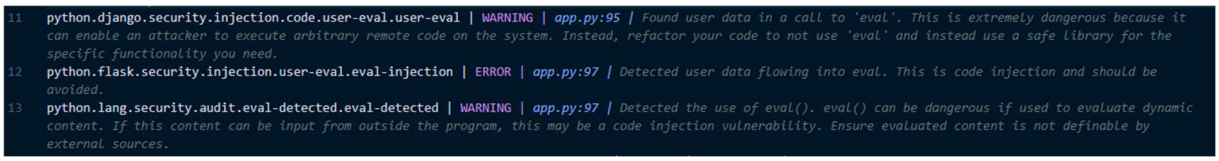
### 3.6.2 Semgrep

O Semgrep também identificou a vulnerabilidade de execução arbitrária por meio de três regras distintas, localizadas nas linhas 95 e 97 do arquivo app.py.

Por meio de duas destas três regras, o Semgrep notificou como *Warning* a possibilidade de existência desta vulnerabilidade, pois apenas detectando o eval ele já foi capaz de identificar o risco. Por outro lado, na terceira regra, e de mais necessidade de visibilidade, ele gerou um alerta como *Error* dado que, identificou

dados externos fluindo para dentro do método eval, justamente gerou como *Error* pois isto se caracteriza a presença nítida da vulnerabilidade.

**Figura 28** - Detecção Uso de eval Bandit.




**Fonte:** Desenvolvido pelo autor (2025).

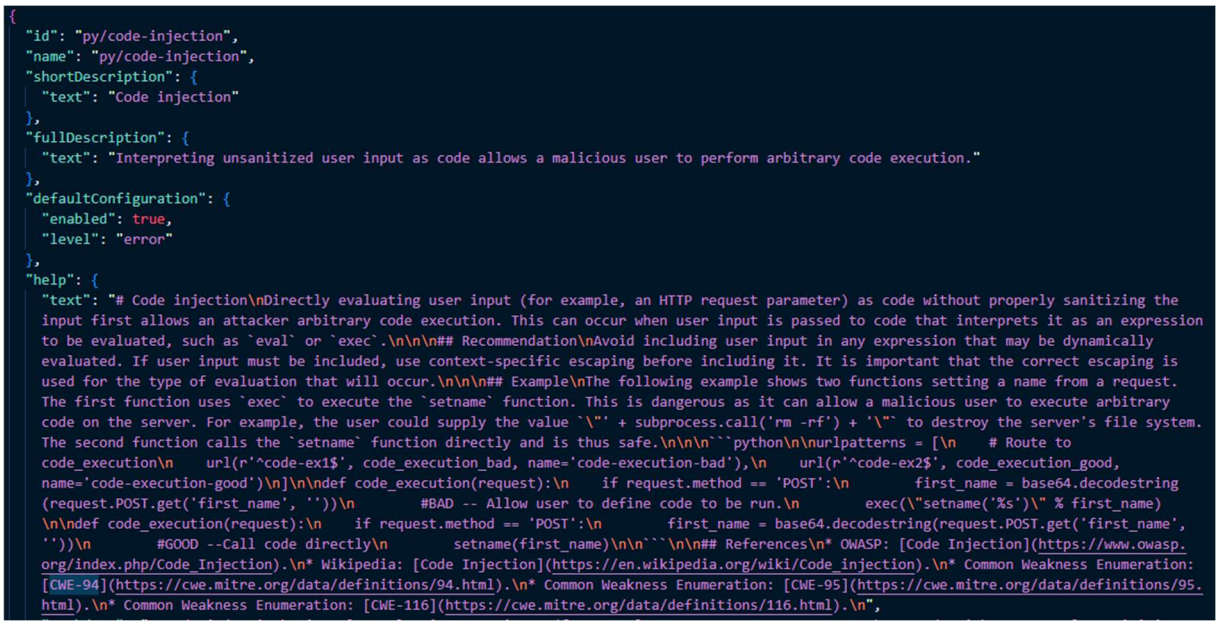
Conclui-se como TP o resultado de Semgrep para esta vulnerabilidade dado que, ele foi capaz não apenas de notificar a possibilidade e com isso recomendações para evitá-la, como também notificar de forma clara a presença dela no código.

### 3.6.3 CodeQl

O CodeQL detectou a mesma vulnerabilidade por meio da query py/code-injection, classificada como Error e mapeada para os identificadores CWE-94 (Code Injection) e CWE-95 (Improper Neutralization of Directives in Dynamically Evaluated Code). A regra define o problema como a interpretação de entrada não sanitizada como código, o que permite que usuários maliciosos executem comandos arbitrários.

Segundo a descrição oficial, isso ocorre quando a aplicação inclui diretamente dados de usuário em uma expressão avaliada por funções como eval() ou exec(), sem qualquer tratamento.

**Figura 29** -Detecção Uso de eval Bandit.



```
{
  "id": "py/code-injection",
  "name": "py/code-injection",
  "shortDescription": {
    "text": "Code injection"
  },
  "fullDescription": {
    "text": "Interpreting unsanitized user input as code allows a malicious user to perform arbitrary code execution."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "error"
  },
  "help": {
    "text": "# Code injection\nDirectly evaluating user input (for example, an HTTP request parameter) as code without properly sanitizing the
    input first allows an attacker arbitrary code execution. This can occur when user input is passed to code that interprets it as an expression
    to be evaluated, such as `eval` or `exec`.\n\n\n## Recommendation\nAvoid including user input in any expression that may be dynamically
    evaluated. If user input must be included, use context-specific escaping before including it. It is important that the correct escaping is
    used for the type of evaluation that will occur.\n\n\n## Example\nThe following example shows two functions setting a name from a request.
    The first function uses `exec` to execute the `setname` function. This is dangerous as it can allow a malicious user to execute arbitrary
    code on the server. For example, the user could supply the value `\"' + subprocess.call('rm -rf') + '\"` to destroy the server's file system.
    The second function calls the `setname` function directly and is thus safe.\n\n\n```python\n\nurlpatterns = [\n    # Route to
    code_execution\n    url(r'^code-ex1$', code_execution_bad, name='code-execution-bad'),\n    url(r'^code-ex2$', code_execution_good,
    name='code-execution-good')\n]\n\ndef code_execution(request):\n    if request.method == 'POST':\n        first_name = base64.decodestring
    (request.POST.get('first_name', ''))\n        #BAD -- Allow user to define code to be run.\n        exec(\"setname('%s')\" % first_name)
    \n\ndef code_execution(request):\n    if request.method == 'POST':\n        first_name = base64.decodestring(request.POST.get('first_name',
    ''))\n        #GOOD --Call code directly\n        setname(first_name)\n\n```\n\n## References\n* OWASP: [Code Injection](https://www.owasp.
    org/index.php/Code_Injection).\n* Wikipedia: [Code Injection](https://en.wikipedia.org/wiki/Code_injection).\n* Common Weakness Enumeration:
    [CWE-94](https://cwe.mitre.org/data/definitions/94.html).\n* Common Weakness Enumeration: [CWE-95](https://cwe.mitre.org/data/definitions/95.
    html).\n* Common Weakness Enumeration: [CWE-116](https://cwe.mitre.org/data/definitions/116.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

O exemplo fornecido na documentação demonstra como um valor passado ao exec() pode ser manipulado para executar instruções arbitrárias, reforçando o risco de RCE.

Assim, a detecção do CodeQL também foi classificada como TP, confirmando a presença e gravidade da vulnerabilidade.

### 3.6.4 Conclusão CWE-94

As três ferramentas Bandit, Semgrep e CodeQL identificaram corretamente a vulnerabilidade de execução de código arbitrário no uso da função eval() com dados externos. As três ferramentas convergiram quanto à natureza e à gravidade da vulnerabilidade, classificando-a como TP.

Essa coerência reforça que o uso de eval() com dados não validados constitui uma falha crítica de segurança e deve ser totalmente evitado em ambientes de produção, sendo substituído por métodos seguros e restritivos como ast.literal_eval() ou por lógicas de conversão controladas.

## 3.7 Criptografia Fraca

Para esta vulnerabilidade, as três ferramentas foram capazes de realizar detecções, será destrinchado individualmente neste parágrafo sobre estes resultados obtidos e quais conclusões pôde-se tomar.

### 3.7.1 Bandit

Por meio da regra B324:hashlib interna da ferramenta, o Bandit detectou o algoritmo *hash* MD5 sendo utilizado no código-fonte, algo que já é considerado ultrapassado quando o assunto é segurança em senhas. Este evento detectado foi considerado de alta confiança, alta severidade e associado à CWE-327 pela ferramenta.

**Figura 30** - Detecção de criptografia fraca Bandit.

```
>> Issue: [B324:hashlib] Use of weak MD5 hash for security. Consider usedforsecurity=False
   Severity: High   Confidence: High
   CWE: CWE-327 (https://cwe.mitre.org/data/definitions/327.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/plugins/b324_hashlib.html
   Location: ./app.py:105:11
104         # MD5 usado como "hash de senha" (inadequado)
105         return hashlib.md5(password.encode()).hexdigest()
106
```

**Fonte:** Desenvolvido pelo autor (2025).

A função hashlib.md5(password.encode()).hexdigest() é utilizada para calcular o *hash* de uma senha, conforme indicado pelo comentário do próprio código. O Bandit considerou esta prática insegura, dado o fato que este algoritmo atualmente, já é obsoleto e vulnerável a ataques de colisão e força bruta. Sendo assim, incapaz de garantir a base da segurança, confidencialidade e integridade destes dados.

### 3.7.2 Semgrep

O Semgrep por meio de duas de suas regras internas, observou o uso desta fraca criptografia presente no código:

**Figura 31** - Detecção de criptografia fraca Semgrep.

```
python.lang.security.insecure-hash-algorithms-md5.insecure-hash-algorithm-md5 | WARNING | app.py:105 | Detected MD5 hash algorithm which is considered insecure.
MD5 is not collision resistant and is therefore not suitable as a cryptographic signature. Use SHA256 or SHA3 instead.
python.lang.security.audit.md5-used-as-password.md5-used-as-password | WARNING | app.py:105 | It looks like MD5 is used as a password hash. MD5 is not
considered a secure password hash because it can be cracked by an attacker in a short amount of time. Use a suitable password hashing function such as scrypt.
You can use `hashlib.scrypt`.
```

**Fonte:** Desenvolvido pelo autor (2025).

A primeira regra detectou a presença de um algoritmo MD5 e foi sinalizada como *warning,* pois até então, o Semgrep apenas o analisou como um algoritmo inseguro, e por enquanto não levou em consideração que ele estaria sendo utilizado como método de criptografia para senhas. Como recomendação a ferramenta trouxe os modelos SHA 256 ou SHA 3.

Por outro lado, na segunda regra, o Semgrep foi direto ao ponto principal da questão, a utilização deste algoritmo na criptografia de senhas, trazendo pontos importantes sobre a vulnerabilidade em questão, no caso, dizendo que este algoritmo não é seguro o suficiente contra ataques simples como o de colisão e pode ser quebrado facilmente pelo atacante em um período curto. Para recomendações, a ferramenta trouxe a possibilidade de usar funções *hashes* especificas e seguras para senhas como a *scrypt* podendo utilizar a biblioteca “hashlib.scrypt”.

### 3.7.3 CodeQl

O CodeQL foi capaz de identificar a vulnerabilidade por meio da regra interna da ferramenta py/weak-sensitive-data-hashing, classificada como *Warning* e classificada para os identificadores CWE-327 e CWE-328.

**Figura 32** - Detecção de criptografia fraca CodeQl.


```
{
  "id": "py/weak-sensitive-data-hashing",
  "name": "py/weak-sensitive-data-hashing",
  "shortDescription": {
    "text": "Use of a broken or weak cryptographic hashing algorithm on sensitive data"
  },
  "fullDescription": {
    "text": "Using broken or weak cryptographic hashing algorithms can compromise security."
  },
  "defaultConfiguration": {
    "enabled": true,
    "level": "warning"
  },
  "help": {
    "text": "# Use of a broken or weak cryptographic hashing algorithm on sensitive data\nUsing a broken or weak cryptographic hash function can leave data vulnerable,
    and should not be used in security related code.\n\nA strong cryptographic hash function should be resistant to:\n\n* pre-image attacks: if you know a hash value `h
    (x)`, you should not be able to easily find the input `x`.\n* collision attacks: if you know a hash value `h(x)`, you should not be able to easily find a different
    input `y` with the same hash value `h(x) = h(y)`.\nIn cases with a limited input space, such as for passwords, the hash function also needs to be computationally
    expensive to be resistant to brute-force attacks. Passwords should also have an unique salt applied before hashing, but that is not considered by this query.\n\nAs
    an example, both MD5 and SHA-1 are known to be vulnerable to collision attacks.\n\nSince it's OK to use a weak cryptographic hash function in a non-security
    context, this query only alerts when these are used to hash sensitive data (such as passwords, certificates, usernames).\n\nUse of broken or weak cryptographic
    algorithms that are not hashing algorithms, is handled by the `py/weak-cryptographic-algorithm` query.\n\n\n## Recommendation\nEnsure that you use a strong, modern
    cryptographic hash function:\n\n* such as Argon2, scrypt, bcrypt, or PBKDF2 for passwords and other data with limited input space.\n* such as SHA-2, or SHA-3 in
    other cases.\n\n## Example\nThe following example shows two functions for checking whether the hash of a certificate matches a known value -- to prevent tampering.
    The first function uses MD5 that is known to be vulnerable to collision attacks. The second function uses SHA-256 that is a strong cryptographic hashing function.
    \n\n\n```python\nimport hashlib\n\ndef certificate_matches_known_hash_bad(certificate, known_hash):\n    hash = hashlib.md5(certificate).hexdigest() # BAD\n
    return hash == known_hash\n\ndef certificate_matches_known_hash_good(certificate, known_hash):\n    hash = hashlib.sha256(certificate).hexdigest() # GOOD\n
    return hash == known_hash\n\n```\n\n## Example\nThe following example shows two functions for hashing passwords. The first function uses SHA-256 to hash passwords.
    Although SHA-256 is a strong cryptographic hash function, it is not suitable for password hashing since it is not computationally expensive.\n\n\n```python\nimport
    hashlib\n\ndef get_password_hash(password: str, salt: str):\n    return hashlib.sha256(password + salt).hexdigest() # BAD\n\n```\nThe second function uses Argon2
    (through the `argon2-cffi` PyPI package), which is a strong password hashing algorithm (and includes a per-password salt by default).\n\n\n```python\nfrom argon2
    import PasswordHasher\n\ndef get_initial_hash(password: str):\n    ph = PasswordHasher()\n    return ph.hash(password) # GOOD\n\ndef check_password(password: str,
    known_hash):\n    ph = PasswordHasher()\n    return ph.verify(known_hash, password) # GOOD\n\n```\n\n## References\n* OWASP: [Password Storage Cheat Sheet](https://
    cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html)\n* Common Weakness Enumeration: [CWE-327](https://cwe.mitre.org/data/definitions/327.
    html).\n* Common Weakness Enumeration: [CWE-328](https://cwe.mitre.org/data/definitions/328.html).\n* Common Weakness Enumeration: [CWE-916](https://cwe.mitre.org/
    data/definitions/916.html).\n",
```


**Fonte:** Desenvolvido pelo autor (2025).

Esta regra explica como o uso de algoritmos fracos podem comprometer a integridade de dados altamente sensíveis, neste caso ele explica a fraqueza quando são utilizados em contextos de autenticação, assinaturas digitais e no caso utilizado como exemplo neste trabalho, armazenamento de senhas. O CodeQL faz uma análise contextual que distingue usos de *hashes* seguros e inseguros, sinalizando apenas quando o algoritmo é aplicado sobre dados sensíveis.

A ferramenta recomenda o uso de funções criptográficas robustas como SHA-256, SHA-3 e, para senhas, o uso de algoritmos ainda mais seguros para estes tipos de dados, como Argon2, bcrypt, scrypt ou PBKDF2.

### 3.7.4 Conclusão Criptografia Fraca.

Pôde-se concluir por meio das análises dos relatórios destas ferramentas testadas, que todas obtiveram o resultado TP, ainda que, cada ferramenta trouxe sua forma individual de explicação e recomendação.

O Bandit apresentou a detecção direta ao ponto, com alta severidade e confiança, destacando o uso indevido de MD5 como função de *hash* para senhas. O Semgrep reforçou o achado por meio de duas regras complementares, abordando tanto a fragilidade do algoritmo quanto o contexto de uso em senhas

CodeQl além de trazer recomendações de outros algoritmos para criptografia, foi capaz ainda de realizar recomendações de algoritmos ainda mais robustos quando o assunto tratar de senhas e outros tipos de dados altamente sensíveis.

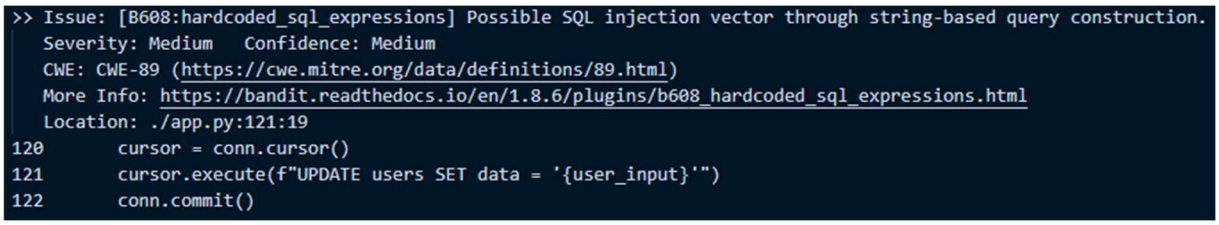
## *3.8* Sem validação de *input*

Nesta vulnerabilidade duas das três ferramentas foram capazes de realizar detecções quanto a vulnerabilidades no trecho do código, contudo, ocorreram algumas detecções diferentes por partes de cada uma, onde será dissertado de forma individual quais foram.

### 3.8.1 Bandit

O Bandit realizou a detecção de uma construção de instrução SQL via interpolação de strings com conteúdo controlado pelo usuário.

**Figura 33** - Detecção de sem validação de *input* Bandit.



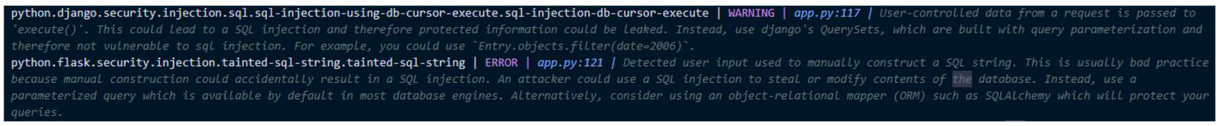

**Fonte:** Desenvolvido pelo autor (2025).

Este padrão foi classificado pela ferramenta como possível SQL *injection*, pois o usuário pode manipular diretamente o conteúdo da variável “user_input” que será passado diretamente na query de *update* do SQL.

### 3.8.2 Semgrep

O semgrep por meio de duas regras internas foi capaz de detectar vulnerabilidades neste mesmo trecho de código:

**Figura 34** - Detecção de sem validação de *input* Semgrep.




**Fonte:** Desenvolvido pelo autor (2025).

A primeira regra sinalizou que dados controlados pelo usuário chegam a *execute*() é um alerta genérico com sugestão de usar QuerySets/ORM do Django para parametrização automática. A classificação *Warning* indica que é um padrão perigoso, mas depende do contexto.

Já a segunda regra detectou especificamente a construção manual da *string* SQL usando o *input* do usuário. Essa regra é mais decisiva quanto à vulnerabilidade existente e classifica como *Error*, que identifica a prática de montar a *query* manualmente.

### 3.8.3 Conclusão sem validação de *input*

Houve resultados significativos por parte de Bandit e Semgrep para os trechos de códigos com esta vulnerabilidade, contudo, ambas as ferramentas detectaram as

falhas como possível SQL *injection*, CWE-89. Resultando ainda assim como TPs, por mais que a vulnerabilidade proposta CWE-20 não tenha sido identificada, as ferramentas foram capazes de detectar as falhas ali por meio de outra vulnerabilidade altamente conhecida.

CodeQl, por outro lado, não detectou nenhuma incidência de falha neste trecho do código, nas linhas 115 a 124 do app.py, resultando assim em um FN.

## Considerações finais

Este trabalho teve como foco principal analisar os resultados obtidos por três ferramentas SAST, sendo elas Bandit, Semgrep e CodeQL. Por meio do desenvolvimento e teste de códigos propositalmente vulneráveis na linguagem Python. O objetivo específico foi verificar a capacidade dessas ferramentas em identificar vulnerabilidades conhecidas e compreender como cada uma as descreve e classifica em seus relatórios de análise.

Durante o desenvolvimento, obteve-se sucesso tanto na implementação dos códigos vulneráveis quanto na execução das ferramentas dentro de uma *pipeline* CI, configurada no ambiente GitHub Actions. As execuções resultaram em achados distintos para cada vulnerabilidade, apresentando diferenças significativas na forma de detecção e no nível de detalhamento apresentado por cada ferramenta.

O Bandit demonstrou-se eficaz e direto em suas análises, identificando com precisão diversas vulnerabilidades e correlacionando-as de forma explícita aos identificadores CWE. Sua abordagem baseada em padrões fixos e regras simples o torna ágil e confiável para detecções clássicas, como *hardcoded credentials*, uso de eval() e algoritmos criptográficos obsoletos. Contudo, sua limitação está na ausência de uma análise contextual mais profunda, o que pode reduzir sua eficácia em casos mais complexos de fluxo de dados.

O Semgrep, por sua vez, apresentou maior flexibilidade e amplitude, sendo capaz de detectar vulnerabilidades por diferentes caminhos de análise. Ele se destacou ao aplicar regras específicas para *frameworks*, como Flask e Django, além de regras genéricas da linguagem Python, o que ampliou consideravelmente sua cobertura. Essa característica permitiu que uma mesma vulnerabilidade fosse identificada sob diferentes perspectivas, resultando em uma análise mais rica e contextualizada. Um grande fator positivo também desta ferramenta foi sempre trazer recomendações diretas sobre como evitar aquela vulnerabilidade em um código.

Já o CodeQL apresentou um altíssimo nível de profundidade analítica. Por meio de sua abordagem semântica, foi capaz de rastrear o fluxo de dados contaminados desde a origem até o ponto de exploração, identificando vulnerabilidades com grande precisão. Além disso, seus relatórios se destacaram por conter explicações detalhadas, exemplos de mitigação da vulnerabilidade e referências diretas aos

identificadores CWE, fornecendo uma visão bastante completa e didática do problema de segurança.

De modo geral, observou-se que algumas vulnerabilidades foram detectadas por apenas uma ou duas ferramentas, enquanto a grande maioria foi identificada pelas três, permitindo uma análise comparativa aprofundada sobre o comportamento, a precisão e a abrangência de cada solução no contexto de segurança de aplicações em pipelines automatizados.

## REFERÊNCIAS


BANDIT. **Configuration - Bandit Documentation**. OpenStack Security Project. Disponível em: <https://bandit.readthedocs.io/en/latest/config.html>. Acesso em: 7 set. 2025.

CODEQL. **About CodeQL queries.** CodeQL Documentation. Disponível em: <https://codeql.github.com/docs/writing-codeql-queries/about-codeql-queries/>. Acesso em: 20 set. 2025.

COMMUNICATION TEAM. **Como integrar o Semgrep no CI/CD e enviar os resultados para a Conviso Platform.** Conviso Blog, 25 maio 2023. Disponível em: <https://blog.convisoappsec.com/como-integrar-o-semgrep-no-ci-cd-e-enviar-os-resultados-para-a-conviso-platform/\>. Acesso em: 23 set. 2025.

GIÃO, Hugo da; FLORES, André; PEREIRA, Rui; CUNHA, Jácome. **Chronicles of CI/CD:** A Deep Dive into its Usage Over Time. 2024. Disponível em: <https://doi.org/10.48550/arXiv.2402.17588>. Acesso em: 17 out. 2024.

JANI, Yash. **Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development.** International Journal of Science and Research (IJSR), v. 12, n. 6, p. 2984-2987, jun. 2023. Disponível em: <https://www.ijsr.net>. Acesso em: 17 out. 2024. DOI: 10.21275/SR24716120535. Acesso em: 17 out. 2024.

MAAYAN, David Gilad. **Dynamic Application Security Testing.** Computer.org, 2023. Disponível em: <https://www.computer.org/publications/tech-news/trends/dynamic-application-security-testing>. Acesso em: 29 out. 2024.

NGUYEN, Bao Quan. **Improving the quality of CodeGrade testing system using Semgrep.** Bachelor's thesis – Lappeenranta-Lahti University of Technology LUT, 2025. Disponível em: <https://urn.fi/URN:NBN:fi-fe2025051442634\>. Acesso em: 23 set. 2025.

NGUYEN-DUC, Anh; DO, Manh Viet; HONG, Quan Luong; KHAC, Kiem Nguyen; QUANG, Anh Nguyen. **On the adoption of static analysis for software security assessment:** A case study of an open-source e-government project. Computers & Security, v. 111, p. 102470, 2021. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167404821002947>. Acesso em: 28 out. 2024.

NIST. **SP 800-63B – Digital Identity Guidelines.** Disponível em: <https://pages.nist.gov/800-63-3/>. Acesso em: 16 out. 2025.

NUTALAPATI, Venkat. **Automated Security Testing for Mobile Apps:** Tools, Techniques, and Best Practices. International Research Journal of Engineering & Applied Sciences (IRJEAS), v. 11, n. 1, p. 26-31, jan.-mar. 2023. Disponível em: <https://doi.org/10.55083/irjeas.2023.v11i01004>. Acesso em: 28 out. 2024.

OWASP Foundation. **Code Injection.** Disponível em: <https://owasp.org/www-community/attacks/Code_Injection>. Acesso em: 22 out. 2025.

OWASP. **Command Injection.** Disponível em: <https://owasp.org/www-community/attacks/Command_Injection>. Acesso em: 15 out. 2025.

OWASP. **Deserialization Cheatsheet.** Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html>. Acesso em: 16 out. 2025.

OWASP. **Input Validation Cheat Sheet.** Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html>. Acesso em: 22 out. 2025.

OWASP. **Password Storage Cheat Sheet.** Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html>. Acesso em: 16 out. 2025.

OWASP. **Path Traversal.** Disponível em: <https://owasp.org/www-community/attacks/Path_Traversal>. Acesso em: 15 out. 2025.

OWASP. **Secrets Management Guidelines.** Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html>. Acesso em: 15 out. 2025.

OWASP. **SQL Injection Prevention Cheat Sheet.** Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>. Acesso em: 15 out. 2025.

SEMGREP. **Writing rules** (Documentação oficial). 2025. Disponível em: <https://semgrep.dev/docs/writing-rules/overview\>. Acesso em: 23 set. 2025.

SHAHIN, Mojtaba; BABAR, Muhammad Ali; ZHU, Liming. **Continuous Integration, Delivery and Deployment:** A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, v. 5, p. 3909-3943, 2017. Disponível em: <https://ieeexplore.ieee.org/document/7884954>. Acesso em: 25 set. 2024.

SMARTTECS. **Code Security with Semgrep.** SmartTECS Cyber Security Blog, 10 fev. 2025. Disponível em: <https://blog.smarttecs.com/posts/2024-006-semgrep/\>. Acesso em: 23 set. 2025.

THULIN, Pontus. **Evaluation of the applicability of security testing techniques in continuous integration environments.** 2015. 83 f. Master's Thesis (Master's degree in Computer and Information Science) – Linköpings Universitet, Linköping, 2015. Disponível em: <http://www.diva-portal.org/smash/get/diva2:784545/FULLTEXT01.pdf>. Acesso em 01 out 2024