**Implementações e Testes do servidor Flask**

Testes com SQLMAP:

🛡️ RELATÓRIO DE SEGURANÇA – SQL INJECTION NA ROTA /consulta

🔴 ANTES DA CORREÇÃO (VULNERÁVEL)

Código vulnerável:

query = f"SELECT nome, profissao, salario FROM pessoas WHERE nome LIKE '%{termo}%'"

c.execute(query)

Problemas identificados:

- Concatenação direta da entrada do usuário (termo) na query.

- Total ausência de parametrização.

- A aplicação não valida o tamanho ou formato da entrada.

- Sanitização ausente, expondo o HTML a injeção de scripts (XSS).

Testes realizados com sqlmap:

sqlmap -u "http://127.0.0.1:5000/consulta" --data="termo=teste" --batch --level=5 --risk=3 --flush-session

Resultados:

- Parâmetro vulnerável: termo (POST)

- Banco de dados identificado: SQLite

- Técnicas exploradas com sucesso:

• Boolean-based blind:

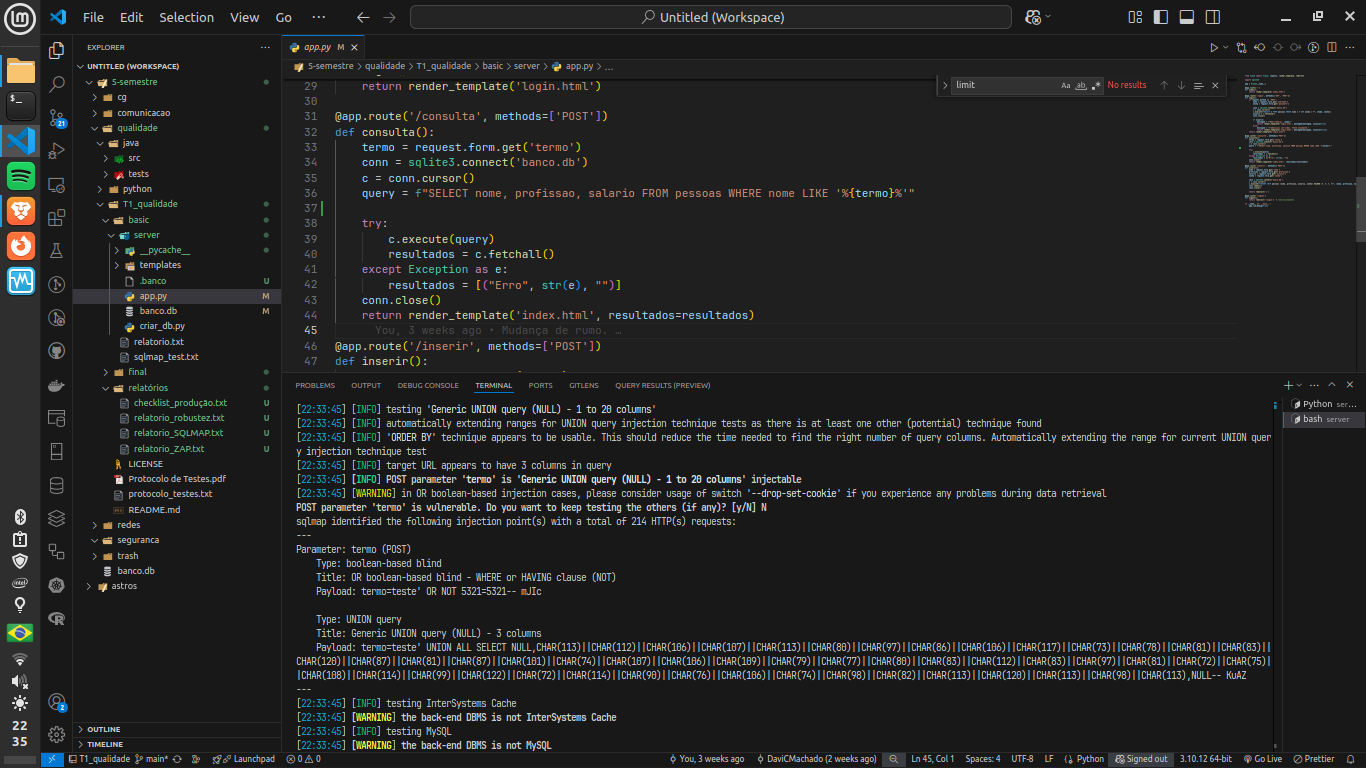
termo=-1984' OR 4854=4854--

• Time-based blind:

termo=teste' OR 6220=LIKE(CHAR(...),UPPER(HEX(RANDOMBLOB(...))))--

• UNION-based:

termo=teste' UNION ALL SELECT NULL,NULL,CHAR(...)--



✅ DEPOIS DA CORREÇÃO (SEGURO)

Código corrigido:

c.execute("SELECT nome, profissao, salario FROM pessoas WHERE nome LIKE ?", ('%' + termo + '%',))

Medidas implementadas:

1. Parametrização segura da query (uso de ? com tupla).

2. Validação da entrada do usuário (vazia ou muito longa).

3. Sanitização com html.escape() antes de exibir no HTML.

4. Tratamento de exceções com try/except para evitar mensagens sensíveis.

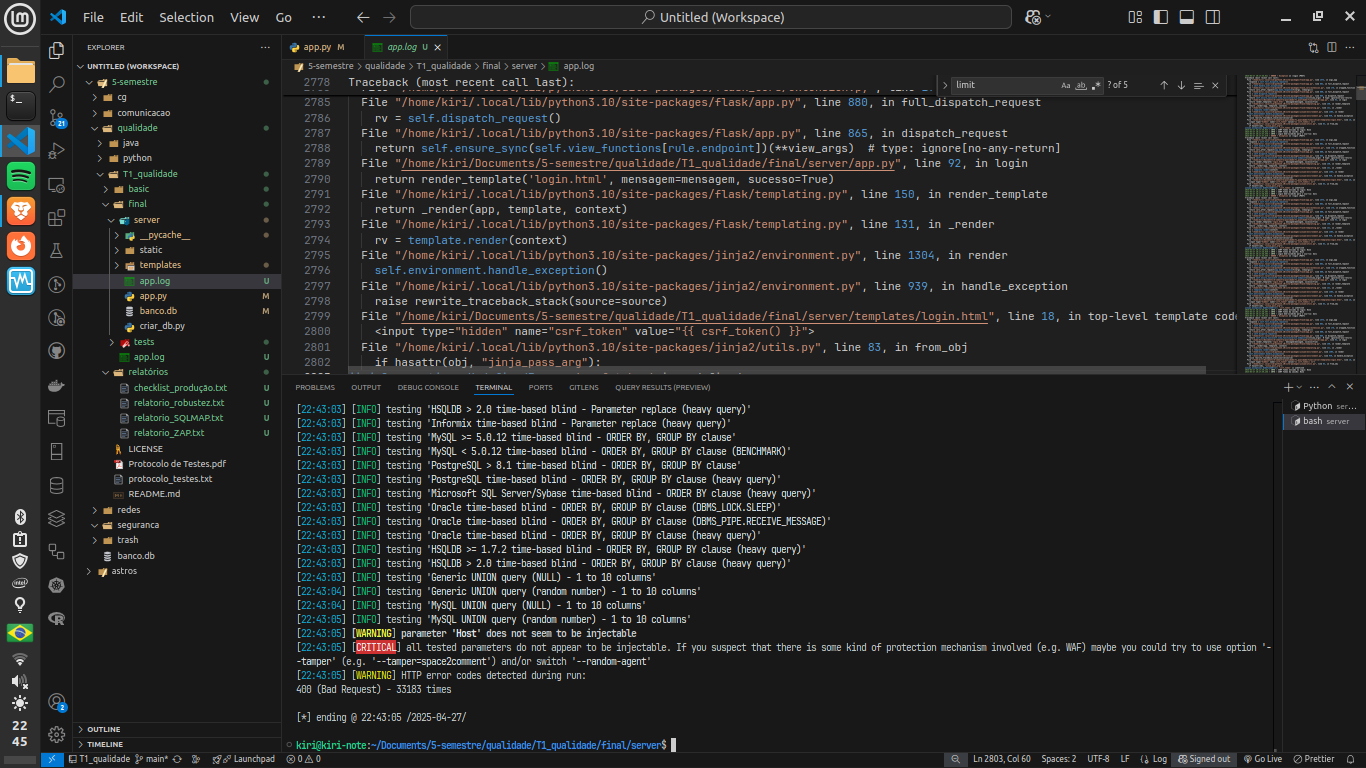
5. Log de acessos para auditoria futura.

Reexecução dos testes com sqlmap:

Resultados obtidos:

[WARNING] POST parameter 'termo' does not seem to be injectable

[CRITICAL] all tested parameters do not appear to be injectable.



📌 CONCLUSÃO

- A vulnerabilidade inicial permitia extração e manipulação arbitrária de dados via injeção SQL.

- Após as correções, a aplicação passou a seguir boas práticas de segurança:

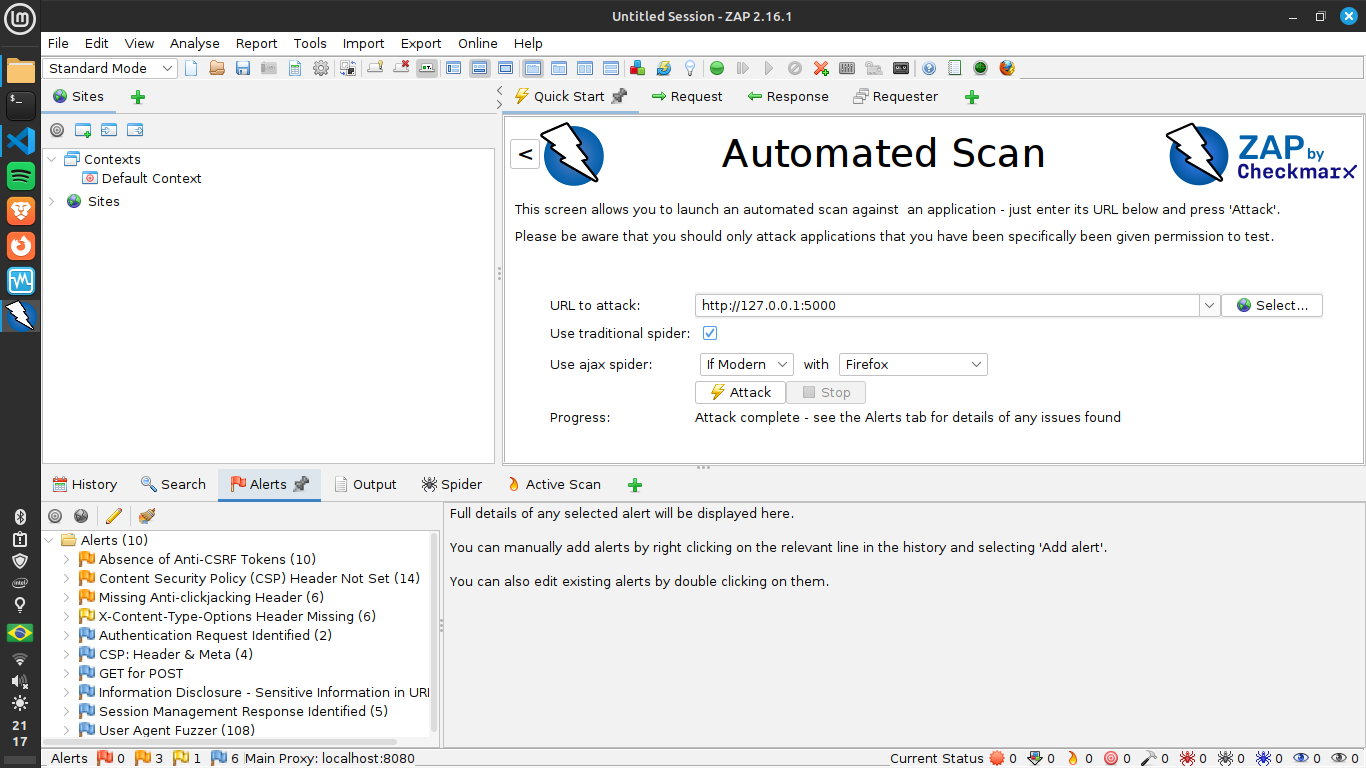
• Uso de parâmetros no SQL.

• Validação e sanitização adequadas.

- O sistema se tornou resistente às técnicas utilizadas pelo sqlmap, como boolean, time-based e union.

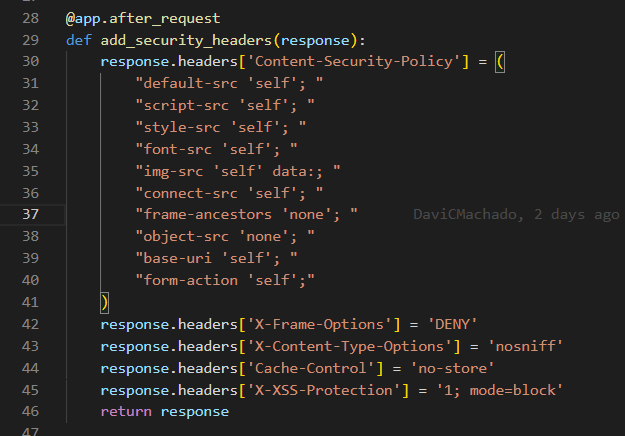
Testes com OWASP ZAP

Antes da implementação de CSP, CSRF e Cookies de sessão:



4 Alertas de segurança, sendo 3 de risco médio, mostrando que o servidor estava suscetível a ataques de XSS e Clickjacking.

Medidas foram implementadas no código do servidor, definindo regras e políticas de segurança.



### **@app.after\_request**

Esse decorator é usado para definir uma função que será executada **após** o processamento de cada requisição. Isso permite modificar a resposta antes que ela seja enviada de volta ao cliente. No caso do código, ele está sendo usado para adicionar cabeçalhos de segurança à resposta HTTP.

#### **Content-Security-Policy**

Esse cabeçalho (CSP) controla quais recursos (scripts, imagens, fontes, etc.) podem ser carregados e executados no navegador, ajudando a evitar ataques como **Cross-Site Scripting (XSS)**. A política definida aqui é bem restritiva:

* **default-src 'self';**: Por padrão, só recursos do mesmo domínio da aplicação podem ser carregados.
* **script-src 'self';**: Apenas scripts do mesmo domínio são permitidos.
* **style-src 'self';**: Apenas estilos CSS do mesmo domínio são permitidos.
* **font-src 'self';**: Somente fontes do mesmo domínio podem ser carregadas.
* **img-src 'self' data:;**: Somente imagens do mesmo domínio ou imagens codificadas em base64 (usando data:) podem ser carregadas.
* **connect-src 'self';**: As conexões (AJAX, WebSockets, etc.) só podem ser feitas para o mesmo domínio.
* **frame-ancestors 'none';**: Nenhum conteúdo pode ser incorporado em iframes de outros domínios, prevenindo ataques de **clickjacking**.
* **object-src 'none';**: Nenhum conteúdo object, embed ou applet pode ser carregado, para evitar ataques através desses elementos.
* **base-uri 'self';**: O <base> HTML (que define o caminho base para URLs relativas) só pode apontar para o mesmo domínio.
* **form-action 'self';**: Os formulários só podem ser enviados para o mesmo domínio, prevenindo redirecionamentos maliciosos.

#### **X-Frame-Options**

Esse cabeçalho impede que sua página seja carregada dentro de um iframe de outro site, ajudando a prevenir **clickjacking**. O valor DENY significa que a página não pode ser carregada em nenhum iframe.

#### **X-Content-Type-Options**

Esse cabeçalho impede que o navegador tente adivinhar o tipo de conteúdo de um arquivo, forçando o navegador a respeitar o tipo MIME especificado. O valor nosniff ajuda a prevenir ataques de **MIME sniffing**, onde um atacante pode enganar o navegador para que ele interprete um arquivo de maneira incorreta (exemplo: tratar um arquivo malicioso como se fosse inofensivo).

#### **Cache-Control**

Esse cabeçalho instrui o navegador a não armazenar a resposta em cache. O valor no-store garante que a resposta não será armazenada, o que é importante para garantir que informações sensíveis não fiquem armazenadas no cache do navegador, principalmente em páginas que contenham dados privados.

#### **X-XSS-Protection**

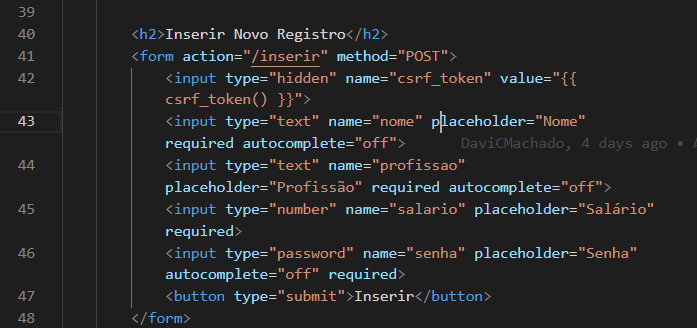
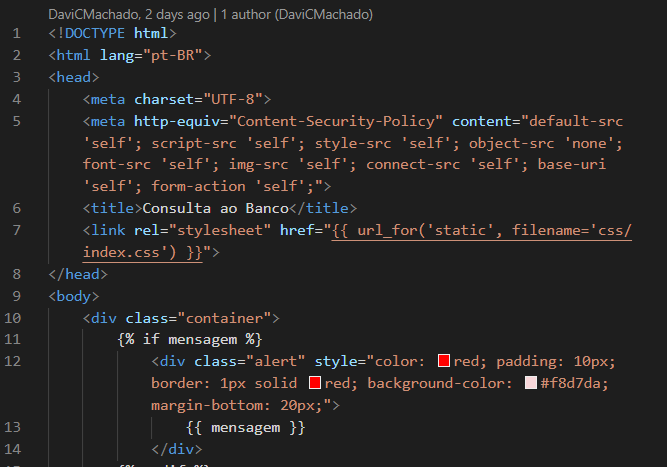
Esse cabeçalho ativa a proteção contra **Cross-Site Scripting (XSS)** nos navegadores que suportam. O valor 1; mode=block faz com que o navegador tente bloquear a execução de scripts maliciosos em caso de detecção de um ataque XSS. Se um XSS for detectado, a página será bloqueada e não renderizada.

### **3. Fluxo de execução**

O código é executado da seguinte maneira:

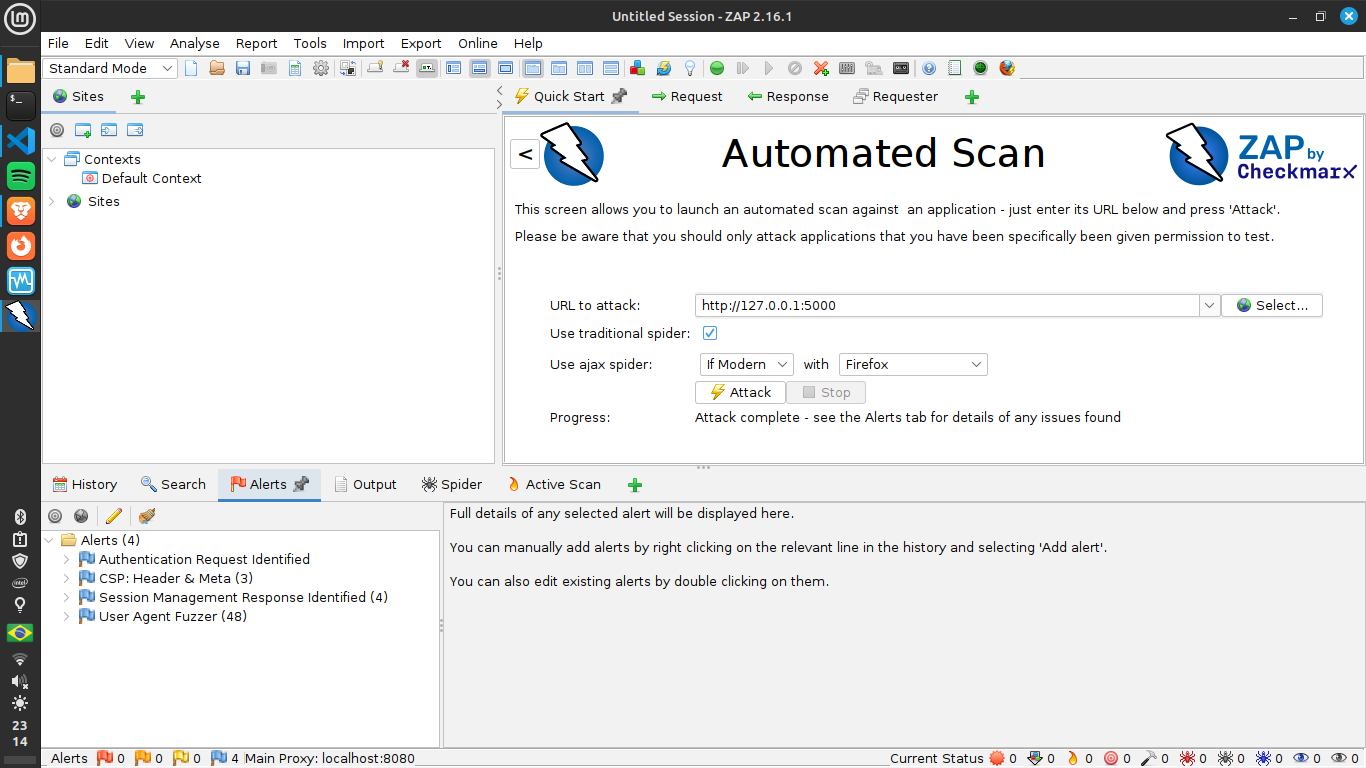
1. Quando uma requisição é recebida, o Flask processa essa requisição normalmente.
2. Depois que a resposta é gerada, a função add\_security\_headers é chamada, antes de enviar a resposta ao cliente.
3. Os cabeçalhos de segurança são adicionados à resposta.
4. A resposta, agora com os cabeçalhos de segurança, é enviada de volta ao cliente.

Medidas implementadas no HTML:



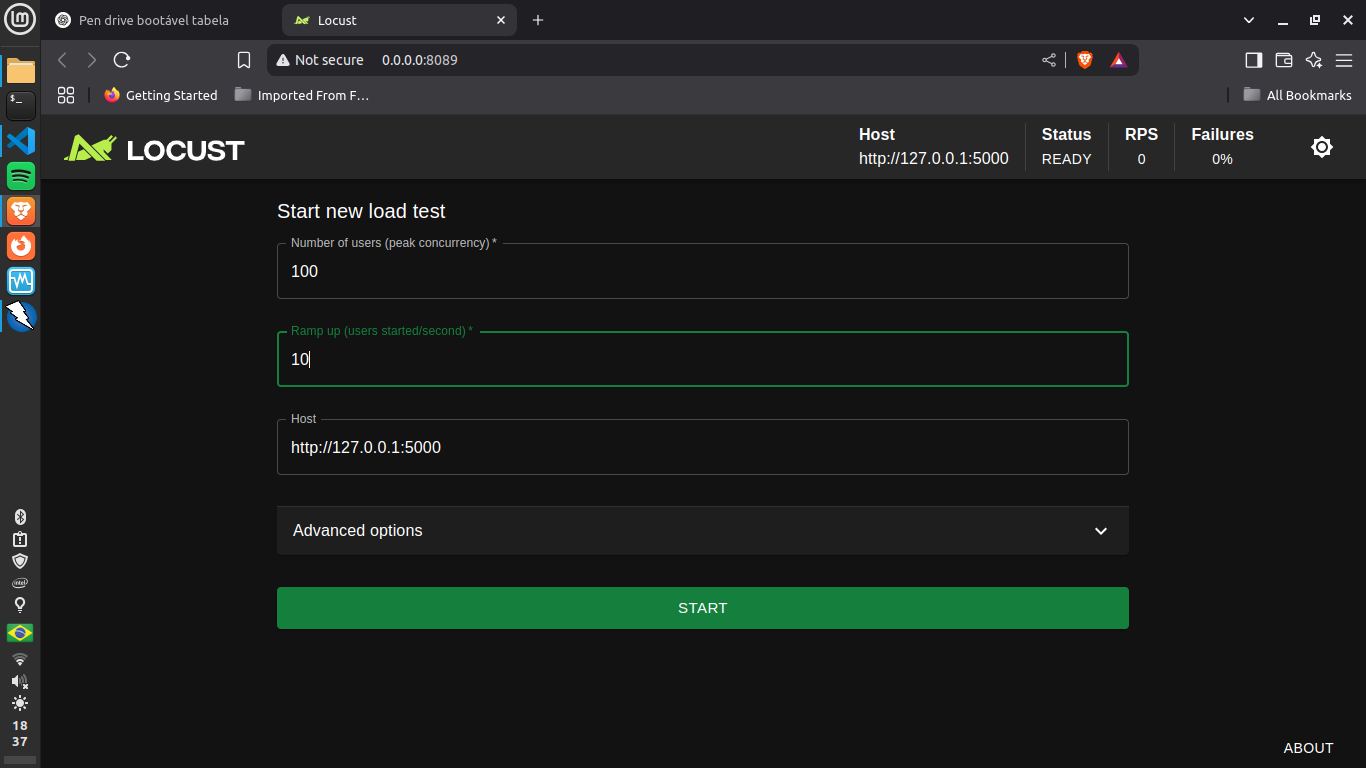
Adição de uma metatag definindo o CSP semelhante ao esperado pelo servidor e uso de input hidden do token csrf de sessão única.

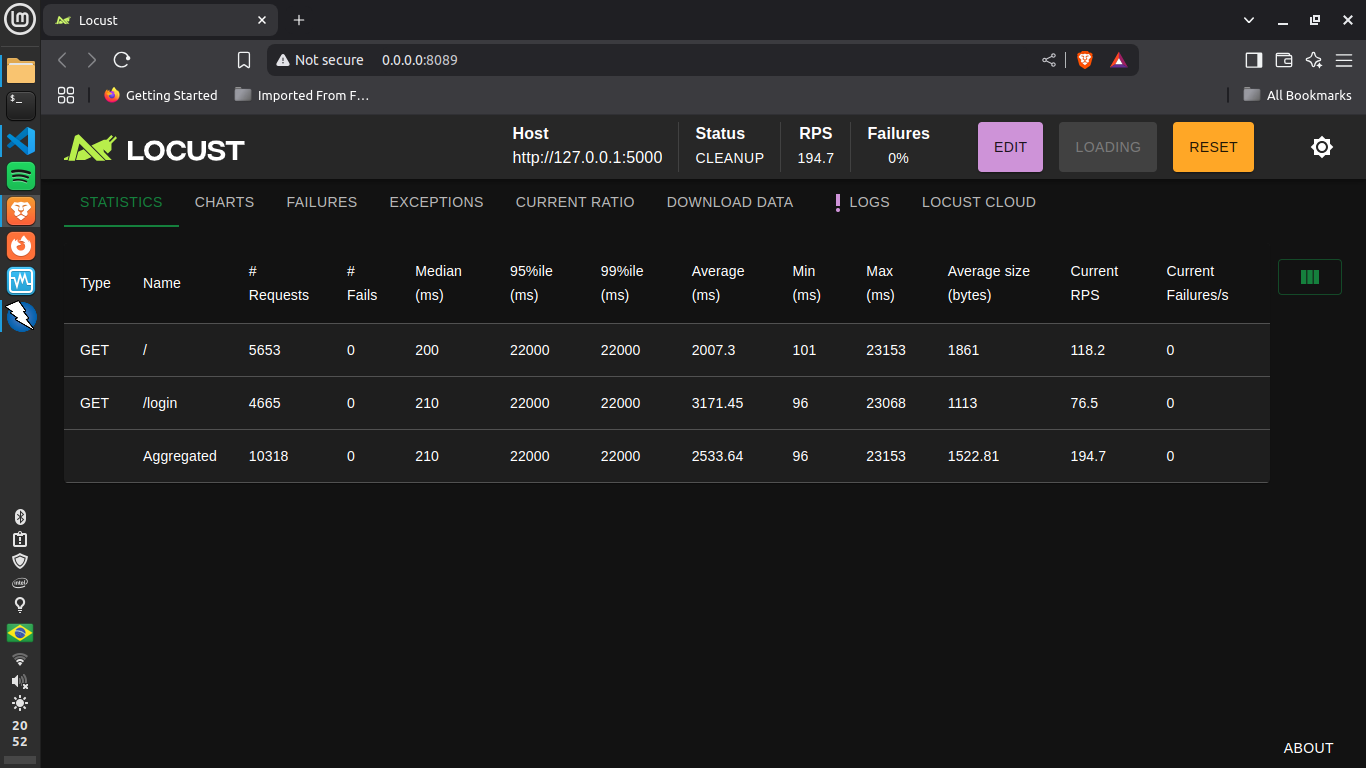
Após implementar medidas de segurança:



Nenhum alerta de risco, apenas alertas informacionais.

Testes de Carga (Resiliência)





Usando a ferramenta LOCUST testamos os seguintes casos:

a) Testes de Carga (Load Testing)

Teste 1

MAX de usuarios: 100

Incremento de Usuarios por segundo: 10

Tempo entre cada requisição: 3-6 seg

Resultados:

Média | Mínimo | Máximo

26 ms | 4 ms | 113 ms

Zero falhas

Teste 2

MAX de usuarios: 200

Incremento de Usuarios por segundo: 20

Tempo entre cada requisição: 3-6 seg

Resultados:

Média | Mínimo | Máximo

45 ms | 4 ms | 269 ms

Zero falhas

Teste 3

MAX de usuarios: 500

Incremento de Usuarios por segundo: 50

Tempo entre cada requisição: 3-6 seg

Resultados:

Média | Mínimo | Máximo

93 ms | 4 ms | 611 ms

Zero falhas

b) Testes de Resiliência (Stress Testing)

Teste 1

MAX de usuarios: 1000

Incremento de Usuarios por segundo: 100

Tempo entre cada requisição: 1-3 seg

Resultados:

Média | Mínimo | Máximo

2527 ms | 96 ms | 23152 ms

Zero falhas

Teste 2

MAX de usuarios: 500

Incremento de Usuarios por segundo: 0 (já começa com 500 users)

Tempo entre cada requisição: 3-6 seg

Resultados:

Média | Mínimo | Máximo

121 ms | 4 ms | 4139 ms

Zero falhas

Conclusão

Os testes realizados demonstraram que o servidor é capaz de lidar

com altas quantidades de requisições simultâneas sem falhas,

mantendo uma boa performance mesmo em condições de carga extrema.

A média de tempo de resposta foi adequada em todos os testes,

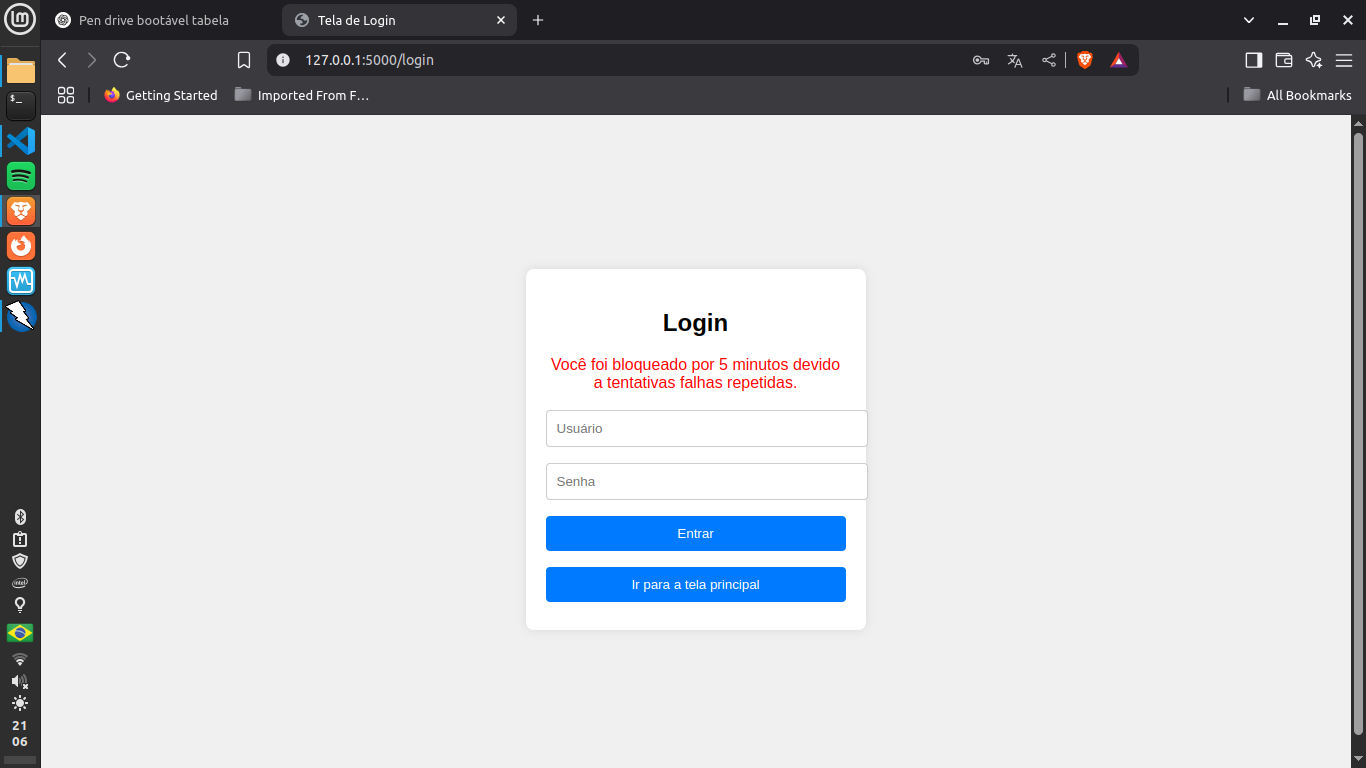
com uma variação aceitável no máximo de requisições por segundo.

Outras implementações:

O sistema possui um limitador de tentativas de login, limitando cada sessão a 5 tentativas por minuto, baseado no IP do usuário.

Um mesmo usuário pode errar até 3 vezes consecutivas.

Se passar disso, o usuário sofre time-out de 5 minutos.



O sistema também possui um sistema de log que registra acessos aos recursos do servidor por parte dos usuários e também erros e exceções.

