INSTITUTO FEDERAL GOIANO - CAMPUS MORRINHOS

CURSO SUPERIOR DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ADRIANO COSTA ARAUJO MORAIS

ANÁLISE DE VULNERABILIDADES DO OWASP TOP 10 RELACIONADAS AO CWE PARA CODIFICAÇÃO SEGURA DE APLICAÇÕES WEB.

MORRINHOS - GO

2026

ADRIANO COSTA ARAUJO MORAIS

ANÁLISE DE VULNERABILIDADES DO OWASP TOP 10 RELACIONADAS AO CWE PARA CODIFICAÇÃO SEGURA DE APLICAÇÕES WEB.

Monografia apresentada ao Curso Superior de Bacharelado em Ciência da Computação do Instituto Federal Goiano – Campus Morrinhos, como requisito parcial para obtenção de título de Bacharel em Ciência da Computação.

Área de concentração: Segurança da Informação.

Orientadora: Profª MSc Ana Maria Martins Carvalho.

MORRINHOS - GO

2026

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas – SIBI/IF Goiano Campus Morrinhos

M827a Morais, Adriano Costa Araujo.
Análise de vulnerabilidades do OWASP TOP 10 relacionadas ao CWE para codificação segura de aplicações WEB. / Adriano Costa Araujo Morais. – Morrinhos, GO: IF Goiano, 2026.
81 f. : il. color.

Orientadora: Ma. Ana Maria Martins Carvalho.

Trabalho de conclusão de curso (graduação) – Instituto Federal Goiano Campus Morrinhos, Bacharelado em Ciências da Computação, 2026.

1. Falhas de Software. 2. SAST. 3. DAST I. Carvalho, Ana Maria Martins. II. Instituto Federal Goiano. III. Título.

CDU 004.7:004.451

Fonte: Elaborado pela Bibliotecária-documentalista Morgana Guimarães, CRB1/2837

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO

PARA DISPONIBILIZAR PRODUÇÕES TÉCNICO-CIENTÍFICAS NO REPOSITÓRIO INSTITUCIONAL DO IF GOIANO

Com base no disposto na Lei Federal nº 9.610, de 19 de fevereiro de 1998, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano a disponibilizar gratuitamente o documento em formato digital no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

IDENTIFICAÇÃO DA PRODUÇÃO TÉCNICO-CIENTÍFICA

- ☐ Tese (doutorado)
- ☐ Dissertação (mestrado)
- ☐ Monografia (especialização)
- ☑ TCC (graduação)
- ☐ Artigo científico
- ☐ Capítulo de livro
- ☐ Livro
- ☐ Trabalho apresentado em evento
- ☐ Produto técnico e educacional - Tipo:

Nome completo do autor:
Adriano Costa Araujo Morais

Matrícula:
2019104201940017

Título do trabalho:
ANÁLISE DE VULNERABILIDADES DO OWASP TOP 10 RELACIONADAS AO CWE PARA CODIFICAÇÃO SEGURA DE APLICAÇÕES WEB.

RESTRIÇÕES DE ACESSO AO DOCUMENTO

Documento confidencial: ☑ Não ☐ Sim, justifique:

Informe a data que poderá ser disponibilizado no RIIF Goiano: 13 /04 /2026

O documento está sujeito a registro de patente? ☐ Sim ☑ Não

O documento pode vir a ser publicado como livro? ☐ Sim ☑ Não

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O(a) referido(a) autor(a) declara:

• Que o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;

• Que obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autoria, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia Goiano os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;

• Que cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia Goiano.

Morrinhos - GO — Local
13 /04 /2026 — Data

Documento assinado digitalmente
gov.br
ADRIANO COSTA ARAUJO MORAIS
Data: 13/04/2026 13:51:42-0300
Verifique em https://validar.iti.gov.br

Assinatura do autor e/ou detentor dos direitos autorais

Ciente e de acordo:

Documento assinado digitalmente
gov.br
ANA MARIA MARTINS CARVALHO
Data: 13/04/2026 12:39:56-0300
Verifique em https://validar.iti.gov.br

Assinatura do(a) orientador(a)

ADRIANO COSTA ARAUJO MORAIS

ANÁLISE DE VULNERABILIDADES DO OWASP TOP 10 RELACIONADAS AO CWE PARA CODIFICAÇÃO SEGURA DE APLICAÇÕES WEB.

Data da defesa: 01/04/2026

Resultado: Aprovado.

BANCA EXAMINADORA ASSINATURAS

MSc Ana Maria Martins Carvalho

Instituto Federal Goiano - Campus Morrinhos-GO

gov.br Documento assinado digitalmente
ANA MARIA MARTINS CARVALHO
Data: 07/04/2026 15:05:30-0300
Verifique em https://validar.iti.gov.br

Dr Antônio Neco de Oliveira

Instituto Federal Goiano - Campus Morrinhos-GO

gov.br Documento assinado digitalmente
ANTONIO NECO DE OLIVEIRA
Data: 07/04/2026 15:54:22-0300
Verifique em https://validar.iti.gov.br

Esp José Pereira Alves

Instituto Federal Goiano - Campus Morrinhos-GO

gov.br Documento assinado digitalmente
JOSE PEREIRA ALVES
Data: 10/04/2026 15:08:11-0300
Verifique em https://validar.iti.gov.br

MORRINHOS - GO

2026

AGRADECIMENTOS

Agradeço à minha mãe, Vera, por sempre me incentivar a me dedicar aos estudos e por ter me ajudado da melhor maneira possível. Agradeço também à minha orientadora, pela amizade e por ter me acompanhado em minha trajetória acadêmica desde o início desta graduação, e a todos que, de alguma forma, colaboraram para que eu pudesse concluí-la.

RESUMO

A consolidação das aplicações *web* e a complexidade dos sistemas modernos tornaram a segurança da informação um requisito crítico. Vulnerabilidades no código-fonte configuram-se como uma das principais portas de entrada para ataques cibernéticos, visto que frequentemente as aplicações são lançadas com falhas de *software* que resultam em vulnerabilidades de segurança exploráveis. A persistência de falhas já conhecidas evidencia uma lacuna entre o conhecimento de segurança e a prática de desenvolvimento. Diante disso, este trabalho tem como objetivo demonstrar como as principais falhas de *software* impactam a segurança de aplicações web, estabelecendo uma relação com as quatro primeiras categorias de vulnerabilidades do OWASP *Top* 10. Para isso, foram utilizadas ferramentas de análise estática (SAST) e dinâmica (DAST), adotando uma abordagem híbrida com o intuito de aprimorar os resultados, reduzir falsos positivos e validar as vulnerabilidades identificadas que estão presentes no ambiente fornecido pela OWASP, o *Juice Shop*. Por fim, o trabalho busca conscientizar sobre a forma como essas falhas podem ser exploradas por usuários mal-intencionados e reforçar a importância da codificação segura como parte integrante do cotidiano de desenvolvedores ou daqueles envolvidos no ciclo de vida de um *software*.

Palavras-chave: Codificação Segura, OWASP *Top* 10, Falhas de *Software*, SAST, DAST.

ABSTRACT

The consolidation of web applications and the complexity of modern systems have made information security a critical requirement. Vulnerabilities in source code are one of the main entry points for cyberattacks, since applications are frequently launched with software flaws that result in exploitable security vulnerabilities. The persistence of known flaws highlights a gap between security knowledge and development practice. Therefore, this work aims to demonstrate how the main software flaws impact the security of web applications, establishing a relationship with the first four vulnerability categories of the OWASP Top 10. To this end, static (SAST) and dynamic (DAST) analysis tools were used, adopting a hybrid approach to improve results, reduce false positives, and validate the identified vulnerabilities present in the environment provided by OWASP, Juice Shop. Finally, this work aims to raise awareness about how these flaws can be exploited by malicious users and to reinforce the importance of secure coding as an integral part of the daily lives of developers or those involved in the software lifecycle.

Keywords: Secure Coding, OWASP Top 10, Software Weaknesses, SAST, DAST.

LISTA DE SIGLAS

ABAC *Attribute-Based Access Control*

CTF *Capture the flag*

CVE *Common Vulnerabilities and Exposures*

CWE *Common Weakness Enumeration*

DAC *Discretionary Access Control*

GUIDs *Globally Unique Identifiers*

HMAC *Hash-based Message Authentication Code*

JSON *JavaScript Object Notation*

JWT *JSON Web Token*

MIME *Multipurpose Internet Mail Extensions*

ORM *Object-Relational Mapping*

OWASP *Open Worldwide Application Security Project*

PTES *Penetration Testing Execution Standard*

RBAC *Role-Based Access Control*

ReBAC *Relationship-based access control*

SaaS *Software as a Service*

SDLC *Software Development Life Cycle*

ZAP *Zed Attack Proxy*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A medida que o acesso à *Internet* se tornou mais amplo e acessível, a criação e o uso de aplicações *web* aumentaram significativamente. Atualmente, essas aplicações estão cada vez mais presentes no cotidiano, impulsionadas pela popularização do modelo SaaS (*Software as a Service*), que viabilizou a criação de diferentes tipos de aplicações diretamente no ambiente *web*, ampliando seu alcance e diversificando o perfil de usuários.

Entretanto, esse cenário de crescimento acelerado das aplicações *web* também amplia a superfície de ataque, pois diferentemente do modelo tradicional de desenvolvimento, no qual o ciclo de vida de um *software* poderia levar meses ou até anos, essas aplicações *web* modernas são desenvolvidas em ciclos muito mais curtos. Essa redução no tempo de desenvolvimento, aliada à crescente complexidade dos sistemas, aumenta a probabilidade da introdução de falhas de segurança durante a implementação. De acordo com a pesquisa, os ataques cibernéticos globais cresceram 21% no segundo trimestre de 2025, segundo estudo realizado pela *Check Point Research*. A pesquisa mostrou que embora a tendência de alta atinja quase todos os setores e regiões, a Europa registrou o crescimento mais expressivo, com um salto de 22% em relação ao ano de 2024 (SECURITY LEADER, 2025).

Diante desse cenário, as empresas percebem, mais do que nunca, a necessidade de investir na segurança dessas aplicações. Uma das organizações mais relevantes nesse contexto é a OWASP (*Open Worldwide Application Security Project*), que contribui para a segurança das aplicações de diversas formas. Um de seus principais projetos é o OWASP *Top* 10, uma lista que classifica as vulnerabilidades mais críticas em aplicações *web*.

De acordo com a OWASP (2021b), essa classificação é elaborada com base em dados estatísticos de CVEs (*Common Vulnerabilities and Exposures*) e contribuições da comunidade global de profissionais de segurança, servindo como padrão de referência para o mercado.

As vulnerabilidades em aplicações podem estar presentes por várias razões: *design* inseguro, infraestrutura insegura, erros de codificação, autenticação fraca e falha em testar condições incomuns ou inesperadas (CENTER FOR INTERNET SECURITY, 2023). Da mesma forma, existem diversas abordagens para fortalecer a segurança das aplicações, dentre as quais se destacam auditorias, monitoramento de eventos e testes de penetração (*pentests*).

Contudo, para além dessas práticas, o desenvolvimento seguro se mostra a forma mais eficaz de reduzir tais ameaças, visto que muitas vulnerabilidades exploradas por usuários mal-intencionados têm origem na fase de desenvolvimento, quando trechos de código vulneráveis são introduzidos. Apesar disso, grande parte dos trabalhos existentes focam apenas na exploração das vulnerabilidades existentes nas aplicações, sem um aprofundamento adequado de suas causas ou em como mitigá-las.

Dessa forma, este trabalho apresenta uma análise das principais falhas de *software* listadas pelo CWE(*Common Weakness Enumeration*) subjacentes às quatro primeiras categorias de vulnerabilidades do *OWASP Top* 10*,* mostrando como um usuário mal intencionado as explora, assim como boas práticas de codificação segura e soluções recomendadas de grandes organizações que tem como foco à segurança. Essa análise é ambientada no *Juice Shop*, uma aplicação propositalmente vulnerável que, embora desenvolvida para fins educacionais, incorpora falhas e comportamentos negligentes contidos em aplicações reais, com o intuito de introduzir a codificação segura no cotidiano do desenvolvimento de *software*.

Este trabalho está estruturado em nove capítulos. O Capítulo 2 apresenta os objetivos. O Capítulo 3 aborda os trabalhos correlatos. O Capítulo 4 apresenta o referencial teórico. O Capítulo 5 descreve a metodologia utilizada. O Capítulo 6 apresenta o mapeamento realizado entre as falhas de *software* e as categorias do *OWASP Top* 10. O Capítulo 7 detalha a análise das vulnerabilidades identificadas no OWASP *Juice Shop*. O Capítulo 8 aborda o desenvolvimento do trabalho, e, por fim, o Capítulo 9 apresenta as conclusões.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Demonstrar como as falhas no processo de desenvolvimento de *software* resultam em vulnerabilidades, explorando a versão mais recente do OWASP *Top* 10, detalhando métodos de exploração dessas vulnerabilidades e propondo diretrizes e exemplos de implementação de código seguro para o aprimoramento da resiliência e segurança dos *softwares*.

2.2 OBJETIVOS ESPECÍFICOS

- Justificar a importância de aderir à segurança desde as etapas iniciais do ciclo de desenvolvimento de *software*.
- Mapear as principais falhas de *software* do CWE (*Common Weakness Enumeration*), e relacioná-las às quatro primeiras categorias do OWASP *Top* 10 de 2021.
- Utilizar o OWASP *Juice Shop* como ambiente prático para a aplicação de conceitos de desenvolvimento seguro.
- Empregar ferramentas de SAST (*Static Application Security Testing*) e DAST (*Dynamic Application Security Testing*) para a detecção de vulnerabilidades no ambiente.
- Expor e analisar as vulnerabilidades identificadas no OWASP *Juice Shop*, correlacionando-as com as categorias abordadas do *Top* 10.
- Apresentar boas práticas de codificação segura e propor soluções de código seguro para mitigar as vulnerabilidades identificadas.

3 TRABALHOS CORRELATOS

Diversos estudos recentes têm explorado a importância da detecção e mitigação de vulnerabilidades em aplicações *Web*, destacando tanto os benefícios do uso de ferramentas automatizadas quanto os desafios enfrentados durante a análise de segurança.

O trabalho de Li (2020) apresenta a primeira matriz que correlaciona as 10 principais vulnerabilidades do OWASP *Top* 10 de 2017 com os 25 erros de *software* mais críticos do CWE/SANS, resultando em uma estrutura de segurança que auxilia as equipes de desenvolvimento na revisão de vulnerabilidades e na correção de código. O autor conduz um estudo de caso envolvendo a varredura de vulnerabilidades em um aplicativo de detecção de *malware* para dispositivos móveis, utilizando uma ferramenta SAST da Checkmarx, à qual é uma solução de varredura estática de código-fonte de última geração para identificar falhas e vulnerabilidades. A matriz CWE/SANS foi empregada para otimizar a análise dos resultados detectados pela ferramenta Checkmarx. Na primeira execução, a varredura gerou alertas relacionados principalmente a dependências externas, e não ao código desenvolvido pelo autor. Após o filtro desses alertas e a realização de uma nova varredura, as vulnerabilidades detectadas foram mapeadas na matriz desenvolvida, otimizando o processo de correção e priorização. Embora o estudo tenha sido direcionado a um aplicativo móvel de análise de segurança, o mapeamento entre OWASP e CWE/SANS apresentado é altamente relevante também no contexto de aplicações *web*, pois oferece uma abordagem sistemática para relacionar vulnerabilidades às suas causas fundamentais. O foco principal do trabalho está no mapeamento e na correlação das categorias de vulnerabilidades, o que é bem explorado pelo autor. No entanto, as falhas específicas são abordadas de forma limitada, o que reduz a profundidade técnica. Além disso, a pesquisa se baseia na versão de 2017 do OWASP *Top* 10, o que abre espaço para novas investigações utilizando as versões mais recentes da classificação.

Jesus (2022) busca apresentar as vulnerabilidades listadas pelo OWASP *Top* 10 de 2021, demonstrando como alguns ataques são realizados por invasores, seus impactos e as formas de mitigação e prevenção contra ataques futuros. Para isso, o autor utiliza laboratórios de testes vulneráveis e emprega duas ferramentas principais: o OWASP ZAP, com foco na funcionalidade de varredura automatizada, e o Burp Suite, utilizando seu *proxy* para interceptar requisições e respostas, além de realizar ataques de força bruta. Apesar de abordar aspectos importantes, o trabalho apresenta uma exploração limitada em relação às vulnerabilidades do *Top* 10, analisando apenas algumas vulnerabilidades e abrangendo poucas classificações. Além disso, os testes foram realizados exclusivamente em laboratórios intencionalmente vulneráveis, o que reduz a representatividade em cenários reais.

Em Franzese (2023), o autor apresenta um estudo de caso sobre as principais vulnerabilidades da plataforma Moodle, identificadas com base na lista OWASP *Top* 10 de 2021. O Moodle é uma aplicação *web* amplamente utilizada por mais de 316 milhões de usuários. O objetivo do autor é verificar se a plataforma Moodle está segura, identificando potenciais vulnerabilidades e avaliando o risco que elas representam para os usuários e administradores do sistema. Para isso,

inicialmente foi realizada uma varredura automatizada utilizando a ferramenta Zed *Attack Proxy* (ZAP), que permitiu gerar um relatório contendo detalhes sobre possíveis vulnerabilidades detectadas. A partir desse relatório, outras ferramentas foram empregadas para conduzir uma análise de risco mais profunda, determinando se os alertas apontados eram vulnerabilidades reais ou falsos positivos. Após constatar quais de fato eram vulnerabilidades, o autor realizou também uma análise de *logs*, destacando comportamentos e evidências que permitiam relacionar certas atividades a técnicas de ataque específicas. As contribuições de seu trabalho são relevantes, pois explicam e demonstram situações recorrentes relacionadas a falhas de *software* e à identificação de vulnerabilidades em aplicações *web*. O autor também ressalta a importância de adoção de medidas preventivas, monitoramento constante e análise regular dos *logs* de acesso da aplicação. Entre as limitações do trabalho, destaca-se o fato de que, embora mencione falhas associadas às categorias do CWE, essa discussão é feita de maneira breve. Além disso, o estudo depende majoritariamente de uma única ferramenta para detecção inicial das vulnerabilidades, o que pode restringir a abrangência da análise e deixar de identificar problemas que outras ferramentas poderiam revelar.

No trabalho de Garcia (2023), o estudo do autor aborda a eficácia dos testes de penetração no contexto da segurança de aplicações *Web*. O objetivo é demonstrar que a realização de testes de penetração em aplicativos *Web* agrega valor ao processo de garantia de segurança, evidenciando que os sistemas estão protegidos contra possíveis ataques. Os testes foram executados em uma aplicação real, explorada por meio do recurso de *spidering* da ferramenta OWASP ZAP. Ao final, o autor conclui que os resultados obtidos pelos testes de penetração são de extrema importância, pois permitem avaliar as vulnerabilidades existentes, reforçando que outras medidas de segurança devem ser adotadas em conjunto com os testes.

Torres (2023) apresenta uma análise das vulnerabilidades encontradas nos portais eletrônicos das 102 câmaras municipais alagoanas, classificadas conforme o critério de criticidade da OWASP. O objetivo foi verificar se existia uma correlação entre o número de vulnerabilidades identificadas com o Produto Interno Bruto (PIB) municipal e o uso do portal de código aberto Interlegis. Em seu trabalho o autor reuniu as URLs e as empresas/instituições desenvolvedoras dos portais e empregou a ferramenta WAPITI como *scanner* de vulnerabilidades para analisar todos os sites. Ao final dessa etapa, constatou-se que, em 9 dos 100 portais analisados, não foram encontradas vulnerabilidades. No total, foram detectadas 667 vulnerabilidades, resultando em uma média de aproximadamente 7,3 por portal. Os resultados indicam que o PIB municipal não apresenta correlação com o número de vulnerabilidades, pois os portais das cidades com maior PIB exibiram, em média, um número maior de falhas do que aqueles de municípios com menor PIB. Além disso, observou-se que o orçamento das câmaras legislativas não é proporcional às medidas de segurança adotadas em seus portais eletrônicos. Com o objetivo de aumentar a confiabilidade dos resultados, o autor sugere, para estudos futuros, a utilização de mais de uma ferramenta de análise.

Todos os trabalhos utilizados fizeram contribuições relevantes para a área de segurança de aplicações, seja por meio da análise de comportamentos maliciosos,

exploração de vulnerabilidades, identificação de falhas ou o uso do OWASP *Top* 10 como referência em uma de suas versões. Entretanto, alguns temas importantes para uso prático ainda são pouco aprofundados, especialmente no que diz respeito a estudos aplicados em cenários mais próximos da realidade.

O OWASP *Juice Shop* é uma aplicação intencionalmente vulnerável, amplamente utilizada principalmente como cobaia de ferramentas e de treinamentos de segurança, porém não é diferente das que estão nó mercado, tendo uma arquitetura moderna e falhas semelhantes às encontradas em aplicações reais. Nesse contexto, o *Juice Shop* é ideal para demonstrar como os incidentes de segurança ocorrem no dia à dia. Diante disso, este trabalho busca preencher algumas lacunas pouco exploradas e desenvolvidas, oferecendo uma abordagem que integra mapeamento teórico, exploração prática e recomendações de codificação segura.

Para isso, inicialmente foi realizado um mapeamento entre falhas de software classificadas pelo CWE e as vulnerabilidades do OWASP *Top* 10 de 2021, evidenciando como erros de desenvolvimento podem resultar em vulnerabilidades críticas. Em seguida, são apresentadas técnicas de exploração e a perspectiva do atacante durante a análise dessas falhas, pois conhecer à forma como esses agentes atuam é fundamental para implementar medidas de mitigação eficazes. Por fim, são discutidas técnicas de codificação segura aplicáveis a funcionalidades frequentemente suscetíveis a exploração quando implementadas sem critérios adequados de segurança.

4 REFERENCIAL TEÓRICO

4.1 OWASP

A *Open Worldwide Application Security Project* (OWASP) é uma fundação sem fins lucrativos que trabalha para melhorar a segurança de *softwares*, por meio de seus projetos de *software* de código aberto liderados pela comunidade. Todos os projetos, ferramentas, documentações e padrões são gratuitos e abertos a qualquer pessoa interessada em aprimorar a segurança de aplicações.

Além disso, a OWASP realiza conferências educacionais e possui treinamentos líderes do setor, para permitir que os desenvolvedores criem *softwares* melhores e que os profissionais de segurança tornem o *software* mundial mais seguro (OWASP, 2025a).

4.2 OWASP *TOP* 10

O OWASP *Top* 10 é um documento de referência voltado à conscientização de desenvolvedores e profissionais de segurança de aplicações *web*. Ele representa um amplo consenso sobre os riscos de segurança mais críticos para aplicações *Web*. Foi criado em 2003 para auxiliar organizações e desenvolvedores, servindo como um ponto de partida para o desenvolvimento seguro. Ao longo dos anos, evoluiu para um pseudo-padrão usado como base para conformidade, educação e ferramentas de fornecedores (OWASP, 2025b).

Esse documento tornou-se fundamental, atualizado periodicamente com base em dados coletados da indústria, é utilizado como referência por diversos trabalhos e organizações, ajudando a reduzir as superfícies de ataque mais exploradas por cibercriminosos. Consiste em uma lista das 10 principais categorias de vulnerabilidades mais críticas.

Neste trabalho, o OWASP *Top* 10 é utilizado como referência para correlacionar vulnerabilidades com falhas de *software*, as quais, em muitos casos, são introduzidas devido a implementações inseguras.

4.3 CODIFICAÇÃO SEGURA

Codificação segura é a prática disciplinada de desenvolver *software* de forma a prevenir vulnerabilidades, assegurando que os erros de implementação não se transformem em falhas exploráveis. A codificação segura é um tema de elevada relevância, sendo abordada em certificações reconhecidas, como a CISSP, e parte dos controles críticos do *CIS Controls*. Esses controles servem, inclusive, como base para a conformidade com regulamentações e normas do setor, como PCI DSS, HIPAA, GDPR, LGPD, entre outras.

Conhecer as fragilidades que resultam em vulnerabilidades significa que os desenvolvedores de software, projetistas de hardware e arquitetos de segurança podem eliminá-las antes da implementação, quando é muito mais fácil e barato fazê-lo (CWE, 2024).

4.4 CWE

A *Common Weakness Enumeration* (CWE) é uma lista, desenvolvida pela comunidade, que cataloga tipos comuns de fraquezas em *softwares* e *hardwares* que podem resultar em problemas de segurança. Uma fraqueza é definida como uma condição em um *software*, *firmware*, *hardware* ou serviço que, sob determinadas circunstâncias, pode contribuir para a introdução de vulnerabilidades (CWE, 2024).

O CWE é mantido pela MITRE Corporation, e uma de suas principais contribuições é apresentar as fraquezas em um nível mais técnico e detalhado, evidenciando como podem ser introduzidas durante o ciclo de desenvolvimento de *software*, seja por erros de codificação, falhas de *design* ou uso de bibliotecas vulneráveis. Cada fraqueza listada no CWE recebe uma identificação única e inclui informações sobre causas, consequências, exemplos de código e técnicas de mitigação. Dessa forma, o CWE funciona como uma base de conhecimento, voltada para às partes que querem compreender como e por que uma falha ocorre.

4.5 OWASP *JUICE SHOP*

De acordo com a própria OWASP, o OWASP *Juice Shop* é provavelmente a aplicação *web* insegura mais moderna e sofisticada, podendo ser usada em treinamentos de segurança, demonstrações de conscientização, competições do tipo *Capture The Flag* (CTF) e como ambiente de testes para ferramentas de segurança.

A aplicação foi desenvolvida utilizando tecnologias modernas, como Node.js, Express e Angular, e contém uma ampla variedade de desafios de *hacking*, distribuídos em diferentes níveis de dificuldade. Cada desafio explora uma ou mais vulnerabilidades subjacentes, frequentemente alinhadas às categorias do OWASP *Top* 10 (OWASP, 2025c).

4.6 SERVIDOR DE PROXY

Em redes de computadores, um *proxy* é um servidor que age como intermediário para requisições de clientes que solicitam recursos de outros servidores (WIKIPÉDIA, 2024). Neste trabalho, o termo *proxy* é utilizado para descrever o comportamento realizado por algumas das ferramentas empregadas que interceptam requisições e respostas HTTP/HTTPS antes que estas sejam efetivamente enviadas ao servidor, possibilitando sua análise e modificação durante os testes de segurança.

4.7 FALSOS POSITIVOS EM SEGURANÇA DE APLICAÇÕES

Segundo o glossário do Instituto Nacional de Padrões e Tecnologia (NIST), um falso positivo se trata de um alerta que indica incorretamente a presença de uma vulnerabilidade. No contexto da segurança de aplicações, falsos positivos são comuns em ferramentas de varredura automatizada, que frequentemente identificam comportamentos suspeitos que não representam, de fato, uma falha explorável.

4.8 FERRAMENTAS DE VARREDURA DE SEGURANÇA

Ferramentas de varredura de segurança são amplamente utilizadas para identificar vulnerabilidades, falhas de configuração e comportamentos inseguros em aplicações e infraestruturas. Essas ferramentas automatizam parte do processo de avaliação de segurança, permitindo detectar problemas de forma mais rápida e sistemática, especialmente em aplicações de grande porte.

O objetivo principal de um *scanner* é a enumeração de vulnerabilidades presentes em redes, *hosts* e aplicativos.

O Zed Attack *Proxy* (ZAP) é uma ferramenta de teste de penetração gratuita e de código aberto. Em sua essência, o ZAP atua como um *proxy* interceptando e inspecionado às requisições e respostas que são trocadas entre o navegador e o servidor *web* durante à execução da aplicação.

Semgrep, utilizada como uma solução SAST para realizar a análise estática do código-fonte, essa ferramenta usa regras para detecção de padrões de códigos vulneráveis, *bugs* e regras personalizadas.

5 METODOLOGIA

Para o desenvolvimento deste trabalho foi realizado um estudo de caso para mapear as principais falhas de *software* listadas no CWE, aplicado às principais vulnerabilidades de segurança, que estão presentes no ambiente do OWASP *Juice Shop*. Usando como referência para classificar às vulnerabilidades encontradas a criticidade das categorias listadas no OWASP *Top* 10 de 2021.

Buscando apresentar um conteúdo relevante e com a profundidade necessária para aplicação prática, este trabalho concentrou-se nas falhas de *software* que resultam em vulnerabilidades classificadas entre as quatro primeiras categorias do OWASP *Top* 10 (2021).

A escolha do *Juice Shop* foi devido à sua natureza propositalmente vulnerável e por oferecer uma ampla variedade de funcionalidades que permitem a demonstração prática das vulnerabilidades originadas principalmente na etapa de implementação do SDLC (*Software Development Life Cycle*). No Capítulo 6, foi feita uma associação entre às falhas de *software* mais comuns que originam às vulnerabilidades de segurança que estão classificadas entre as quatro primeiras categorias de vulnerabilidades do OWASP *Top* 10.

Após essa etapa, o próximo passo foi identificar quais vulnerabilidades a aplicação possuía, realizando uma etapa de análise de vulnerabilidades. De acordo com o PTES (*Penetration Testing Execution Standard*), essa etapa envolve o uso de *scanners*, ferramentas que detectam falhas passíveis de exploração por atacantes.

Isso foi feito utilizando métodos ativos e passivos, o teste ativo envolve interação direta com o componente em teste, nesta categoria se enquadra ferramentas automatizadas, *scanners* de aplicativos da *web* (também chamado de *crawlers*) e muitos outros. Já nos testes passivos são feitas atividades como análise de metadados e monitoramento de tráfego, atividades que não necessitam de uma interação direta com o alvo. Neste trabalho foi adotada uma abordagem híbrida utilizando os dois métodos.

Para a aplicação do método ativo foram utilizados dois tipos distintos de *scanners*. Inicialmente, foi empregada a ferramenta Semgrep, utilizada para realizar a análise estática do código-fonte. Em seguida, foi utilizada a ferramenta OWASP ZAP, tanto por meio de sua funcionalidade de varredura automatizada quanto o modo de exploração manual, com a finalidade de analisar páginas da aplicação que exigem autenticação.

Após a análise dos resultados obtidos pelas ferramentas, foi possível identificar falhas comuns que resultaram em vulnerabilidades como SQL *Injection*, *Insecure Direct Object Reference* (IDOR), *Cross-Site Scripting* (XSS), entre outras. Para verificar se as falhas eram, de fato, exploráveis ou se tratavam de falsos positivos, cada uma delas foi testada, aplicando técnicas de exploração comumente empregadas por atacantes.

Por fim, com base nos pontos vulneráveis identificados, foram consultadas recomendações e boas práticas provenientes de organizações reconhecidas na área de segurança da informação, tais como a OWASP, NIST e CWE, com o intuito de propor medidas de mitigação adequadas, fundamentadas na adoção de práticas de codificação segura.

6 MAPEAMENTO ENTRE CATEGORIAS DE FALHAS (CWE) E VULNERABILIDADES (OWASP *TOP* 10)

A relação entre CWE e OWASP emerge do fato de que ambos os referenciais se complementam em níveis distintos. O OWASP Top 10 apresenta categorias gerais de vulnerabilidades (como A03: *Injection* ou A04: *Insecure Design*), enquanto o CWE descreve essas mesmas falhas em sua forma técnica e específica, por meio de identificadores e cenários concretos. Desse modo, pode-se entender que para cada categoria do OWASP existe um conjunto de CWEs subjacentes que, se não mitigados, causam essa vulnerabilidade.

O mapeamento entre ambos, portanto, é essencial para integrar a perspectiva estratégica de gestão de riscos com a execução prática de controles técnicos, estabelecendo um elo entre teoria, prática e mitigação efetiva de ameaças.

A Tabela 1 representa as quatro primeiras categorias do OWASP *Top* 10 exploradas, juntamente com os CWEs associados às falhas identificadas nos desafios resolvidos:

Tabela 1 – Mapeamento entre vulnerabilidades do OWASP Top 10 e fraquezas catalogadas no CWE

Classificação OWASP	Vulnerabilidade OWASP	CWEs
A01	Controle de Acesso Quebrado	CWE-20: Validação de entrada inadequada CWE-287: Autenticação imprópria CWE-352: Falsificação de solicitação entre sites (CSRF) CWE-862: Autorização ausente CWE-863: Autorização incorreta
A02	Falhas Criptográficas	CWE-200: Exposição de informações confidenciais a um ator não autorizado CWE-306: Autenticação ausente para função crítica CWE-326: Força de criptografia inadequada CWE-862: Autorização ausente CWE-916: Uso de *hash* de senha com esforço computacional insuficiente

A03	Injeção	CWE-20: Validação de entrada inadequada CWE-89: Neutralização imprópria de elementos especiais usados em um comando SQL ('Injeção de SQL')
A04	*Design* Inseguro	CWE-209: Geração de mensagem de erro contendo informações sensíveis CWE-213: Exposição de informações sensíveis devido a políticas incompatíveis CWE-269: Gestão Inadequada de Privilégios

O CWE descreve cada uma dessas falhas de *software* como:

- CWE-20: O produto recebe entrada ou dados, mas não valida ou valida incorretamente se a entrada tem as propriedades necessárias para processar os dados de forma segura e correta;
- CWE-89: O produto constrói todo ou parte de um comando SQL usando entradas influenciadas externamente de um componente *upstream*, mas não neutraliza ou neutraliza incorretamente elementos especiais que poderiam modificar o comando SQL pretendido quando este é enviado a um componente *downstream*;
- CWE-200: O produto expõe informações confidenciais a um agente que não está explicitamente autorizado a ter acesso a essas informações;
- CWE-209: O produto gera uma mensagem de erro que inclui informações confidenciais sobre seu ambiente, usuários ou dados associados;
- CWE-213: A funcionalidade pretendida do produto expõe informações a determinados atores de acordo com a política de segurança do desenvolvedor, mas essas informações são consideradas confidenciais de acordo com as políticas de segurança pretendidas de outras partes interessadas, como o administrador do produto, usuários ou outras pessoas cujas informações estão sendo processadas;
- CWE-269: O produto não atribui, modifica, rastreia ou verifica corretamente os privilégios de um ator, criando uma esfera de controle não intencional para esse ator;
- CWE-287: Quando um ator afirma ter uma determinada identidade, o produto não prova ou prova insuficientemente que a afirmação está correta;
- CWE-306: O produto não realiza nenhuma autenticação para funcionalidades que exijam uma identidade de usuário comprovada ou que consumam uma quantidade significativa de recursos;
- CWE-326: O produto armazena ou transmite dados confidenciais usando um esquema de criptografia que é teoricamente sólido, mas não é forte o suficiente para o nível de proteção necessário;
- CWE-352: O aplicativo da *web* não verifica, ou não consegue verificar,

suficientemente se uma solicitação foi fornecida intencionalmente pelo usuário que enviou a solicitação, o que pode ter se originado de um agente não autorizado;

- CWE-862: O produto não executa uma verificação de autorização quando um ator tenta acessar um recurso ou executar uma ação;
- CWE-863: O produto executa uma verificação de autorização quando um ator tenta acessar um recurso ou executar uma ação, mas não executa a verificação corretamente;
- CWE-916: O produto gera um *hash* para uma senha, mas usa um esquema que não fornece um nível suficiente de esforço computacional que tornaria os ataques de quebra de senha inviáveis ou caros.

7 ANÁLISE DE VULNERABILIDADES NO OWASP *JUICE SHOP*

Análise de vulnerabilidade é o processo de descoberta de falhas em sistemas e aplicações que podem ser exploradas por um invasor. Essas falhas podem variar desde configurações incorretas de *host* e serviço até *design* inseguro de aplicações. Embora o processo usado para procurar falhas varie e dependa muito do componente específico testado, alguns princípios fundamentais se aplicam ao processo (PTES, 2014). Os testes aplicados dependem do nível de acesso a um sistema, ao ter o acesso ao código-fonte é possível fazer análises do código em busca de padrões e falhas, um atacante dependendo do nível de exposição da aplicação pode ou não ter acesso ao código-fonte, caso não tenha o escopo dos testes irá ser limitado aos recursos que um usuário teria disponível normalmente.

Essa etapa de análise foi fundamental para a resolução dos desafios, pois muitos anti-padrões, trechos de código vulneráveis e bibliotecas desatualizadas foram detectados, orientando os pontos de exploração. Como mencionado anteriormente foram utilizadas duas ferramentas, uma de análise estática do código-fonte (Semgrep) e um *scanner* de segurança dinâmica (ZAP).

A integração do Semgrep com o repositório do projeto foi realizada por meio de autenticação via GitHub, onde está hospedado o projeto do OWASP *Juice Shop*. Como mostrado na Figura 1, o Semgrep disponibiliza múltiplas opções de autenticação e fontes de código.

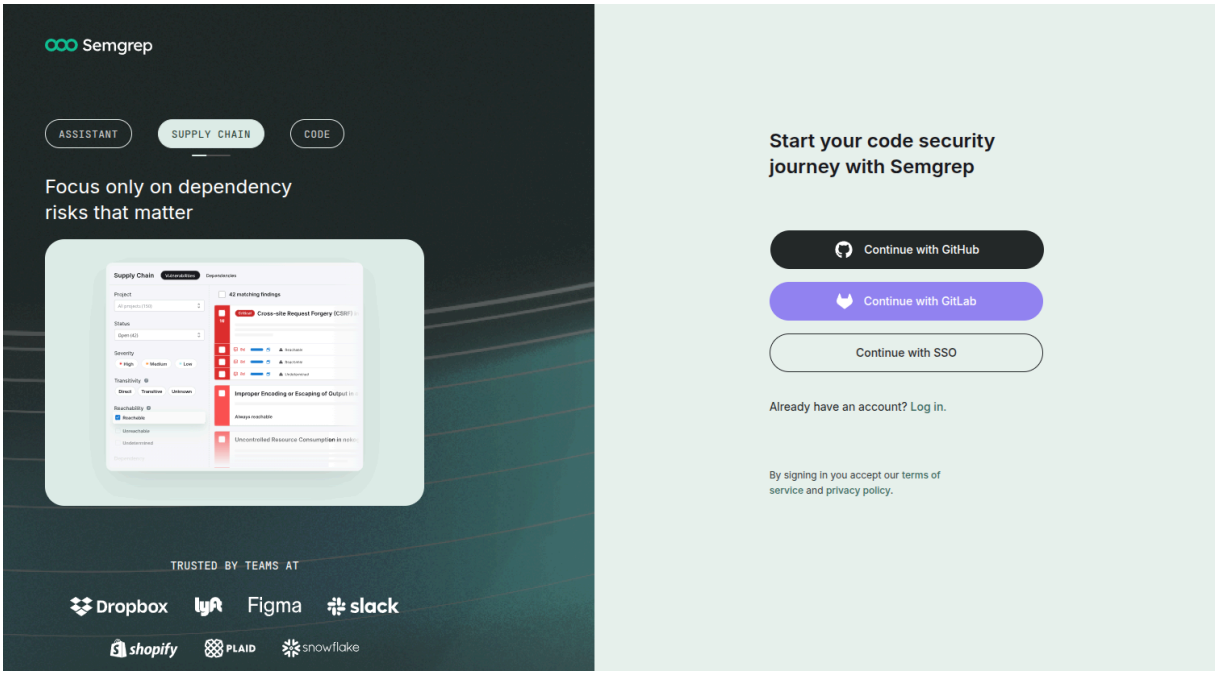

Figura 1 – Página de integração do Semgrep com provedores de código.

Ao conectar o Semgrep ao GitHub, é necessário especificar quais repositórios serão analisados ou permitir acesso a todos os repositórios da conta (ver Figura 2).

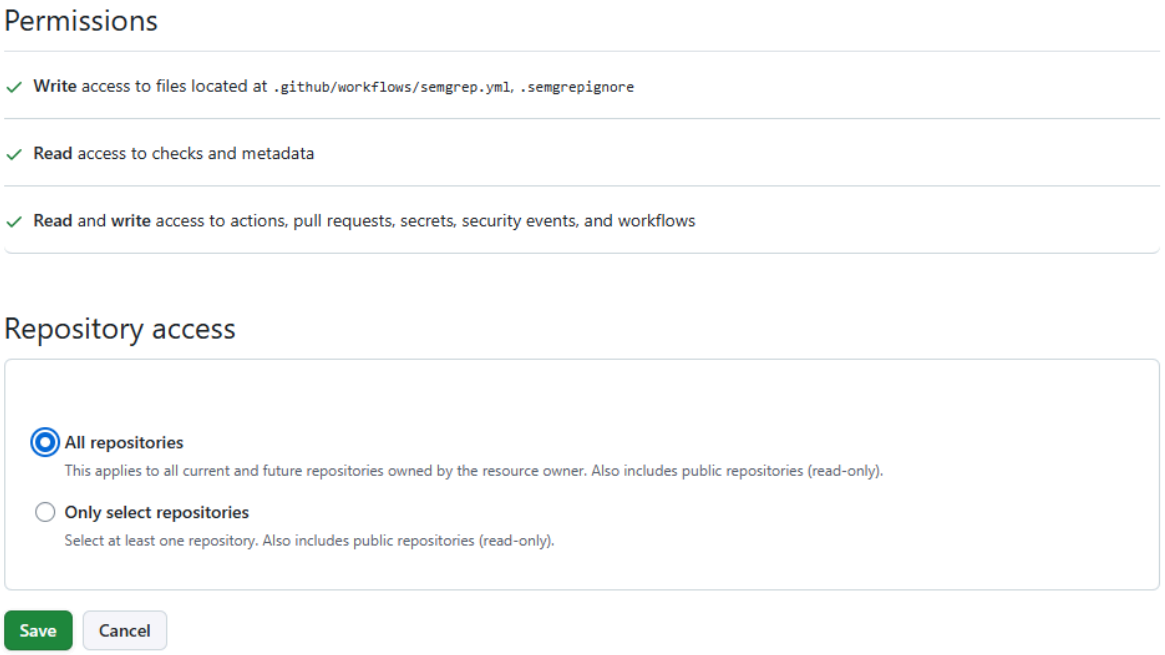

Figura 2 – Página de permissão de acesso aos repositórios do GitHub.

Após a integração, à aplicação redireciona o usuário para um painel (*Dashboard*) que ainda está vazio. No menu lateral esquerdo ao clicar em *Projects*, uma página com duas abas é aberta. Os repositórios autorizados aparecem na aba *Not Scanning*, selecionando algum dos repositórios listados, um botão chamado *Enable Managed Scans* será habilitado conforme ilustra a Figura 3, clicando nele o repositório é movido para a aba *Scanning* e o processo de análise do código é iniciado.

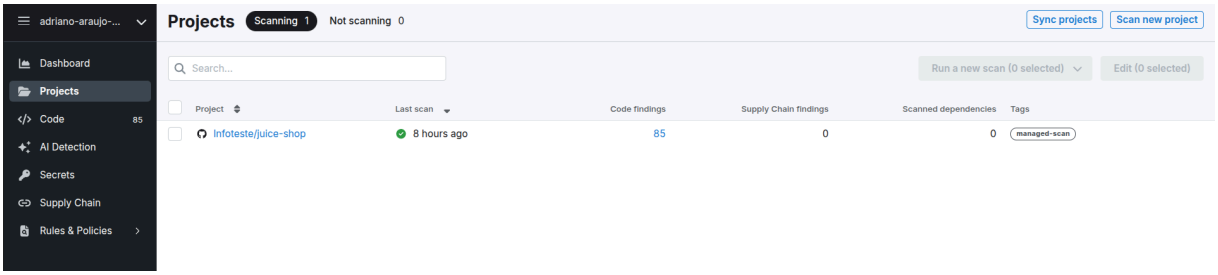

Figura 3 – Habilitação de varreduras gerenciadas no painel do Semgrep.

Concluído o processo de análise do código, é possível acessar a página de detalhes do repositório, onde são listados todos os problemas identificados, com trechos de código, severidade e sugestões de correção conforme é mostrado na Figura 4.

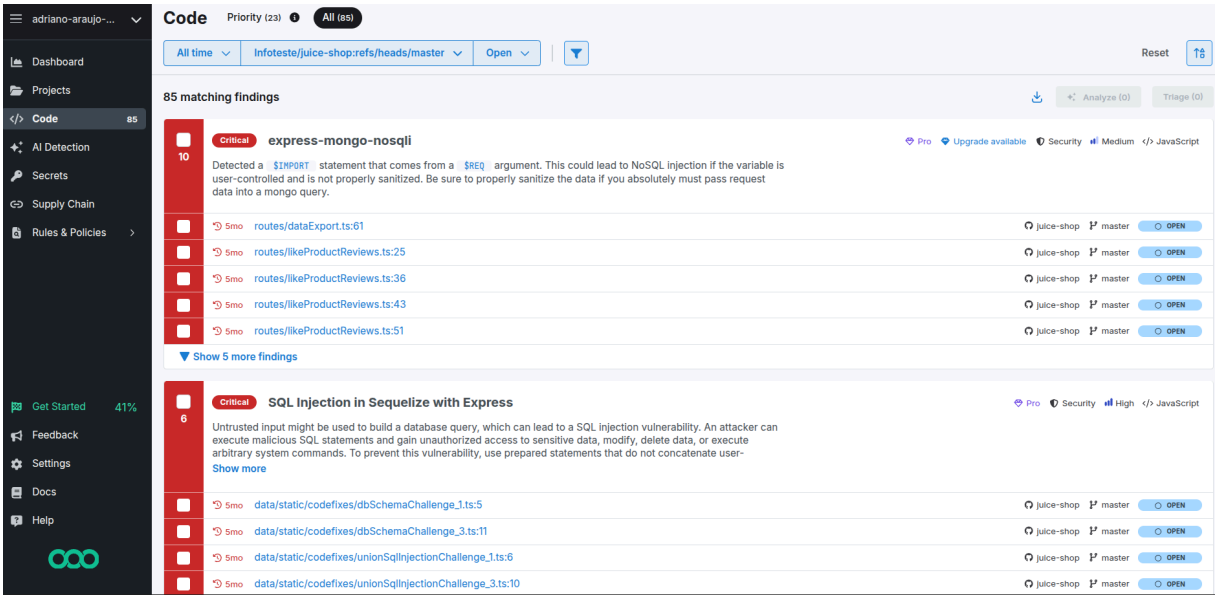

Figura 4 – Página de detalhes de uma varredura no Semgrep.

Para utilizar o OWASP ZAP, foi necessário realizar à sua instalação acessando o site oficial da ferramenta, há um botão de *Download* na página inicial (ver Figura 5).

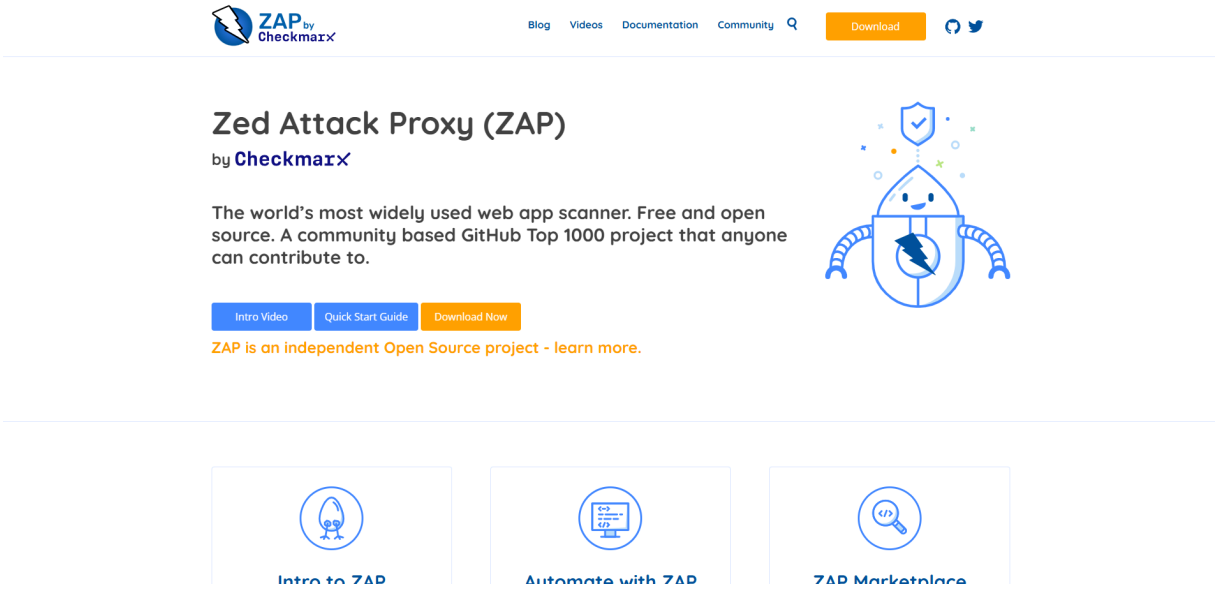

Figura 5 – Página inicial do site do OWASP ZAP.

Clicando no botão *Download* é exibida uma página com diversas opções de instaladores (ver Figura 6) para múltiplos sistemas operacionais, para a realização deste trabalho, foi utilizada a opção de executável para o sistema operacional Windows. Após o *Download*, basta dar um duplo clique para iniciar a instalação e seguir as instruções do instalador.

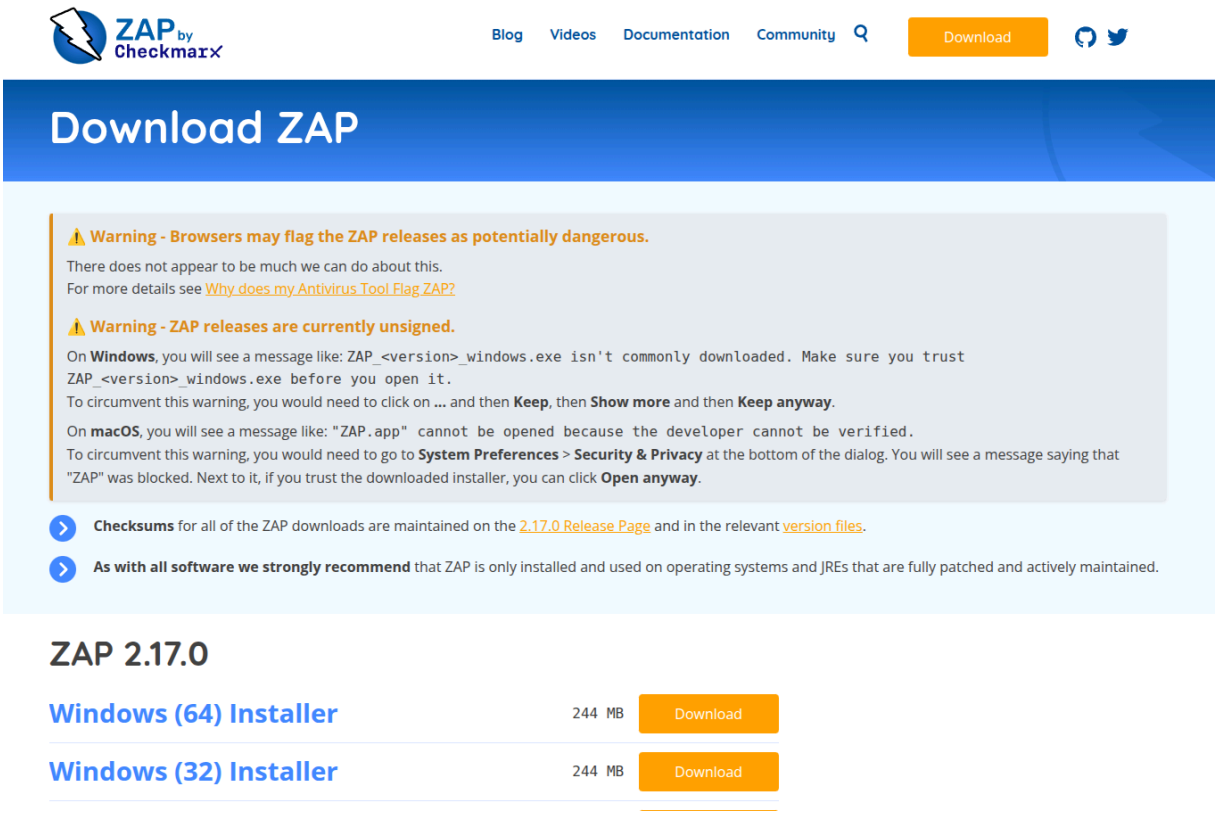

Figura 6 – Página de *download* do OWASP ZAP.

Concluída a instalação, ao iniciar a ferramenta, o usuário é questionado se deseja persistir com a sessão atual ou prosseguir sem salvá-la. Após selecionar uma das opções, a página inicial do ZAP é exibida, apresentando suas duas principais funcionalidades, conforme ilustrado na Figura 7.

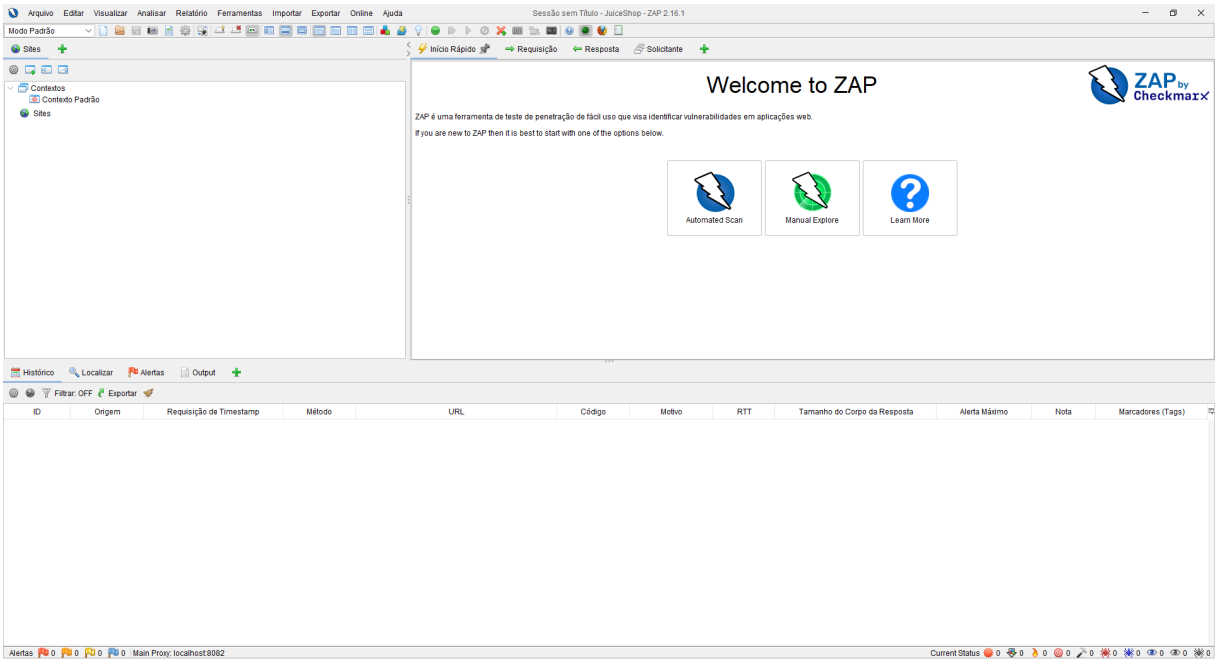

Figura 7 – Página inicial da ferramenta ZAP.

A primeira funcionalidade trata-se de uma varredura automatizada (*Automated Scan*). A varredura automatizada é composta por diferentes etapas. A primeira consiste no mapeamento das páginas da aplicação a partir da URL informada, indexando as rotas e realizando um escaneamento passivo de todas as solicitações e respostas enviadas por *proxy* através de um mecanismo do ZAP conhecido como *web crawling* ou *web spidering*.

O ZAP oferece dois tipos de *spiders* para essa etapa:

1. *Spider* tradicional – descobre links examinando o HTML contido nas respostas HTTP;
2. Spider para aplicações AJAX – projetado especificamente para explorar aplicações que utilizam JavaScript para gerar *links* e carregar conteúdo.

Em seguida, o ZAP usará o *scanner* ativo para tentar explorar todas as páginas, funcionalidades e parâmetros descobertos (ZAP, 2025).

A segunda funcionalidade é a exploração manual (*Manual Explore*), em que o usuário navega pela aplicação de forma natural, enquanto o ZAP escaneia passivamente todas as requisições e respostas feitas durante esse fluxo, identificando possíveis vulnerabilidades e emitindo novos alertas. Os *spiders* são uma ótima maneira de explorar seu *site* básico, mas devem ser combinados com a exploração manual para serem mais eficazes. Os *spiders*, por exemplo, inseriram apenas dados básicos padrões em formulários do seu aplicativo *web*, enquanto um usuário pode inserir informações mais relevantes, o que, por sua vez, pode expor mais partes do aplicativo ao ZAP (ZAP, 2025).

Com a URL da aplicação *web* definida, bastou clicar em Ataque para iniciar a varredura automatizada. Como o *Juice Shop* estava sendo executado localmente, foi necessário também incluir a porta na URL, conforme mostrado na Figura 8.

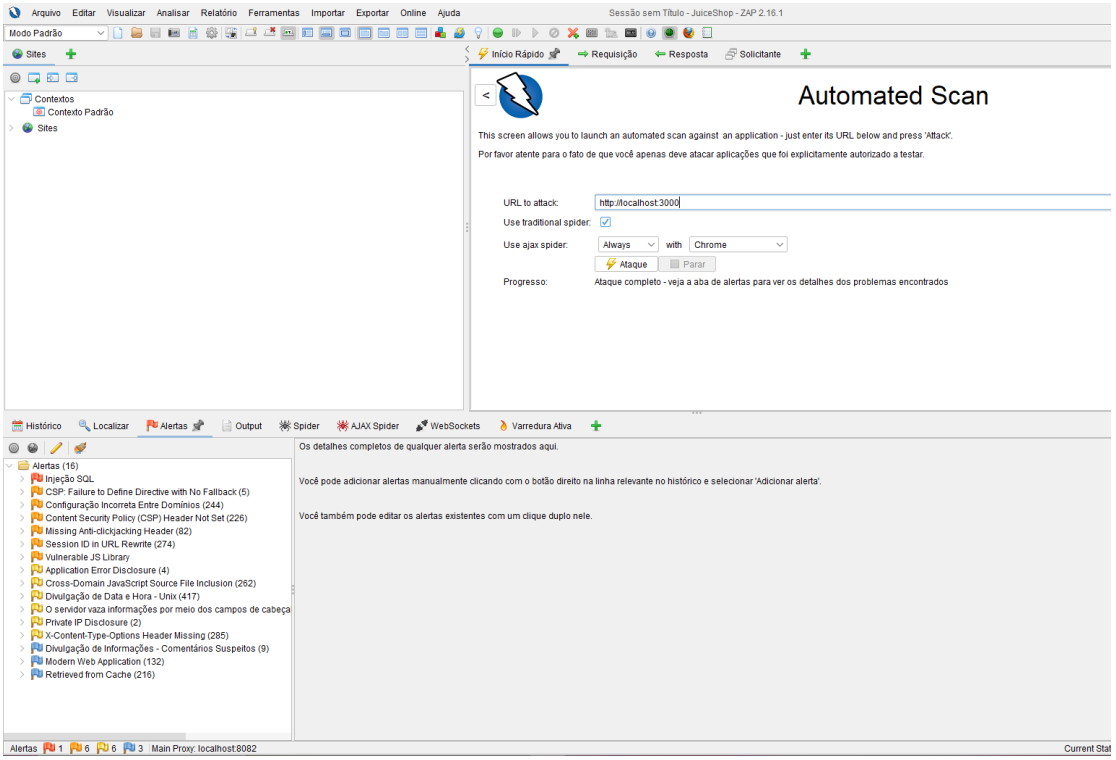

Figura 8 – Configuração da varredura automatizada.

Após a conclusão das etapas da varredura automatizada, o ZAP registrou diversos alertas, incluindo alguns classificados como de alto risco, conforme mostra a Figura 9.

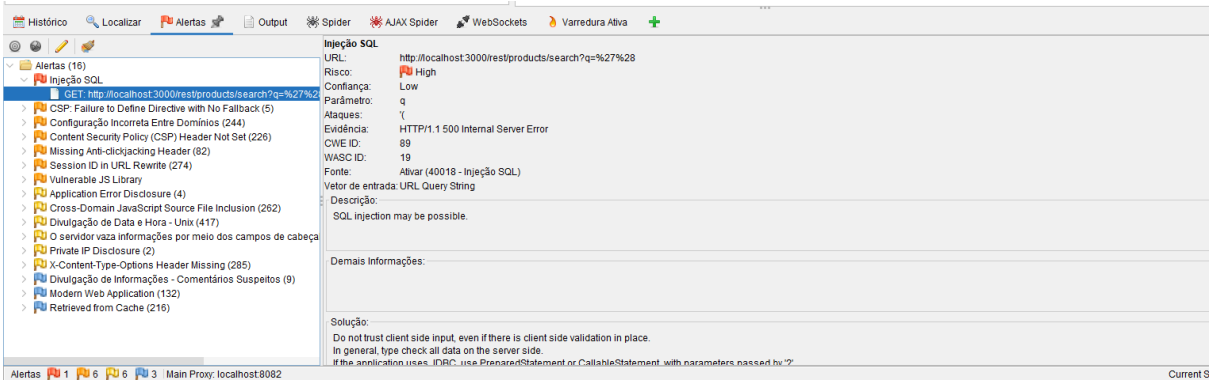

Figura 9 – Alertas gerados após a varredura automatizada.

Em seguida, iniciou-se a exploração manual. Nessa etapa, foi necessário informar novamente a URL, à porta da aplicação e selecionar um navegador compatível (ver Figura 10).

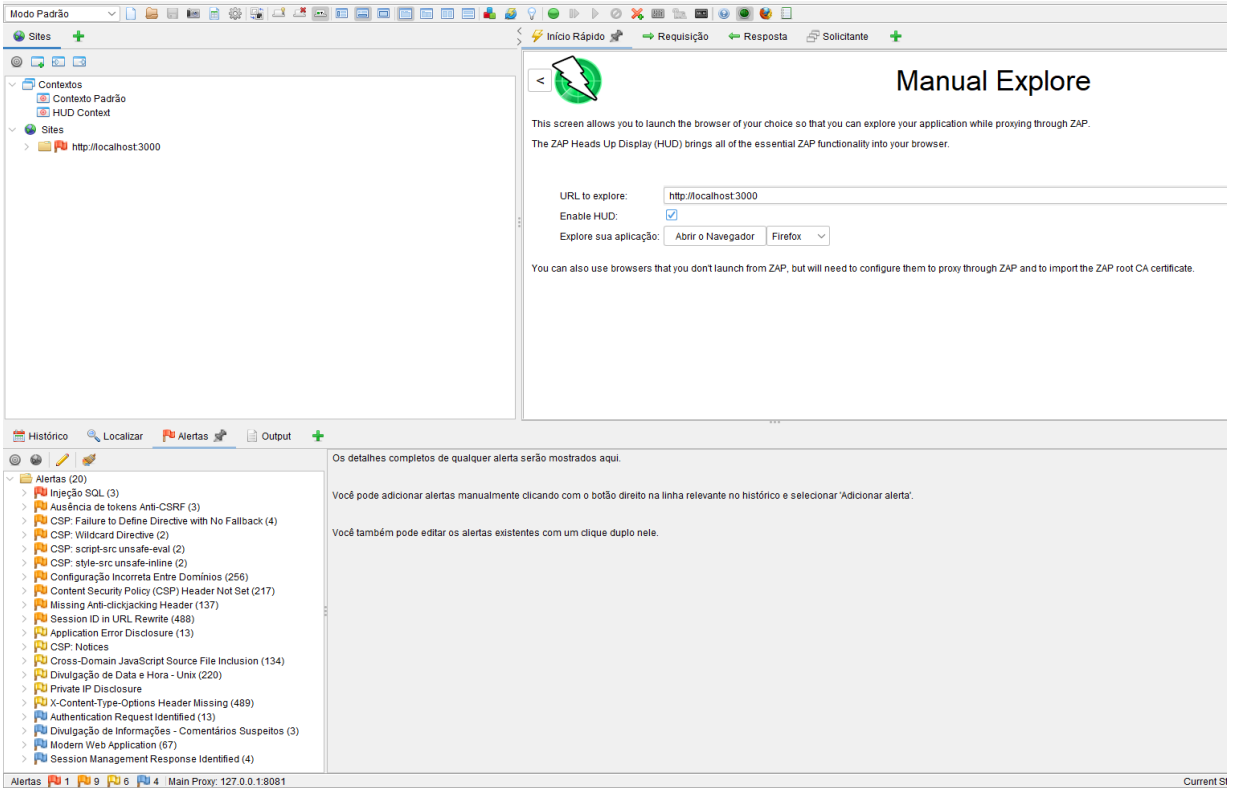

Figura 10 – Configuração da exploração manual.

Ao abrir o navegador, a aplicação é carregada com o **HUD (*Head-Up Display*)** do ZAP, ilustrado na Figura 11.

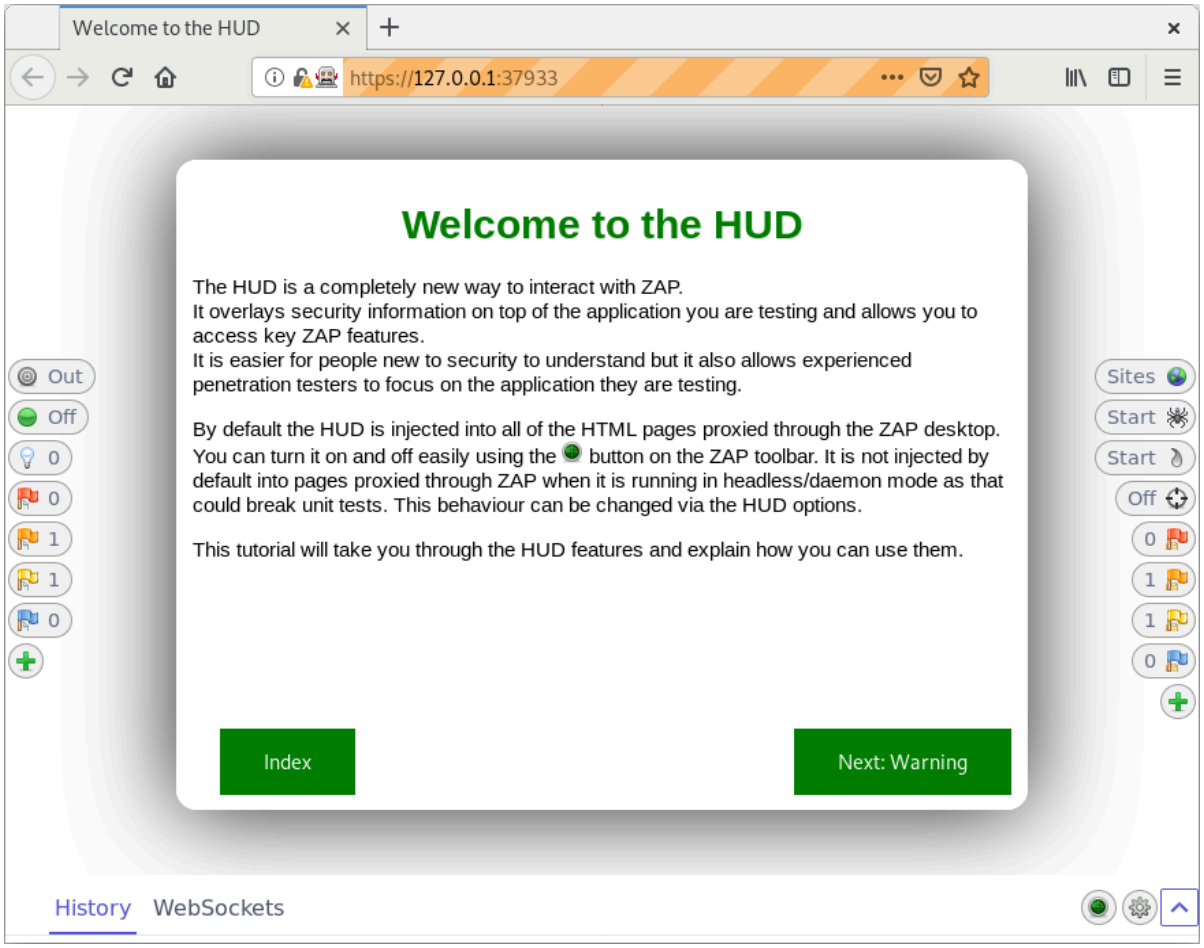

Figura 11 – Interface HUD do OWASP ZAP.

Autor: ZAP, 2025

A exploração manual é ideal para testar áreas da aplicação que exigem autenticação, não alcançadas pela varredura automatizada.

Após explorar toda a aplicação do *Juice Shop* e retornar ao painel do ZAP, alguns novos alertas foram gerados, indicando a presença de vulnerabilidades confirmadas e também de possíveis vulnerabilidades. Ao clicar em um alerta é possível visualizar informações detalhadas, como o nível de risco, a descrição da vulnerabilidade, *links* para documentações, recomendações específicas e o *payload* utilizado durante o ataque, conforme mostrado na Figura 12.

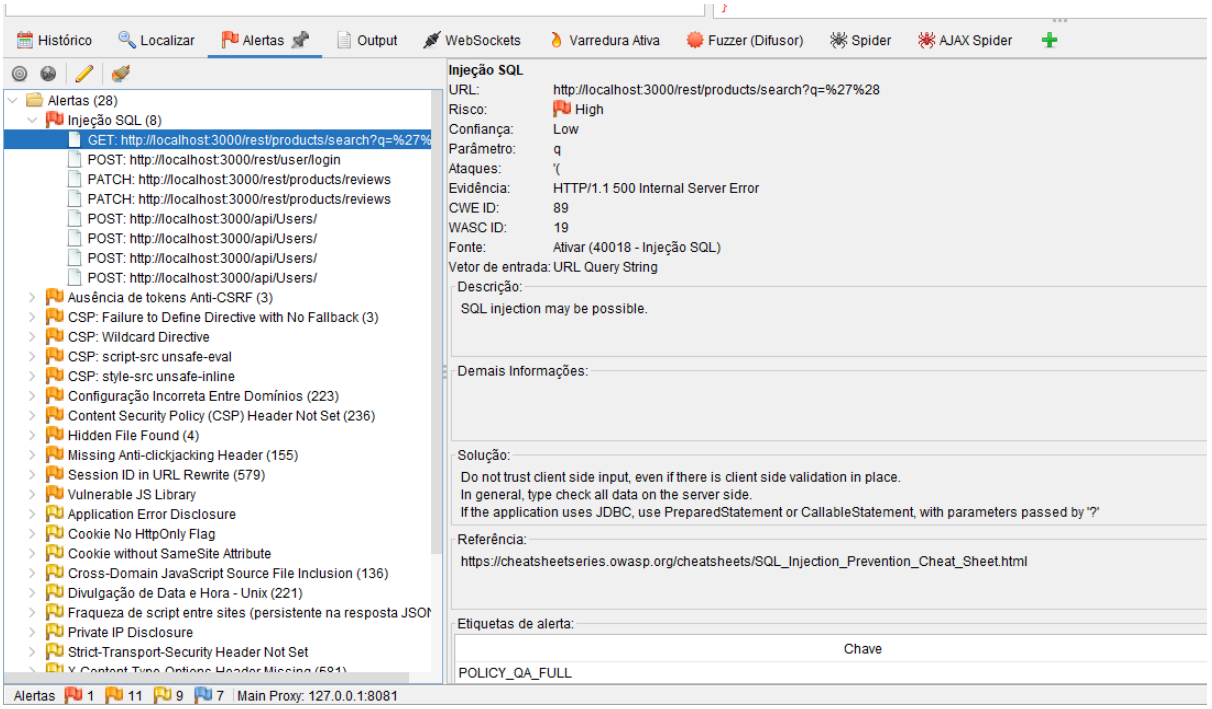

Figura 12 – Detalhes dos alertas identificados após a exploração manual.

Algumas das vulnerabilidades detectadas durante as etapas de exploração com o ZAP, geraram alertas contendo os *payloads* utilizados durante os testes.

Os alertas produzidos por ambas às ferramentas foram fundamentais para a etapa de correção e mitigação das falhas abordadas no Capítulo 8, servindo como base para a análise e compreensão das vulnerabilidades.

Essa abordagem híbrida, combinando ferramentas SAST e DAST, permite compreender tanto o comportamento estático quanto o comportamento dinâmico da aplicação, possibilitando uma visão mais completa de suas possíveis fragilidades.

8 DESENVOLVIMENTO

O OWASP *Juice Shop* utiliza um modelo de gamificação, no qual cada desafio concluído corresponde a uma falha de segurança que foi explorada com sucesso. Nesse sentido, as vulnerabilidades identificadas na etapa anterior serão analisadas para estabelecer a relação com os desafios correspondentes.

O processo para cada desafio consiste na identificação das falhas que possibilitaram a exploração, a descrição de como um atacante poderia se beneficiar dessas falhas juntamente com os recursos empregados na exploração e, por fim, a apresentação das técnicas de mitigação, acompanhadas de boas práticas de desenvolvimento seguro.

É importante destacar que as técnicas de mitigação descritas são independentes de linguagem de programação ou *framework* específico. O conceito de segurança é o que importa, cada *framework* e linguagem terá seus próprios recursos e sintaxes para lidar com as falhas apresentadas.

8.1 A01 - CONTROLE DE ACESSO QUEBRADO (*BROKEN ACCESS CONTROL*)

Os desafios explorados nesta seção, representam algumas das causas mais recorrentes das vulnerabilidades relacionadas à quebra de controle de acesso.

8.1.1 Desafio - Alterar o nome de um usuário executando Cross-Site *Request Forgery* de outra origem.

Este desafio tem como objetivo permitir que seja explorada uma falha de CSRF (*Cross-Site Request Forgery*) para alterar os dados de um usuário autenticado.

Falhas de *software*:

Essa vulnerabilidade está associada ao CWE-352, que descreve situações em que o aplicativo da *web* não verifica, ou não consegue verificar, suficientemente se uma solicitação foi fornecida intencionalmente pelo usuário que enviou a solicitação, o que pode ter se originado de um agente não autorizado (CWE, 2024).

Exploração da Vulnerabilidade:

À falha ocorre devido à forma como o *token* JWT (JSON *Web Token*) é manipulado. Após o usuário autenticar-se, um *token* JWT é retornado no corpo da resposta HTTP como indicado na Figura 13.

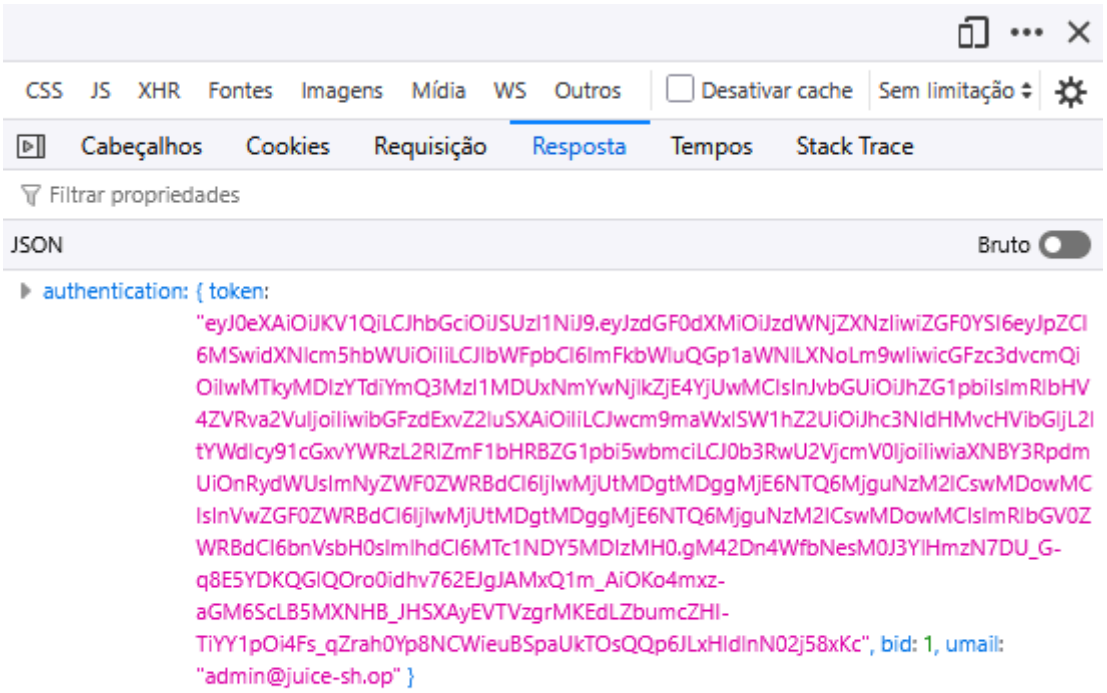

Figura 13 – Corpo da resposta da requisição de *login*

Nesta aplicação o *frontend* fica responsável por armazenar o *token* nos *cookies* do navegador. Essa prática é insegura, pois o *cookie* é salvo sem o atributo SameSite, deixando sua configuração a critério do navegador. O atributo SameSite é uma medida de segurança que restringe o envio de *cookies* em requisições entre *sites*. Quando não definido ou configurado de forma inadequada, o navegador pode enviar *cookies* de autenticação em solicitações originadas de páginas maliciosas. A Figura 14 apresenta o painel de alertas da ferramenta OWASP ZAP, no qual foi detectada a ausência desse atributo na definição dos *cookies*.

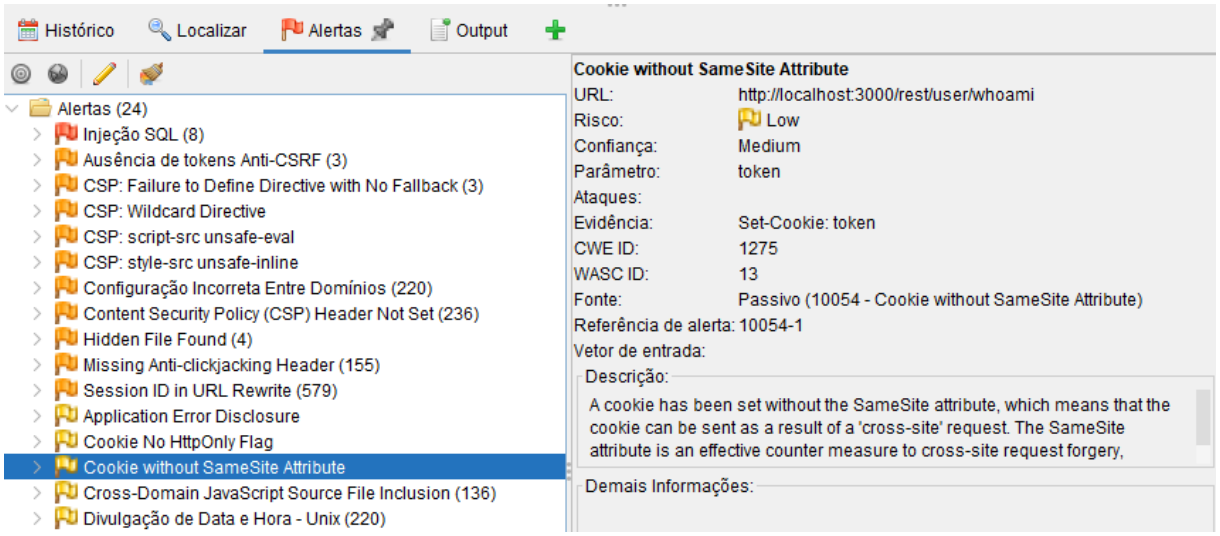

Figura 14 – Painel de alertas do OWASP ZAP

Ao definir esse atributo diretamente no cabeçalho de resposta da requisição (Set-Cookie), o servidor garante que o *cookie* não será enviado junto com

solicitações *cross-site*, protegendo assim contra ataques de falsificação de solicitações entre sites (CSRF). A seguir, na Figura 15, pode ser observado que nenhum *cookie*, juntamente com suas propriedades de segurança, está sendo definido no cabeçalho de resposta da requisição.

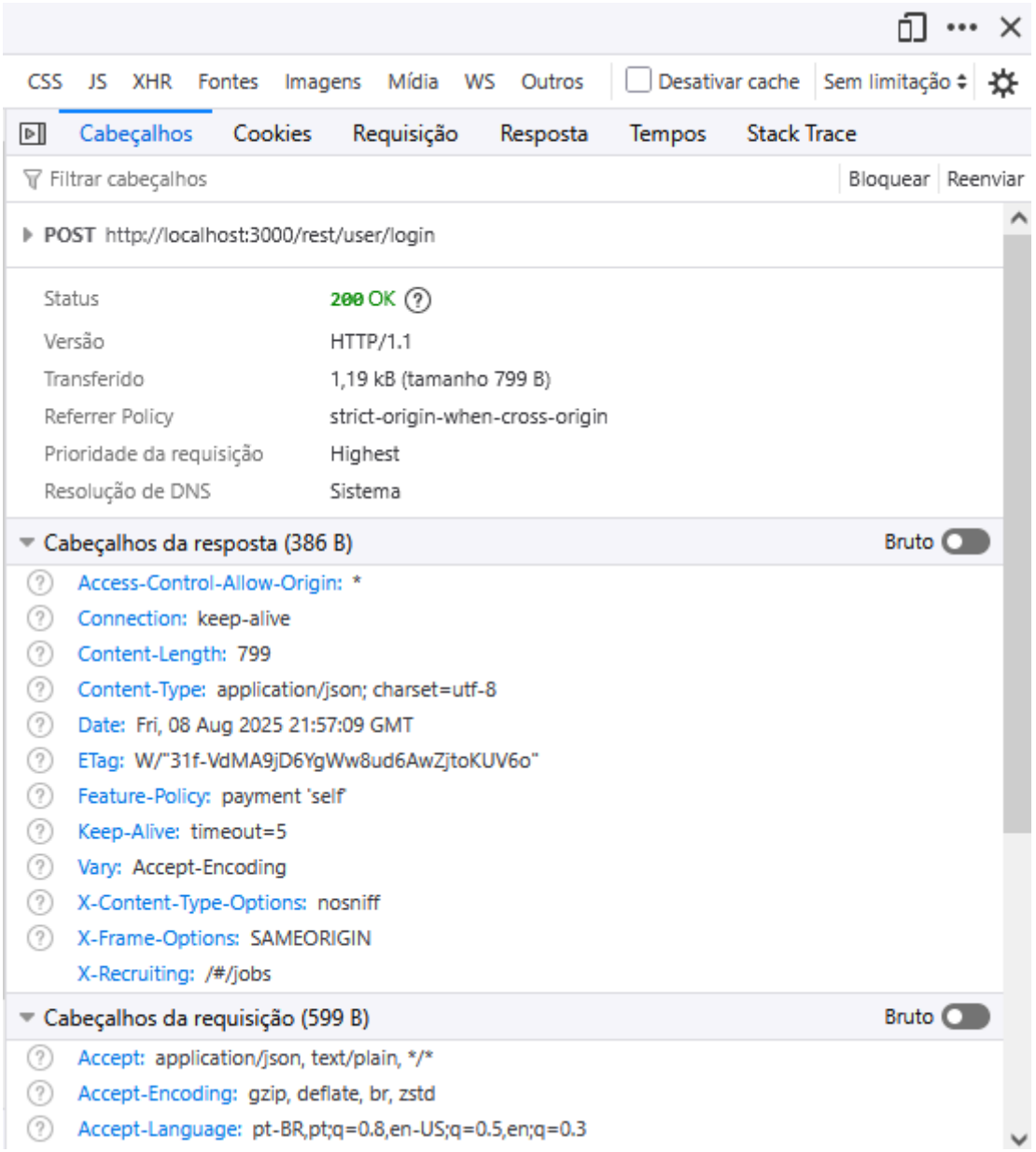

Figura 15 – Cabeçalhos da requisição de *login*

Um usuário mal intencionado pode explorar esse comportamento criando uma página maliciosa que contém um formulário oculto ou código JavaScript, projetado para enviar uma requisição forjada de alteração de dados ao *Juice Shop*. Se a vítima já estiver autenticada na aplicação, ao acessar a página maliciosa, o navegador inclui automaticamente os *cookies* nessa solicitação entre *sites*, em razão da ausência de uma política adequada para o atributo SameSite.

Dessa forma, a solicitação parecerá legítima para o servidor, que processará a alteração dos dados do usuário sem qualquer validação adicional. Esse tipo de ataque é comumente disseminado por meio de *e-mails* de *phishing* ou *links* inseridos em sites comprometidos.

Esse comportamento do navegador pode variar de acordo com a versão utilizada, já que, atualmente, a maioria dos navegadores modernos adota políticas padrão para reduzir o risco desse tipo de ataque, configurando automaticamente os *cookies* com o atributo SameSite definido como Lax ou Strict.

Correção e Mitigação:

Existem diversas maneiras de prevenir ataques CSRF, como a utilização de *tokens* Anti-CSRF, a verificação do cabeçalho de origem da solicitação, a configuração do atributo SameSite, entre outras. O ideal é combinar esses recursos para obter uma defesa mais robusta. Na Figura 16 os alertas gerados pelo OWASP ZAP, listam a ausência de *tokens* Anti-CSRF, destacando a relevância da adoção dessas práticas preventivas.

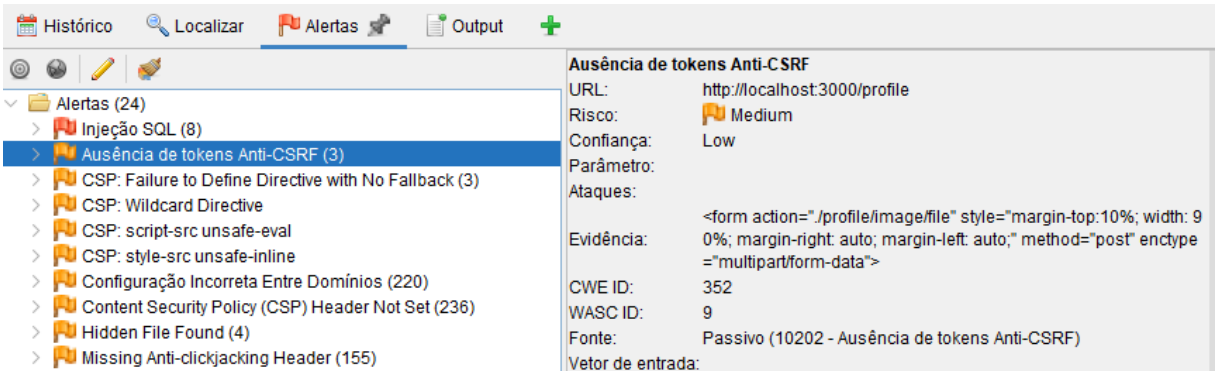

Figura 16 – Alerta de ausência de *tokens* Anti-CSRF no painel do OWASP ZAP

No entanto, isso demanda maior conhecimento técnico e a implementação de novas funcionalidades, considerando o que já existe na aplicação, uma medida viável foi corrigir a forma como o atributo SameSite é definido. A mitigação adequada envolve configurar corretamente os *cookies* de autenticação no *backend*, evitando que o *frontend* seja responsável por essa tarefa.

O cabeçalho de resposta HTTP deve definir o *token* através da função Set-Cookie, com atributos de segurança configurados, como HttpOnly, *Secure* e SameSite. O valor atribuído ao SameSite deve ser definido de acordo com o contexto da aplicação. Por exemplo, se houver subdomínios que necessitam do *token* ou integração com outros sites, definir como Strict poderá atrapalhar essa funcionalidade. Um exemplo de como essa configuração pode ser definida no *backend* (utilizando a sintaxe do Express/Node.js) é mostrado a seguir:

```
response.cookie('token', token, { expires: new Date(Date.now() + 8 * 60 * 60 * 1000),
sameSite: 'strict', httpOnly: true })
```

8.1.2 Desafio - Publique algum *feedback* em nome de outro usuário

O *Juice Shop* possui um recurso que permite criar *feedbacks* que são exibidos em uma página pública de avaliações. Esses *feedbacks* podem ser enviados tanto de forma anônima quanto autenticada. Contudo, o recurso não possui validação suficiente para impedir que um usuário se passe por outro.

Falhas de *software*:

Essa vulnerabilidade está associada a duas falhas principais, à falta de verificação de autorização (CWE-862) e validação de identidade (CWE-287).

Nesse caso, o usuário malicioso consegue criar um *feedback* se passando por outro usuário, pois a aplicação não valida se o autor da requisição é, de fato, o mesmo que está sendo atrelado ao comentário. Pois, o que identifica o autor do *feedback* é um parâmetro chamado UserId, enviado no corpo da requisição, conforme ilustrado na Figura 17.

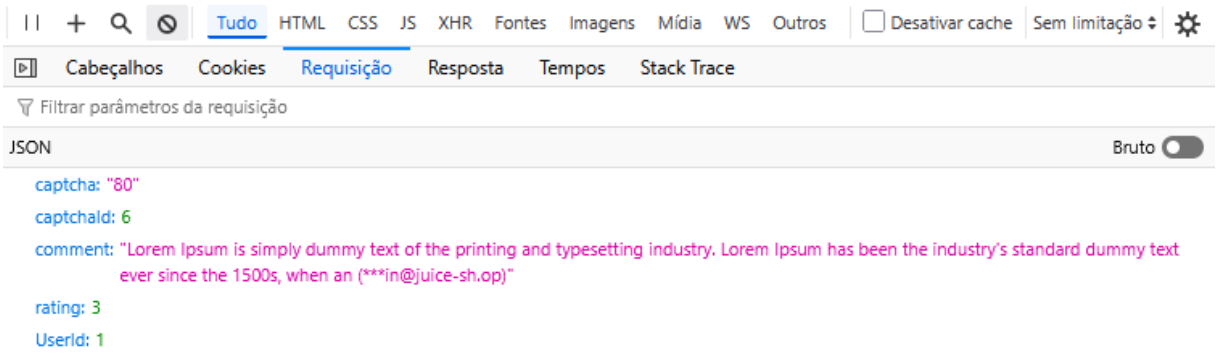

Figura 17 – Corpo da requisição de novo feedback

Exploração da Vulnerabilidade:

Para explorar essa falha, basta analisar as diferenças entre as requisições realizadas quando se cria um *feedback* autenticado e de forma anônima. A única diferença relevante é o valor do parâmetro UserId que é enviado quando o usuário está autenticado na aplicação. Existem diversas formas de obter o UserId de outros usuários, sendo a mais fácil inspecionar outras áreas da própria aplicação que realizem requisições contendo o UserId , na própria listagem pública de *feedbacks* é possível descobrir esses dados, assim como em outras áreas que listam usuários.

Outras maneiras que costumam ser utilizadas pelos atacantes é procurar por vazamentos de informações da respectiva aplicação e utilizar ataques de força bruta para descobrir identificadores válidos. Uma vez que o UserId de outro usuário foi obtido, foi então incluído no corpo da requisição que cria um *feedback*. A aplicação, sem qualquer validação adicional, simplesmente executa essa requisição, sem retornar qualquer tipo de erro.

Correção e Mitigação:

Ao analisar como à funcionalidade de *feedbacks* foi implementada, foi possível identificar diversas oportunidades de melhoria. A primeira delas é a forma como o UserId é gerado. Atualmente, ele é uma sequência numérica auto incremental, essa prática é desaconselhada, pois, além de previsível, contribui para a exploração de vulnerabilidades, permitindo que atacantes deduzam facilmente identificadores válidos. Para resolver esse problema, a recomendação é utilizar GUIDs (*Globally Unique Identifiers*). Por serem identificadores mais complexos e aleatórios, o uso deles dificultam, e em alguns casos tornam praticamente impossível para um atacante adivinhar valores válidos, aumentando a segurança do sistema.

No entanto, mesmo com identificadores complexos, a implementação de verificações de controle de acesso é fundamental. Isso leva à outro problema encontrado na arquitetura: a requisição que cria o *feedback* inclui o UserId do autor.

A melhor prática em operações que dependem dos dados do usuário autenticado é recuperar essas informações diretamente no servidor, e não confiar nos dados enviados pelo cliente.

Com o usuário autenticado na aplicação as requisições realizadas por ele contém um *token* que identifica unicamente esse usuário e que é transmitido a cada solicitação, o ideal, então, seria que no lado do servidor esse *token* fosse usado para recuperar os dados desse usuário e, então, criar o *feedback* utilizando o UserId dos dados recuperados.

Essa abordagem resolve dois problemas de uma só vez, evita que um usuário mal-intencionado se passe por outro, já que ele não consegue manipular o UserId e fortalece o controle de acesso, pois impede a inserção de um *feedback* em nome de outro usuário, garantindo que apenas o próprio usuário autenticado possa criar *feedbacks* vinculados à sua conta.

8.1.3 Desafio - Coloque um produto adicional no carrinho de compras de outro usuário

Este desafio além de explorar uma vulnerabilidade de controle de acesso quebrado, também envolve outras vulnerabilidades presentes na aplicação.

Falhas de *software*:

As vulnerabilidades exploradas neste desafio são semelhantes às do anterior, porém apresentam algumas diferenças importantes. Neste caso, há uma verificação de autorização, mas ela é implementada de forma incorreta (CWE-863). Além disso, para que o ataque seja bem-sucedido, é necessário explorar outra categoria de falha relacionada à validação incorreta de entradas (CWE-20).

Exploração da Vulnerabilidade:

O primeiro passo para compreender o ataque consiste em entender como a funcionalidade de adicionar um produto ao carrinho de compras funciona. A Figura 18 mostra o corpo da requisição que é responsável por essa ação.

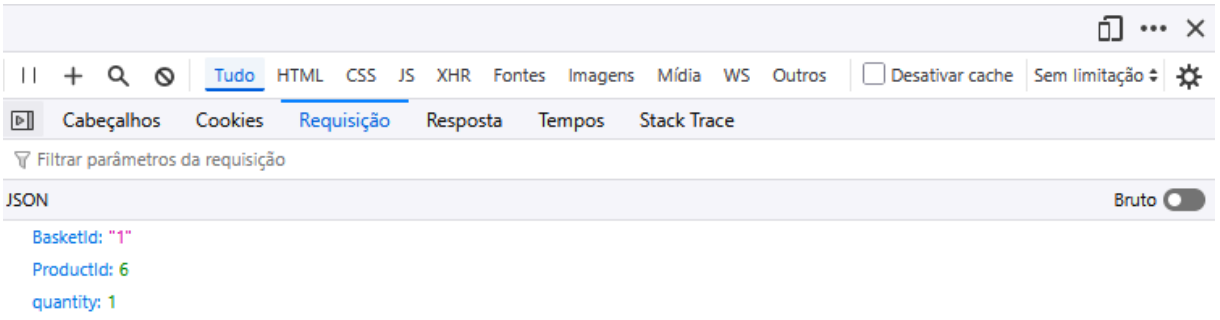

Figura 18 – Corpo da requisição de inserção de produto no carrinho de compras

À primeira vista, o único parâmetro de destaque é o BasketId, apesar do seu nome não ser descritivo, ao observar as chamadas realizadas pela aplicação na área do carrinho de compras, nota-se que o mesmo identificador é utilizado em uma requisição que lista os produtos já adicionados no carrinho. A Figura 19 mostra a requisição responsável pela listagem, e a Figura 20, os dados retornados por ela.

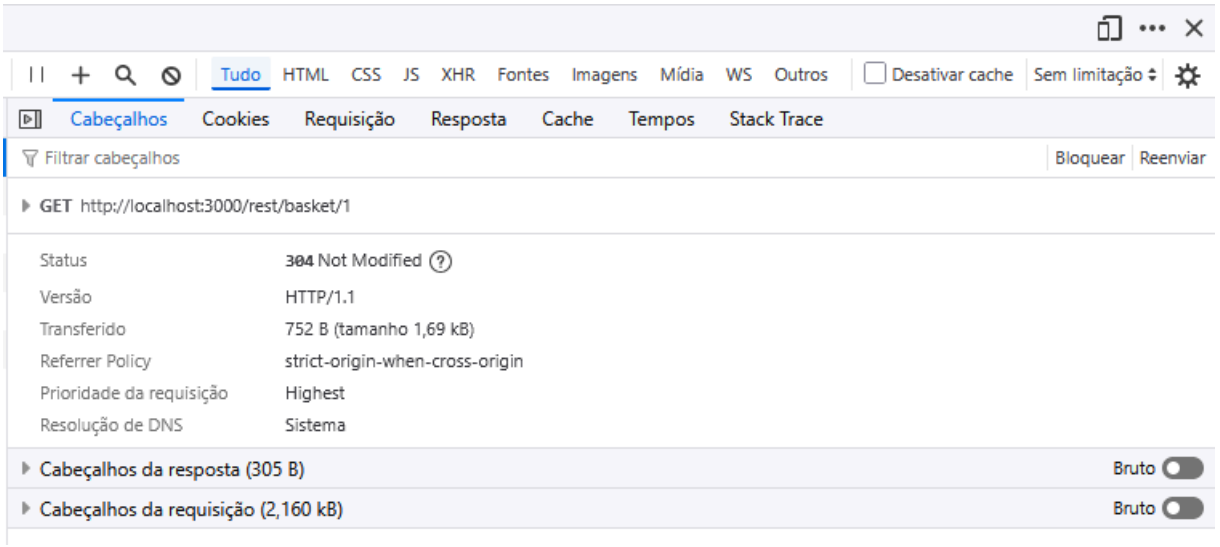

Figura 19 – Cabeçalho da rota de listagem dos produtos no carrinho

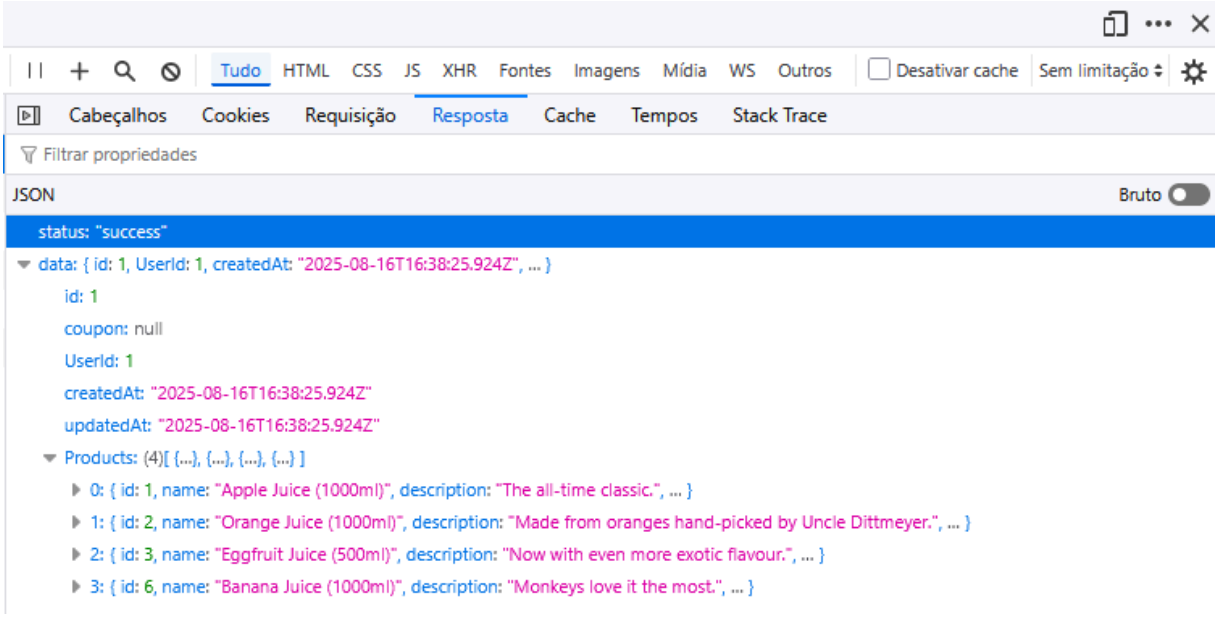

Figura 20 – Dados retornados pela requisição

Ao identificar que essa rota retorna os produtos de um determinado BasketId e que esse identificador parece ser uma sequência numérica auto incremental, foi realizado um ataque de força bruta para verificar se a aplicação retornaria os produtos do carrinho de outro usuário. Como esperado, à aplicação retornou os dados de outro carrinho, confirmando que o identificador era válido e auto incremental, além de revelar uma vulnerabilidade de IDOR.

De posse do identificador do carrinho de compras de outro usuário, o próximo passo foi tentar utilizá-lo na requisição de inserção de produtos no carrinho. No entanto, a tentativa resultou em um erro HTTP 401 *Unauthorized*, como exibido na Figura 21.

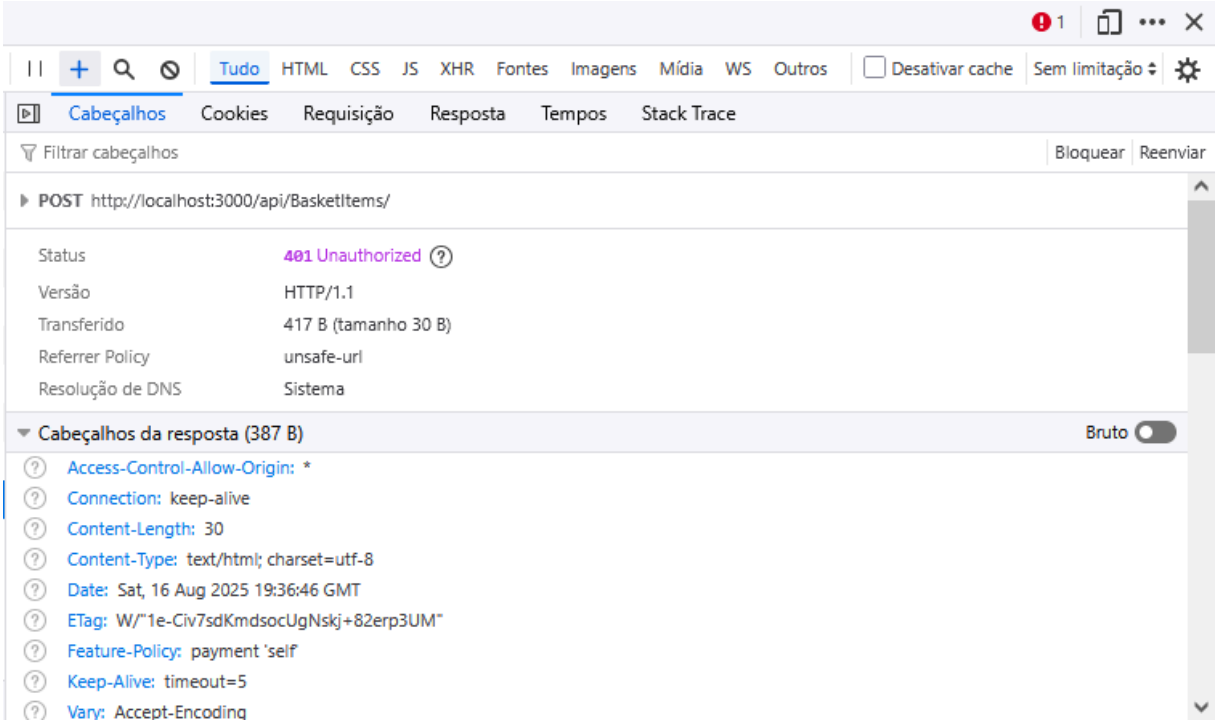

Figura 21 – Requisição de inserção de produto recusada

Esse erro indica que existe algum tipo de controle de autorização nessa funcionalidade, sabendo disso, foram aplicadas diferentes técnicas comumente utilizadas por atacantes na tentativa de contornar esse controle, até que uma delas obteve êxito: a poluição de parâmetros HTTP. Essa técnica consiste em enviar o mesmo parâmetro com valores diferentes para ver como a aplicação lida com a ambiguidade.

Neste caso, o parâmetro BasketId foi enviado duas vezes: uma com o identificador do carrinho do atacante e outra com o identificador de outro usuário. A aplicação, então, processou o primeiro parâmetro, que pertencia ao usuário autenticado, mas usou o valor do segundo parâmetro para realizar a inserção. Esse comportamento é ilustrado na Figura 22.

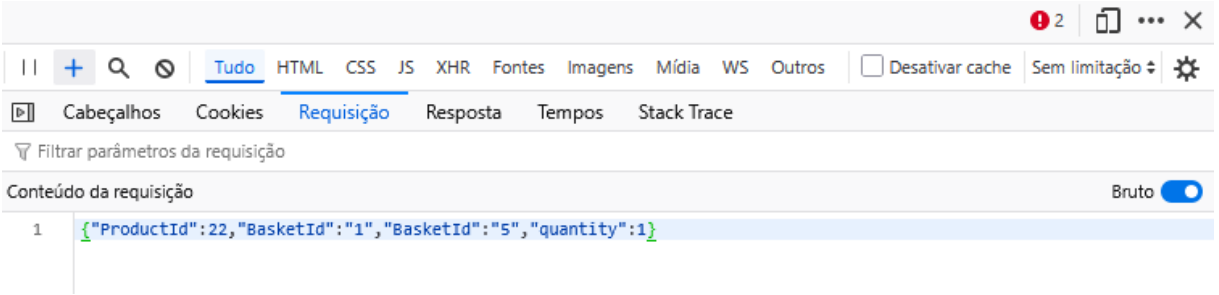

Figura 22 – Corpo da requisição poluído

A comunicação entre cliente e servidor ocorre por meio do formato JSON (*JavaScript Object Notation*), embora o JSON não proíba explicitamente a utilização de múltiplos parâmetros com o mesmo nome, essa é considerada uma má prática de

desenvolvimento. O comportamento do servidor ao processar esses dados depende do *parser* da linguagem ou do *framework* utilizado. Muitos deles podem aceitar a primeira ocorrência do parâmetro, a última ou em alguns casos, gerar erros.

A exploração dessa vulnerabilidade só foi possível devido a uma implementação falha no servidor, que recebia a requisição e transformava o JSON em uma lista de parâmetros sem verificar se havia múltiplos valores para a mesma chave.

Enquanto um usuário comum, em um fluxo normal de utilização, não alteraria os parâmetros da requisição, um atacante busca e explora justamente as inconsistências e possíveis falhas cometidas durante a etapa de desenvolvimento.

Correção e Mitigação:

A prática de utilizar identificadores baseados em sequências numéricas auto incrementais é desencorajada, sendo recomendada a adoção de identificadores mais complexos como GUIDs. A falha de IDOR que expôs dados de outro usuário ocorreu devido à ausência ou a falha do controle de autorização. O ideal é que as permissões sejam verificadas corretamente em cada solicitação do usuário. Dependendo do *framework* adotado, essa verificação pode ser implementada por meio de filtros, *middlewares* ou camadas de interceptação.

O principal é que exista um mecanismo intermediário de controle, responsável por autorizar ou bloquear a execução de uma ação ou a recuperação de dados, garantindo que apenas usuários devidamente autorizados possam acessar os recursos.

Uma boa prática adicional consiste em garantir que as verificações de autorização sejam realizadas no local correto, embora tais verificações no *frontend* possam ser permitidas para melhorar a experiência do usuário, elas nunca devem ser o fator decisivo para conceder ou negar acesso a um recurso, a lógica do lado do cliente costuma ser fácil de contornar. As verificações de controle de acesso devem ser realizadas no lado do servidor (OWASP, 2025b).

Por fim, à vulnerabilidade que permitiu a utilização da técnica de poluição de parâmetros HTTP para inserir um produto no carrinho de outro usuário foi criada por uma falha de implementação.O servidor do *Juice Shop* utiliza JavaScript, e por padrão, o *parser* dessa linguagem que converte JSON para um objeto JavaScript pega a última ocorrência de uma chave duplicada. Nesse caso específico, o simples uso do recurso nativo da linguagem, já trataria essa inconsistência automaticamente.

8.2 A02 - FALHAS CRIPTOGRÁFICAS (*CRYPTOGRAPHIC FAILURES*)

Nesta seção os próximos desafios representam algumas das causas mais comuns de vulnerabilidades categorizadas pelo OWASP como falhas criptográficas, englobando tanto a exposição de dados sensíveis quanto problemas decorrentes de implementações criptográficas inadequadas.

8.2.1 Desafio - Acessar o arquivo de backup esquecido de um desenvolvedor

O desafio a seguir tem como objetivo explorar uma rota que não é intencionalmente divulgada na aplicação, mas que foi identificada durante a etapa de análise de vulnerabilidades. Essa rota fornece acesso a arquivos internos que deveriam ser restritos, configurando uma falha crítica de segurança.

Falhas de *software*:

De forma geral, essa vulnerabilidade está associada a três categorias do CWE. Primeiramente, a rota expõe informações sensíveis sem a devida proteção (CWE-200). Além disso, não há qualquer mecanismo de controle de acesso que impeça usuários não autorizados de obter esses arquivos (CWE-862), tampouco é necessário estar autenticado para acessá-los (CWE-306).

Por fim, a aplicação realiza apenas uma validação superficial do nome do arquivo solicitado, sem efetuar uma verificação de autorização robusta, permitindo que um atacante acesse arquivos confidenciais.

Exploração da Vulnerabilidade:

A rota que foi explorada nesse desafio não é acessada diretamente no fluxo comum de um usuário. Navegando pelo *Juice Shop*, é possível acessar os termos de uso em uma página que tem o caminho /ftp/legal.md. Ao tentar acessar o diretório anterior, /ftp, são listados todos os arquivos e pastas do diretório /ftp no servidor, conforme mostrado na Figura 23.

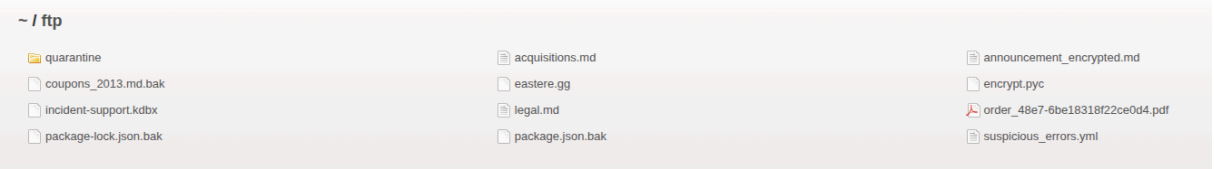

Figura 23 – Página com arquivos do diretório ftp do servidor

Essa exposição do diretório ocorre devido à utilização de um *middleware* no servidor que torna o conteúdo do diretório /ftp acessível e indexável. Após a análise do código fonte realizada pela ferramenta Semgrep, foi identificado o trecho responsável por essa configuração. A ferramenta classificou a falha como um risco de nível médio, conforme ilustrado nas Figuras 24 e 25.

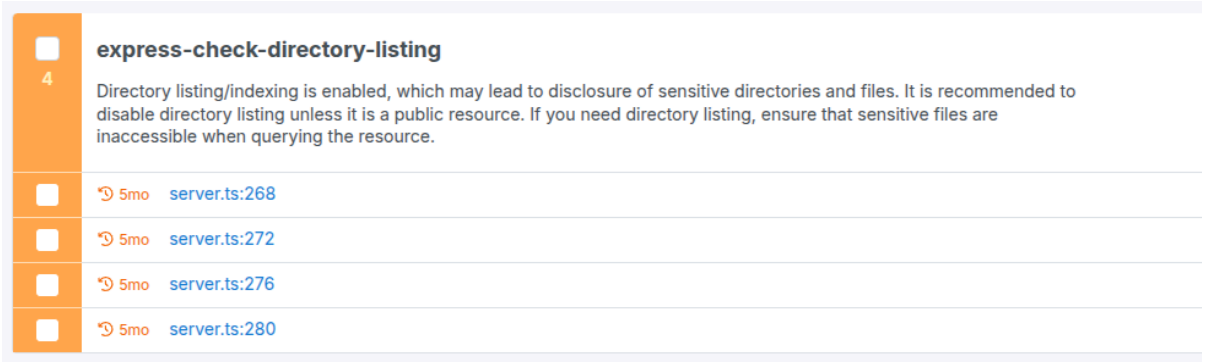

Figura 24 – Alerta de listagem de diretório identificado pelo Semgrep

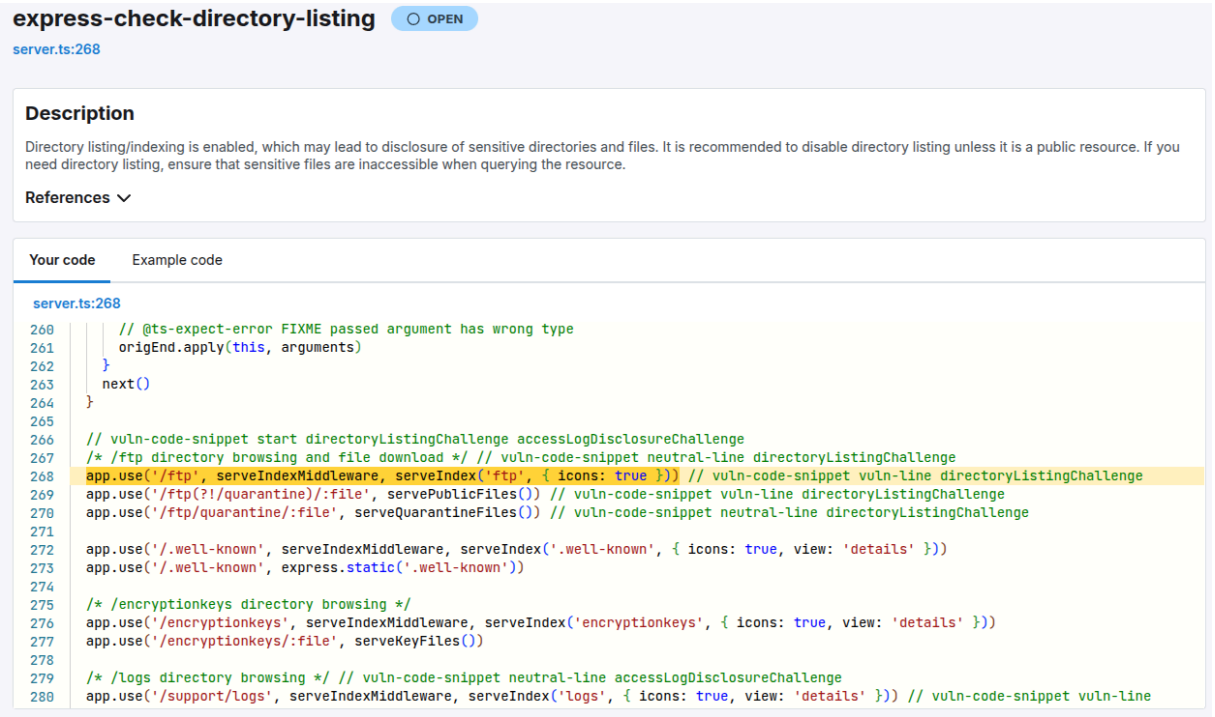

Figura 25 – Análise detalhada do alerta de exposição de diretório

Ao tentar acessar alguns dos arquivos listados nessa rota, o servidor retorna um código HTTP 403 (*Forbidden*) no cabeçalho da resposta, acompanhado de uma mensagem de erro, conforme ilustrado nas Figuras 26 e 27.

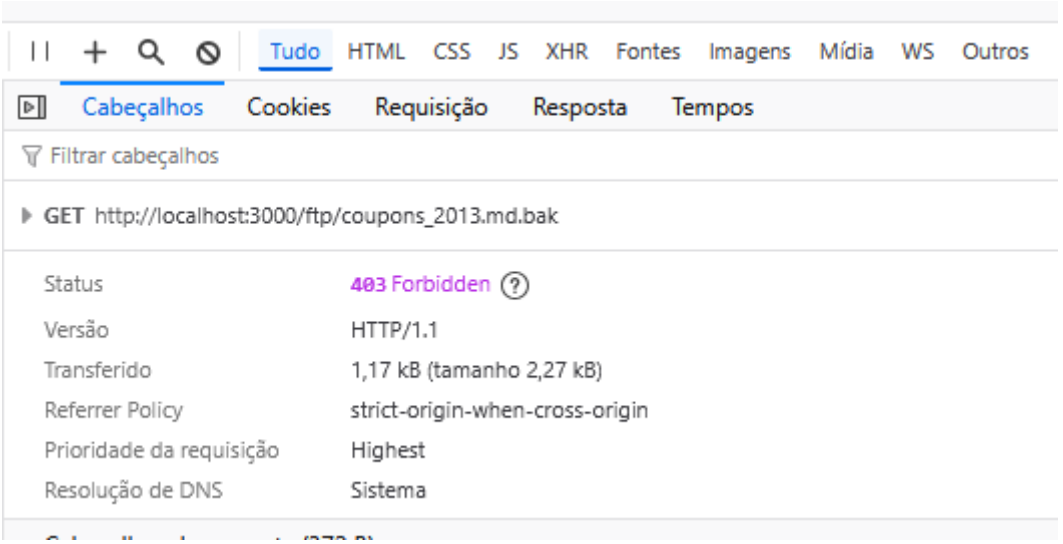

Figura 26 – Cabeçalho de resposta da requisição de recuperação de arquivo

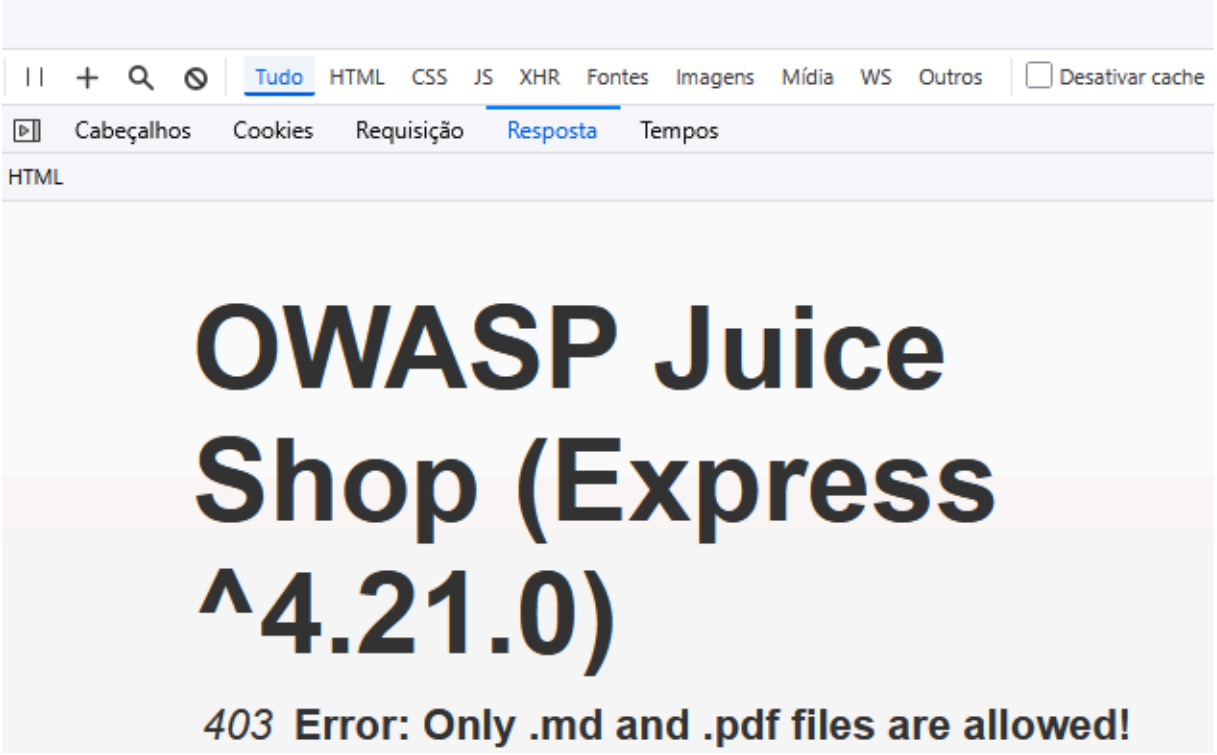

Figura 27 – Mensagem de erro retornado ao tentar recuperar um arquivo

Ao analisar a mensagem de erro retornada pela requisição, foi possível identificar informações sensíveis sobre o servidor, incluindo a versão do *framework* utilizado e a regra de validação aplicada, que indica explicitamente que apenas arquivos com as extensões .md e .pdf são aceitos.

Com base nessas informações, foi utilizada a ferramenta Burp Suite, amplamente empregada em testes de segurança de aplicações *web*, que permite

interceptar, modificar e analisar requisições HTTP, além de auxiliar na identificação de vulnerabilidades. Essa ferramenta facilita a realização de diversos tipos de ataques, entre eles o *fuzzing*, técnica na qual diferentes valores são enviados como entrada em parâmetros de requisições ao servidor, com o objetivo de observar como a aplicação reage a esses dados inesperados ou manipulados. Após algumas tentativas, foi possível notar que o servidor retornava respostas de erro distintas conforme os valores passados para a rota.

A partir disso, aplicou-se uma das técnicas mais comuns utilizadas por atacantes: o *Null Byte Injection*. O *null byte* é tradicionalmente usado para indicar o fim de uma *string*. Assim, a estratégia consistiu em enviar um nome de arquivo que não terminava em uma extensão permitida, mas que incluía um *null byte* seguido de uma extensão válida.

Dessa forma, o sistema de arquivos do servidor interpretaria o *null byte* como o marcador de fim de nome do arquivo, permitindo a recuperação do arquivo solicitado, enquanto a validação superficial leria a parte da extensão que vinha após o *null byte*, consideraria o nome do arquivo como válido e concederia o acesso.

No entanto, essa tentativa resultou apenas em um erro com status HTTP 400 (*Bad Request*). Para contornar essa barreira, foi utilizada outra técnica, o *Double Encoding*. Como o *null byte* é passado codificado pela URL %00, essa técnica o codifica uma segunda vez, transformando-o em %2500.

Essa abordagem é eficaz porque, se o servidor aplicar a decodificação de URL mais de uma vez, o valor %2500 é decodificado para %00, que por sua vez é decodificado para o *null byte* (\x00). Isso permite que a injeção ignore as validações de tipo de arquivo.

Essa técnica, que é comumente usada para contornar filtros de segurança e WAFs(*Web Application Firewalls*), funcionou perfeitamente. Ela não apenas contornou a validação do servidor, mas também permitiu a recuperação de qualquer arquivo dentro do diretório /ftp.

Correção e Mitigação:

A correção desse desafio envolve a resolução de dois problemas principais: a exposição de arquivos do servidor e a falta de controle de acesso.

Uma solução inicial seria disponibilizar os termos de uso em uma página HTML estática, em vez de recuperar esse arquivo via rota pública. No entanto, como o objetivo do desafio é explorar a exposição não intencional de arquivos, a abordagem deve se concentrar em como gerenciar o acesso ao diretório /ftp de forma segura.

O primeiro passo é remover a parte do código que expõe o diretório inteiro ao cliente. A exposição de diretórios é considerada uma má prática de segurança e, neste caso específico do *Juice Shop*, nem sequer é um recurso necessário para o funcionamento da aplicação.

Para controlar o acesso aos arquivos, uma estratégia seria adotar uma lista de bloqueios(*blacklist*) que contém os arquivos que não podem ser acessados, essa abordagem é falha, pois depende da atualização constante da lista. Uma estratégia mais robusta é adotar uma lista de permissões (*whitelist*), que vai conter apenas os arquivos explicitamente autorizados.

Outro aspecto indispensável é a validação do caminho absoluto final do arquivo solicitado. Antes de retornar qualquer conteúdo, o sistema deve assegurar que a resolução do caminho permaneça dentro de um diretório autorizado. Essa prática mitiga tentativas de *Path Traversa*l, impedindo que requisições maliciosas que contenham sequências como ../../ não obtenham acesso a arquivos externos ao diretório autorizado.

Antes de qualquer validação de permissões, é necessário decodificar e normalizar o nome do arquivo solicitado. Essa etapa de decodificação é essencial para mitigar ataques baseados em *Null Byte Injection* e variações de *Path Traversal*.

Por exemplo, em um ataque que utilize o caminho //ftp/dev.zip%00.md, o servidor deve primeiramente decodificar à URL, transformando %00 em um *null byte*. Em seguida, deve-se aplicar uma higienização nesse caminho removendo caracteres ou sequências potencialmente maliciosas, como o *null byte* e/ou tentativas de navegação de diretório. Após esse processo, o caminho resultante seria normalizado para //ftp/dev.zip. Somente então as seguintes validações devem ser aplicadas:

1. Validar se o caminho absoluto final está contido em um diretório autorizado;

2. Confirmar se o arquivo solicitado está presente na lista de permissões;

3. Verificar se a extensão do arquivo corresponde à um formato esperado.

Essa combinação de lista de permissões, validação de caminho absoluto e normalização prévia, representa uma barreira eficaz contra ataques de injeção, acesso não autorizado e exposição acidental de arquivos.

8.2.2 Desafio - Crie um código de cupom que lhe dê um desconto de pelo menos 80%

Nos desafios anteriores, as vulnerabilidades identificadas representavam um potencial risco de perda financeira para a loja. No entanto, neste caso específico, esse desafio representa uma ameaça direta e com potencial de impacto significativo nas receitas, afetando não apenas a segurança da aplicação, mas também seus lucros.

Falhas de *software*:

A falha na lógica dos cupons ocorre porque a aplicação extrai as informações diretamente do código fornecido pelo usuário, em vez de utilizá-lo apenas como uma referência. Além disso, a forma como a funcionalidade foi implementada revela uma

tentativa ineficaz de ocultar o formato dos cupons por meio de uma simples codificação, em vez de empregar algum mecanismo criptográfico adequado (CWE-326).

Exploração da Vulnerabilidade:

A primeira abordagem para exploração consistiu na análise de cupons antigos, já expirados, encontrados em perfis públicos da loja em redes sociais. Essa análise permitiu descobrir alguns cupons válidos, pois foram identificados alguns padrões quando comparados meses de anos anteriores e o valor do desconto. Apesar disso, nenhum dos cupons encontrados atendia ao requisito de conceder pelo menos 80% de desconto.

A identificação desses padrões, somada ao fato de os cupons possuírem sempre um tamanho fixo, indicavam que o sistema não utiliza criptografia, mas sim algum tipo de codificação ou, possivelmente, um algoritmo de *hash* para gerar os cupons.

Com o objetivo de compreender melhor o funcionamento interno dessa funcionalidade, foram buscadas informações adicionais em mensagens de erro e arquivos expostos que pudessem conter dados sensíveis. Esse processo levou ao aproveitamento de uma vulnerabilidade já explorada anteriormente na rota /ftp, nela continha um arquivo package.json. Esse tipo de arquivo contém a lista de dependências da aplicação, incluindo bibliotecas utilizadas e suas versões.

Entre as dependências listadas, destacou-se a biblioteca z85, utilizada para a codificação de dados no padrão ZeroMQ Z85 que é uma variação do ASCII85. Para validar a hipótese de que essa biblioteca estava relacionada à geração dos cupons, foi aplicada à sua função de decodificação a um cupom já expirado. O resultado revelou o formato dos cupons, estruturados como: mês + ano + valor do desconto.

Em posse do formato dos cupons antes de serem codificados, um novo cupom foi gerado para o mês e ano atuais, com um valor de 80% de desconto. Em seguida, a função de codificação da biblioteca foi aplicada gerando um código de cupom válido e funcional. As Figuras 28 e 29 ilustram, respectivamente, a requisição de aplicação do cupom e a resposta da aplicação, confirmando a exploração bem-sucedida da vulnerabilidade.

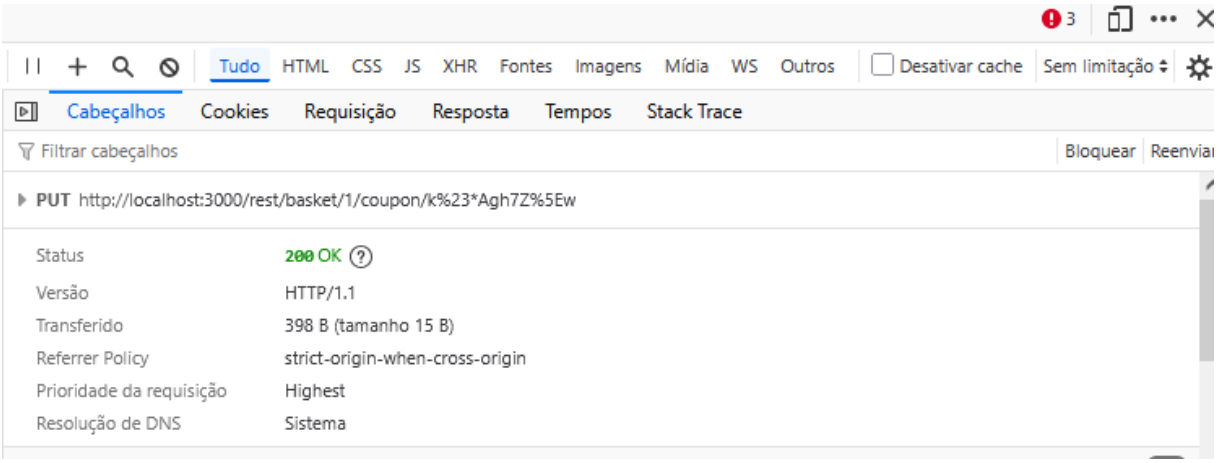

Figura 28 – Requisição de aplicação de cupom com 80% de desconto

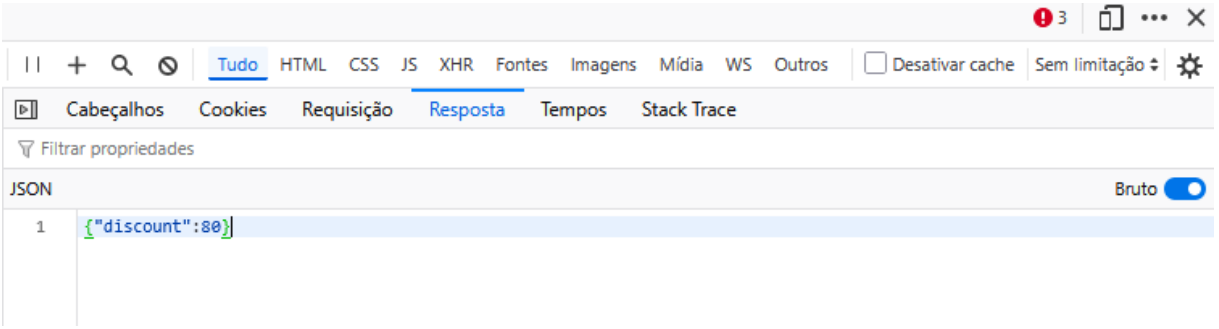

Figura 29 – Resposta da requisição de aplicação do cupom

Correção e Mitigação:

Uma maneira mais segura de utilizar essa funcionalidade é refatorar a lógica, centralizando a criação e a validação dos cupons no servidor, implementando uma tabela de cupons no banco de dados para armazenar informações essenciais, como:

- Identificador;
- Código do cupom;
- Valor do desconto;
- Número máximo de usos;
- Data de validade;
- *Status* de expiração.

Assim, o código do cupom não contém a lógica do desconto, apenas serve de referência para recuperar as informações corretas de forma segura a partir do banco de dados**.** O código pode assumir diferentes formatos, sendo tanto um texto legível e amigável para o usuário quanto uma sequência criptograficamente segura de caracteres pseudo aleatórios, que reduz o risco de adivinhação ou enumeração.

Essa abordagem é flexível e pode ser implementada tanto em bancos de dados relacionais quanto em bancos NoSQL, garantindo que a lógica de validação do cupom esteja sempre sob o controle do servidor.

8.2.3 Desafio - Problemas criptográficos

Este desafio não consiste em explorar diretamente uma falha funcional, mas sim em identificar uma vulnerabilidade grave relacionada à forma como é realizado o armazenamento de senhas.

Falhas de *software*:

É um princípio fundamental de segurança que senhas nunca devem ser armazenadas em texto simples no banco de dados. Na maioria dos casos, as senhas não devem ser criptografadas, mas sim convertidas para um formato de *hash* seguro, pois, diferentemente da criptografia, os *hashes* são funções unidirecionais e não podem ser revertidos para o valor original.

Em um processo de autenticação, por exemplo, a senha fornecida pelo usuário é transformada em um *hash*, e o sistema compara esse valor com o *hash* armazenado. Entretanto, esse procedimento só é seguro se forem utilizados algoritmos de *hash* adequados.

No caso do *Juice Shop*, às senhas processadas utilizam o algoritmo MD5, que é considerado há muitos anos obsoleto e inseguro para os padrões atuais de segurança (CWE-916).

Exploração da Vulnerabilidade:

Se a aplicação não possuir mecanismos contra ataques *offlines* e de força bruta, um atacante pode descobrir as senhas com relativa facilidade, especialmente com o uso de GPUs, que permitem ataques em larga escala e paralelizados.

As principais técnicas utilizadas para quebrar senhas armazenadas com algoritmos de *hash* fracos, como o MD5, incluem:

- Uso de listas de senhas comprometidas de outros vazamentos;
- Ataques de força bruta (*brute force*), tentando todas as combinações possíveis;
- Ataques de dicionário, explorando listas de palavras ou senhas mais comuns (*wordlists*).

Correção e Mitigação:

A abordagem ideal é utilizar algoritmos que exijam alto consumo de recursos computacionais, dificultando tentativas de quebra em caso de vazamento do banco de dados. Dessa forma, mesmo que as senhas sejam expostas, o tempo necessário para decifrá-las em ataques *offline* será significativamente maior. No entanto, é importante encontrar um equilíbrio entre segurança e desempenho, para evitar que o próprio processo de autenticação possa esgotar os recursos do servidor.

Alguns dos algoritmos de *hash* mais recomendados incluem:

- Argon2id: Atualmente, o padrão recomendado pelo OWASP;
- Scrypt: Um algoritmo projetado para ser resistente a ataques com *hardware* especializado;
- PBKDF2: Um algoritmo mais antigo, mas ainda considerado seguro se configurado com um número elevado de iterações.

Além da escolha do algoritmo, existem vários mecanismos adicionais para fortalecer o esquema de *hashes* de senhas, sendo às principais:

- Salt: é uma sequência de caracteres aleatórios e únicos adicionados à senha antes da aplicação do *hash*. O uso desse mecanismo garante que senhas idênticas resultem em *hashes* diferentes, dificultando ataques baseados em tabelas pré-computadas (*rainbow tables*);

- *Pepper***:** trata-se de um segredo adicional armazenado fora do banco de dados. Uma estratégia comum consiste em utilizar esse segredo em conjunto com um HMAC (*Hash-based Message Authentication Code*). O HMAC é um mecanismo criptográfico que combina uma função de *hash* com uma chave secreta para produzir um valor de autenticação criptográfica. Nesse contexto, o *pepper* atua como essa chave secreta, sendo aplicado sobre o *hash* da senha para gerar um novo valor criptográfico. Isso torna o processo mais seguro, pois, mesmo que os hashes das senhas sejam comprometidos, um atacante não conseguirá reproduzir ou validar corretamente os valores sem acesso a essa chave secreta.

O uso desses mecanismos é tão importante quanto a escolha de um algoritmo de *hash* adequado, pois reduzem consideravelmente o impacto de um possível vazamento de credenciais.

8.3 A03 - INJEÇÃO (*INJECTION*)

Nesta seção os desafios representam algumas das causas mais comuns que levam às vulnerabilidades de injeção nas aplicações, em sua maioria sendo problemas de validação de entradas que acarretam em vulnerabilidades de SQL *Injection*, NoSQL e XSS.

8.3.1 Desafio - Registre-se como um usuário com privilégios de administrador

Para concluir este desafio, é necessário explorar uma falha na aplicação que permite que um atacante se registre como um usuário com permissões administrativas.

Falhas de *software*:

À vulnerabilidade deste desafio decorre de uma validação imprópria de entradas, relacionada diretamente com a categoria (CWE-20), onde os dados de

entrada não foram validados suficientemente. No qual os dados fornecidos pelo usuário não são devidamente verificados pelo sistema antes de serem processados.

Exploração da Vulnerabilidade:

Como indicado pelo nome do desafio, a falha está presente no processo de registro de novos usuários. O primeiro passo foi analisar os dados transmitidos na requisição responsável por essa ação. Conforme mostrado na Figura 30, após o registro é retornado um objeto chamado *data*, que contém diversas propriedades, incluindo uma chamada *role*, sugerindo a existência de um controle de acesso baseado em funções.

O RBAC(*Role-Based Access Control*) é um dos principais modelos de controle de acesso utilizados. É um modelo de controle baseado em funções, onde cada usuário é associado a uma função, como membro, gerente e administrador e, cada função define diferentes níveis de permissão dentro da aplicação.

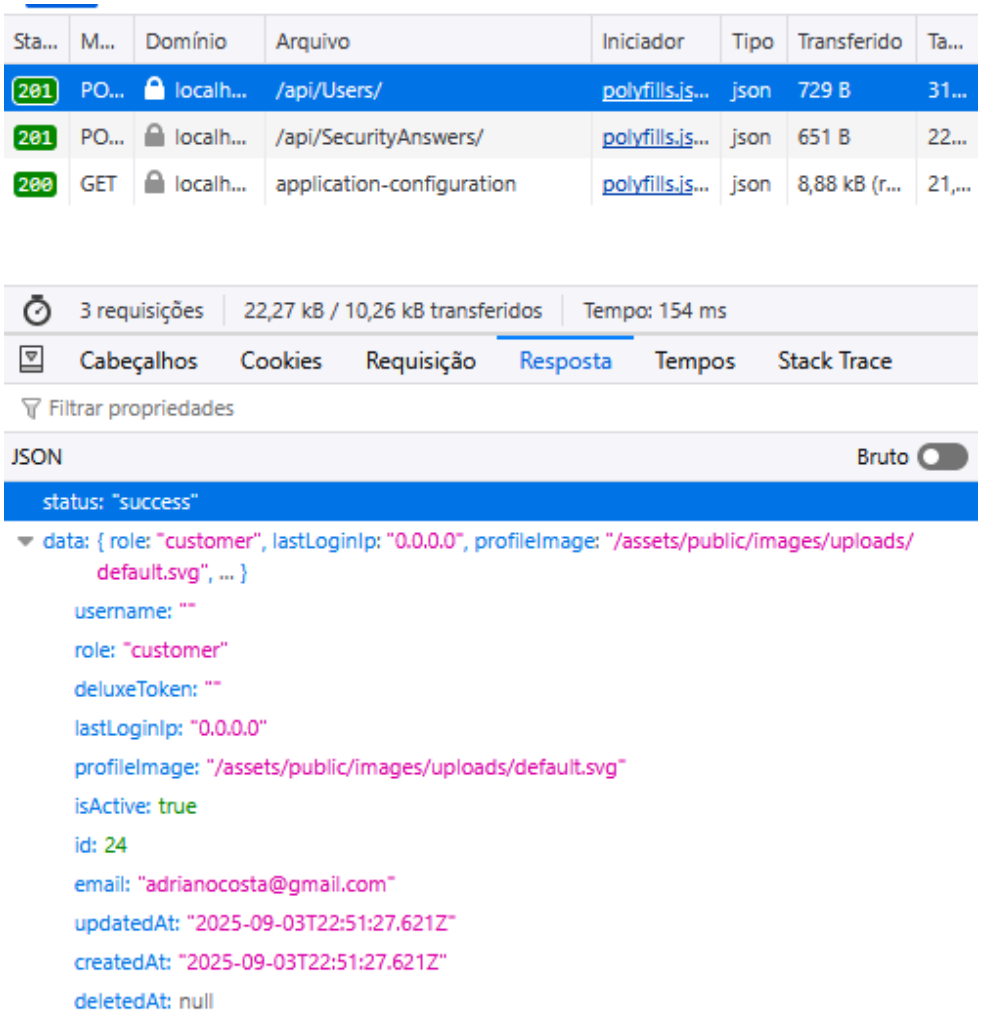

Figura 30 – Requisição de registro de um usuário

Em seguida, o código fonte da aplicação foi inspecionado por meio das ferramentas de desenvolvedor do navegador para tentar obter mais informações do

uso da propriedade *role*. Observou-se que essa propriedade era de fato utilizada em alguns locais do código para comparações de permissões, conforme ilustrado na Figura 31.

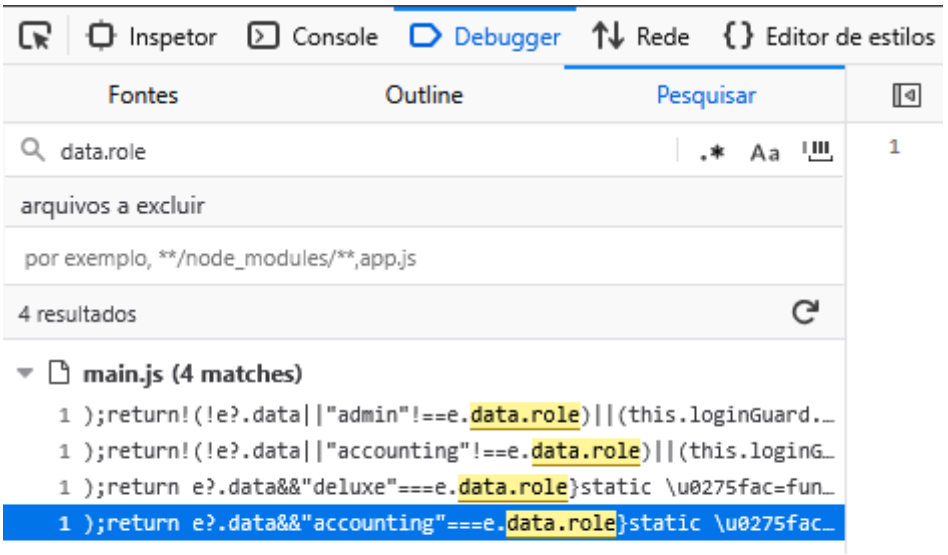

Figura 31 – Código fonte

Sabendo das possíveis funções de usuário, o passo seguinte foi testar se a requisição de registro permitia a inclusão de uma propriedade adicional *role* com o valor "admin". A Figura 32 mostra a resposta dessa requisição, que retornou os dados de um novo usuário criado com privilégios administrativos, sem que a aplicação apresentasse qualquer tipo de erro ou validação adicional.

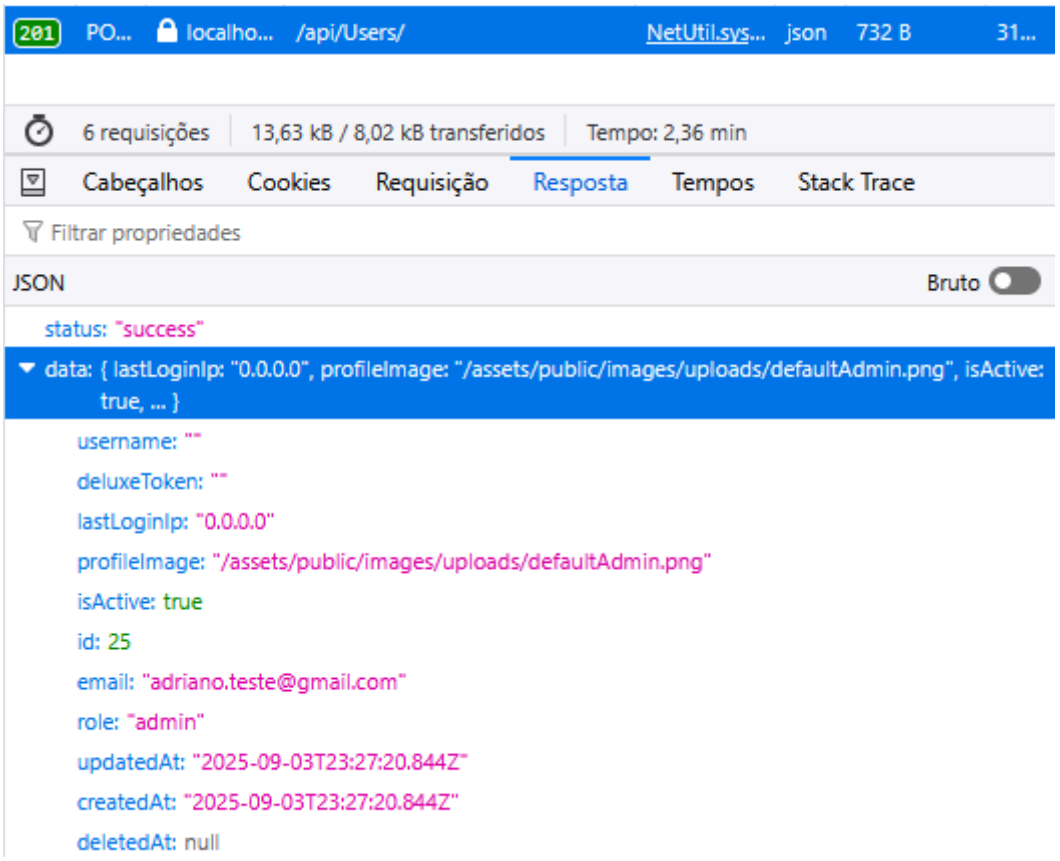

Figura 32 – Requisição de registro com privilégio administrativo

Correção e Mitigação:

A vulnerabilidade explorada é conhecida como *Mass Assignment* (Atribuição em Massa) (OWASP CHEAT SHEET SERIES, 2025b). Ela ocorre quando o servidor converte automaticamente os dados enviados pelo cliente em propriedades internas, geralmente associados à um banco de dados, sem verificar a sensibilidade ou se o usuário deveria ser capaz de manipulá-los.

No caso do *Juice Shop*, essa vulnerabilidade ocorreu porque os parâmetros enviados na requisição estavam diretamente associados às colunas da tabela de usuários no banco de dados, sem qualquer validação ou filtragem. A aplicação confiava apenas em uma segurança baseada na obscuridade, o que permitiu que campos adicionais fossem injetados no formulário de registro e persistidos no banco de dados, comprometendo a integridade e o controle de acesso do sistema.

Uma solução eficaz é utilizar uma lista de permissões, definindo explicitamente quais campos são esperados e podem ser persistidos no banco de dados. A própria documentação da OWASP recomenda essa prática e sugere o uso de bibliotecas como Underscore.js, que disponibiliza funções para extrair apenas propriedades específicas de um objeto. Essa biblioteca é apenas um exemplo voltado para às tecnologias utilizadas neste trabalho, o ponto essencial é garantir que apenas propriedades explicitamente permitidas sejam processadas.

Por fim, é crucial que essas práticas sejam implementadas em conjunto com uma validação rigorosa no servidor, de forma a evitar a dependência exclusiva de controles no lado do cliente. Essa abordagem cria uma camada de defesa robusta, garantindo que o atacante não possa injetar novos dados.

8.3.2 Desafio - Carregar um arquivo maior que 100 KB e carregue um arquivo que não tenha extensão .pdf ou .zip

Esse desafio aborda a importância da validação no lado do servidor, destacando os riscos de depender exclusivamente das verificações realizadas no lado do cliente, que podem ser facilmente contornadas por um atacante.

Falhas de *software*:

Assim como no desafio anterior, no qual foi possível realizar o registro com privilégios administrativos, a falha deste está relacionada à validação imprópria de entradas (CWE-20), entretanto, neste caso, existe uma validação, porém ela é falha e não é aplicada de maneira eficiente.

Exploração da Vulnerabilidade:

A área de suporte da aplicação só é acessível apenas para usuários autenticados e disponibiliza um formulário para envio de uma mensagem acompanhada de um arquivo. Apenas um arquivo com extensão .zip ou .pdf pode ser enviado e o tamanho máximo permitido é 100 KB.

Apesar dessas restrições serem difíceis de burlar diretamente pelo navegador, uma análise mais detalhada revelou que esse processo de submissão é dividido em duas requisições distintas: a primeira responsável pelo *upload* do arquivo e a segunda pelo envio da mensagem ao suporte.

A rota de *upload* do arquivo pode ser identificada pela propriedade *Content-Type: multipart/form-data* no cabeçalho da requisição. Esse é um indicador típico de envio de arquivos em formulários, conforme ilustrado na Figura 33.

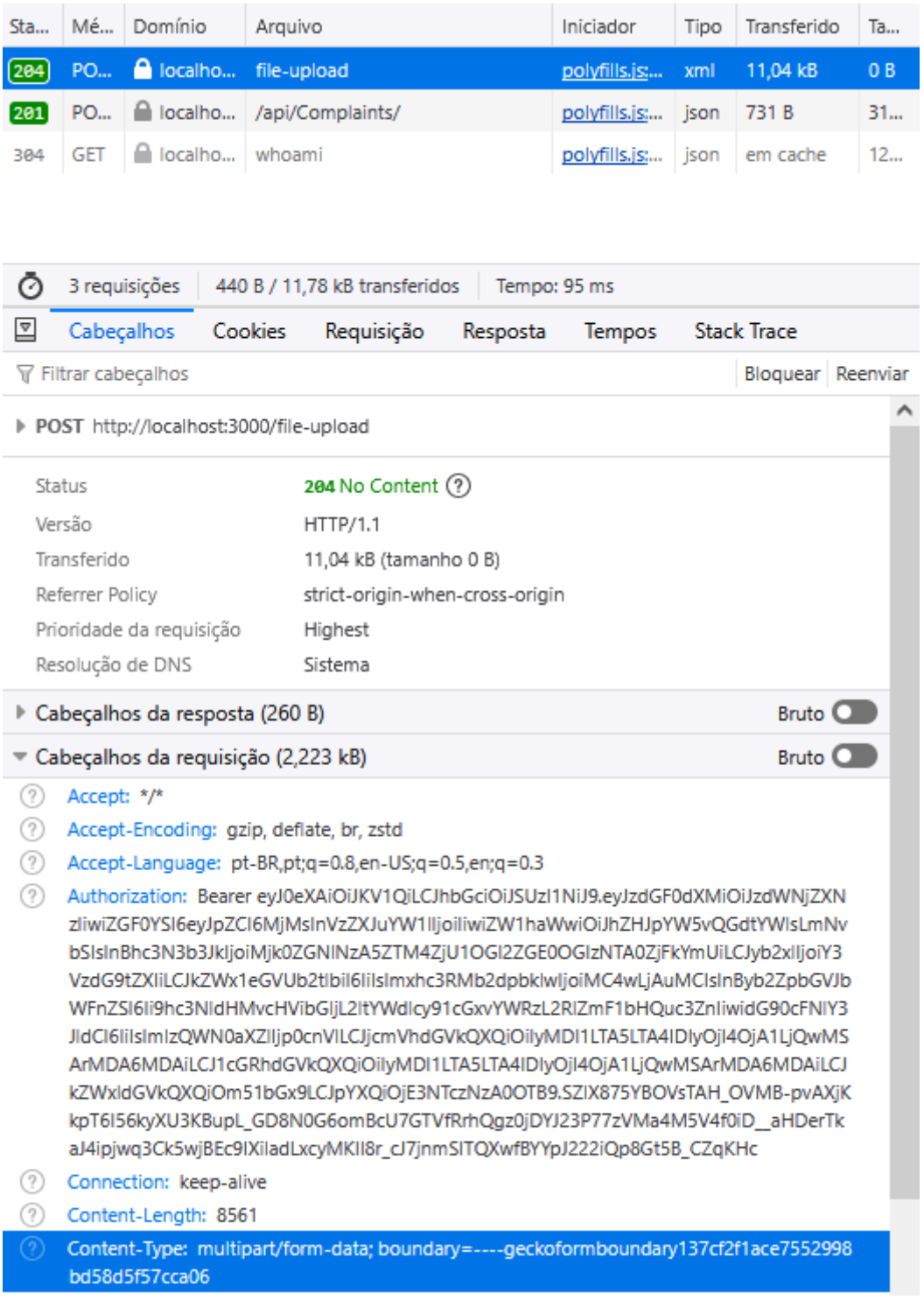

Figura 33 – Cabeçalho da requisição de *upload* de arquivo

Com o uso de ferramentas de teste de API, que dispensam o uso do cliente convencional, tornou-se possível manipular manualmente essas requisições e testar o envio de arquivos com diferentes tamanhos e extensões. Essas ferramentas permitem contornar as validações impostas no cliente e avaliar se o servidor aplica a mesma validação.

Esses testes revelaram algumas falhas:

- A rota de *upload* não exige autenticação: o envio pode ser realizado sem a presença de um *token* válido no cabeçalho;
- Existe apenas uma limitação de tamanho;
- Não há validação do formato do arquivo: qualquer tipo de arquivo pode ser enviado, independentemente da extensão.

Essas falhas demonstram que as restrições exibidas na aplicação não eram aplicadas no servidor, mas apenas no lado do cliente.

Correção e Mitigação:

No contexto da segurança em funcionalidades de *upload* de arquivos, é essencial que todas as validações e controles sejam implementados no lado do servidor, em vez de depender apenas da validação no cliente. Os principais pontos de correção incluem:

- Exigir autenticação e autorização: o envio de arquivos deve estar vinculado a um usuário autenticado e se a funcionalidade exigir, o servidor também deve validar as permissões para garantir que apenas usuários autorizados possam realizar o *upload*;
- Validação robusta do arquivo: à extensão dos arquivos deve ser validada com uma solução semelhante a que foi aplicada no problema da rota /ftp, utilizando uma lista de permissões, definindo quais extensões são aceitas. Da mesma forma, o tipo MIME do arquivo deve ser verificado e comparado com o *Content-Type* declarado no cabeçalho da requisição;
- Restrição de tamanho: os limites de tamanho devem ser implementados tanto no cliente quanto no servidor. Independentemente do *framework* utilizado, o servidor deve possuir parâmetros configurados para restringir o tamanho das requisições, prevenindo ataques de negação de serviço (DoS) que utilizam arquivos grandes;
- Armazenamento seguro: os arquivos recebidos não devem ser armazenados dentro do diretório *root* da aplicação. Além disso, devem ser gravados com nomes gerados automaticamente (como *hashes* ou UUIDs), prevenindo previsibilidade.

Essa é uma abordagem de defesa em profundidade, garantindo que apenas arquivos legítimos sejam processados pela aplicação, mitigando riscos de injeção, execução indevida ou acesso não autorizado.

8.3.3 Desafio - Recuperar uma lista de todas as credenciais do usuário por meio de injeção de SQL

A injeção de SQL (SQL *Injection***)** é uma vulnerabilidade clássica, uma das falhas mais antigas e recorrentes em aplicações *web*, catalogada nas edições anteriores do *OWASP Top* 10. Nesse desafio o objetivo consiste em explorar uma falha para extrair credenciais armazenadas no banco de dados.

Falhas de *software*:

Assim como a maioria dos problemas de injeção, a injeção de SQL ocorre devido à validação e sanitização inadequadas dos dados de entrada. Essa vulnerabilidade está diretamente relacionada às categorias CWE-20 e CWE-89.

O problema surge quando uma entrada do usuário, em vez de ser tratada como dado, é interpretada como parte do comando SQL, permitindo a manipulação da consulta original.

Exploração da Vulnerabilidade:

Nesse desafio, a vulnerabilidade foi encontrada em um parâmetro de uma rota que lista produtos na página inicial, a função principal desse parâmetro era filtrar produtos por nome. A possível vulnerabilidade foi detectada durante a etapa de análise de vulnerabilidades pela ferramenta OWASP ZAP, como mostrado na Figura 34.

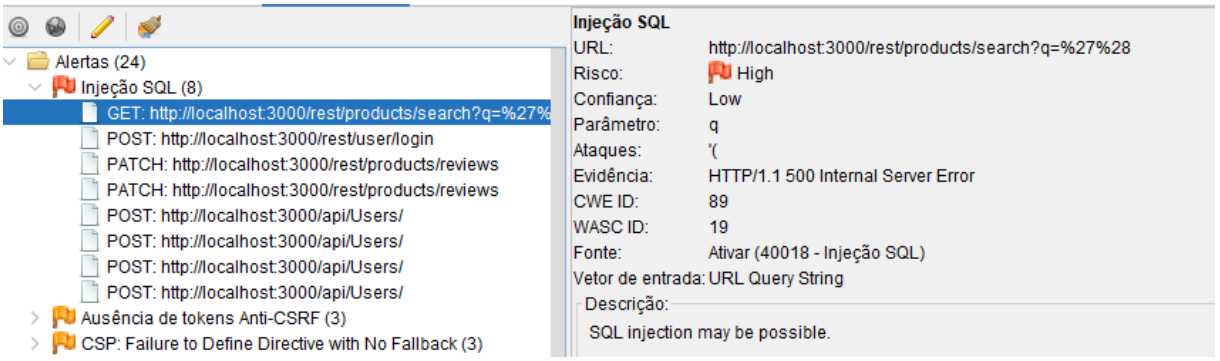

Figura 34 – Alerta de SQL *Injection* da ferramenta OWASP ZAP.

O uso dessa ferramenta acelerou a detecção, já que a análise manual de cada requisição e de suas reações a diferentes entradas levaria muito mais tempo.

Ao utilizar o mesmo valor enviado pela ferramenta no parâmetro da rota, foi possível obter mais informações sobre a consulta e a estrutura do banco de dados. A resposta de erro, exibida na Figura 35, forneceu informações cruciais sobre a estrutura da consulta SQL.

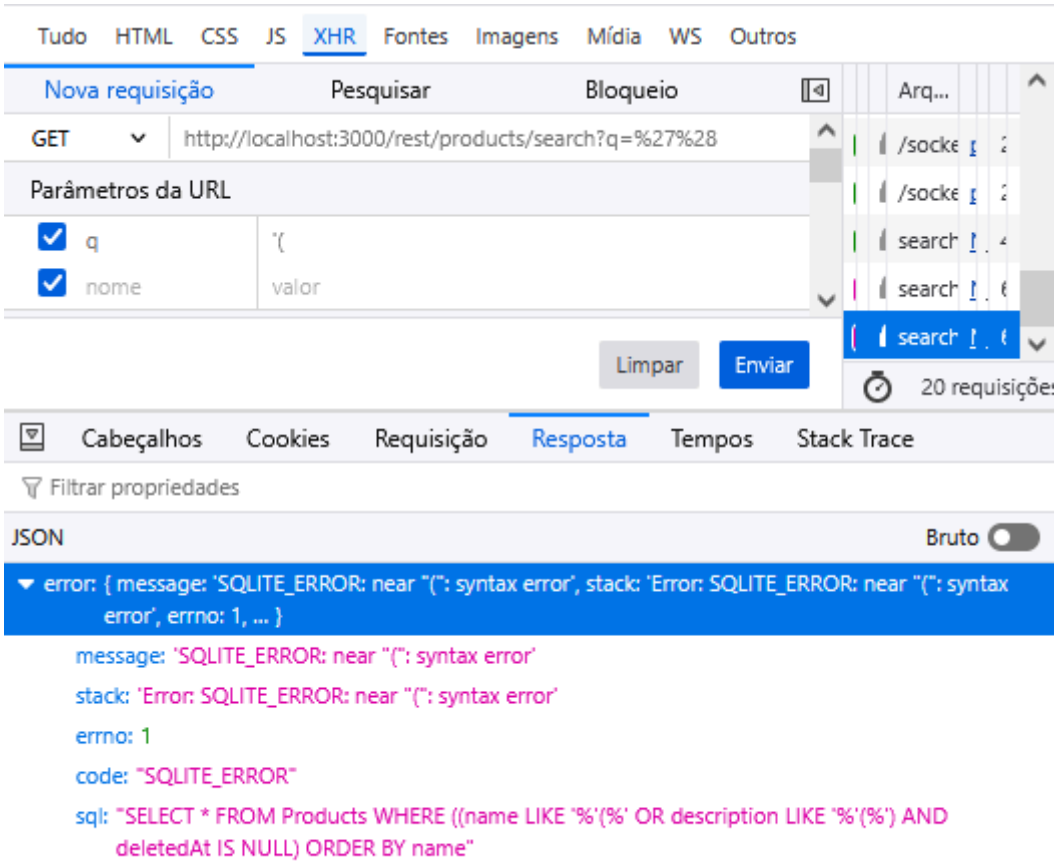

Figura 35 – Resposta da requisição de listagem de produtos.

Com base nas informações do erro, o próximo passo foi construir um *payload* para injetar na rota e extrair dados da tabela de usuários. Para isso utilizou-se à cláusula *UNION*, que permite combinar os resultados de múltiplas consultas em uma única.

Para que a operação *UNION* funcione, as consultas precisam retornar o mesmo número de colunas e ter tipos de dados compatíveis. Na Figura 36 é possível observar os dados retornados em um fluxo normal da aplicação, onde cada produto é um objeto com 9 propriedades, a suposição inicial foi de que a tabela de produtos possuía 9 colunas.

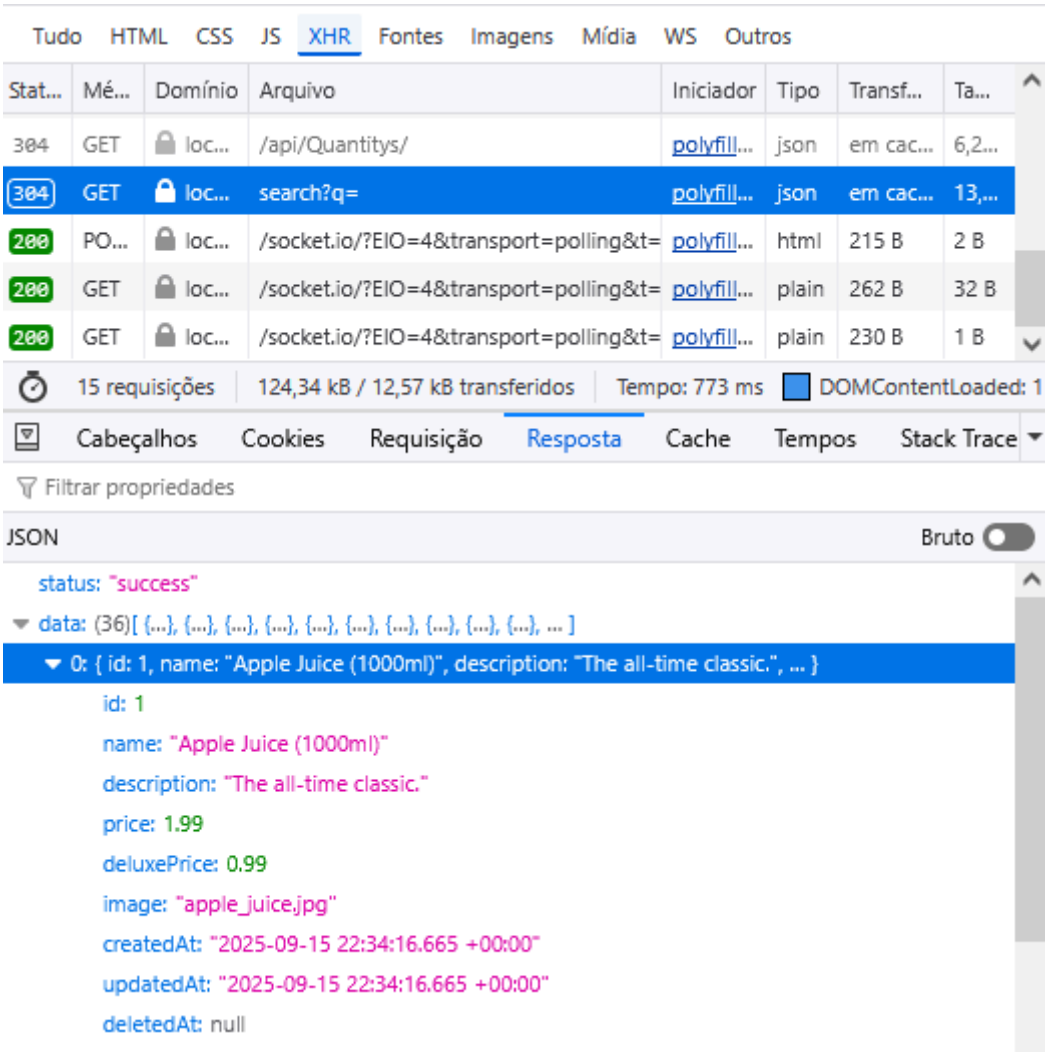

Figura 36 – Requisição de produtos.

Para confirmar o número exato de colunas, foi utilizada uma das técnicas mais simples, comumente utilizada por atacantes usando à clausula *ORDER BY*. Essa técnica funciona injetando um valor que representa um índice de coluna, como 'ORDER BY 1 -- , ' ORDER BY 2 --, ' ORDER BY 3 -- e assim consecutivamente.

Quando um índice maior que o total de colunas é usado, o banco de dados retorna um erro, como mostrado na Figura 37, confirmando a quantidade.

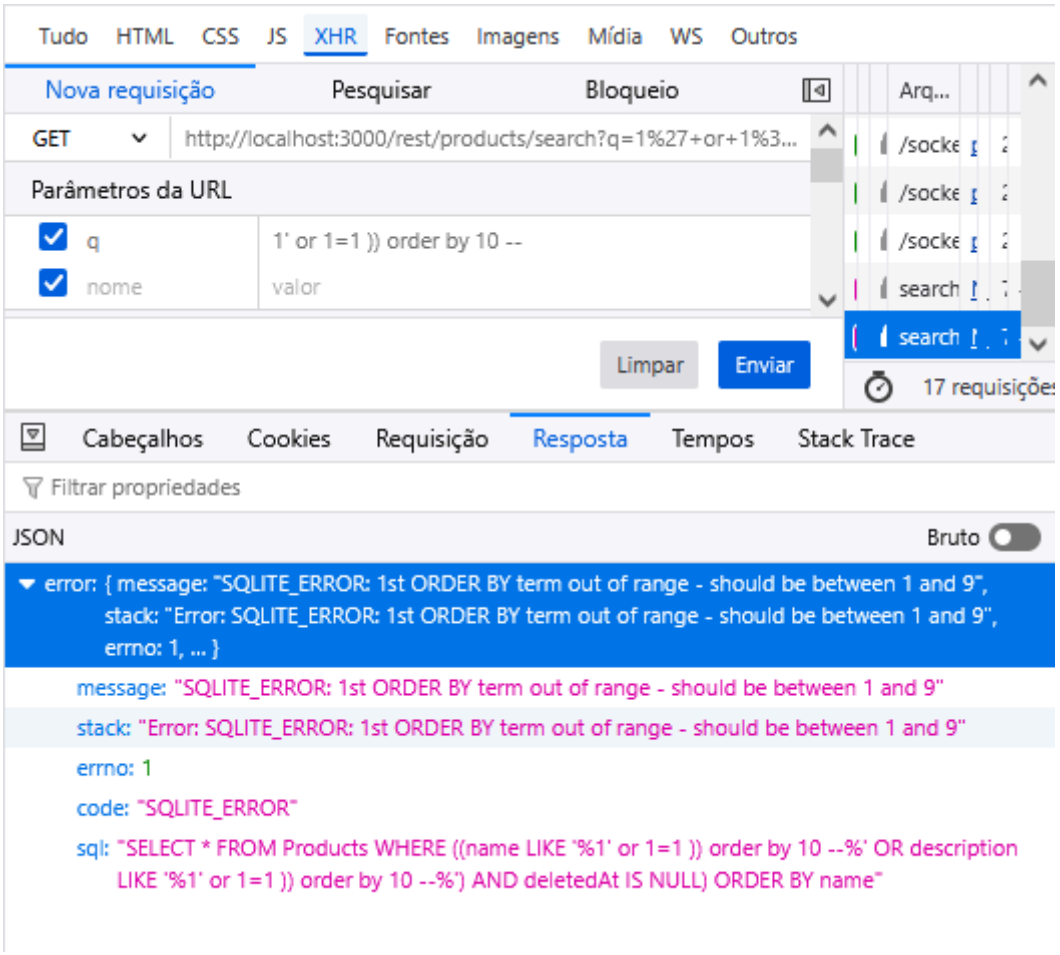

Figura 37 – Erro retornado com um número de colunas inválido

Com o número de colunas confirmado, o *payload* final foi criado para unir a consulta de produtos a tabela de usuários, extraindo campos sensíveis como nome de usuário, senha e *email*.

Payload final:

1' or 1=1)) UNION select username, password, email, null, null, null, null, null, null from USERS --

Ao injetar este *payload*, os registros da tabela de usuários foram retornados pela aplicação e exibidos, conforme ilustrado na Figura 38.

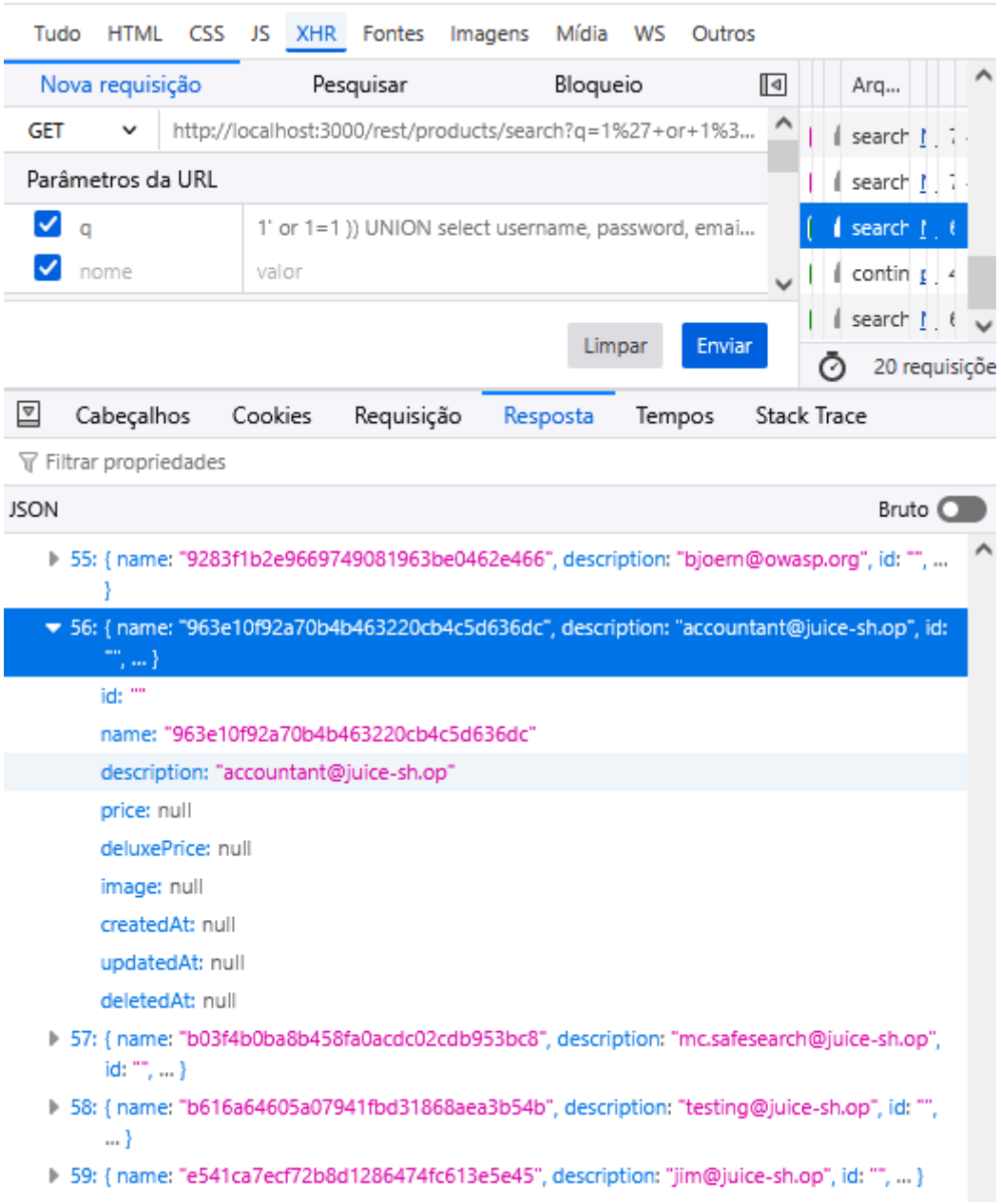

Figura 38 – Dados da tabela de usuários recuperados

Esse resultado demonstra um alto impacto na aplicação, expondo credenciais e consequente risco de acesso não autorizado e o comprometimento de contas.

Correção e Mitigação:

O trecho de código responsável pela vulnerabilidade de injeção de SQL é:

models.sequelize.query(`SELECT * FROM Products WHERE ((name LIKE '%${criteria}%' OR description LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER BY name`)

Essa prática de criar consultas SQL por meio da concatenação de *strings* é totalmente desencorajada. Existem muitas maneiras de evitar isso, e a própria aplicação já utiliza uma delas: o uso de ORMs, porém a maneira como a consulta foi escrita a torna vulnerável.

Em alguns casos específicos, pode ser necessário usar consultas SQL "brutas". Nesse caso, por exemplo, o ORM *Sequelize* usado na aplicação, disponibiliza o método *query*, que permite a execução de comandos SQL diretos. No entanto, mesmo essa abordagem pode ser feita de forma segura, sem permitir que um usuário mal-intencionado injete código malicioso.

Uma forma correta de fazer isso é por meio de consultas parametrizadas. Nessas consultas, os parâmetros não são tratados como parte da instrução SQL, mas sim como dados. Dessa forma, qualquer entrada do usuário é tratada apenas como um valor, impedindo a injeção de código.

Esse é um exemplo de consulta parametrizada com o *Sequelize*:

```
models.sequelize.query('SELECT * FROM Products WHERE ((name LIKE :name OR description LIKE :description) AND deletedAt IS NULL) ORDER BY name', { replacements: { name: criteria, description: criteria }, type: QueryTypes.SELECT})
```

Os ORMs são uma forma de abstrair as interações com o banco de dados e oferecem diversas funcionalidades para operações simples e complexas. Um exemplo usando uma funcionalidade nativa do ORM sem necessidade de escrever um código SQL diretamente:

```
ProductModel.findAll({
    where: { [Op.or]: [
        { name: { [Op.like]: `%${criteria}%` } },
        { description: { [Op.like]: `%${criteria}%` } }
    ]}
})
```

Além disso, outros meios de defesa podem ser aplicados, como procedimentos armazenados e o escape das entradas do usuário. Contudo, para a maioria dos cenários, o uso seguro do ORM em conjunto com a higienização das entradas já cria uma defesa bastante robusta.

8.4 A04 - DESIGN INSEGURO (*INSECURE DESIGN*)

O *Design* Inseguro é uma categoria de vulnerabilidades que não pode ser resolvida apenas com codificação segura. No entanto, é fundamental que um desenvolvedor saiba identificar esses cenários para que eles sejam discutidos e não resultem em vulnerabilidades exploráveis.

Existe uma diferença entre *design* inseguro e uma implementação insegura, um *design* inseguro não pode ser corrigido por uma implementação perfeita, pois, por definição, os controles de segurança necessários nunca foram criados para defender contra ataques específicos (OWASP, 2021a).

8.4.1 Desafio - Redefinir a senha de um usuário através do mecanismo Esqueceu a senha

Este é um dos muitos desafios relacionados à recuperação de senhas explorando à fragilidade do mecanismo de perguntas e respostas em que, em alguns casos, às respostas já estão expostas dentro da própria aplicação de alguma maneira.

Falhas de *software*:

Essa falha pode ser diretamente mapeada para o CWE-213, que se refere à exposição de informações sensíveis, que é exatamente o problema abordado neste desafio.

Exploração da Vulnerabilidade:

Ao criar uma conta na aplicação, é necessário informar uma pergunta de segurança para ser usada na recuperação da senha. No entanto, essa é uma estratégia desencorajada pelo NIST 800-63b (2017), OWASP ASVS e OWASP *Top* 10, pois perguntas e respostas não podem ser consideradas provas de identidade, pois mais de uma pessoa pode saber as respostas (OWASP CHEAT SHEET SERIES, 2025c).

Ao acessar a área de recuperação de senha e fornecer um endereço de e-mail válido, caso esse *e-mail* pertença a uma conta existente o sistema exibe a pergunta de segurança associada, como mostrado na Figura 39.

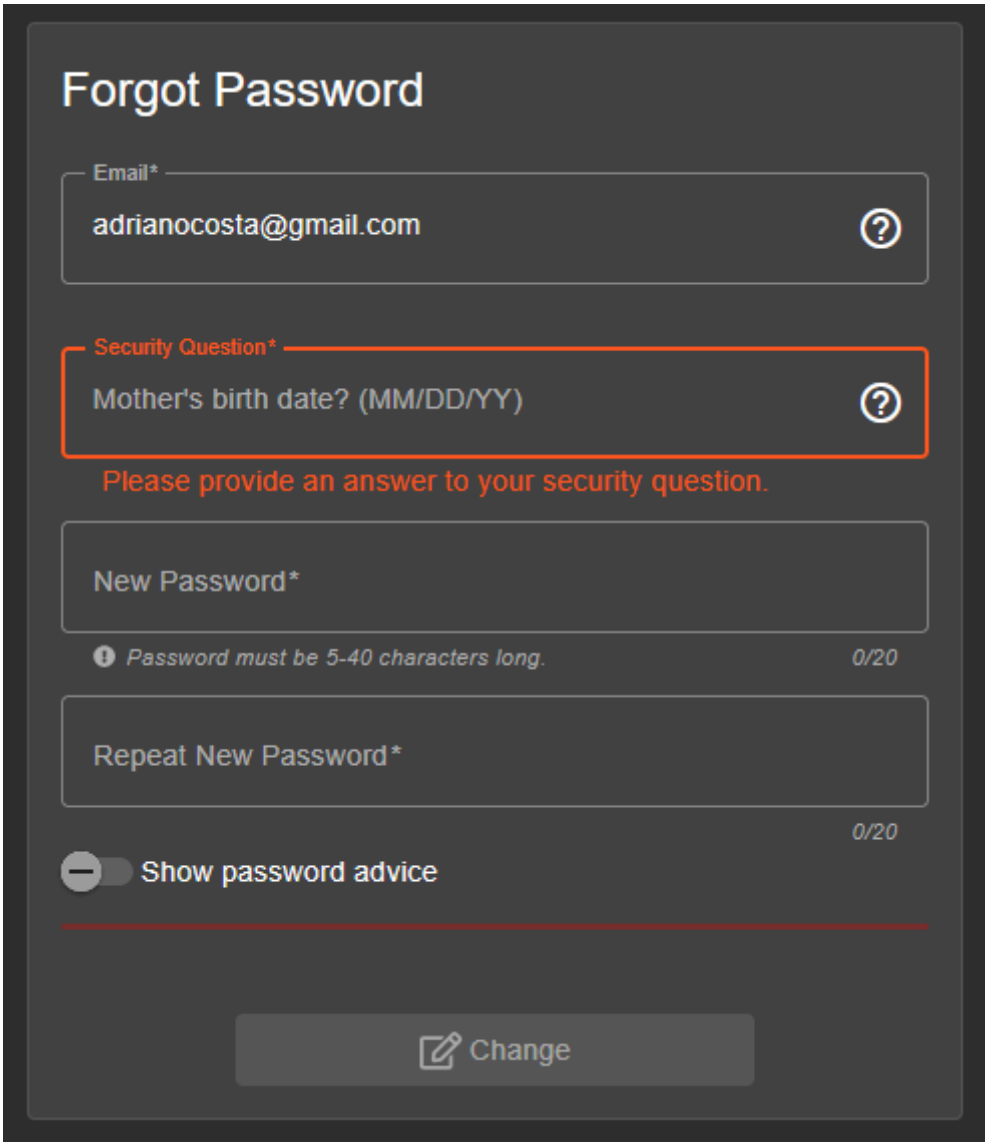

Figura 39 – Pergunta de segurança exposta na área de recuperação de senha

Esse comportamento pode ser usado para a enumeração de contas, pois quando um *e-mail* inválido é inserido, nenhuma pergunta de segurança é exibida. Além disso, a falta de um controle que bloqueie essa ação após um número de tentativas dá a um usuário mal-intencionado a possibilidade de testar inúmeros *e-mails* e descobrir quais contas existem no sistema.

Correção e Mitigação:

Uma das melhores maneiras de identificar um *design* inseguro é através da modelagem de ameaças, ela tem como objetivo identificar e mitigar falhas no projeto:

- É um processo que permite antecipar possíveis ataques e vulnerabilidades de segurança antes mesmo que o código seja escrito;

- A modelagem de ameaças força as equipes a pensarem como um atacante desde o início do projeto, permitindo a incorporação de controles de segurança no *design*, em vez de tentar adicioná-los posteriormente.

Em resumo é uma análise sistemática de como uma parte do sistema pode ser atacado, quais seriam as consequências e quais às contramedidas precisam ser implementadas.

Uma solução seria substituir as perguntas de segurança por mecanismos mais robustos como *tokens* de redefinição de uso único enviados por *e-mail*, verificação multifator ou adotar uma solução padrão de alguma arquitetura segura. Um padrão de arquitetura seguro é uma solução padrão que foi revisada e reforçada contra ameaças de segurança conhecidas.

Além disso, é importante ressaltar que perguntas e respostas de segurança são um dos métodos menos seguros e eficientes para recuperação de senhas. Uma pesquisa realizada em 2015, baseada em dados reais do Google, concluiu que cerca de 40% dos usuários não conseguiam se lembrar das suas respostas. Essa taxa de sucesso é significativamente inferior à de mecanismos alternativos, como códigos de redefinição enviados por SMS, que têm uma taxa de sucesso acima de 80% (BONNEAU et al., 2015).

Apesar de as perguntas de segurança não serem reconhecidas como um método de autenticação válido pelo NIST, elas ainda podem ser usadas em alguns casos, desde que implementadas com segurança e como parte de um processo de verificação multifator.

8.4.2 Desafio - Faça *login* com o contador (inexistente) sem nunca ter registrado esse usuário.

Esse desafio, assim como muitos outros, apresenta uma predisposição à exploração, pois uma exceção não tratada no código expôs dados sensíveis, incluindo a estrutura da consulta SQL e outras informações internas da aplicação.

Falhas de *software*:

Embora a vulnerabilidade principal seja uma falha clássica de injeção SQL, que poderia ter sido evitada com boas práticas de validação e parametrização de consultas, esse desafio também permite expor um problema crítico de configuração e *design*: a Geração de Mensagens de Erro com Informações Sensíveis (CWE-209).

Essa falha é semelhante à identificada no desafio de recuperação de uma lista de credenciais do banco de dados, porém mais grave, pois mensagens de erro sem tratamento podem revelar diretórios completos, regras de negócio, credenciais e outras informações internas, servindo como um mapa para um atacante.

Exploração da Vulnerabilidade:

A funcionalidade de autenticação, assim como a de registro, é suscetível a injeção de SQL. Neste caso, a identificação é ainda mais simples, ao enviar um *payload* comumente utilizado em testes de penetração, a aplicação retorna uma mensagem de erro contendo a estrutura completa da consulta SQL executada, como ilustrado na Figura 40.

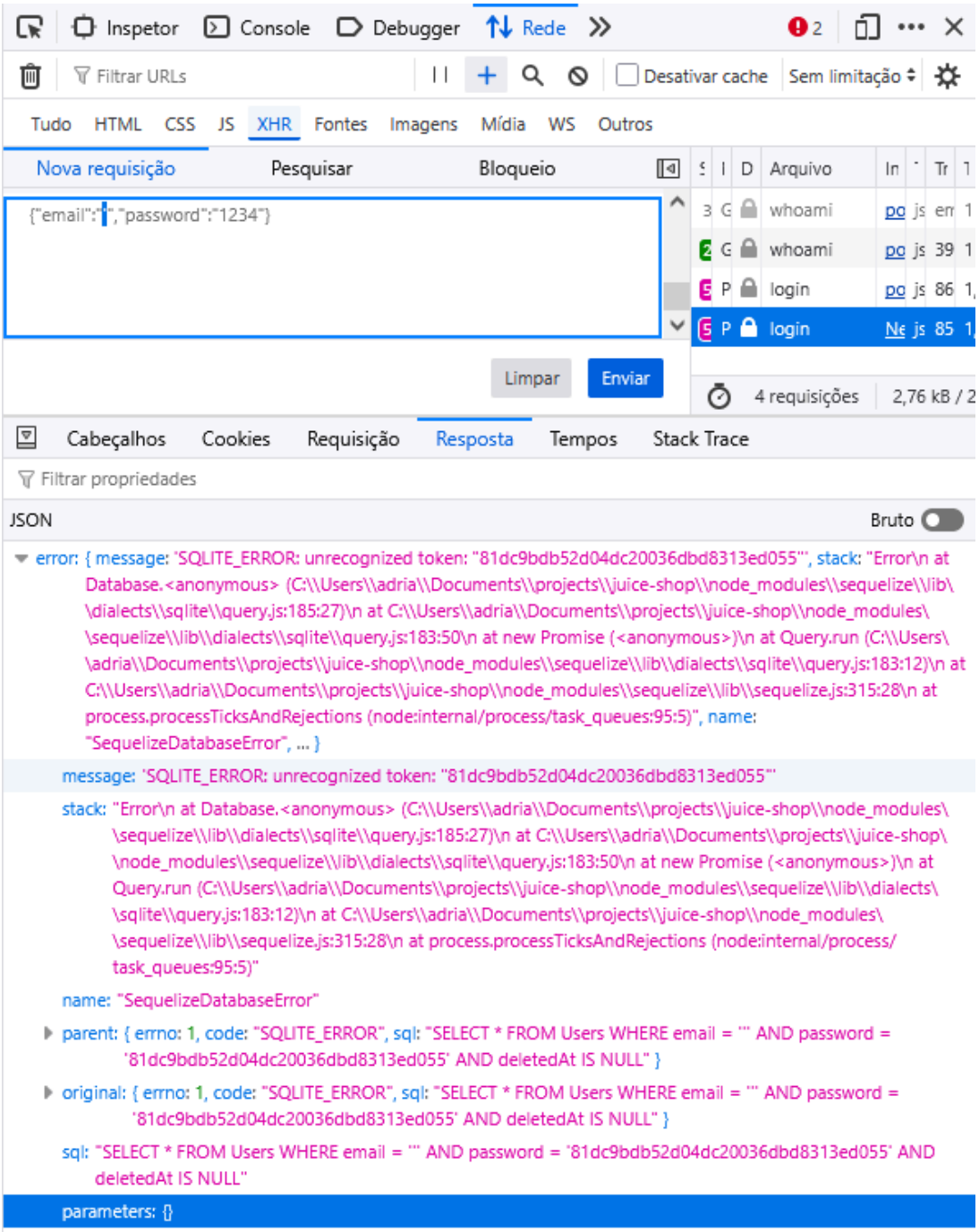

Figura 40 – Mensagem de erro da requisição de autenticação

A partir das informações retornadas na mensagem de erro, é possível identificar dados sensíveis, como a estrutura de diretórios do servidor, o formato da consulta SQL, o tipo de banco de dados utilizado e a própria exceção.

Um exemplo do cenário em que isso pode acontecer pode ser visto na Figura 41, onde qualquer exceção ou erro que ocorrer dentro das chaves do bloco *try* vai ser retornada pela aplicação.

```
try {
/.../
}
catch (Exception e) {
System.out.println(e);
}
```

Figura 41 – Exemplo de *bad code*.

Correção e Mitigação:

A utilização de modelagem de ameaças para funcionalidades de autenticação é, sem dúvida, essencial. Contudo, em cenários onde não há um profissional de segurança para auxiliar na avaliação e no desenho dos controles, uma das medidas mais eficazes e imediatas é implementar um tratamento adequado dos erros gerados pela aplicação.

O OWASP CHEAT SHEET SERIES (2025a), fornece um guia para implementar o tratamento seguro de erros. Embora o material apresente exemplos de código em diferentes tecnologias, o ponto central é o conceito, que pode ser aplicado independentemente da tecnologia utilizada. O objetivo é implementar um manipulador global de erros, configurado como parte da arquitetura da aplicação. Com isso, quando um erro inesperado ocorre, a aplicação retorna ao usuário apenas uma mensagem genérica, sem revelar detalhes internos. O ideal é que essas informações completas sobre o erro sejam registradas exclusivamente no servidor para uma auditoria posterior.

Além disso, é recomendado que a resposta enviada ao cliente contenha uma mensagem genérica, porém informativa o suficiente para indicar o tipo de problema ocorrido.

O retorno de uma mensagem genérica, acompanhado apenas do código HTTP, nem sempre é suficiente para que o cliente compreenda corretamente o tipo de erro ocorrido. Para isso, podemos utilizar um identificador único de erro.

Por exemplo, em um cenário de autenticação no qual o usuário insere credenciais incorretas, a aplicação deve indicar que houve uma falha, mas sem informar quais credenciais estão erradas, pois isto é considerado um antipadrão que pode ser utilizado para enumeração de contas. Nessa situação, uma resposta apropriada poderia incluir:

- Status HTTP 401 (Unauthorized);

- Uma mensagem genérica, como *"Credenciais inválidas!"*;
- Um identificador, chave ou código interno, como *"INVALID_LOGIN_CREDENTIALS"*, conhecido tanto pelo servidor quanto pelo cliente.

Esse tratamento mantém a segurança, evita vazamento de informações sensíveis e ainda permite que o cliente trate os erros de forma consistente e previsível.

8.4.3 Desafio - Altere o href do *link* na descrição do produto O-Saft.

Este desafio apresenta uma falha clássica de controle de acesso quebrado. Entretanto, a exploração realizada permitiu identificar um problema que vai além de um erro pontual em um trecho isolado de código, indicando uma deficiência estrutural na arquitetura de segurança da aplicação

Falhas de *software*:

A vulnerabilidade principal, que permite a um usuário mal-intencionado modificar os dados de um produto, decorre da aplicação inconsistente dos controles de permissão nos *endpoints* da API.

Essa condição pode ser mapeada para a categoria CWE-269, na qual o sistema não restringe corretamente as ações que cada usuário está autorizado a realizar.

Exploração da Vulnerabilidade:

A resolução desse desafio é relativamente simples, basta obter o identificador de um produto, o qual é retornado pela própria aplicação em diversas requisições, como ao visualizar detalhes e avaliações.

De posse do identificador, realizou-se uma requisição para o *endpoint* de atualização do produto, enviando um objeto JSON com a propriedade *description* alterada. Esse *endpoint* já havia sido identificado durante a etapa de reconhecimento realizada com o OWASP ZAP, conforme ilustrado na Figura 42.

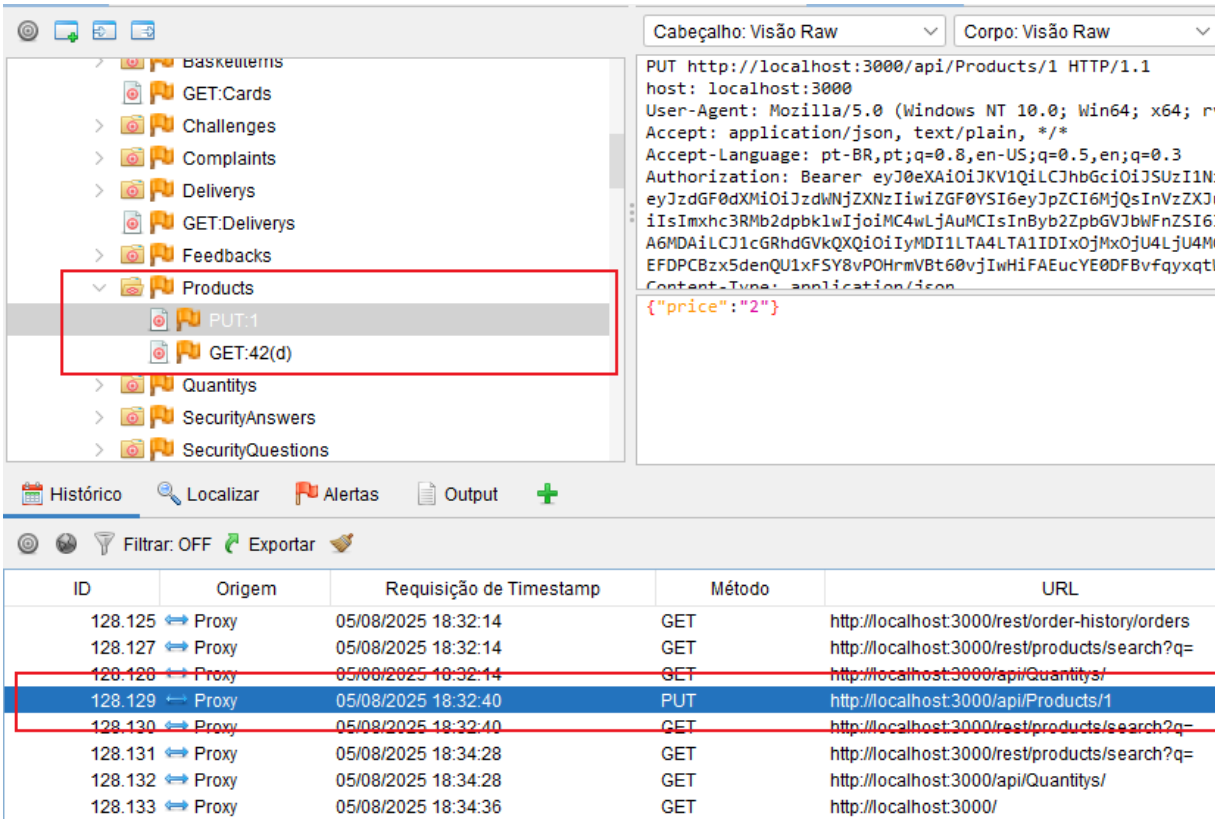

Figura 42 – Histórico de requisições capturadas pelo OWASP ZAP.

O foco principal desta análise não está na exploração em si, mas no motivo pelo qual ela é possível. Ao examinar o código do servidor responsável pela definição das rotas, constatou-se que não há uma estrutura centralizada para o controle de permissões.

O controle de acesso é realizado por meio de *middlewares* aplicados individualmente em cada rota. Essa abordagem depende diretamente do desenvolvedor incluir o *middleware* na declaração da rota, e isso introduz um risco significativo, pois o simples esquecimento resulta em uma proteção inconsistente. Esse comportamento pode ser observado no trecho de código do *Juice Shop* abaixo:

```
app.get('/rest/products/:id/reviews', showProductReviews())

app.put('/rest/products/:id/reviews', createProductReviews())

app.patch('/rest/products/reviews', security.isAuthorized(), updateProductReviews())

app.post('/rest/products/reviews', security.isAuthorized(), likeProductReviews())
```

Nesse exemplo, apenas algumas rotas possuem o *middleware* security.isAuthorized(), enquanto outras executam ações sensíveis sem qualquer verificação de permissão.

De acordo com o OWASP CHEAT SHEET SERIES (2025a), as permissões devem ser validadas corretamente em cada requisição, independentemente de terem sido iniciadas por um script AJAX, pelo servidor ou por qualquer outra fonte. A tecnologia utilizada para realizar essas verificações deve permitir uma configuração global para toda a aplicação, em vez de exigir que cada método ou classe seja aplicado individualmente. Esse cenário reflete exatamente a situação observada no *Juice Shop*.

Correção e Mitigação:

Seguindo as recomendações da OWASP para a centralização da lógica de controle de acesso, o primeiro passo é definir o modelo de autorização adequado para a aplicação.

Além do RBAC, mencionado anteriormente, existem outros dois amplamente utilizados:

- ReBAC(*Relationship-based access control*): É um modelo que usa os relacionamentos entre os objetos para determinar o acesso. Diferentemente do RBAC, onde dois usuários com as mesmas funções têm permissões idênticas, o ReBAC avalia o vínculo. Um exemplo clássico é a edição de comentários, em que apenas o autor do comentário ou um usuário com privilégios administrativos pode modificá-lo. É conceitualmente semelhante às ACLs de sistemas operacionais;

- ABAC(*Attribute-Based Access Control*): É um modelo mais amplo e granular, que considera atributos do usuário (função, departamento), do recurso (*status*, sensibilidade) e do ambiente (horário, localização/IP). Ele engloba outros modelos de autorização, como RBAC e ReBAC (ver Figura 43), uma vez que funções e relacionamentos também podem ser tratados como atributos, sendo ideal para cenários que exigem um controle de permissão flexível.

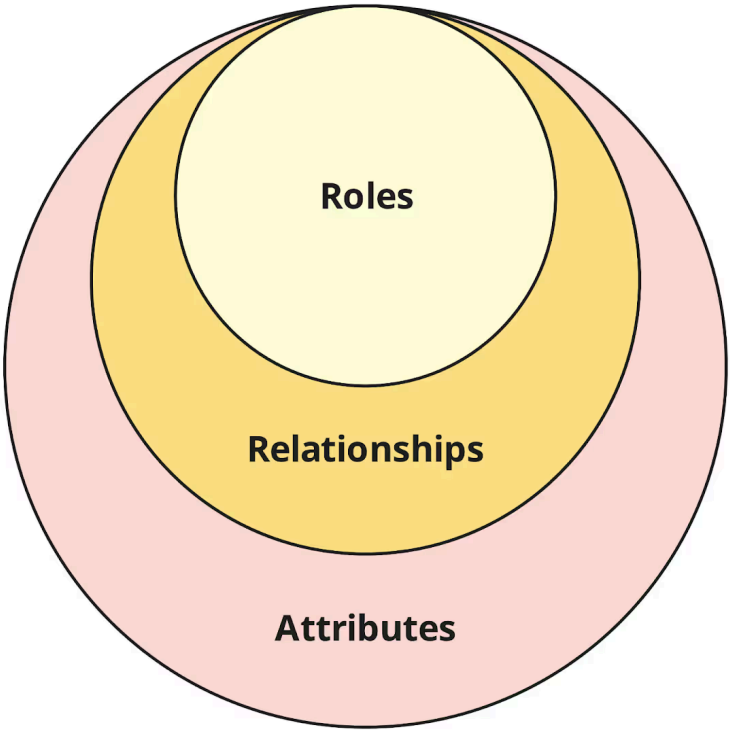

Figura 43 – A hierarquia de funções, relacionamentos e atributos.
Fonte - OSOHQ (2021).

Para definir qual modelo de controle de acesso será utilizado, é necessário avaliar as necessidades de cada aplicação.

O modelo baseado em funções (RBAC) costuma ser a primeira escolha pela simplicidade. No entanto, ele falha em cenários de escalonamento horizontal de privilégios, e à medida que o sistema evolui, torna-se necessário criar novas funções para atender a requisitos específicos, o que compromete sua escalabilidade e pode resultar em permissões excessivas, violando o princípio do menor privilégio, uma vez que é mais fácil conceder permissões adicionais aos usuários do que revogar às já atribuídas.

Nesse contexto, um modelo baseado em relacionamentos (ReBAC) parece ser um meio termo ideal para o *Juice Shop*, especialmente por oferecer proteção contra escalonamento horizontal de privilégios. Considerando que o *Juice Shop* possui funcionalidades como perfis de usuários, comentários e publicações, esses itens devem ser modificados apenas por seus respectivos autores ou por usuários com permissões administrativas.

O ABAC também é uma ótima opção, pois, além de englobar as capacidades do ReBAC, permite a aplicação de regras ainda mais precisas, considerando múltiplos atributos e contextos.

Independentemente do modelo escolhido, o crucial é centralizar à logica de verificação de permissões. Como o *Juice Shop* utiliza tanto banco de dados SQL

quanto NoSQL, ambas as tecnologias precisam ser abrangidas, evitando a necessidade de declarações manuais de autorização em cada *endpoint* da aplicação.

9 CONCLUSÃO

Este trabalho evidenciou que a segurança de aplicações *web* está diretamente relacionada à qualidade das decisões tomadas durante o desenvolvimento do *software*. A exploração dos desafios demonstrou que muitas falhas não decorrem apenas de erros pontuais de implementação, mas de decisões estruturais inadequadas, especialmente no que se refere ao controle de acesso e à validação de dados. Ainda assim, grande parte dessas falhas recorrentes podem ser mitigadas ou evitadas por meio da adoção de padrões de *design* seguros e boas práticas de codificação.

Além da abordagem prática adotada como foco principal, o estudo também apresenta contribuições para a área de segurança de aplicações, destacando o modo de pensar do atacante ao explorar vetores que, muitas vezes, não são previstos durante a fase de desenvolvimento. A manipulação de dados em requisições por meio de ferramentas de interceptação, como o Burp Suite, e a execução de varreduras utilizando diferentes categorias de *scanners*.

Durante a análise, as principais dificuldades encontradas relacionaram-se à exploração de vulnerabilidades que exigiam um conhecimento aprofundado sobre o processamento de dados no lado do servidor. Em certos casos, foi necessário o encadeamento de múltiplas falhas para concluir uma exploração com sucesso. A correlação entre falhas de *software* e vulnerabilidades mostrou-se essencial para a identificação precisa dos vetores de ataque.

Considerando que muitas organizações não possuem profissionais especializados em segurança para auxiliar no ciclo de desenvolvimento, a adoção de práticas de codificação segura torna-se um pilar fundamental para a criação de aplicações seguras e resilientes.

Como possibilidade de trabalhos futuros, sugere-se a análise de vulnerabilidades em outros ambientes, utilizando versões atualizadas do *OWASP Top* 10 e novas falhas catalogadas pelo CWE, com o propósito de ampliar o escopo das análises realizadas. Assim como o uso de ferramentas de análise mais avançadas, como o teste interativo de segurança (IAST), possibilitando a

identificação de novas vulnerabilidades e a expansão dos resultados obtidos nesta pesquisa.

REFERÊNCIAS

BONNEAU, J.; BURSZTEIN, E.; CARON, I.; JACKSON, R.; WILLIAMSON, M. **Secrets, Lies, and Account Recovery: Lessons from the Use of Personal Knowledge Questions at Google.** 2015. Disponivel em: https://research.google/pubs/secrets-lies-and-account-recovery-lessons-from-the-use-of-personal-knowledge-questions-at-google. Acesso em 17 de novembro de 2025.

CENTER FOR INTERNET SECURITY. **CIS Critical Security Controls v8.** 2023. Disponível em: https://learn.cisecurity.org/cis-controls-download-v8. Acesso em: 18 mar. 2026.

CWE. **CWE-352: Cross-Site Request Forgery (CSRF).** 2024. Disponível em: https://cwe.mitre.org/data/definitions/352.html. Acesso em 12 de outubro de 2025.

CWE. **About CWE.** 2024. Disponível em: https://cwe.mitre.org/about/index.html. Acesso em 21 de fevereiro de 2026.

GARCIA, O. C. D. S. **A importância do teste de penetração na avaliação das vulnerabilidades de uma plataforma Web.** Instituto Superior de Tecnologias Avançadas. Lisboa: 2023.

FRANZESE, L. F. **Um Estudo das Vulnerabilidades do OWASP Top 10 na plataforma Moodle.** Universidade Federal de Uberlândia. Uberlândia: 2023.

JESUS, G. C. DE. **Segurança da Informação em Aplicações Web.** Instituto Federal de Educação, Ciência e Tecnologia de Goiás. Uruaçu: 2022.

LI, J. **Vulnerabilities Mapping based on OWASP-SANS: A Survey for Static Application Security Testing (SAST).** Department of Electrical and Electronic Engineering. London, UK: 2020.

NIST. **NIST Special Publication 800-63B.** 2017. Disponível em: https://pages.nist.gov/800-63-3/sp800-63b.html#sec5. Acesso em 31 de dezembro de 2025.

OSOHQ. **Relationship-Based Access Control (ReBAC).** 2021. Disponível em: https://www.osohq.com/academy/relationship-based-access-control-rebac. Acesso em 31 de dezembro de 2025.

OWASP. **A04 Design Inseguro - OWASP Top 10:2021.** 2021a. Disponível em: https://owasp.org/Top10/2021/pt-BR/A04_2021-Insecure_Design/. Acesso em 16 de março de 2026.

OWASP. **Introduction - OWASP Top 10:2021.** 2021b. Disponível em: https://owasp.org/Top10/2021/A00_2021_Introduction/#how-the-data-is-used-for-selecting-categories. Acesso em 16 de março de 2026.

OWASP. **Open Web Application Security Project.** 2025a. Disponível em: https://owasp.org/about/. Acesso em 31 de dezembro de 2025.

OWASP. **OWASP Top Ten Web Application Security Risks.** 2025b. Disponível em: https://owasp.org/www-project-top-ten/. Acesso em 17 de Fevereiro de 2026.

OWASP. **Juice Shop.** 2025c. Disponível em: https://owasp.org/www-project-juice-shop/. Acesso em 12 de outubro de 2025.

OWASP CHEAT SHEET SERIES. **Error Handling Cheat Sheet.** 2025a Disponível em: https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html. Acesso em 04 de dezembro de 2025.

OWASP CHEAT SHEET SERIES. **Mass Assignment Cheat Sheet.** 2025b. Disponível em: https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html/. Acesso em 12 de outubro de 2025.

OWASP CHEAT SHEET SERIES. **Choosing and Using Security Questions.** 2025c. Disponível em: https://cheatsheetseries.owasp.org/cheatsheets/Choosing_and_Using_Security_Questions_Cheat_Sheet.html. Acesso em 16 de março de 2026.

PTES, PENETRATION TESTING EXECUTION STANDARD. **Vulnerability Analysis.** 2014. Disponível em: http://www.pentest-standard.org/index.php/Vulnerability_Analysis. Acesso em 12 de outubro de 2025.

SECURITY LEADER. **Ataques cibernéticos globais crescem 21% no segundo trimestre de 2025, aponta estudo.** 2025. Disponível em: https://securityleaders.com.br/ataques-ciberneticos-globais-crescem-21-no-segundo-trimestre-de-2025-aponta-estudo/. Acesso em 31 de dezembro de 2025.

TORRES, E. V. V. **Análise de vulnerabilidades dos portais web das câmaras municipais alagoanas.** Instituto Federal de Alagoas. Arapiraca: 2023.

WIKIPÉDIA. **Proxy.** 2024. Disponível em: https://pt.wikipedia.org/wiki/Proxy. Acesso em 31 de dezembro de 2025.

ZAP. **Zed Attack Proxy.** 2025. Disponível em: https://www.zaproxy.org/getting-started/. Acesso em 20 de novembro de 2025.