**Escola Superior de Tecnologia e Gestão**

**Licenciatura em Engenharia Informática**

**Laboratório de Programação**

**Ano Letivo 2024/25**

**Trabalhos Laboratoriais nº 8**

**Elaborado em: 2024/11/25**

**António Dinis a2021157297**

**Mariana Magalhães a2022147454**

**Tânia Martinho a2021153931**

**Índice**

[Lista de Figuras ii](#_Toc183446809)

[Lista de Tabelas ii](#_Toc183446810)

[1 Introdução 1](#_Toc183446811)

[1.1 Análise da evolução das vulnerabilidades comuns 1](#_Toc183446812)

[1.1.1 Análise de Tendências: 2](#_Toc183446813)

[1.1.2 Mudanças Significativas 3](#_Toc183446814)

[2 Métodos 4](#_Toc183446815)

[2.1 Auditoria manual de segurança 4](#_Toc183446816)

[2.2 Auditoria automática com o OWASP ZAP e SonarQube 6](#_Toc183446817)

[2.2.1 OWASP ZAP 6](#_Toc183446818)

[2.2.2 Análise do relatório gerado: 10](#_Toc183446819)

[2.2.3 SonarQube 11](#_Toc183446820)

[3 Resultados e Discussões 18](#_Toc183446821)

[3.1 Ações de remediação 18](#_Toc183446822)

[4 Conclusão 22](#_Toc183446823)

[5 Referências 23](#_Toc183446824)

# Lista de Figuras

[Figura 1: gráfico das mudanças significativas 3](#_Toc183446784)

[Figura 2: validação e sanitização de entradas do utilizador 4](#_Toc183446785)

[Figura 3: Controlo de acesso a recursos 4](#_Toc183446786)

[Figura 4: Gestão de sessões e cookies 5](#_Toc183446787)

[Figura 5: Armazenamento e exposição de dados sensíveis 5](#_Toc183446788)

[Figura 6: instalação do OWASP Zap e SonarQube no NixOS 6](file:///C:\Users\TaniaMartinho\Desktop\ENG.%20INF\ESTGOH\4º%20ANO\1º%20semestre\LP\Relatório%20-%20TP08.docx#_Toc183446789)

[Figura 7: scan de segurança zap – 1 6](#_Toc183446790)

[Figura 8: scan de segurança zap – 2 7](#_Toc183446791)

[Figura 9: scan de segurança zap – 3 7](#_Toc183446792)

[Figura 10: scan de segurança zap – 4 8](#_Toc183446793)

[Figura 11: scan de segurança zap – 5 8](#_Toc183446794)

[Figura 12: scan de segurança zap – 6 8](#_Toc183446795)

[Figura 13: scan de segurança zap – 7 9](#_Toc183446796)

[Figura 14: relatório gerado - 1 9](#_Toc183446797)

[Figura 15: relatório gerado – 2 10](#_Toc183446798)

[Figura 16: SonarQube – 1 11](#_Toc183446799)

[Figura 17: SonarQube – 2 11](#_Toc183446800)

[Figura 18: SonarQube – 3 12](#_Toc183446801)

[Figura 19: SonarQube – 4 12](#_Toc183446802)

[Figura 20: SonarQube – 5 13](#_Toc183446803)

[Figura 21: SonarQube – 6 13](#_Toc183446804)

[Figura 22: SonarQube – 7 14](#_Toc183446805)

[Figura 23: SonarQube – 8 14](#_Toc183446806)

[Figura 24: SonarQube – 9 15](#_Toc183446807)

[Figura 25: SonarQube – 10 15](#_Toc183446808)

# Lista de Tabelas

[Tabela 1: Tabela Comparativa das Principais Vulnerabilidades 2](#_Toc183446777)

[Tabela 2: vulnerabilidades 5](#_Toc183446778)

[Tabela 3: Resultados manuais e automáticos 17](#_Toc183446779)

# Introdução

Esta atividade laboratorial tem como principal objetivo compreender e aplicar os fundamentos de segurança no desenvolvimento de software. Dado o crescente impacto das vulnerabilidades em sistemas computacionais, é essencial que dominemos as práticas seguras de programação, assegurando a proteção de dados e a robustez das aplicações.

Ao longo desta atividade, iremos explorar vulnerabilidades comuns, como SQL Injection, XSS e CSRF, analisando o impacto que estas podem causar em sistemas reais. Além disso, serão introduzidas ferramentas e técnicas para auditoria e remediação de falhas, promovendo uma abordagem prática e sistemática para a melhoria contínua da segurança.

Por fim, será realizada uma análise das tendências na área de segurança de software, utilizando dados como o CWE Top 25, com o intuito de compreender a evolução das vulnerabilidades e adaptar as práticas de desenvolvimento aos desafios contemporâneos.

A aquisição destas competências visa preparar-nos para enfrentar um cenário tecnológico em constante mudança, onde a segurança desempenha um papel crucial.

Nesta atividade laboratorial utilizamos as seguintes ferramentas: O OWASP ZAP que é uma ferramenta de código aberto projetada para realizar testes de segurança em aplicações web. Focada na identificação de vulnerabilidades como SQL Injection, Cross-Site Scripting (XSS), e outras ameaças presentes na camada de interação com o utilizador, o ZAP automatiza ataques simulados e fornece relatórios detalhados sobre as fragilidades encontradas. É amplamente utilizado em processos de pentesting e como parte de pipelines CI/CD para garantir a segurança contínua de aplicações; Por outro lado, o SonarQube é uma plataforma poderosa de análise estática de código. Ele avalia a qualidade do código com base em métricas como segurança, manutenibilidade, eficiência e cobertura de testes. No contexto de segurança, o SonarQube identifica vulnerabilidades de segurança no código-fonte, como má configuração de variáveis sensíveis, e deteta vulnerabilidades relacionadas ao CWE (Common Weakness Enumeration), garantindo conformidade com padrões reconhecidos.

Esta atividade está dividida em quatro partes fundamentais, sendo estas: análise da evolução das vulnerabilidades comuns; auditoria manual de segurança; auditoria automática com o OWASP ZAP e SonarQube; e por fim ações de remediação.

## Análise da evolução das vulnerabilidades comuns

O panorama das vulnerabilidades de segurança em software está em constantemente em transformação, impulsionado pela evolução tecnológica, mudanças nas práticas de desenvolvimento e o surgimento de novas técnicas de ataque. Analisar a evolução dessas vulnerabilidades é essencial para compreender tendências, identificar áreas críticas de risco e desenvolver estratégias de mitigação mais eficazes.

Uma das principais referências para essa análise é o CWE Top 25, uma lista atualizada anualmente que destaca as vulnerabilidades de software mais frequentes e críticas, como SQL Injection, Cross-Site Scripting (XSS) e falhas de validação de entradas. Esta lista, baseada em dados reais, fornece uma visão abrangente das ameaças emergentes e persistentes, permitindo que desenvolvedores e organizações priorizem seus esforços de segurança.

Para tal criámos uma tabela comparativa, com os dados que baixamos do site <https://cwe.mitre.org/>, como podemos ver na **Tabela 1**.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CWE | Vulnerabilidade | Descrição | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 |
| CWE-119 | Buffer Overflow | Operações de memória que podem ler ou escrever fora dos limites do buffer | 1 | 1 | 1 | 1 | 1 | 1 |
| CWE-79 | Cross-site Scripting (XSS) | Falha na validação de entrada que permite injeção de scripts maliciosos | 2 | 4 | 3 | 3 | 2 | 2 |
| CWE-89 | SQL Injection | Inserção de comandos SQL maliciosos por meio de entradas não validadas | 6 | 6 | 7 | 6 | 3 | 3 |
| CWE-200 | Information Exposure | Exposição não intencional de informações sensíveis | 4 | 3 | 5 | 4 | 4 | 4 |
| CWE-125 | Out-of-bounds Read | Leitura de dados fora dos limites do buffer pretendido | 5 | 5 | 4 | 5 | 5 | 5 |
| CWE-416 | Use After Free | Uso de memória após a sua liberação | 7 | 7 | 6 | 7 | 6 | 6 |
| CWE-78 | OS Command Injection | Injeção de comandos do sistema operativo | 11 | 11 | 13 | 11 | 7 | 7 |
| CWE-287 | Authentication Issues | Problemas na verificação de identidade | 13 | 15 | 14 | 12 | 8 | 8 |

Tabela 1: Tabela Comparativa das Principais Vulnerabilidades

### Análise de Tendências:

**Vulnerabilidades Persistentes**

* Buffer Overflow (CWE-119) manteve consistentemente a primeira posição ao longo dos anos, demonstrando que problemas fundamentais de gestão de memória continuam a ser críticos.
* Cross-site Scripting (CWE-79) e SQL Injection (CWE-89) permaneceram consistentemente no top 10, indicando que vulnerabilidades relacionadas à validação de entrada continuam a ser um desafio significativo.

**Tendências Emergentes**

* Authentication Issues (CWE-287) mostrou uma tendência de subida constante, movendo-se da posição 13 em 2019 para a posição 8 em 2024, refletindo a crescente importância da segurança de autenticação.
* OS Command Injection (CWE-78) também demonstrou uma tendência de ascensão, especialmente nos últimos dois anos.

### Mudanças Significativas

* Houve uma maior ênfase em vulnerabilidades relacionadas à autenticação e gestão de sessões nos últimos anos.
* Vulnerabilidades relacionadas com vazamento de informações mantiveram-se consistentemente importantes.

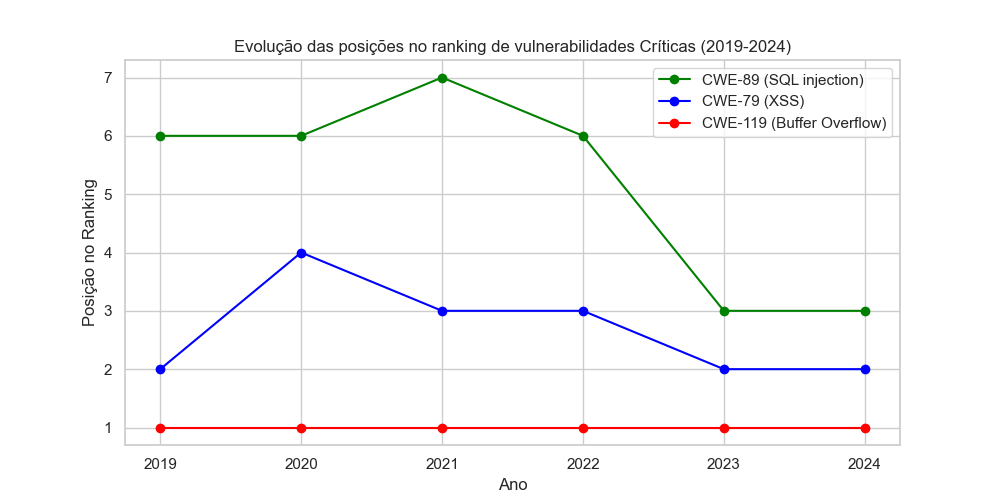


Figura 1: gráfico das mudanças significativas

*Explicação do gráfico da* **Figura 1***:*

Assim sendo e resumindo o **CWE-119** (Buffer Overflow) lidera consistentemente o ranking de vulnerabilidades mais perigosas entre 2019 e 2024, refletindo a complexidade de mitigar este problema em sistemas críticos. Vulnerabilidades relacionadas à validação de entradas de utilizadores, como o **CWE-79** (XSS) e o **CWE-89** (SQL Injection), permanecem críticas devido à sua ampla exploração em aplicações web modernas.

A subida do **CWE-287** (Authentication Issues) demonstra a relevância crescente de falhas de autenticação, provavelmente devido ao aumento de sistemas distribuídos e APIs. Já vulnerabilidades como o **CWE-78** (OS Command Injection) mostram sinais de melhoria, indicando uma adoção mais ampla de práticas seguras.

# Métodos

## Auditoria manual de segurança

A segurança no desenvolvimento de software é um dos pilares fundamentais para garantir a proteção de dados e a integridade de sistemas em ambientes digitais.

Esta análise tem como objetivo avaliar o código fornecido sob a perspetiva de práticas seguras de programação, identificando potenciais vulnerabilidades e propondo melhorias para mitigar riscos.

A análise será conduzida com foco em quatro aspetos cruciais, sendo dado um exemplo em cada uma delas:

* Validação e sanitização de entradas do utilizador, para evitar ataques como SQL Injection e outras formas de exploração de dados manipulados.
  + **Análise:** O código insere diretamente os dados fornecidos pelo utilizador nas consultas SQL utilizando **f-strings**. Isso deixa a aplicação vulnerável a ataques de **SQL Injection**, como podemos ver na **Figura 2**.

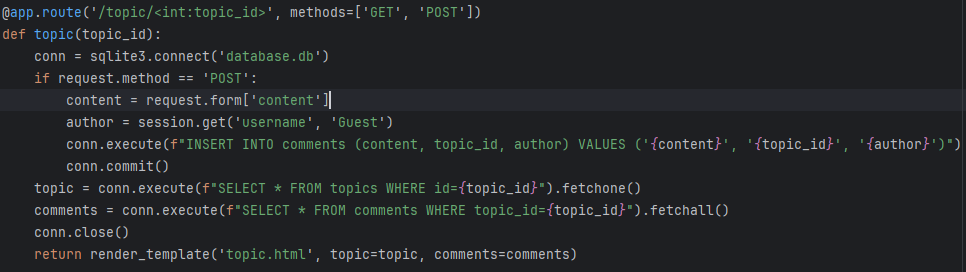


Figura 2: validação e sanitização de entradas do utilizador

* Controlo de acesso a recursos, assegurando que utilizadores autenticados acedem apenas a recursos para os quais têm permissão.
  + **Análise:** Algumas funcionalidades, como a criação de tópicos (/new\_topic) ou edição de comentários/tópicos, verificam o acesso, verificando a presença do utilizador na sessão. Neste exemplo, não garante que o utilizador tenha permissão para modificar ou eliminar recursos (ex.: editar um tópico de outro autor), (**Figura 3**).

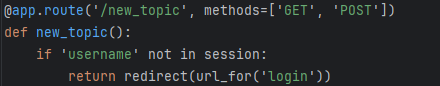


Figura 3: Controlo de acesso a recursos

* Gestão de sessões e cookies, com especial atenção à proteção de informações armazenadas na sessão e à configuração de cookies.
  + **Análise:** A secret\_key está armazenada diretamente no código, tornando-a vulnerável caso o repositório de código for acedido por terceiros (**Figura 4**).



Figura 4: Gestão de sessões e cookies

* Armazenamento e exposição de dados sensíveis, como passwords, enfatizando a necessidade de técnicas de proteção robustas para evitar a sua exposição**.**
  + **Análise:** As passwords são armazenadas como texto simples na base de dados, expondo os utilizadores em caso de violação de segurança (**Figura 5**).

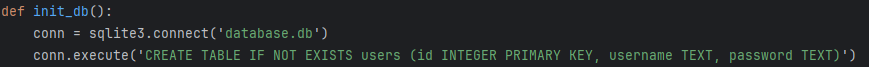


Figura 5: Armazenamento e exposição de dados sensíveis

Com esta abordagem, pretende-se identificar fragilidades no código e propor soluções que fortaleçam a sua resiliência a ameaças, promovendo um desenvolvimento mais seguro e alinhado com boas práticas de segurança, realizamos uma tabela para ser mais percetível a visualização da vulnerabilidade, (**Tabela 2**).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vulnerabilidade | CWE | Impacto | Probabilidade de Exploração | Proposta Inicial de Remediação |
| Uso de f-strings para consultas SQL  app.py, linha 32 | CWE-89 | SQL Injection compromete a integridade dos dados. (**Alto**) | **Alto** | Substituir consultas SQL diretas por prepared statements ou consultas parametrizadas. |
| Falta de controlo de acesso  app.py, linha 62 a 66 | CWE-862 | Usuários podem modificar recursos de outros. (**Alto**) | **Médio** | Implementar verificações de permissões para garantir que o utilizador só pode modificar os seus recursos. |
| secret\_key exposta no código  app.py, linha 6 | CWE-312 | Possibilidade de roubo e manipulação de sessões. (**Médio**) | **Alto** | Armazenar a secret\_key em variáveis de ambiente e carregá-la dinamicamente no código. |
| Armazenamento de passwords em texto simples  app.py, linha 109 | CWE-256 | Passwords facilmente comprometidas em caso de violação. (**Médio**) | **Médio** | Utilizar hashing seguro, como bcrypt ou argon2, para armazenar passwords de forma protegida. |

Tabela 2: vulnerabilidades

## Auditoria automática com o OWASP ZAP e SonarQube

A auditoria de segurança automatizada é uma etapa fundamental no desenvolvimento de software para identificar vulnerabilidades e melhorar a qualidade do código de forma eficiente e contínua.

Ferramentas como o OWASP ZAP (Zed Attack Proxy) e o SonarQube desempenham papéis complementares nesse processo, permitindo uma análise abrangente de aplicações web.

Ao integrar o OWASP ZAP e o SonarQube, é possível obter uma abordagem abrangente para auditoria de aplicações, combinando a análise dinâmica (testes em execução) com a análise estática (código-fonte), otimizando tanto a identificação quanto a mitigação de riscos. Este processo melhora a resiliência das aplicações e promove a entrega de software seguro e confiável.

Para tal começamos por instalar e configurar as ferramentas OWASP ZAP e SonarQube, como utilizámos o NixOs, então realizamos o código apresentado na **Figura 6**.

Figura 6: instalação do OWASP Zap e SonarQube no NixOS

environment.systemPackages = with pkgs; [

zap

sonar-scanner-cli

];

### OWASP ZAP

Seguidamente realizamos o próximo passo que envolve duas etapas principais: realizar o scan de segurança e analisar o relatório gerado.

Para realizar o scan de segurança foi necessário identificar vulnerabilidades na aplicação, como SQL Injection, Cross-Site Scripting (XSS), falhas de autenticação ou configuração inadequada de segurança, para tal realizamos um conjunto de passos, como mostra na **Figura 7** à **Figura 13**.

Persistência da sessão zap:

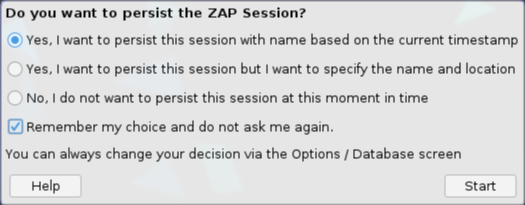


Figura 7: scan de segurança zap – 1

Janela default do zap:

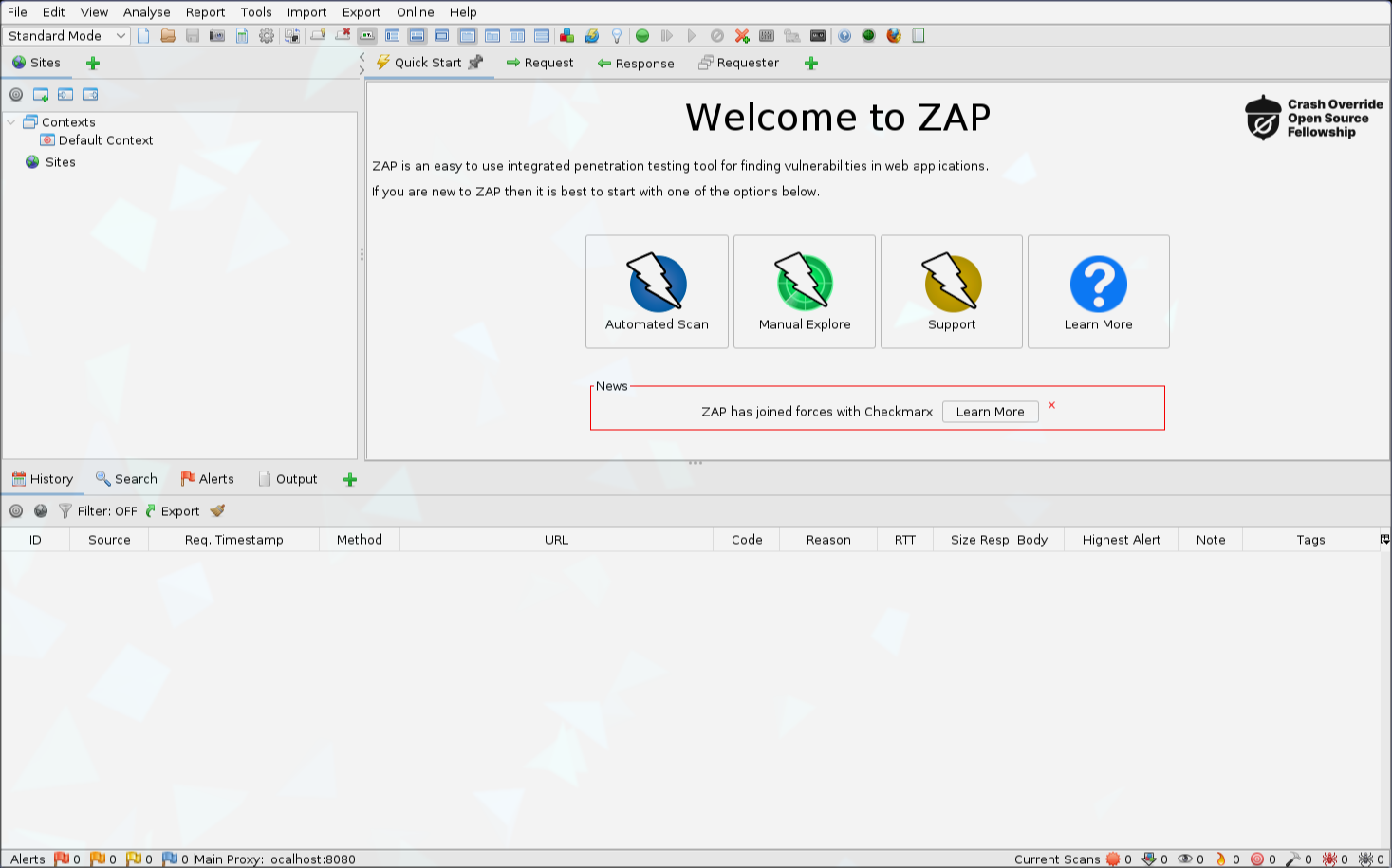


Figura 8: scan de segurança zap – 2

Introdução do url para atacar:

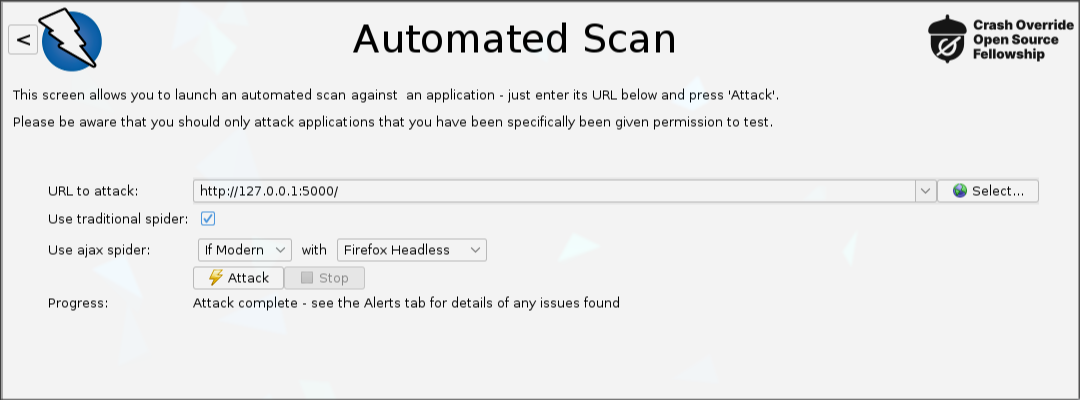


Figura 9: scan de segurança zap – 3

Spider:

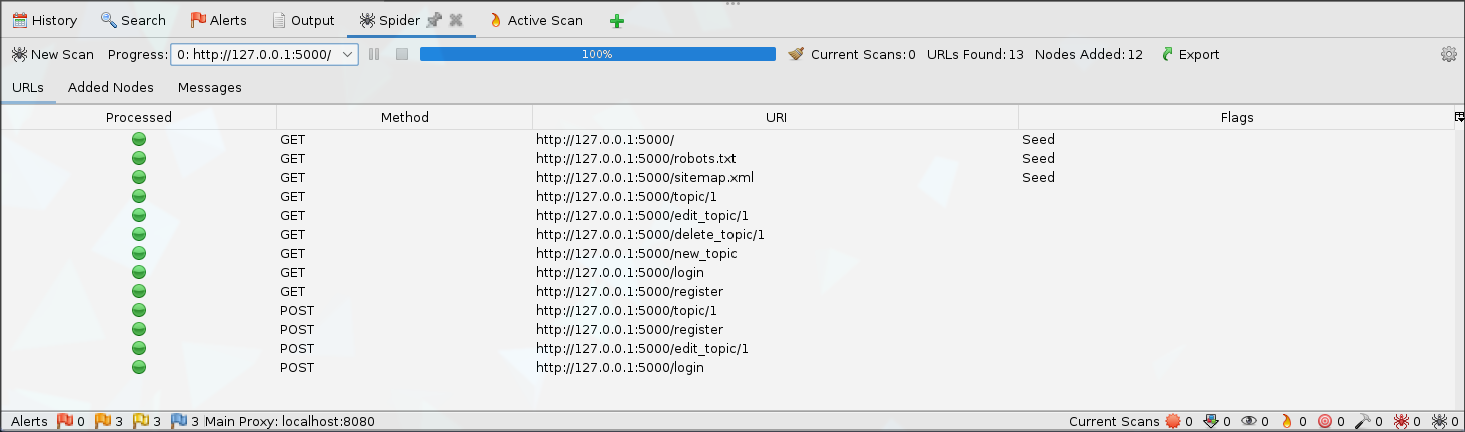


Figura 10: scan de segurança zap – 4

Active Scans:

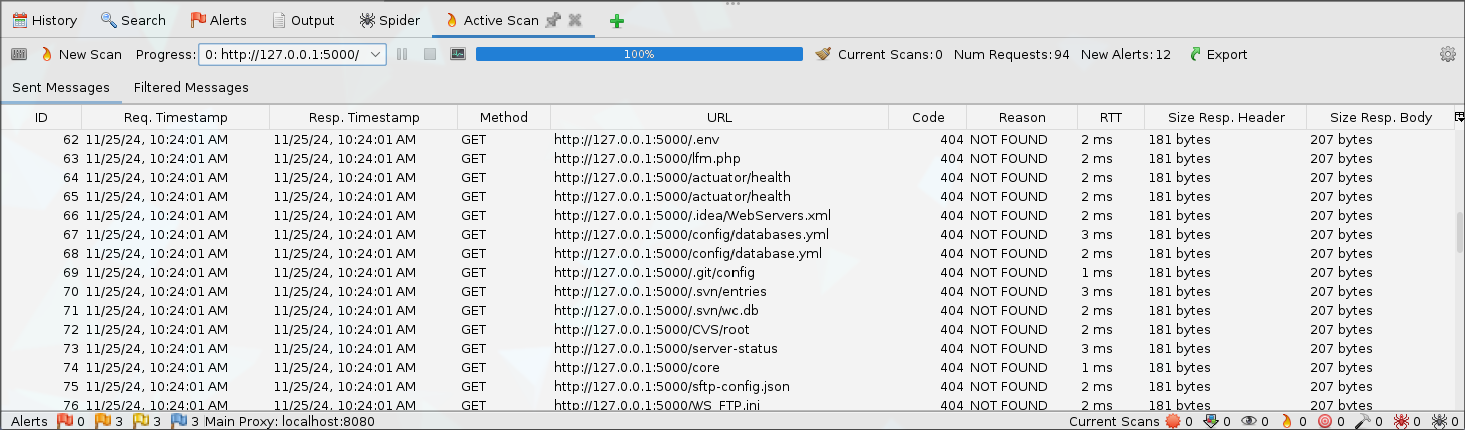


Figura 11: scan de segurança zap – 5

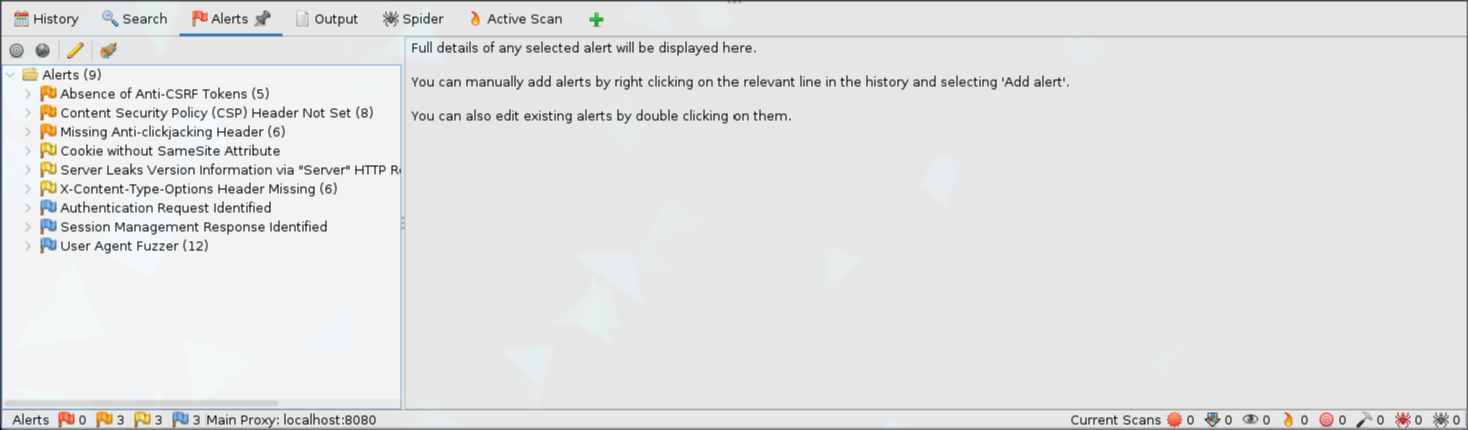
Alertas:

Figura 12: scan de segurança zap – 6

Firefox proxy settings:

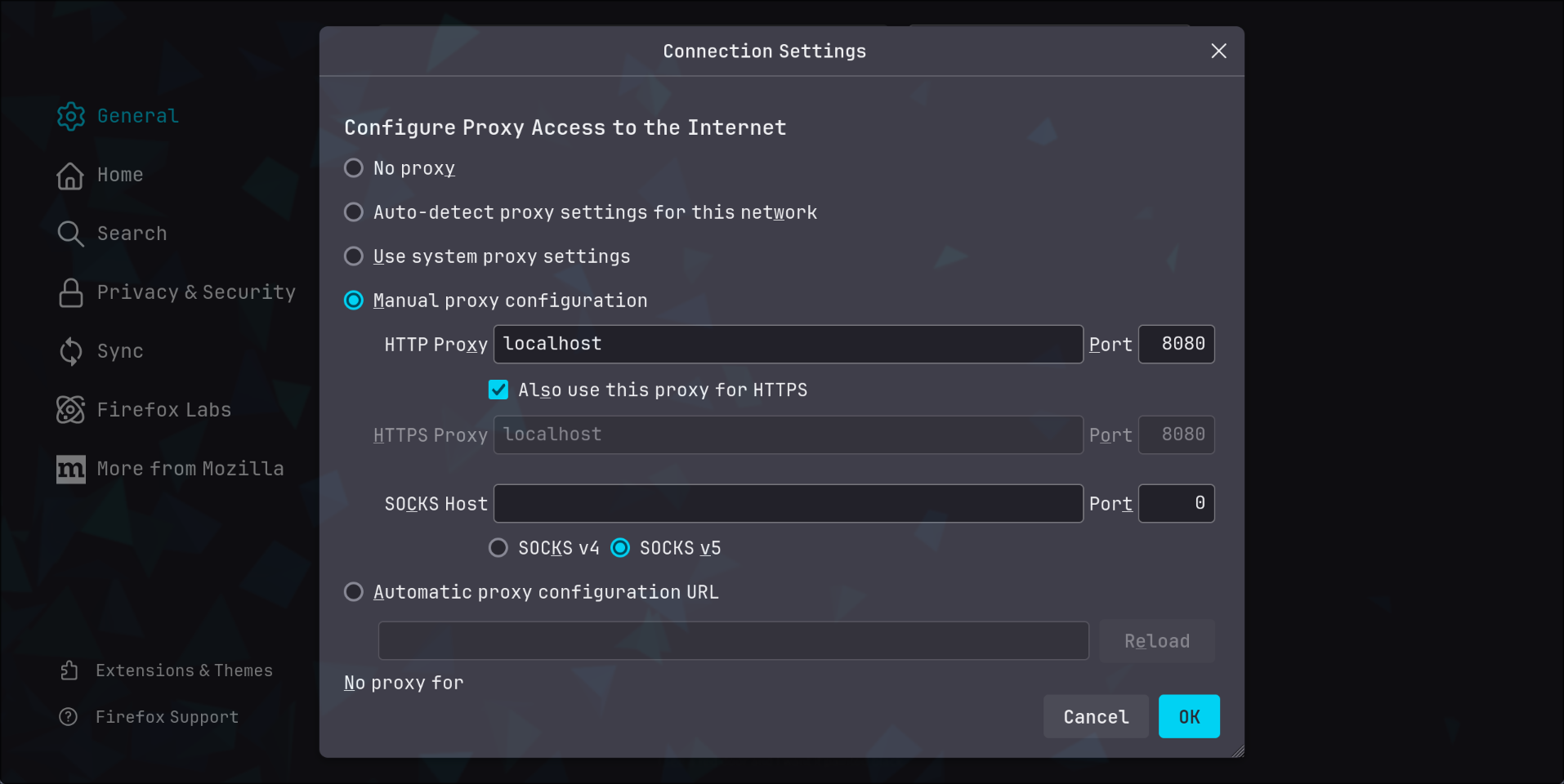


Figura 13: scan de segurança zap – 7

Após o scan de segurança, o OWASP ZAP geramos um relatório que contem todas as vulnerabilidades detetadas durante o processo de análise. Para exportar o relatório (**Figura 14** e **Figura 15**):

Relatório gerado:

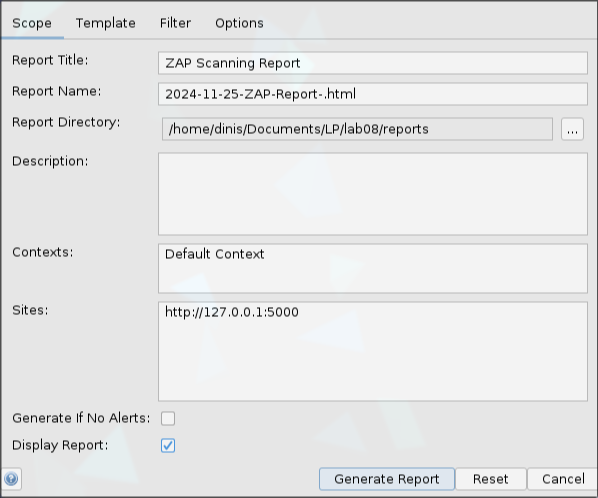


Figura 14: relatório gerado - 1

Resultado:



Figura 15: relatório gerado – 2

### Análise do relatório gerado:

**Vulnerabilidades:**

* SQL Injection (SQLi): Vulnerabilidade onde comandos SQL maliciosos podem ser injetados em consultas, comprometendo a integridade do banco de dados.
* Cross-Site Scripting (XSS): Quando um atacante injeta scripts maliciosos em páginas web vistas por outros usuários, permitindo o roubo de informações sensíveis ou execução de ações não autorizadas.
* Cross-Site Request Forgery (CSRF): Exploração onde o atacante induz um usuário autenticado a realizar ações indesejadas em um site.
* Broken Authentication: Falhas na autenticação de usuários, podendo permitir o acesso não autorizado.
* Sensitive Data Exposure: Dados sensíveis não são adequadamente protegidos, expondo informações como senhas e dados financeiros.

**Gravidade:**

* Alta: Exposição de dados sensíveis, falhas críticas de autenticação, etc.
* Média: Vulnerabilidades como XSS, onde a exploração é possível, mas geralmente exige interação do usuário.
* Baixa: Problemas de segurança menos críticos, como cabeçalhos HTTP mal configurados.

**Localização da Vulnerabilidade:**

O relatório informa onde a vulnerabilidade foi detetada, como:

* Endpoint: URL ou página afetada pela vulnerabilidade.
* Parâmetros: Quais parâmetros ou campos da aplicação estão expostos a riscos (exemplo: username, password em um formulário de login).

### SonarQube

O objetivo desta etapa é utilizar o SonarQube para realizar uma auditoria completa do código da aplicação fornecida. A análise concentra-se na identificação de vulnerabilidades críticas, como injeções de código, e problemas relacionados à manutenibilidade e eficiência do software. O relatório gerado será examinado detalhadamente, contribuindo para a priorização de correções e reforço da segurança e qualidade do projeto, podemos ver a realização desta etapa, na **Figura 16** à **Figura 25**.

Página inicial do SonarQube, (username: Admin; password: Admin123):

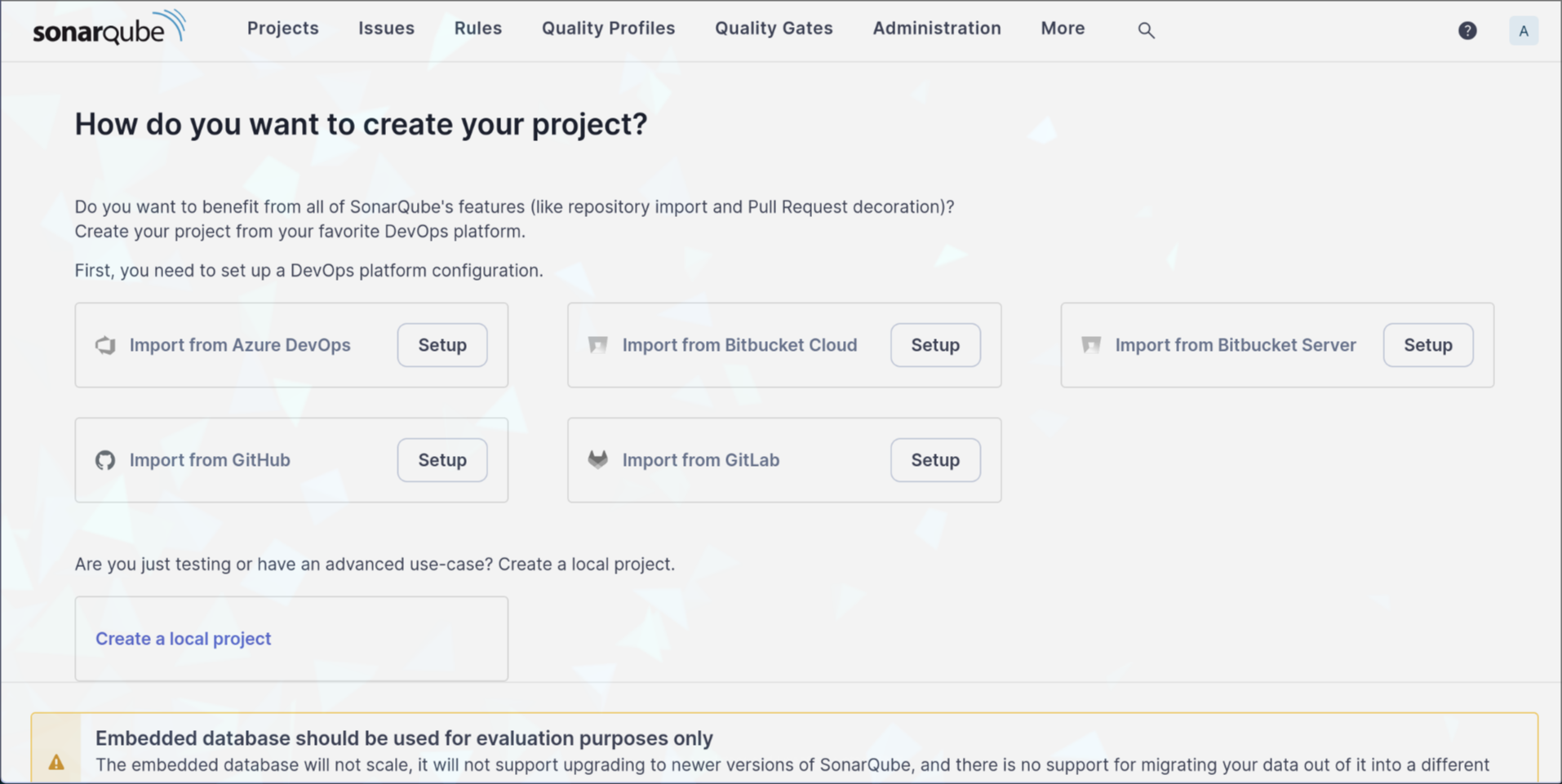


Figura 16: SonarQube – 1

De seguida criamos um projeto:

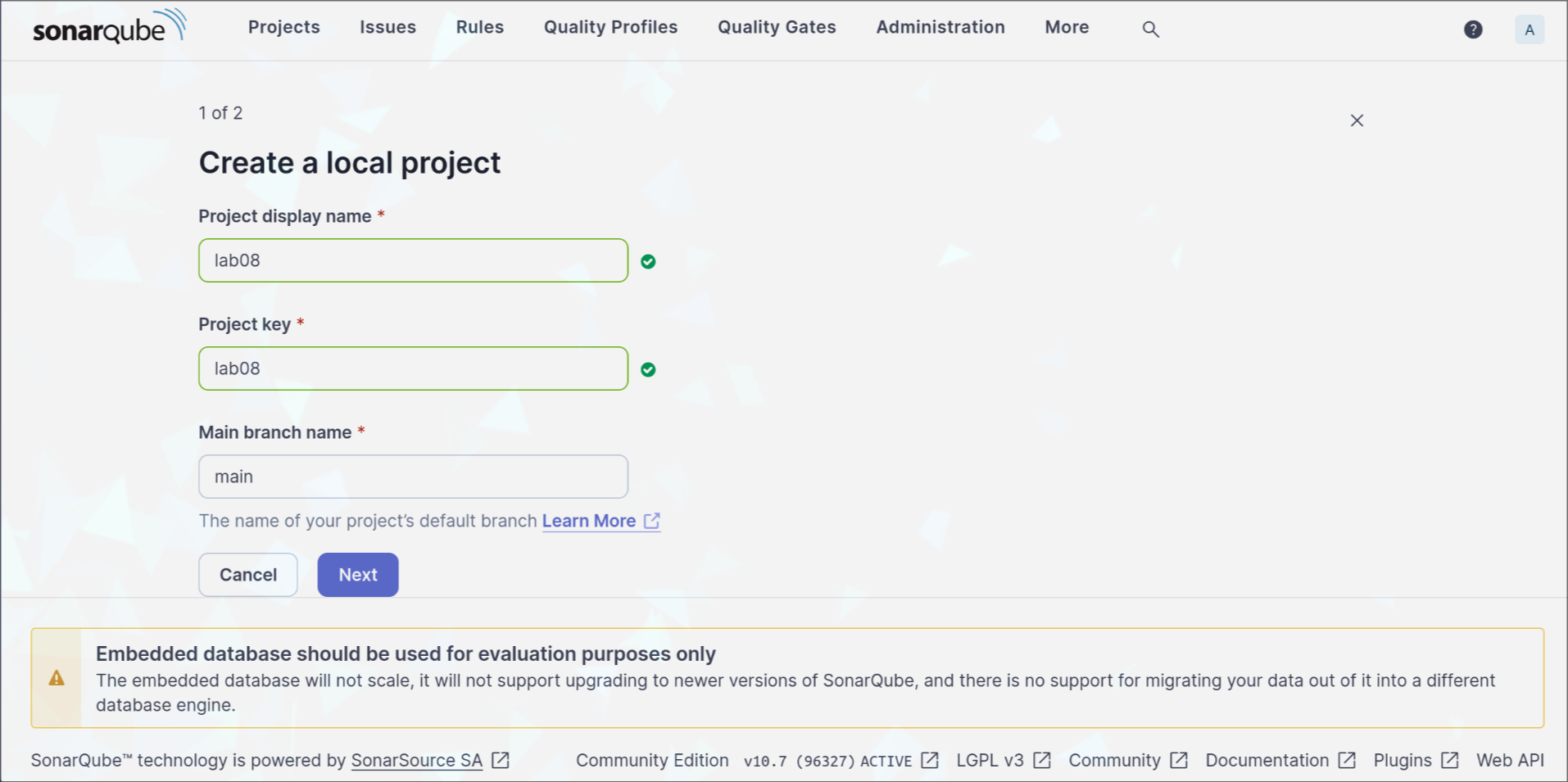


Figura 17: SonarQube – 2

Posteriormente escolhemos global settings:

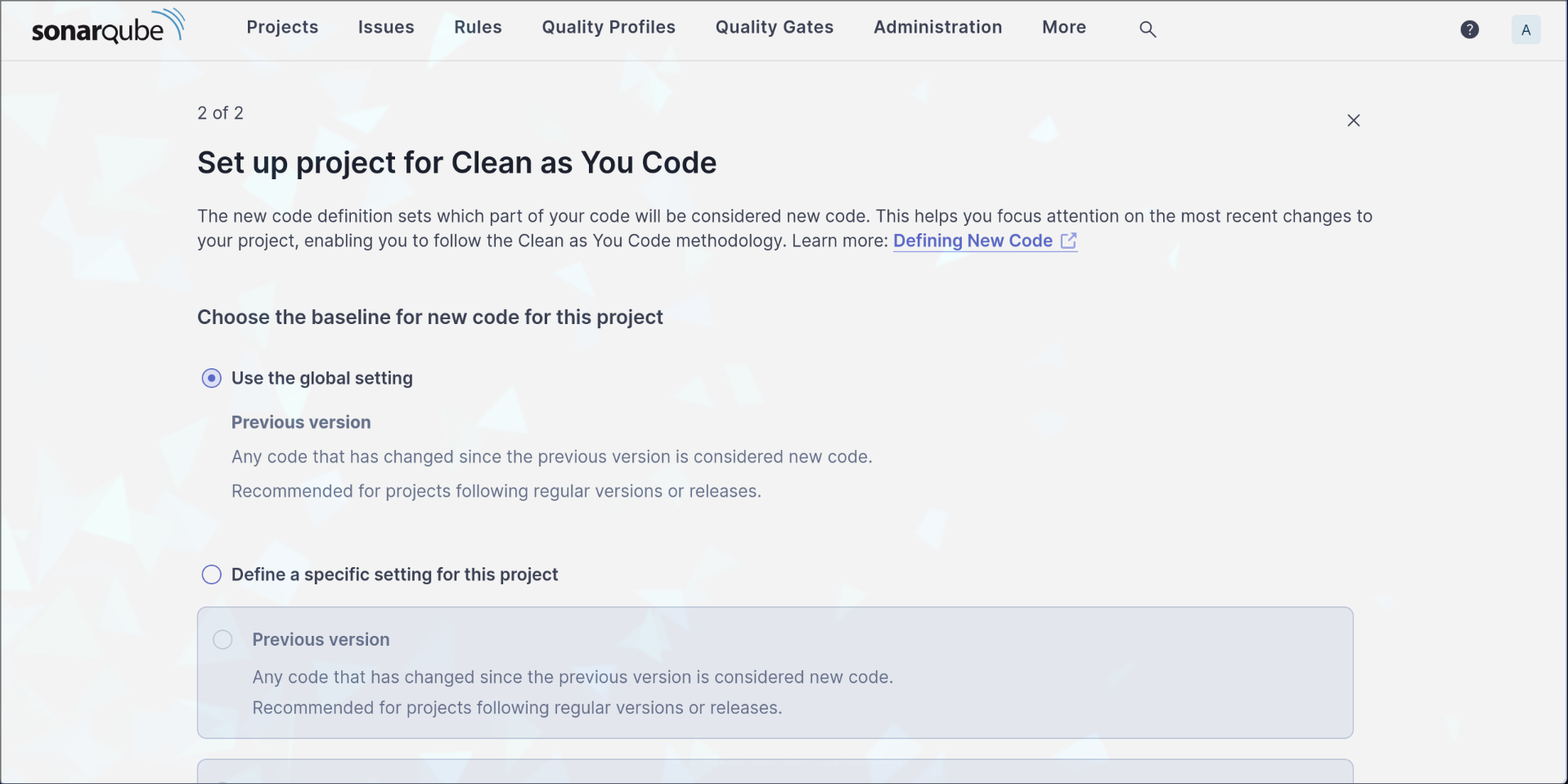


Figura 18: SonarQube – 3

Seguidamente escolhemos a opção Locally:

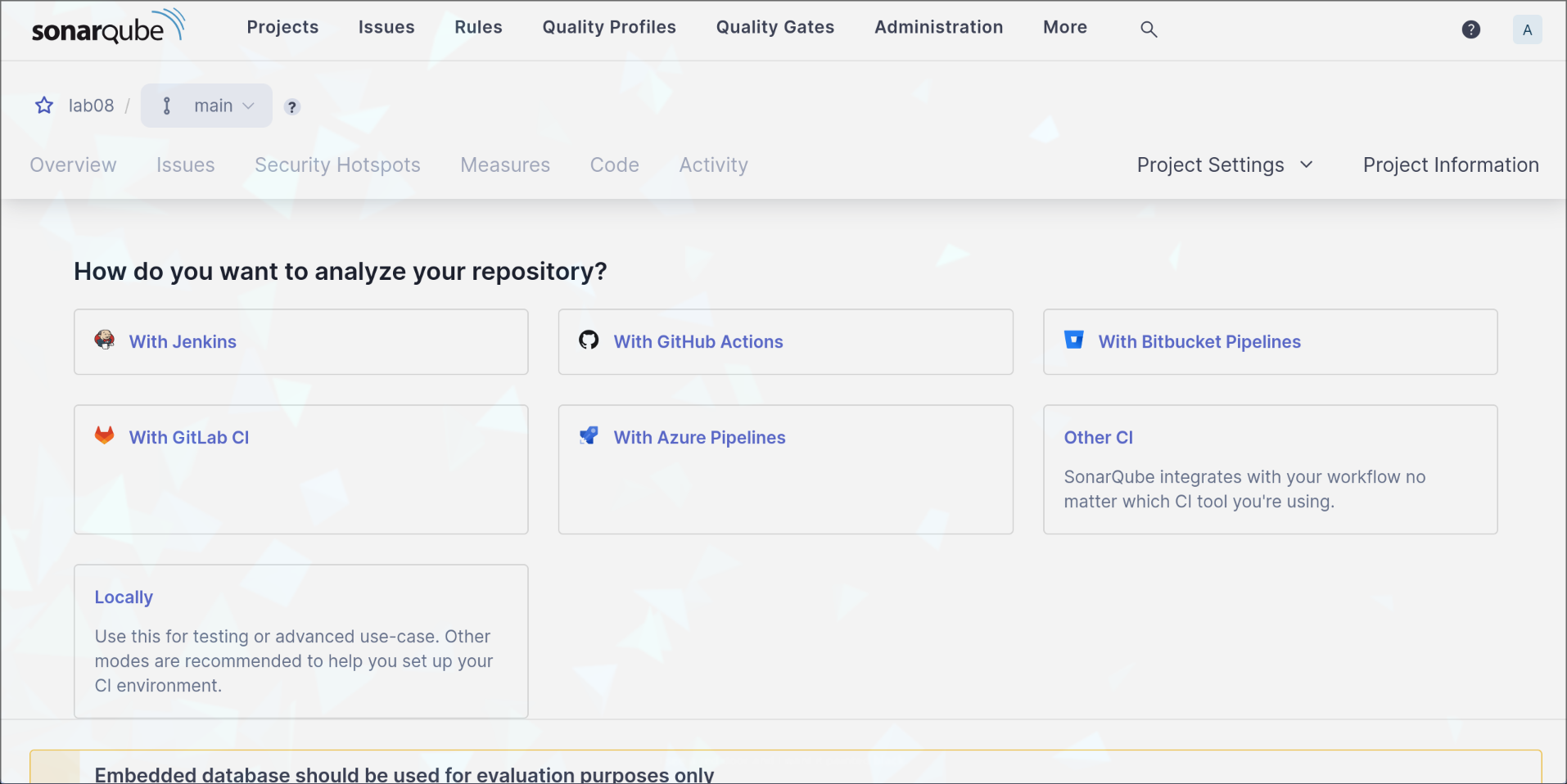


Figura 19: SonarQube – 4

Opção generate:

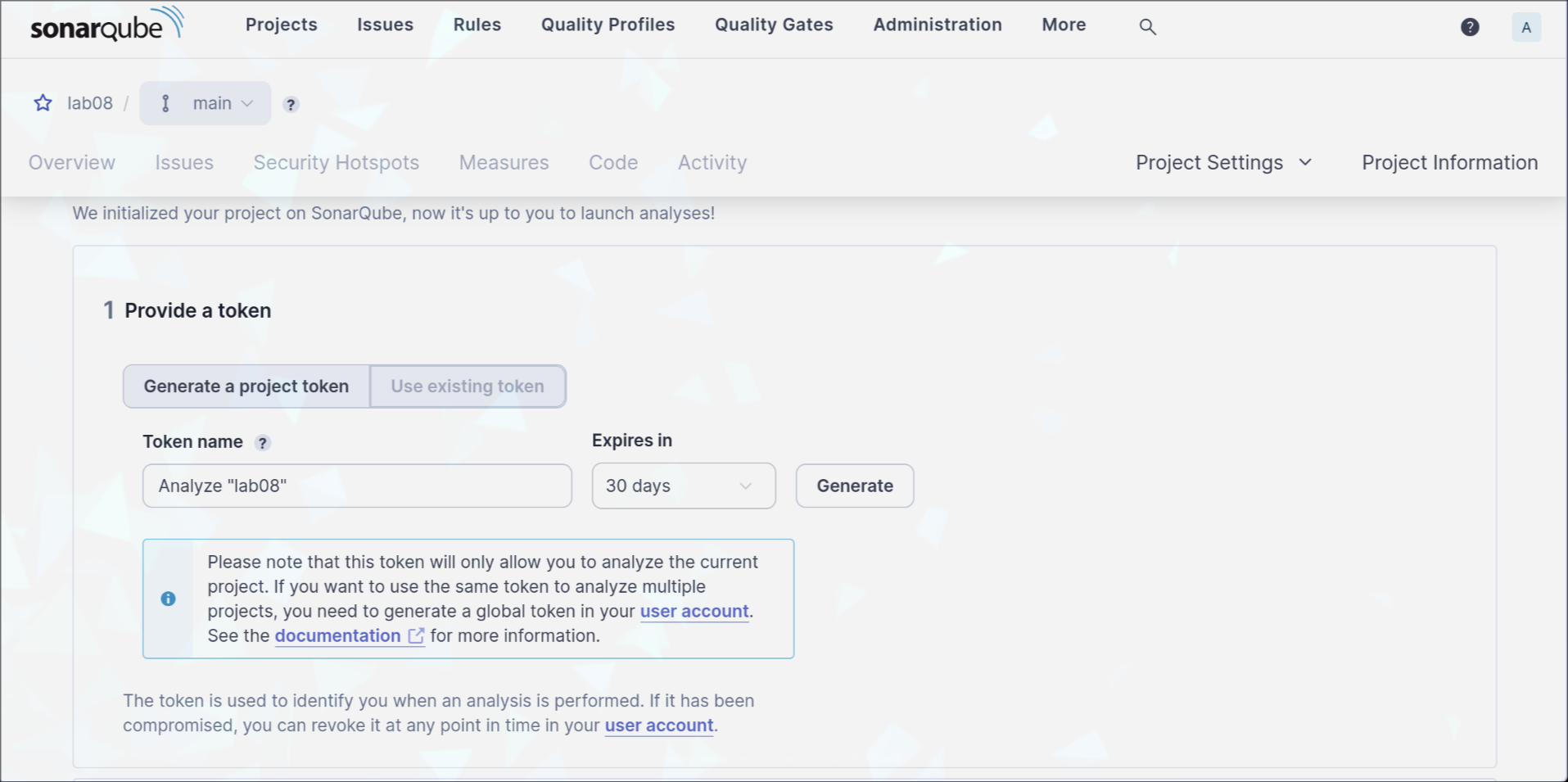


Figura 20: SonarQube – 5

Opção continue:

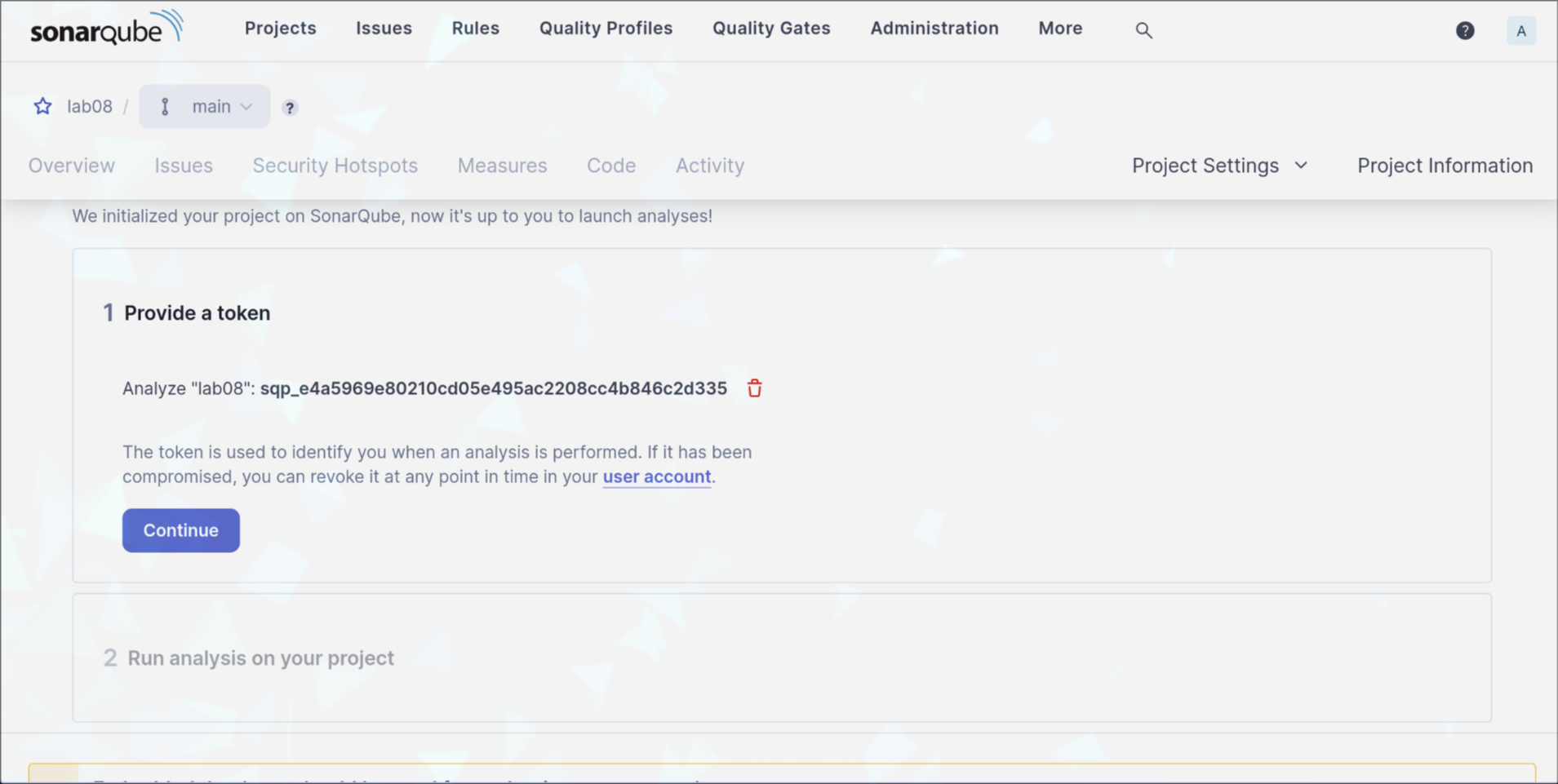


Figura 21: SonarQube – 6

Escolher other:

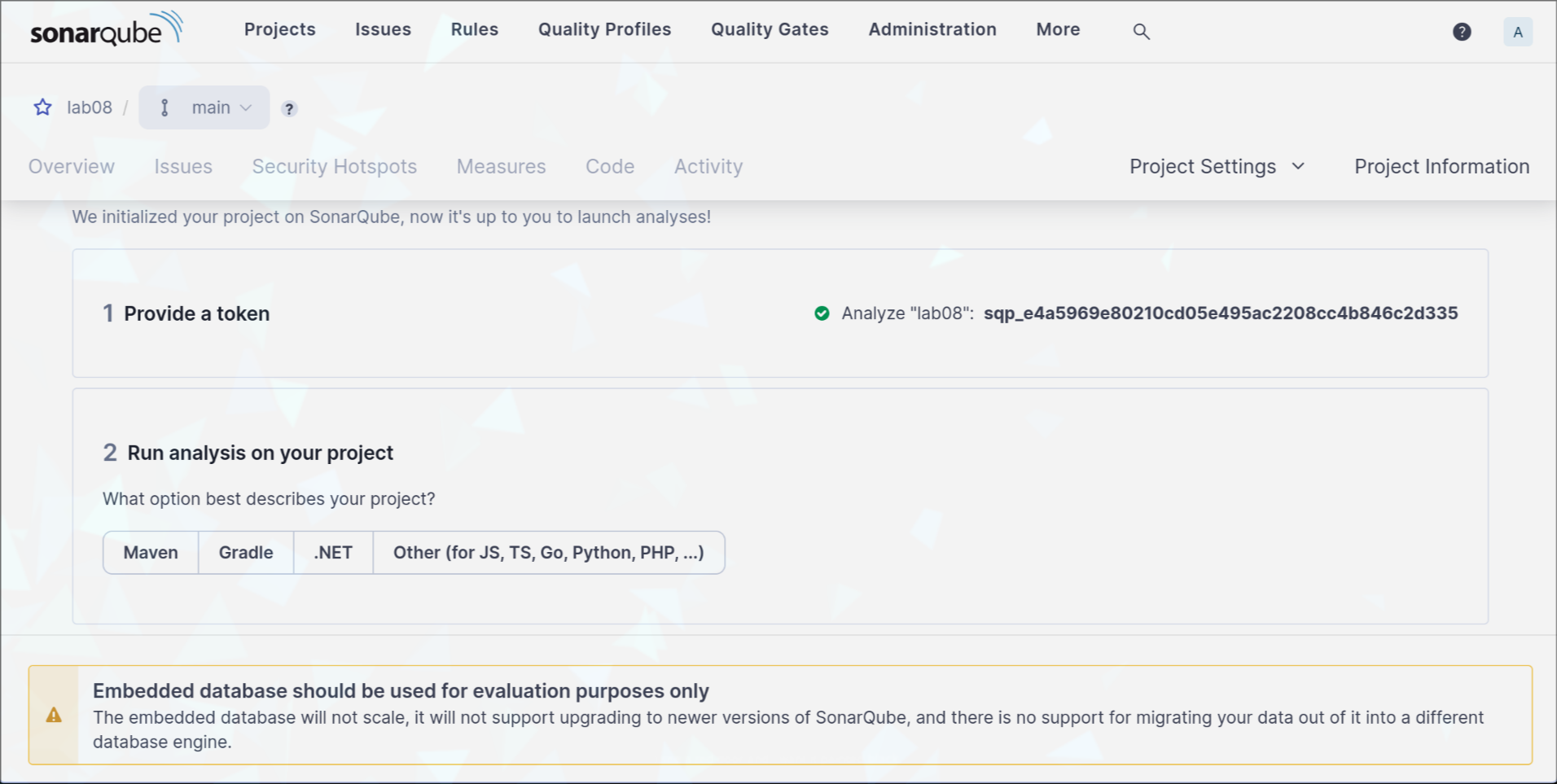


Figura 22: SonarQube – 7

Escolher o sistema, copiar e correr o código providenciado, o scanner processará os arquivos do projeto e enviará os dados para o servidor do SonarQube.:

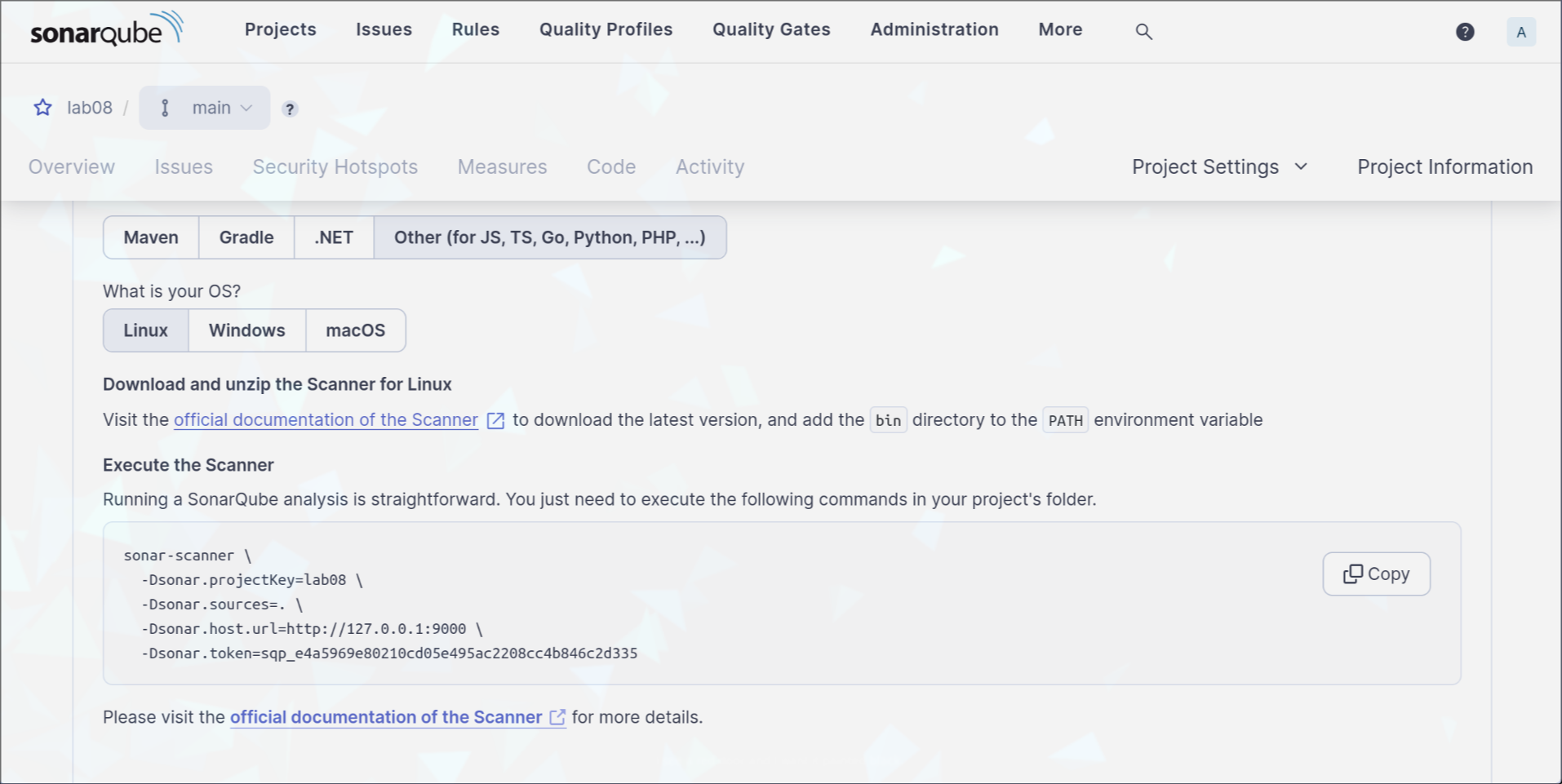


Figura 23: SonarQube – 8

Execução do código:

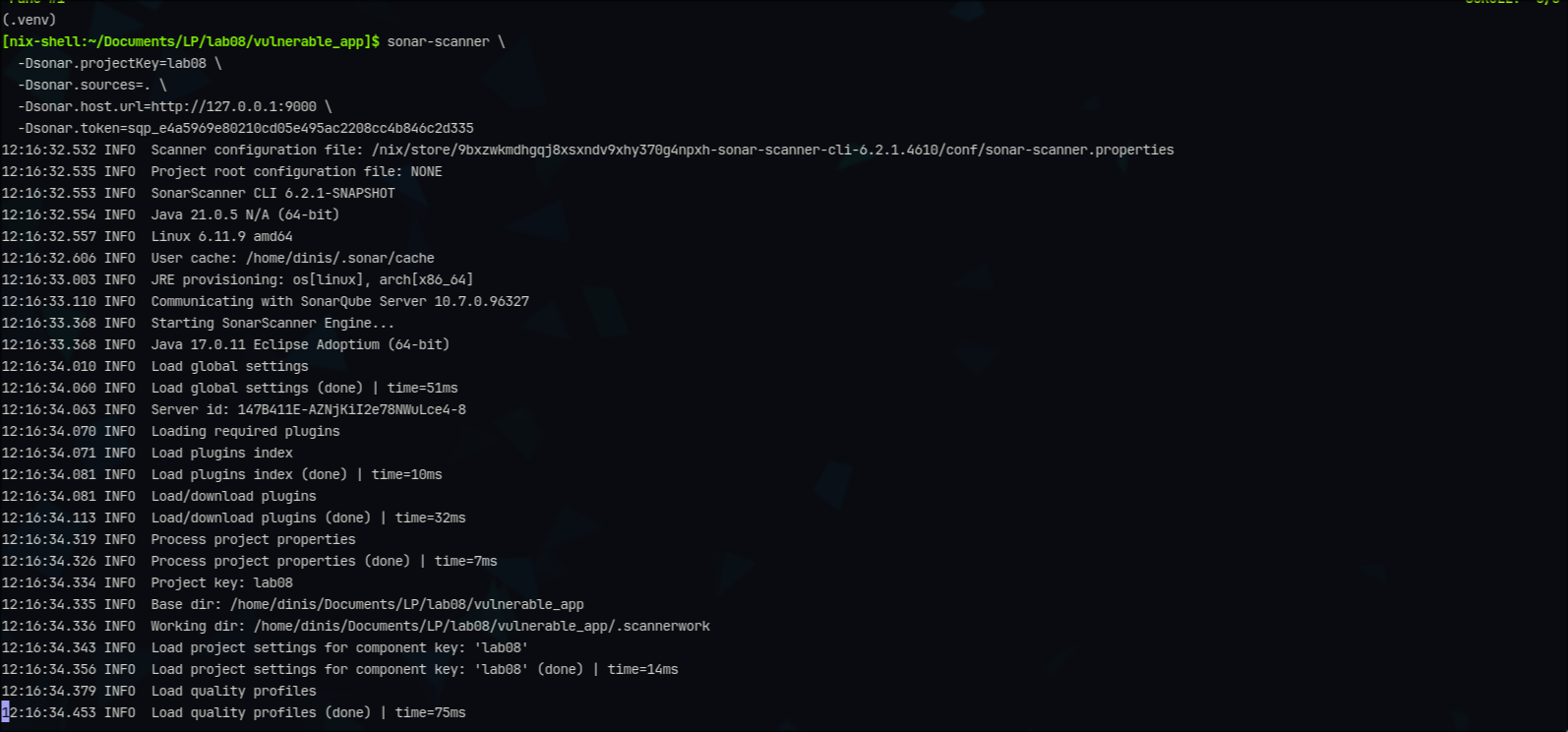


Figura 24: SonarQube – 9

Análise concluída:

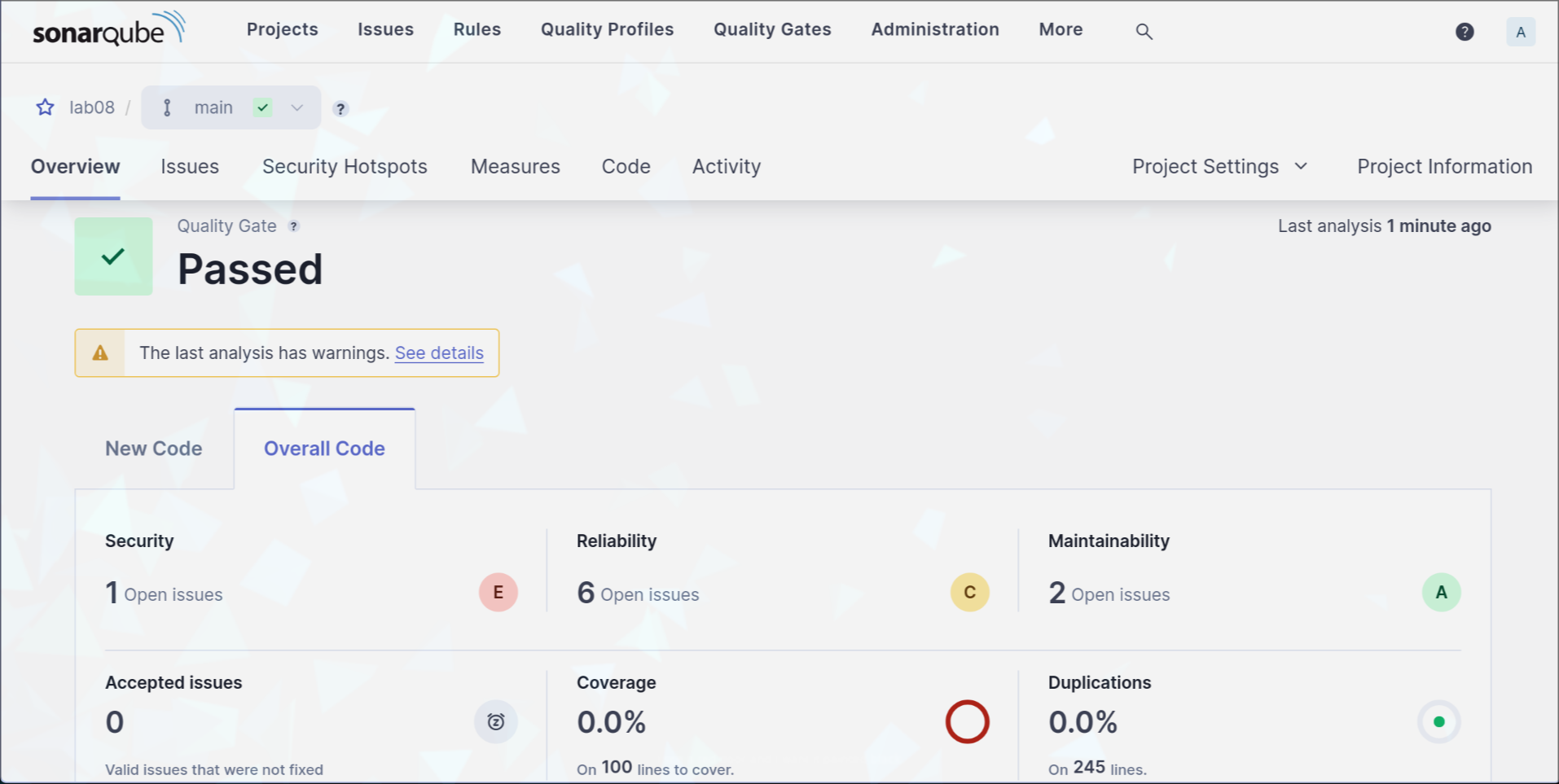


Figura 25: SonarQube – 10

Após a execução do **SonarScanner**, o relatório gerado pelo SonarQube apresenta uma visão detalhada sobre as vulnerabilidades de segurança e problemas de qualidade encontrados no código-fonte. A análise é dividida em duas categorias principais: **Segurança** e **Qualidade do Código**

**Segurança:**

SQL Injection:

* Localização: Função de login no endpoint */login*.
* Descrição: Strings concatenadas diretamente nas consultas SQL sem parametrização. Isso expõe a aplicação a injeções de código malicioso.
* Gravidade: Alta.

Cross-Site Scripting (XSS):

* Localização: Campos de comentário no endpoint */topic/<int:topic\_id>.*
* Descrição: Dados de entrada do utilizador são exibidos sem escape adequado, permitindo a execução de scripts maliciosos.
* Gravidade: Alta.

Armazenamento de Senhas em Texto Puro:

* Localização: Função de registo no endpoint */register*.
* Descrição: As senhas são armazenadas diretamente no banco de dados sem hash ou encriptação, tornando-as vulneráveis em caso de vazamento.
* Gravidade: Alta.

Validação Inadequada de Entradas:

* Localização: Campos de entrada em vários endpoints *(/login, /register, /new\_topic).*
* Descrição: Falta de validação robusta nos dados de entrada, permitindo que utilizadores enviem valores inesperados.
* Gravidade: Média.

**Problemas de Qualidade do Código**

**Duplicação de Código**

* **Localização**: Métodos em */edit\_comment e /edit\_topic*.
* **Descrição**: Blocos de código repetidos entre funções de edição de comentários e tópicos.
* **Impacto**: Médio.

**Uso Insuficiente de Comentários**

* **Localização**: Código crítico, como operações em base de dados.
* **Descrição**: Ausência de comentários explicativos, dificultando a compreensão do funcionamento.
* **Impacto**: Baixo.

**Dependências Desatualizadas**

* **Localização**: Arquivo de dependências do projeto.
* **Descrição**: Uso de versões antigas de bibliotecas, que podem conter vulnerabilidades conhecidas.
* **Impacto**: Médio.

Por fim comparamos os resultados das auditorias manuais e automáticas, evidenciando: Vulnerabilidades identificadas em comum; Vulnerabilidades exclusivas de cada método; Falsos positivos ou falsos negativos detetados pelas ferramentas, como podemos ver na **Tabela 3**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Vulnerabilidade** | **Comuns** | **Manual** | **Automática** | **Falsos Positivo/Negativo** |
| ****SQL Injection**** | Sim | Não | Não | Nenhum |
| ****Cross-Site Scripting (XSS)**** | Sim | Não | Não | Nenhum |
| ****Armazenamento de Senhas em Texto Puro**** | Sim | Não | Não | Nenhum |
| ****Validação Inadequada de Entradas**** | Não | Sim | Não | Falso Negativo na Automática |
| ****Dependências Desatualizadas**** | Não | Não | Sim | Nenhum |
| ****Duplicação de Código**** | Não | Não | Sim | Nenhum |
| ****Complexidade Ciclomática Alta**** | Não | Não | Sim | Nenhum |
| ****Uso Insuficiente de Comentários**** | Não | Não | Sim | Nenhum |

Tabela 3: Resultados manuais e automáticos

# Resultados e Discussões

## Ações de remediação

Após a identificação das vulnerabilidades no sistema por meio de auditorias manuais e automáticas, torna-se essencial implementar ações de remediação para mitigar os riscos associados. Estas ações têm como objetivo não apenas corrigir falhas identificadas, mas também reforçar a segurança e melhorar a qualidade do código, prevenindo futuras vulnerabilidades.

Nesta etapa, foram propostas estratégias para remediar os problemas encontrados, alinhadas às melhores práticas de segurança e desenvolvimento seguro, incluindo:

* A aplicação de validação robusta de entradas.
* A adoção de técnicas seguras para armazenamento de dados sensíveis.
* Corrigir problemas de controlo de acessos e gestão de sessões

Estas medidas visam aumentar a resiliência do sistema contra ataques e garantir a conformidade com padrões de segurança amplamente aceitos.

**SQL Injection**

Problema:

* Consultas SQL utilizavam strings concatenadas diretamente com entradas do utilizador.
* Isso permite que um atacante injete comandos SQL maliciosos, comprometendo a integridade e confidencialidade da base de dados.

Solução:

* Utilização de instruções parametrizadas com ?, fornecidas pela biblioteca SQLite, para evitar injeções SQL.

Código:

# Antes

*topic = conn.execute(f"SELECT \* FROM topics WHERE id={topic\_id}").fetchone()*

*comments = conn.execute(f"SELECT \* FROM comments WHERE topic\_id={topic\_id}").fetchall()*

# Depois

*topic = conn.execute("SELECT \* FROM topics WHERE id = ?", (topic\_id,)).fetchone()*

*comments = conn.execute("SELECT \* FROM comments WHERE topic\_id = ?", (topic\_id,)).fetchall()*

**Exposição de Informações Sensíveis**

Problema:

* Senhas dos utilizadores eram armazenadas em texto simples na base de dados, expondo os dados a um risco elevado caso a base de dados fosse comprometida.

Soluções:

* Implementação de hashing das senhas usando bcrypt antes de armazená-las.

Código:

#Antes

*conn.execute(f"INSERT INTO users (username, password) VALUES ('{username}', '{password}')")*

#Depois

*hashed\_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())*

*conn.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, hashed\_password))*

**Controlo de Acesso Inadequado**

Problema:

* Qualquer utilizador, autenticado ou não, podia aceder e modificar recursos críticos, como tópicos e comentários, mesmo que não fosse o autor.

Soluções:

* Implementação de verificações de controle de acesso, garantindo que apenas o autor do recurso possa modificá-lo ou excluí-lo.

Código:

#Antes

*@app.route('/edit\_topic/<int:topic\_id>', methods=['GET', 'POST'])*

*def edit\_topic(topic\_id):*

*conn = sqlite3.connect('database.db')*

*if request.method == 'POST':*

*title = request.form['title']*

*content = request.form['content']*

*conn.execute(f"UPDATE topics SET title='{title}', content='{content}' WHERE id={topic\_id}")*

*conn.commit()*

*conn.close()*

*return redirect(url\_for('index'))*

#Depois

*@app.route('/edit\_topic/<int:topic\_id>', methods=['GET', 'POST'])*

*def edit\_topic(topic\_id):*

*if 'username' not in session:*

*return redirect(url\_for('login'))*

*conn = get\_db()*

*if request.method == 'POST':*

*title = request.form['title']*

*content = request.form['content']*

*author = session['username']*

*topic = conn.execute('SELECT \* FROM topics WHERE id = ? AND author = ?', (topic\_id, author)).fetchone()*

*if not topic:*

*return "Permissão negada", 403*

*conn.execute('UPDATE topics SET title = ?, content = ? WHERE id = ?', (title, content, topic\_id))*

*conn.commit()*

*return redirect(url\_for('index'))*

**Sessões Inseguras**

Problema:

* A chave secreta utilizada para as sessões era uma string estática e previsível ('123456'), facilitando ataques de força bruta ou predição.

Solução:

* Substituir a chave secreta por uma gerada dinamicamente usando os.urandom.

Código:

#Antes

*app.secret\_key = '123456'*

#Depois

*import os*

*app.secret\_key = os.urandom(24)*

**Cross-Site Scripting (XSS)**

Problema:

* Entradas do utilizador não eram sanitizadas antes de serem exibidas nos templates HTML, permitindo que scripts maliciosos fossem injetados.

Solução:

* Garantir a sanitização automática de entradas utilizando os recursos nativos de templating do Jinja2 (exibição com `{{ }}`).

Código:

#Antes

*<td>{{ topic[1] }}</td>*

*<td>{{ topic[2] }}</td>*

#Depois

*<td>{{ topic[1] | e }}</td>*

# Conclusão

Esta atividade laboratorial permitiu uma exploração abrangente das vulnerabilidades de segurança e dos desafios de qualidade no desenvolvimento de software. Através da combinação de auditorias manuais e automáticas, foi possível identificar e classificar diversas falhas, destacando-se SQL Injection, Cross-Site Scripting (XSS), armazenamento inadequado de senhas, e problemas relacionados à manutenibilidade do código, como duplicação e complexidade ciclomática.

A utilização de ferramentas como OWASP ZAP e SonarQube foi essencial para complementar a análise manual, evidenciando a importância de adotar metodologias híbridas que unam a perspicácia humana com a eficiência das ferramentas automatizadas. Além disso, a análise das vulnerabilidades com base no CWE Top 25 proporcionou uma visão sobre os riscos mais críticos, reforçando a relevância de práticas seguras em todas as fases do ciclo de desenvolvimento.

As ações de remediação propostas não apenas abordam as falhas identificadas, mas também estabelecem uma base sólida para a implementação de medidas preventivas e a promoção de um desenvolvimento mais seguro e eficiente. Este trabalho sublinha a necessidade contínua de incorporar a segurança como um componente intrínseco ao processo de desenvolvimento, garantindo sistemas mais confiáveis e resilientes diante das crescentes ameaças no panorama tecnológico.

Em suma, a atividade cumpriu os seus objetivos de consolidar conhecimentos teóricos e práticos sobre segurança em software, capacitando-nos a identificar, analisar e mitigar vulnerabilidades de forma eficaz.

# Referências

“Code Quality and Security | Developers First | SonarSource.” Www.sonarsource.com, [www.sonarsource.com/](http://www.sonarsource.com/).

“CWE - CWE Top 25 Most Dangerous Software Weaknesses.” Cwe.mitre.org, cwe.mitre.org/top25/.

“Introduction · OWASP Cheat Sheet Series.” Owasp.org, 2019, cheatsheetseries.owasp.org/.

OWASP. “OWASP Top Ten.” Owasp.org, OWASP, 2024, owasp.org/www-project-top-ten/.

---. “OWASP ZAP.” Zaproxy.org, 2020, [www.zaproxy.org/](http://www.zaproxy.org/).

Owasp. “Input Validation OWASP Cheat Sheet Series.” Owasp.org, 2019, cheatsheetseries.owasp.org/cheatsheets/Input\_Validation\_Cheat\_Sheet.html.

“ZAP – Configuring Proxies.” Www.zaproxy.org, 2024, [www.zaproxy.org/docs/desktop/start/proxies/.](https://www.zaproxy.org/docs/desktop/start/proxies/.%20) Accessed 25 Nov. 2024.

SonarSource. “Static Code Analysis Using SonarQube : A Step-By-Step Guide.” Sonarsource.com, 5 Aug. 2024, [www.sonarsource.com/learn/static-code-analysis-using-sonarqube/.](https://www.sonarsource.com/learn/static-code-analysis-using-sonarqube/.%20) Accessed 25 Nov. 2024.