DESENVOLVIMENTO DE UMA API SEGURA: APLICAÇÃO DE PRÁTICAS DE SEGURANÇA PARA A PROTEÇÃO DE DADOS SENSÍVEIS

Rogers Billieri Junior[1], André Faria Ruaro[2]

Resumo: A crescente digitalização de serviços e a integração entre sistemas têm ampliado a exposição de dados sensíveis e o risco de ataques cibernéticos. Nesse cenário, o desenvolvimento seguro de APIs Web tornou-se essencial para garantir a integridade das informações. A aplicação de boas práticas e padrões de segurança é fundamental. Este trabalho tem como objetivo demonstrar, de forma prática, a implementação de medidas de segurança em uma API Web desenvolvida em .NET 8, abordando vulnerabilidades críticas. A pesquisa possui caráter aplicado e experimental. Foi construída uma API com camadas de proteção como autenticação via JWT, hashing de dados em repouso, criptografia de dados em movimento, auditoria e limitação de requisições. Cada implementação foi testada em cenários simulados, comparando o comportamento da aplicação antes e depois das correções. As medidas aplicadas mitigaram com sucesso as vulnerabilidades, resultando em uma arquitetura segura e aderente às boas práticas da OWASP Foundation. A documentação comparativa das etapas contribuiu para o caráter didático do estudo e forneceu uma base prática reutilizável para desenvolvedores. Como limitações, destacam-se a ausência de testes em ambiente real e a não abordagem de todas as categorias OWASP. Para trabalhos futuros, recomenda-se ampliar o escopo e integrar ferramentas automatizadas de monitoramento e detecção de intrusões.

Palavras-chave: Segurança da Informação; API; .NET; OWASP; Cibersegurança.

[1]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), rogersbjunior@gmail.com

[2]Orientador. Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), andre.ruaro@unesc.net

ABSTRACT: The increasing digitalization of services and the integration between systems have amplified the exposure of sensitive data and the risk of cyberattacks. In this scenario, the secure development of Web APIs has become essential to guarantee the integrity of information. The application of best practices and security standards is fundamental. This work aims to demonstrate, in a practical way, the implementation of security measures in a Web API developed in .NET 8, addressing critical vulnerabilities. The research is applied and experimental in nature. An API was built with layers of protection such as authentication via JWT, data hashing at rest, data encryption in transit, auditing, and request limiting. Each implementation was tested in simulated scenarios, comparing the application's behavior before and after the corrections. The applied measures successfully mitigated the vulnerabilities, resulting in a secure architecture that adheres to the best practices set forth by the OWASP Foundation. The comparative documentation of the steps contributed to the didactic nature of the study and provided a reusable practical basis for developers. Limitations include the absence of testing in a real environment and the non-coverage of all OWASP categories. For future work, it is recommended to expand the scope and integrate automated monitoring and intrusion detection tools.

Keywords: API; security; cybersecurity; data protection; OWASP; .NET.

1 INTRODUÇÃO

O crescente uso da internet para a integração de sistemas e fornecimento de serviços online, impulsionado pelas APIs Web, tem exposto uma preocupação crítica: a segurança e o gerenciamento de dados sensíveis. Informações pessoais, credenciais de acesso e registros sensíveis são dados frequentemente manipulados por APIs, o que os torna um alvo atrativo para ataques cibernéticos e devem ser protegidos por diversas camadas de segurança para garantir a integridade e a confidencialidade dos serviços.

O compartilhamento e utilização de informações acontecem de forma instantânea e, muitas vezes, sem o devido cuidado (Schirmer D. L. e Thaines, 2021). Esse grande volume de informações, armazenado e trafegado no ambiente digital, tem se tornado um alvo constante de indivíduos e grupos que buscam burlar a segurança de sistemas informatizados para obter benefícios indevidos (Barbosa et al., 2021).

Os ataques cibernéticos simples ou individuais, que causavam e ainda causam, em geral, males controláveis e prejuízos limitados, estão

dando lugar a operações sofisticadas e bem financiadas, capazes de causar danos econômicos e de reputação significativos a vítimas dos setores público e privado, atingindo nações e grandes empresas mundiais (Vianna; Fernandes, 2015).

Relatórios recentes de segurança confirmam essa tendência. A empresa SonicWall revela aumentos significativos no número de ataques cibernéticos utilizando softwares maliciosos, expondo que serviços abertos à internet são constantemente alvejados (Sonic Wall, 2024). Outro relatório feito pela Check Point Software revela um aumento expressivo no número de tentativas de ataques de softwares maliciosos, projetados para roubar informações sensíveis de sistemas comprometidos, e do uso de softwares maliciosos multifuncionais como botnets (Check Point Software, 2025).

Falhas de segurança críticas como falhas em produtos da Fortinet, Joomla! e ownCloud que foram exploradas por ataques maliciosos em 2024 demonstram que APIs mal protegidas podem levar à exposição de informações sensíveis e acessos não autorizados a sistemas corporativos. Além disso, empresas que dependem fortemente de APIs enfrentam um risco crescente de violações que podem comprometer sua reputação e causar danos financeiros severos. A dependência de APIs desenvolvidas sem práticas de segurança adequadas e, muitas vezes, com tecnologias desatualizadas, as torna vulneráveis a ataques como injeções de SQL, roubo de dados e acessos não autorizados.

Além dos riscos técnicos, o cenário de cibersegurança está alinhado a um rigor regulatório crescente. É evidente um aumento na conscientização das organizações sobre o cenário global de segurança e riscos cibernéticos nos últimos anos. Entretanto, muitas dessas instituições acreditam não estar adequadas para gerenciar esses riscos, o que pode acarretar sérios prejuízos à marca, espionagem corporativa, bem como os custos dessas violações (Accenture, 2025). Junto a isso, companhias constantemente subestimam a magnitude de um ataque cibernético e o consequente roubo de dados que pode vir a ocorrer, a demonstrar, muitas vezes, que são mais reativas do que proativas (Buttrick, 2016).

No contexto do avanço das regulamentações de proteção de dados, a União Europeia e o Brasil adotaram legislações específicas para tratar da proteção de dados: o Regulamento Geral de Proteção de Dados (GDPR, na sigla em inglês) e a Lei Geral de Proteção de Dados (LGPD), respectivamente, que enraízam a responsabilidade legal e ética das empresas em assegurar a privacidade e integridade das informações pessoais

(Feiler; Gazaniga; Vieira, 2024). APIs que lidam com dados sensíveis devem, portanto, não apenas estar blindadas contra ameaças externas, mas também em conformidade com essas leis, sob pena de sanções significativas e danos à reputação das organizações.

Considerando isso, a atuação das empresas no contexto digital trouxe consigo a necessidade de criação de mecanismos de regulação e proteção dos dados pessoais daqueles que utilizam serviços, compras ou realizam qualquer tipo de transação online que envolve o fornecimento de informações pessoais. Toda situação ou ação realizada no ambiente virtual faz parte da realidade de qualquer pessoa; portanto, os direitos garantidos no “mundo offline” devem ser assegurados também no espaço virtual. Em virtude disso, é importante apontar que a lei brasileira não protege somente os dados pessoais nos meios digitais (Pinheiro, 2018).

O objetivo deste trabalho, diante desse cenário, é desenvolver uma API Web segura e avaliar medidas de segurança durante o desenvolvimento, demonstrando a importância da segurança em APIs Web por meio da implementação e análise de medidas de proteção contra vulnerabilidades comuns.

Busca-se identificar e combater possíveis ameaças que possam comprometer a integridade da aplicação, implementando tratativas eficazes para mitigar ataques em nível de aplicação. Além disso, o trabalho visa produzir um código base reutilizável, devidamente documentado, que possa servir como referência para futuros projetos. Ademais, pretende-se disseminar práticas de segurança para APIs, contribuindo para a construção de sistemas mais robustos e alinhados às melhores recomendações do mercado.

Ao documentar e validar a aplicação dessas medidas, este projeto oferece uma contribuição prática e acadêmica, servindo como referência para desenvolvedores e profissionais de segurança que enfrentam os mesmos desafios. Através da implementação e validação dessas práticas, o trabalho alinha-se às demandas do mercado, que exige soluções seguras e confiáveis, e contribui para o avanço de estratégias que mitigam os riscos associados à manipulação de dados sensíveis.

Para tanto, o projeto seguiu uma metodologia exploratória e aplicada, fundamentando os principais riscos de segurança em APIs Web e as estratégias de mitigação com base em referências como a fundação Open Web Application Security Project (OWASP). A abordagem aplicada reside no desenvolvimento prático de uma API em um ambiente de teste que ma-

nipule dados sensíveis, na implementação das medidas de segurança escolhidas e na validação da eficácia dessas implementações por meio de testes práticos, demonstrando a mitigação das vulnerabilidades.

A estrutura deste trabalho está dividida em: Capítulo 1, a introdução, que contextualiza o problema, define os objetivos e justifica a pesquisa; Capítulo 2, que apresenta os trabalhos correlatos para embasar a pesquisa; Capítulo 3, que descreve os materiais e métodos, apresentando as tecnologias e a metodologia de desenvolvimento; Capítulo 4, que apresenta a discussão e os resultados obtidos; e o Capítulo 5, que apresenta a conclusão.

2 TRABALHOS CORRELATOS

Para o desenvolvimento deste projeto, foram realizadas pesquisas sobre trabalhos semelhantes para adquirir conhecimento acerca das abordagens já utilizadas na área e destacar as contribuições únicas desta pesquisa.

O estudo ”A Proteção de Dados e Segurança da Informação na Pandemia Covid-19: Contexto Nacional” (Barbosa et al., 2021) analisa o impacto da pandemia da COVID-19 no aumento dos ataques cibernéticos no Brasil. Embora o trabalho não se concentre no desenvolvimento de soluções técnicas, ele fornece um valioso contexto sobre a suscetibilidade de usuários e organizações em cenários de incerteza social. A principal contribuição deste trabalho é a demonstração de como atores maliciosos se aproveitam de situações de pânico para explorar os pontos fracos de segurança. As estatísticas e os dados apresentados neste estudo reforçam a necessidade de medidas de segurança robustas, servindo como base para a justificativa deste projeto e ajudando a entender as vulnerabilidades a serem tratadas.

A tese de mestrado ”Concepção e Desenvolvimento de uma API Rest com Incorporação de Mecanismos de Segurança Aplicacional” (Gonçalves, 2022) apresenta um trabalho semelhante ao propor o desenvolvimento de uma API REST com implementações de segurança, como autenticação, autorização e criptografia. O estudo aprofunda-se nas decisões técnicas de desenvolvimento e foca na documentação do processo. No entanto, o trabalho não demonstra de forma prática os efeitos das implementações de segurança em cenários de ataques simulados. Diferente dessa abordagem, o presente projeto visa não apenas implementar as medidas, mas também documentar o comportamento da API antes e depois de cada

implementação, validando sua eficácia por meio de testes de penetração.

O trabalho ”Desenvolvimento de API Criptográfica Para Comunicação Segura Entre Aplicações Web: Estudo de Caso Aplicado em Web Service Rest” (AOYAMA, 2016) concentra-se na criação de uma API criptográfica destinada a apoiar desenvolvedores com pouca experiência em segurança da informação, atuando como um middleware capaz de selecionar automaticamente a cifra mais adequada para proteger dados em trânsito. Além de apresentar uma solução prática para criptografia, o estudo de Aoyama também serviu como referência e inspiração direta para as implementações criptográficas adotadas neste projeto. Ainda que a criptografia seja um elemento essencial em ambos os trabalhos, o escopo deste projeto é mais abrangente, indo além da proteção de dados em trânsito para contemplar uma arquitetura de segurança mais completa.

3 MATERIAIS E MÉTODOS

Para orientar o desenvolvimento e a avaliação das medidas de segurança deste projeto, adotou-se uma metodologia que combina pesquisa teórica e experimentação prática. Foram analisados padrões e recomendações amplamente utilizados na área, especialmente os da OWASP. Em seguida, essas diretrizes foram aplicadas na construção e teste da API, possibilitando observar e validar na prática a eficácia das técnicas de proteção implementadas.

3.1 OWASP TOP 10

As vulnerabilidades em aplicações web e APIs são sistematicamente categorizadas pela OWASP (Open Web Application Security Project), uma referência global em segurança. A lista ”OWASP Top 10” de 2025, que aponta os riscos de segurança mais críticos, serve de base para as medidas de proteção abordadas neste trabalho. Segundo a OWASP, muitas das falhas mais comuns ocorrem por falta de práticas seguras durante o desenvolvimento.

Entre as ameaças a serem tratadas neste projeto, o Controle de Acesso Quebrado (Broken Access Control) ocorre quando um usuário consegue acessar recursos ou funcionalidades fora de suas permissões, levando à divulgação de dados sensíveis, modificação de informações ou elevação de privilégio. Uma das causas comuns é a falta de validação adequada de permissões em cada requisição de API. Para mitigar esse risco, é essencial implementar mecanismos de autorização robustos e seguir o

princípio do privilégio mínimo.

Outra ameaça são as Falhas Criptográficas (Cryptographic Failures). A falta de criptografia adequada expõe dados sensíveis, como senhas, informações pessoais e registros financeiros, por exemplo o uso de algoritmos fracos ou a ausência de proteção em trânsito e em repouso. A prevenção inclui a aplicação de protocolos de comunicação seguros, como TLS (Transport Layer Security), e a utilização de algoritmos de criptografia fortes e atualizados para proteger dados em repouso, como o Argon2id.

Adicionalmente, a Injeção de Código (Injection) é um ataque que ocorre quando dados não confiáveis são enviados para um interpretador como parte de um comando ou consulta. Um dos exemplos mais conhecidos é a injeção de SQL, na qual um atacante insere um código SQL malicioso em um campo de entrada para manipular o banco de dados. A principal forma de prevenção é a utilização de consultas parametrizadas, que isolam o código dos dados fornecidos pelo usuário.

A categoria de Falhas de Identificação e Autenticação (Identification and Authentication Failures) engloba vulnerabilidades relacionadas a processos de login, senhas e gerenciamento de sessão. As falhas podem permitir ataques de força bruta ou enumeração de contas. Medidas preventivas incluem a implementação de limites de tentativas de login, a utilização de senhas fortes, a autenticação multifator (MFA) e a invalidação correta de sessões após o logout.

Por fim, a categoria Falhas de Monitoramento e Auditoria de Segurança (Security Logging and Monitoring Failures) ressalta que a ausência de um monitoramento eficaz impede a detecção de ataques em tempo hábil. Eventos críticos, como tentativas de login, alterações de dados sensíveis de registros importantes, falhas de acesso e transações de alto valor, devem ser registrados e auditados. A implementação de um sistema de log centralizado é fundamental para uma resposta rápida a incidentes de segurança.

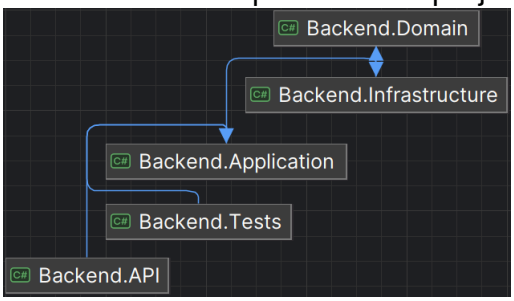
Além dos listados pela OWASP, outros ataques, como a Negação de Serviço (DoS), podem comprometer a disponibilidade de uma API. A implementação de um mecanismo de rate-limiting é uma forma eficaz de mitigar esse risco, limitando o número de requisições que uma fonte pode fazer em um determinado período de tempo.

3.2 TECNOLOGIAS

O desenvolvimento deste trabalho foi realizado com o uso de tecnologias e ferramentas modernas para garantir um ambiente de desenvolvimento eficiente e alinhado às práticas de mercado. A escolha do framework .NET 8, em conjunto com a linguagem C#, oferece uma base robusta e de alto desempenho para a construção da API.

O padrão de design Domain-Driven Design (DDD) promove uma organização mais sólida e coerente do código ao enfatizar a separação clara das responsabilidades dentro da API, de acordo com a Figura 1. Embora o DDD não imponha rigidamente uma arquitetura em camadas, ele incentiva a criação de uma estrutura bem definida, cada uma com funções específicas e regras de dependência próprias. Essa abordagem ajuda a manter o código mais organizado, facilitar a manutenção e reduzir o acoplamento entre módulos. Assim, a hierarquia natural de dependências entre as camadas garante que regras de negócio permaneçam isoladas e protegidas de detalhes de implementação, resultando em um sistema mais robusto, sustentável e alinhado ao domínio de negócio.

Figura 1 - Estrutura da hierarquia dos sub-projetos do código



Fonte: Do autor.

A camada de apresentação, representada pelo sub-projeto Backend.API, guarda as controllers, que contém os métodos da API que os consumidores poderão utilizar, conforme exemplificado na Figura 2.

Figura 2 - Método de uma controller

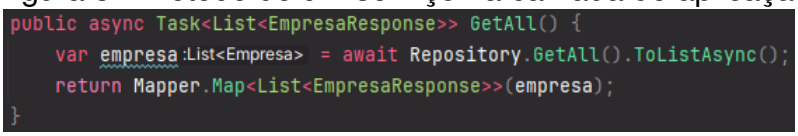
```
[HttpGet]
Rogers Billieri *
public async Task<IActionResult> GetAll() {
    try { return Ok(await _empresaService.GetAll()); }
    catch (Exception e) { return StatusCode((int)HttpStatusCode.BadRequest, e.Message); }
}
```

Fonte: Do autor.

A camada de aplicação (Backend.Application) concentra as regras de negócio e a maior parte da lógica do projeto. Esta utiliza a camada de infraestrutura para obter os registros do banco de dados e efetuar as

operações necessárias. A Figura 3 contém um exemplo da camada de aplicação.

Figura 3 - Método de um serviço na camada de aplicação



Fonte: Do autor.

A camada de infraestrutura (Backend.Infrastructure) define os repositórios das entidades e a conexão com o banco de dados. Ela é dependente apenas da camada de domínio (Backend.Domain), que contém a definição das entidades, enumeradores, interfaces dos repositórios e serviços da camada de aplicação e as classes menores com métodos compartilhados utilizados pelos diversos serviços. Esta é a camada base para o funcionamento da aplicação.

Para o gerenciamento de dados, o banco de dados relacional PostgreSQL foi escolhido devido à sua escalabilidade e confiabilidade. A persistência de dados foi gerenciada pelo Entity Framework Core (EF Core), um Mapeador Objeto-Relacional (ORM) que simplifica a comunicação com o banco de dados, abstraindo a lógica SQL.

Os recursos necessários para a programação incluíram um computador com o sistema operacional Windows e a IDE JetBrains Rider 2024.5. O banco de dados foi gerenciado por meio do sistema DBeaver Community 24.2.2.

3.3 TESTES

Para a validação e demonstração das vulnerabilidades, a ferramenta Postman foi utilizada para realizar as requisições à API e simular ataques.

O desenvolvimento do projeto seguiu uma sequência de etapas bem definidas. Inicialmente, foi feito um levantamento teórico e referencial, seguido pela construção de um ambiente de testes completo, incluindo a API e a modelagem do banco de dados com as entidades necessárias. Com o ambiente-base estabelecido, foram implementadas as medidas de proteção selecionadas, como autenticação robusta, mecanismos de autorização, criptografia de dados e sistemas de log e auditoria.

Para demonstrar a eficácia das implementações, foi adotada a técnica de teste de penetração (*pen-test*), e por fim cenários de ataques

simulados foram executados para documentar o comportamento da API antes e depois das medidas de segurança.

4 RESULTADOS E DISCUSSÃO

Esta seção apresenta a implementação das medidas de segurança na API desenvolvida em .NET 8, seguindo a ordem de mitigação dos riscos de segurança identificados com base na lista da OWASP Foundation. Para cada tópico, é demonstrado o cenário de fragilidade inicial e o resultado após a aplicação das mitigações, com evidências do código implementado e dos testes de penetração realizados.

4.1 DEFINIÇÃO DA API BASE E MAPEAMENTO DE ENTIDADES

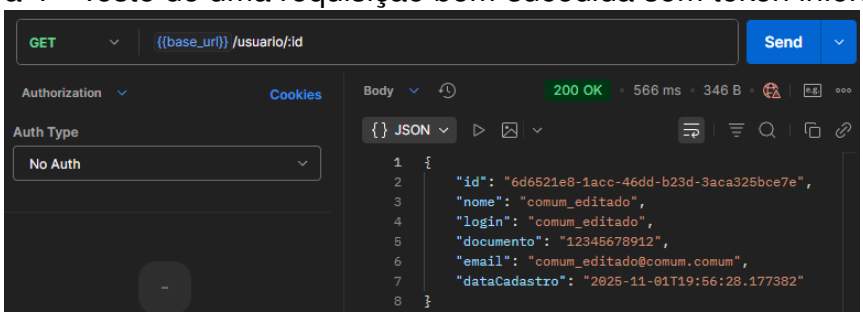
O ponto de partida do trabalho foi o desenvolvimento da API base, sem qualquer implementação de segurança. O projeto inicial foi estruturado com as entidades Empresa, Usuário e Registro. Sem qualquer restrição, é permitida a manipulação de dados sensíveis por meio de requisições simples. A aplicação desta API base serve como o cenário de vulnerabilidade inicial, fundamental para a validação dos testes subsequentes.

4.2 MITIGAÇÃO DA VULNERABILIDADE DE AUTENTICAÇÃO (OWASP: FALHAS DE IDENTIFICAÇÃO E AUTENTICAÇÃO)

Qualquer usuário podia realizar operações de alteração ou exclusão de dados sem comprovar sua identidade. A mitigação dessa falha foi iniciada com a implementação de um sistema de login e a adoção do padrão JSON Web Token (JWT) para autenticação.

No cenário inicial, a requisição a um endpoint de dados críticos era aceita sem a necessidade de um cabeçalho de autorização, expondo a aplicação a acessos não autorizados, conforme a Figura 4.

Figura 4 - Teste de uma requisição bem-sucedida sem token informado



Fonte: Do autor.

O sistema foi aprimorado com o uso do JWT. Ao efetuar o lo-

gin, um token de sessão é gerado e se torna obrigatório nas requisições subsequentes, como mostrado na Figura 5. Ao efetuar o logout, token é invalidado. O uso do atributo [Authorize] (Figura 6) garante que apenas requisições com um *header* de autenticação Bearer token válido sejam processadas, conforme a Figura 7 evidenciando erro quando não é informado o token. Para essa implementação foi utilizada a autenticação e a autorização padrão do .NET. As configurações utilizadas se assemelham aos do trabalho de (Gonçalves, 2022), embora neste tenha sido utilizado o IdentityServer para efetuar a gerenciação do token de acesso em vez de uma implementação manual.

Figura 5 - Método para geração do token de acesso

```
public static string GenerateJwtToken(Guid id) {
    var claims = new[] { // Informações adicionais no token
        new Claim( type: "id", id.ToString()), // ID do usuário
        new Claim( type: JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()) // ID único para o token
    };
    var tokenDescriptor = new SecurityTokenDescriptor { // Credenciais de assinatura com chave secreta
        Subject = new ClaimsIdentity(claims),
        Expires = DateTime.UtcNow.AddSeconds(ExpiresInSeconds), // Validade de 1 hora
        SigningCredentials = new SigningCredentials(GetSecretKey(), algorithm: SecurityAlgorithms.HmacSha256)
    };
    var tokenHandler = new JwtSecurityTokenHandler(); // Gerador de tokens
    return tokenHandler.WriteToken(tokenHandler.CreateToken(tokenDescriptor)); // Retorna como string
}
```

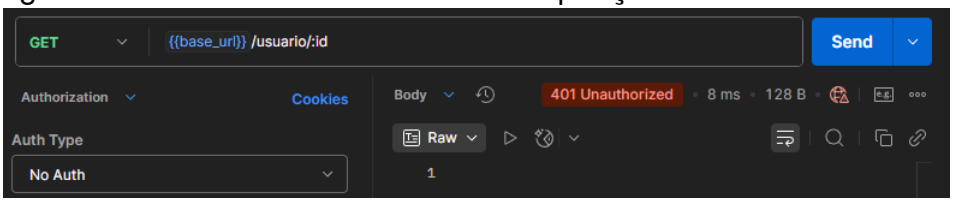
Fonte: Do autor.

Figura 6 - Atributo [Authorize], indicando necessidade de autenticação

```
[Authorize]
[HttpGet]
Rogers Billieri *
public async Task<IActionResult> GetAll() {
```

Fonte: Do autor.

Figura 7 - Teste no Postman de uma requisição sem token informado



Fonte: Do autor.

4.3 MITIGAÇÃO DE INJEÇÃO DE CÓDIGO (OWASP: INJEÇÃO DE SQL)

A injeção de código, particularmente a SQL Injection, é uma das vulnerabilidades mais críticas e conhecidas em aplicações Web. Ela ocorre quando comandos SQL maliciosos são inseridos em campos de entrada de dados e executados pelo banco de dados devido à ausência de tratamento adequado dessas entradas. Essa falha pode permitir desde a leitura indevida de dados até a exclusão completa de tabelas.

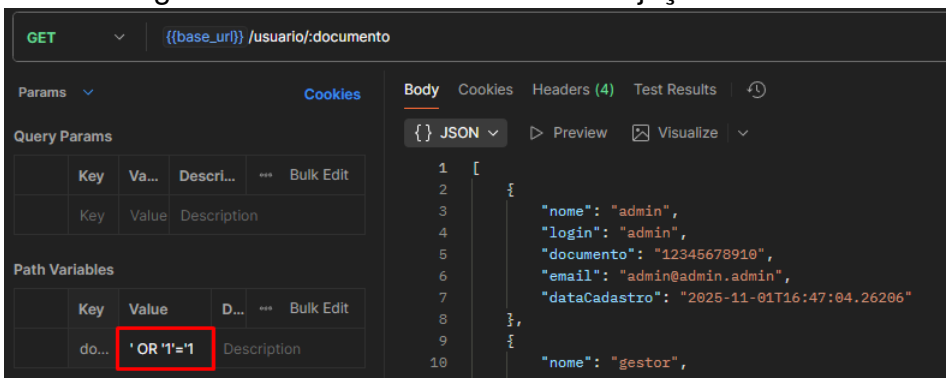
Antes da implementação, as consultas ao banco de dados eram realizadas por meio de comandos SQL concatenados manualmente, como mostra a Figura 8. Isso permitia que um invasor enviasse instruções SQL arbitrárias nos parâmetros de requisição, comprometendo a integridade da base de dados. Por exemplo, um endpoint que recebia o parâmetro email diretamente na consulta poderia ser explorado com uma string maliciosa como: ” ’ OR ’1’=’1’ – ”. Esse tipo de entrada faria o banco de dados retornar todos os registros de usuários, violando completamente o controle de acesso, como evidenciado na Figura 9. Ainda mais grave, a string ” ’ OR ’1’=’1’; DROP TABLE usuarios; –”, dependendo das configurações do banco de dados, poderia apagar a tabela de usuários inteira.

Figura 8 - Trecho de código vulnerável com interpolação de string SQL

```
public async Task<List<UsuarioResponse>> Get(string documento) {
    var sql = $"SELECT documento, nome, email, login, datacadastro FROM usuario WHERE documento = '{documento}'";
    using var conn = new NpgsqlConnection(_connectionString); conn.Open();
    using var cmd = new NpgsqlCommand(sql, conn); using var reader = await cmd.ExecuteReaderAsync();
    var resultados = new List<UsuarioResponse>();
    while (reader.Read()) { resultados.Add(new UsuarioResponse( documento: reader.GetString( ordinal: 0), nome: reader.GetString( ordinal: 1),
            email: reader.GetString( ordinal: 2), login: reader.GetString( ordinal: 3), dataCadastro: reader.GetDateTime( ordinal: 4))); }
    return resultados;
}
```

Fonte: Do autor.

Figura 9 - Teste bem-sucedido de injeção de SQL

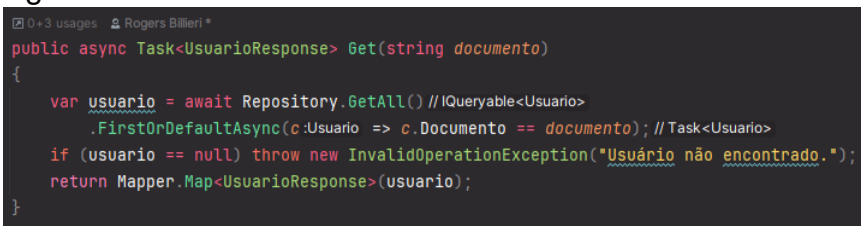


Fonte: Do autor.

Após a implementação, a vulnerabilidade foi mitigada com a utilização exclusiva do Entity Framework Core (EF Core) em conjunto com consultas LINQ para todas as operações de persistência e leitura de dados, conforme a Figura 10.

O EF Core converte automaticamente as consultas LINQ em comandos SQL parametrizados, impossibilitando a injeção de instruções arbitrárias, demonstrado pela Figura 11.

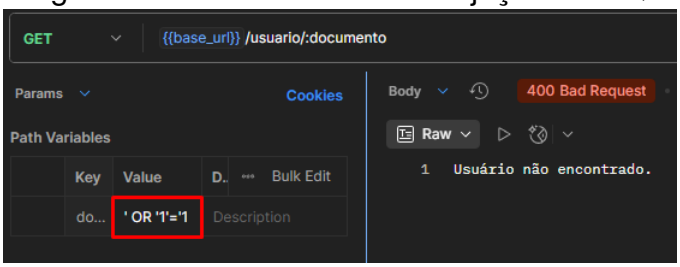
Figura 10 - Consulta ao banco de dados utilizando LINQ e EF



```
0+3 usages  Rogers Billieri *
public async Task<UsuarioResponse> Get(string documento)
{
    var usuario = await Repository.GetAll() // IQueryable<Usuario>
        .FirstOrDefaultAsync(c :Usuario => c.Documento == documento); // Task<Usuario>
    if (usuario == null) throw new InvalidOperationException("Usuário não encontrado.");
    return Mapper.Map<UsuarioResponse>(usuario);
}
```

Fonte: Do autor.

Figura 11 - Tentativa falha de injeção de SQL



Fonte: Do autor.

Junto a isso, foram implementadas validações de entrada de dados como tamanho máximo e caracteres permitidos, garantindo a saúde do banco de dados pelo controle do que será salvo ou não.

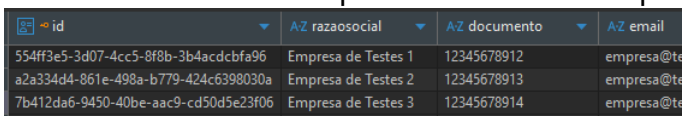
4.4 MITIGAÇÃO DA VULNERABILIDADE DE ID SEQUENCIAL (OWASP: CONTROLE DE ACESSO QUEBRADO - IDOR)

A API base utilizava identificadores inteiros sequenciais (ID tipo int) para as entidades. Essa abordagem facilita a enumeração de recursos e a execução de ataques IDOR (Insecure Direct Object Reference).

Antes da implementação, um atacante poderia deduzir o ID de outros usuários ou registros simplesmente incrementando o ID na URL (/api/registros/1, /api/registros/2, etc.), permitindo a consulta ou verificação de existência inadequada.

Após a implementação, o mapeamento das entidades no EF Core foi modificado para utilizar identificadores globais exclusivos (GUID), que são imprevisíveis. Além disso, a validação de permissão garante que, mesmo que o GUID seja adivinhado, o usuário não consiga acessar o recurso, mitigando duplamente o risco. A Figura 12 mostra o tipo de ID GUID.

Figura 12 - Entidades com tipo Guid como chave primária



id	razaosocial	documento	email
554ff3e5-3d07-4cc5-8f8b-3b4acdcbfa96	Empresa de Testes 1	12345678912	empresa@te
a2a334d4-861e-498a-b779-424c6398030a	Empresa de Testes 2	12345678913	empresa@te
7b412da6-9450-40be-aac9-cd50d5e23f06	Empresa de Testes 3	12345678914	empresa@te

Fonte: Do autor.

4.5 PROTEÇÃO DE DADOS SENSÍVEIS EM REPOUSO (OWASP: FALHAS CRIPTOGRÁFICAS)

A vulnerabilidade inicial envolvia o armazenamento de senhas dos usuários em texto simples ou com um hash inseguro no banco de dados. Para proteger os dados de autenticação em repouso, foi implementado o algoritmo de hashing Argon2id, que é altamente resistente a ataques de força bruta e rainbow tables.

Antes da implementação, a senha era armazenada diretamente em texto simples, como ilustrado no banco de dados pela Figura 13.

Figura 13 - Senha exposta no banco de dados

id	nome	login	documento	senha
9989e0e5-f671-4c51-8eee-1ceaf9b7e6dd	admin	admin	12345678910	Senha1010@
0c008100-cba2-4026-b326-e07b1ee9e105	gestor	gestor	12345678911	Gestor1010@
6d6521e8-1acc-46dd-b23d-3aca325bce7e	comum_editado	comum_editado	12345678912	Comum1010@

Fonte: Do autor.

Após a implementação, exibida na Figura 15, a lógica de registro foi alterada para aplicar o hash da senha utilizando o Argon2id, garantindo que, mesmo em caso de vazamento da base de dados, as credenciais não sejam comprometidas, como mostra a Figura 14.

Figura 14 - Senha cifrada no banco de dados

id	nome	login	documento	senha
9989e0e5-f67	admin	admin	12345678910	ouDS7ikysT1tLkpu5G22e
0c008100-cb	gestor	gestor	12345678911	zg1Cf88+GGf+SzVVg++/
6d6521e8-1a	comum_editad	comum_edita	12345678912	Gen8Bn7FHkr2plkne3Y62

Fonte: Do autor.

Figura 15 - Implementação do hashing com Argon2id

```
public string HashPassword(string password) {
    var salt = new byte[16]; // Salt de 16 bytes
    using (var rng = RandomNumberGenerator.Create()) { rng.GetBytes(salt); }
    var hash:byte[] = HashPassword(password, salt);
    var combinedBytes = new byte[salt.Length + hash.Length]; // Combinar salt e hash
    Array.Copy( sourceArray: salt,  sourceIndex: 0,  destinationArray: combinedBytes,  destinationIndex: 0, salt.Length);
    Array.Copy( sourceArray: hash,  sourceIndex: 0,  destinationArray: combinedBytes,  destinationIndex: salt.Length, hash.Length);
    return Convert.ToBase64String(combinedBytes); // Converter para base64 para armazenar no bd
}
2 usages  Rogers Billieri *
private byte[] HashPassword(string password, byte[] salt) { //DegreeOfParallelism = n# de threads
    return new Argon2id( password: Encoding.UTF8.GetBytes(password)) { Salt = salt, DegreeOfParallelism = 8,
        Iterations = 4, MemorySize = 1024 * 1024 }.GetBytes( bc: 32); //32 = tamanho do hash
}
```

Fonte: Do autor.

4.6 MITIGAÇÃO DE MÉTODOS ABERTOS (OWASP: CONTROLE DE ACESSO QUEBRADO)

Antes da implementação, um usuário autenticado poderia acessar ou modificar recursos de outra empresa ou outro usuário. Esta falha é a

essência do Controle de Acesso Quebrado. O usuário ”A” conseguia alterar dados do usuário ”B” se soubesse o ID (ou GUID) do recurso, pois a lógica de autorização se limitava à validade do token. A Figura 16 demonstra um exemplo de uma edição sem validações.

Figura 16 - Edição sem verificações de permissão

```
public async Task<RegistroResponse> Edit(Guid id, RegistroRequest request) {
    var registro = await Repository.GetById(id).FirstOrDefaultAsync();
    if (registro == null) throw new InvalidOperationException("Registro não encontrado.");
    Mapper.Map(request, registro); await Repository.UpdateAsync(registro); await Repository.SaveAsync();
    return Mapper.Map<RegistroResponse>(registro);
}
```

Fonte: Do autor.

Foram implementados mecanismos de autorização baseada em requisitos e validações granulares na camada de aplicação. Antes de cada operação, o ID do usuário autenticado e suas permissões são verificados, garantindo que ele só possa interagir com dados aos quais tem permissão explícita, exemplificado pelo código contido na Figura 17.

Figura 17 - Filtro pela limitação de acesso do usuário

```
public async Task<RegistroResponse> Edit(Guid id, RegistroRequest request) {
    var registro :IQueryable<Registro> = Repository.GetById(id);
    if (_userContext.Perfil == EPerfilUsuario.Gestor)
        registro = registro.Where(c :Registro => c.Usuario.EmpresaId == _userContext.EmpresaId);
    if (_userContext.Perfil == EPerfilUsuario.Comum)
        registro = registro.Where(c :Registro => c.UsuarioId == _userContext.Id);
    if (registro == null) throw new InvalidOperationException("Registro não encontrado.");
    var registroResult = await registro.FirstOrDefaultAsync();
    Mapper.Map(request, registroResult);
    await Repository.UpdateAsync(registroResult); await Repository.SaveAsync();
    return Mapper.Map<RegistroResponse>(registroResult);
}
```

Fonte: Do autor.

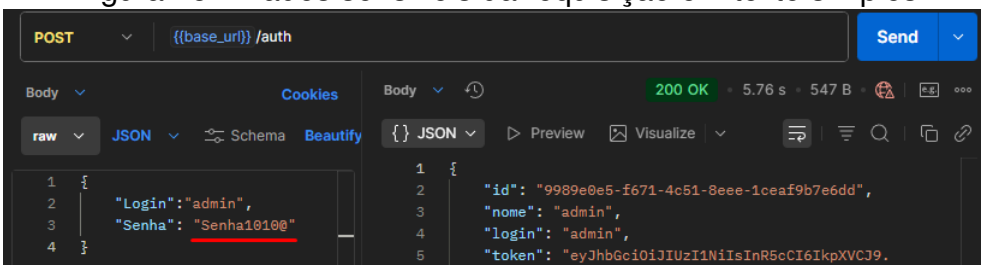
4.7 PROTEÇÃO DE DADOS SENSÍVEIS EM TRÂNSITO (OWASP: FALHAS CRIPTOGRÁFICAS)

A segurança dos dados transmitidos entre o cliente e a API, como credenciais de login ou informações pessoais, é fundamental para prevenir ataques de interceptação (sniffing) e Man-in-the-Middle (MITM). Para demonstrar a capacidade de proteger o conteúdo da requisição antes mesmo da camada de transporte, optou-se pela implementação da criptografia RSA (algoritmo de chave pública e privada) para proteger campos críticos, este algoritmo também é utilizado pelo trabalho de (AOYAMA, 2016), que utilizou para o mesmo propósito de cifrar dados em trânisto.

Antes da implementação, a transmissão de dados era feita em texto simples, tornando-os legíveis caso a comunicação fosse interceptada

(como em um ataque de sniffing em uma rede desprotegida ou na ausência da obrigatoriedade de HTTPS), demonstrado pela Figura 18.

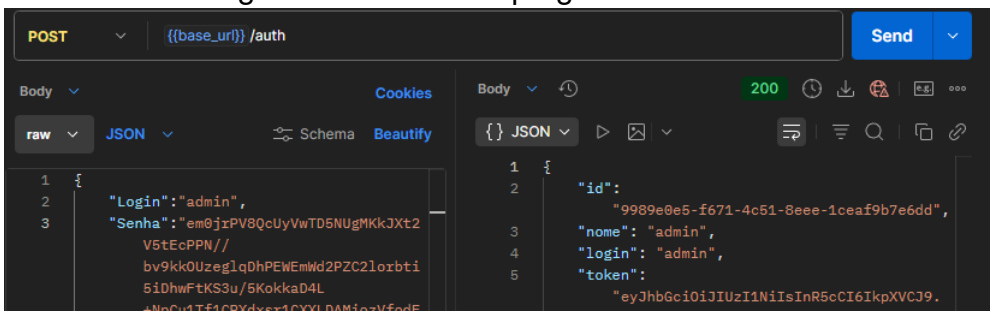
Figura 18 - Dados sensíveis da requisição em texto simples



Fonte: Do autor.

Após a implementação, exemplificada na Figura 20, a aplicação foi configurada para utilizar um par de chaves RSA. O cliente deve criptografar dados sensíveis utilizando a chave pública da API antes de enviar a requisição. Ao receber a requisição, a API utiliza sua chave privada correspondente para decifrar a informação. Essa tratativa demonstra uma camada de criptografia aplicacional que protege os dados do início ao fim do trânsito, assegurando a confidencialidade de informações como a senha do usuário durante o login, ao custo de uma implementação adicional de criptografia pelo lado do cliente. Na Figura 19 vemos uma requisição sendo efetuada com os dados críticos criptografados.

Figura 19 - Senha criptografada no envio



Fonte: Do autor.

Figura 20 - Implementação da criptografia RSA

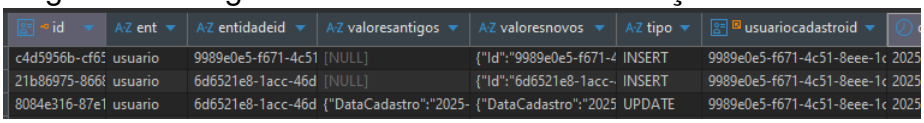
```
public class RsaCryptoService(IConfiguration configuration, ILogger<RsaCryptoService> logger) : IRsaCryptoService {
    private RSA _rsa; private string _privateKey = configuration["RSA:Private"]; private string _publicKey = configuration["RSA:Public"];
    0+1 usages  Rogers Billieri *
    public string Encrypt(string plainText) {
        _rsa = RSA.Create(); _rsa.ImportFromPem( _publicKey);
        var data :byte[] = Encoding.UTF8.GetBytes(plainText);
        return Convert.ToBase64String( inArray: _rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA256));
    }
    0+2 usages  Rogers Billieri *
    public string Decrypt(string base64Encrypted) {
        try {
            _rsa = RSA.Create(); _rsa.ImportFromPem( _privateKey);
            var encryptedData :byte[] = Convert.FromBase64String(base64Encrypted);
            return Encoding.UTF8.GetString( bytes: _rsa.Decrypt(encryptedData, RSAEncryptionPadding.OaepSHA256));
        } catch (Exception e) { logger.LogError(e, e.Message); throw new Exception("Dados inválidos."); }
    }
```

Fonte: Do autor.

4.8 IMPLEMENTAÇÃO DE AUDITORIA E LOGS (OWASP: FALHAS DE MONITORAMENTO E AUDITORIA DE SEGURANÇA)

A API inicial não possuía nenhum mecanismo para registrar ações críticas, impedindo a detecção e a resposta a incidentes de segurança. Foi criada uma Tabela de Auditoria no banco de dados e uma lógica de registro automático de alterações das entidades, exemplificada na Figura 21, permitindo o registro das alterações das entidades às quais o desenvolvedor achar importantes para manter um histórico, registrando as operações de INSERT, UPDATE e DELETE no banco de dados.

Figura 21 - Registro no banco de dados das ações de auditoria

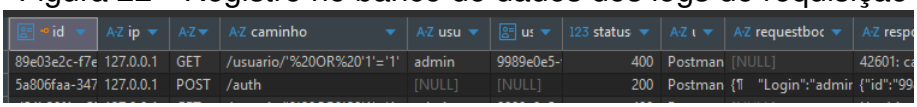


id	ent	entidadeid	valoresantigos	valoresnovos	tipo	usuariocadastroid
c4d5956b-cf6	usuario	9989e0e5-f671-4c51	[NULL]	{"Id":"9989e0e5-f671-4	INSERT	9989e0e5-f671-4c51-8eee-1c 2025
21b86975-866	usuario	6d6521e8-1acc-46d	[NULL]	{"Id":"6d6521e8-1acc-	INSERT	9989e0e5-f671-4c51-8eee-1c 2025
8084e316-87e1	usuario	6d6521e8-1acc-46d	{"DataCadastro":"2025-	{"DataCadastro":"2025	UPDATE	9989e0e5-f671-4c51-8eee-1c 2025

Fonte: Do autor.

Também foi criada uma tabela de Logs de requisições, registrando todas as requisições feitas para a aplicação, como mostra a Figura 22,. Junto a isso, foi criado um atributo que, quando usado em um método, desabilita o log das requisições para aquela função.

Figura 22 - Registro no banco de dados dos logs de requisição

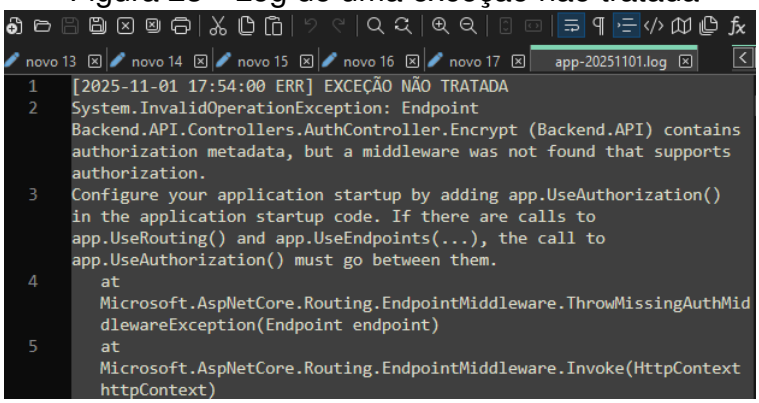


id	ip	caminho	usu	us	status	requestboc	resp
89e03e2c-f7e	127.0.0.1	GET	/usuario/'%20OR%20'1'='1'	admin	9989e0e5-	400 Postman	[NULL] 42601: c
5a806faa-347	127.0.0.1	POST	/auth	[NULL]	[NULL]	200 Postman	{¶ "Login":"admir {"id":"99

Fonte: Do autor.

Foi implementado também o registro de Logs de exceções em arquivo, mantendo um histórico de todos os erros inesperados e não tratados que podem ocorrer durante o funcionamento da aplicação. O desenvolvedor pode controlar as escritas de exceções neste arquivo a partir do código. A Figura 23 revela um caso de exemplo.

Figura 23 - Log de uma exceção não tratada



Fonte: Do autor.

4.9 MITIGAÇÃO DE ATAQUES EM MASSA (NEGAÇÃO DE SERVIÇO - DOS)

Antes da implementação, a API era vulnerável a ataques de Negação de Serviço (DoS) por meio de repetição excessiva de requisições, como tentativas de brute-force contra o endpoint de login.

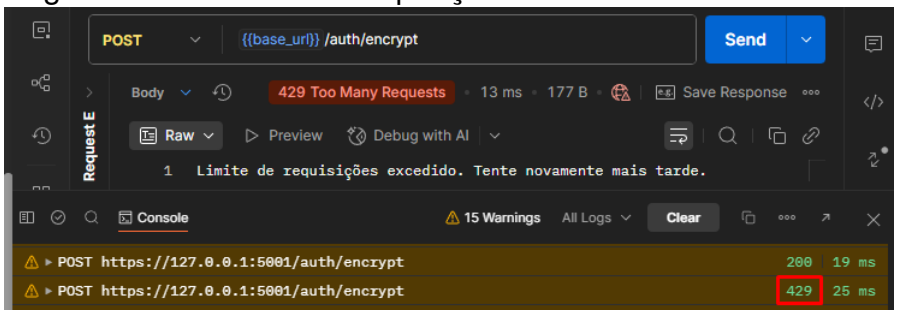
Após a implementação, foi adicionado um mecanismo de Rate Limiting configurável como evidenciado na Figura 24. O Rate Limiting bloqueia temporariamente um endereço IP (ou usuário) que excede um número predefinido de requisições em um curto período. Também foi implementado um contador de logins falhos por erro de senha, bloqueando o usuário na terceira tentativa falha. A Figura 25 demonstra o erro causado quando um usuário tenta efetuar requisições excessivas.

Figura 24 - Configuração do Rate Limiting no .NET 8

```
builder.Services.AddRateLimiter(options => { // Config Global - 10 req por minuto
    options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>( partitioner: httpContext =>
        RateLimitPartition.GetFixedWindowLimiter(
            partitionKey: httpContext.User.Identity?.Name ?? httpContext.Request.Headers.Host.ToString(),
            factory: partition :string => new FixedWindowRateLimiterOptions
            { AutoReplenishment = true, PermitLimit = 10, QueueLimit = 0, Window = TimeSpan.FromMinutes(1) }));
    options.OnRejected = async (context, token) => {
        context.HttpContext.Response.StatusCode = StatusCodes.Status429TooManyRequests;
        await context.HttpContext.Response.WriteAsync("Limite de requisições excedido. Tente novamente mais tarde.", token);
    };
});
```

Fonte: Do autor.

Figura 25 - Excesso de requisições resultando em Status 429



Fonte: Do autor.

Complementando as medidas de Rate Limiting, implementou-se um mecanismo de blacklist de endereços IP que permite bloquear de forma persistente ou temporária IPs maliciosos ou que apresentaram comportamento anômalo (brute force, spam de requisições, scanners). A blacklist é consultada no início do processamento de requisições, garantindo rejeição imediata (código HTTP 403). A lista é gerenciável via banco de dados. Nos testes, IPs adicionados foram imediatamente bloqueados, confirmando a eficácia como uma camada adicional à estratégia de rate limiting.

Dessa maneira, este estudo se aproxima da tese de (Gonçalves, 2022) ao adotar uma abordagem focada na construção de uma API REST segura, fundamentada em boas práticas de desenvolvimento e em mecanismos sólidos de autenticação, autorização e proteção de dados. Assim como o autor, este trabalho reconhece a relevância crescente das APIs no cenário corporativo e a necessidade de estruturar soluções seguras desde as etapas iniciais do projeto. No entanto, a principal diferenciação reside no caráter eminentemente prático e experimental deste estudo: enquanto a dissertação de Gonçalves concentra-se na definição de métricas teóricas e diretrizes de design para orientar o desenvolvimento de APIs robustas, este trabalho se propôs a implementar e testar na prática cada camada de proteção, documentando de forma comparativa o comportamento da aplicação antes e depois das medidas aplicadas. Além disso, enquanto a solução desenvolvida na tese foi integrada ao contexto de um sistema corporativo específico, a API deste trabalho foi desenvolvida de forma modular e reutilizável, com foco na demonstração didática da eficácia das técnicas de segurança em cenários simulados de ataque.

Este trabalho também dialoga com o estudo de (AOYAMA, 2016), uma vez que ambos compartilham a preocupação com a proteção dos dados em trânsito e a necessidade de incorporar segurança desde as fases iniciais do desenvolvimento. No entanto, enquanto o estudo citado concentra-se na criação de uma API especializada em criptografia dinâmica, com seleção automática de algoritmos conforme o cenário de uso, o presente projeto adota uma abordagem mais ampla, integrando múltiplas camadas de segurança além da criptografia, como autenticação, autorização, auditoria e controle de acesso. Assim, ao passo que a outra pesquisa aprofunda-se na otimização criptográfica e na escolha adaptativa de cifras, este trabalho busca demonstrar a aplicação prática e combinada de diversas técnicas de segurança aplicacional dentro de uma arquitetura REST completa e modular.

5 CONCLUSÃO

O presente trabalho teve como objetivo demonstrar, de forma prática e aplicada, a implementação de medidas de segurança em uma API Web desenvolvida em .NET 8, com foco na mitigação das vulnerabilidades mais críticas descritas pela OWASP Foundation, gerando um projeto que pode ser reutilizado por outros desenvolvedores. A partir da construção de uma API funcional, foram aplicadas e testadas camadas de proteção

voltadas à autenticação, autorização, criptografia, auditoria e limitação de requisições, contribuindo para a formação de uma base de código segura e reutilizável.

Entre os pontos positivos, destaca-se a abordagem prática e validada por meio de testes de penetração simulados, o que proporcionou uma avaliação concreta das medidas aplicadas. O uso de tecnologias atuais, como o algoritmo Argon2id para hashing de senhas, o padrão JWT para autenticação e o mecanismo de Rate Limiting integrado ao .NET 8, reforçou a aderência do projeto às boas práticas de segurança e ao estado da arte no desenvolvimento de APIs seguras. Além disso, a documentação do comportamento da aplicação antes e depois das implementações fornece uma contribuição didática relevante, permitindo que outros desenvolvedores compreendam o impacto real de cada medida de proteção.

Como pontos negativos e limitações, o projeto apresentou restrições relacionadas à abrangência das vulnerabilidades tratadas. Apesar de cobrir as principais ameaças listadas pela OWASP, nem todas as categorias foram exploradas em profundidade, como falhas de configuração de segurança e deserialização insegura. Além disso, os testes de penetração foram realizados em ambiente controlado, sem exposição da API a um cenário real de rede, o que limita a generalização dos resultados. Outro ponto a ser considerado é a ausência de uma análise quantitativa mais detalhada do desempenho da aplicação após a aplicação das medidas de segurança, o que poderia reforçar a avaliação do impacto das proteções no tempo de resposta e no consumo de recursos.

As principais contribuições deste estudo incluem o desenvolvimento de uma arquitetura prática e modular de segurança para APIs em .NET, a integração de mecanismos de autenticação e criptografia avançados, e a criação de uma base de código que pode servir como referência para futuras implementações. O trabalho também contribui para a disseminação de práticas seguras no desenvolvimento de software, alinhando-se às diretrizes da Lei Geral de Proteção de Dados (LGPD) e à necessidade crescente de conformidade e proteção de dados sensíveis em ambientes corporativos e governamentais.

Em síntese, o projeto cumpriu o objetivo de demonstrar a aplicabilidade das práticas de segurança no desenvolvimento de APIs, gerando junto a isso uma API Web segura que pode servir como base para projetos de outros desenvolvedores. Apesar das limitações, os resultados obtidos comprovam a relevância e a efetividade das medidas implementadas, ofe-

recendo uma contribuição prática e acadêmica ao campo da segurança da informação e ao desenvolvimento seguro de software.

Como trabalhos futuros, recomenda-se a ampliação do escopo das medidas de segurança para contemplar a proteção contra vulnerabilidades adicionais, como ataques de deserialização e falhas de configuração. A finalização do fluxo do usuário a partir da opção de redefinição de senha por envio de e-mail também é recomendada, juntamente com a implementação de um duplo-fator de autenticação. Sugere-se, ainda, a integração da API com ferramentas automatizadas de monitoramento de segurança e sistemas de detecção de intrusões (IDS/IPS), bem como a aplicação de testes de penetração em ambiente de produção simulado. Outra linha promissora é a utilização de técnicas de machine learning para análise de padrões de requisição e detecção proativa de comportamentos anômalos, reforçando o papel da inteligência artificial na segurança cibernética.

REFERÊNCIAS

ACCENTURE, F. e. **Global Cybersecurity Outlook 2025**. [S.l.], 2025. Accesso em: 14 abr. 2025. Disponível em: <https://www.weforum.org/publications/global-cybersecurity-outlook-2025/>.

AOYAMA, D. D. S. **Desenvolvimento de api criptográfica para comunicação segura entre aplicações web: Estudo de caso aplicado em web service rest**. UNIVEM Aberto, 2016. Accesso em: 14 abr. 2025. Disponível em: <https://aberto.univem.edu.br/handle/11077/1584>.

BARBOSA, J. S. et al. **A proteção de dados e segurança da informação na pandemia covid-19: contexto nacional**. Research, Society and Development, 2021, v. 10, n. 2, p. e40510212557–e40510212557. Accesso em: 25 abr. 2025. Disponível em: <https://rsdjournal.org/rsd/article/view/12557>.

BUTTRICK, H. G. **The skeleton of a data breach: The ethical and legal concerns**. Richmond Journal of Law Technology, 2016. Accesso em: 14 abr. 2025. Disponível em: <https://scholarship.richmond.edu/jolt/vol23/iss1/2/>.

Check Point Software. **The Cyber Security Report 2025**. [S.l.], 2025. Accesso em: 14 abr. 2025. Disponível em: <https://www.checkpoint.com/security-report/?flz-category=items&flz-item=report--cyber-security-report-2025>.

FEILER, A. R.; GAZANIGA, F.; VIEIRA, T. A. M. **O valor fundamental dos dados pessoais: uma análise comparativa entre a lgpd e gdpr sob a ótica da análise econômica do direito**. Revista de Direito,

2024, v. 16, n. 02, p. 01–29. Accesso em: 14 abr. 2025. Disponível em: <https://periodicos.ufv.br/revistadir/article/view/17158>.

GONÇALVES, A. d. S. **Concepção e desenvolvimento de uma API REST com incorporação de mecanismos de segurança aplicacional**. Tese (Doutorado) — Universidade do Minho, 2022. Accesso em: 14 abr. 2025. Disponível em: <https://repositorium.uminho.pt/entities/publication/84e2119b-dcbe-4d5a-aea4-fc5bda3ee2cf>.

PINHEIRO, P. P. **Proteção de dados pessoais: comentários à Lei n. 13.709/2018 (LGPD)**. 1. ed. São Paulo: Saraiva Educação, 2018. Direito à privacidade - Legislação - Brasil; Direitos fundamentais; Proteção de dados - Legislação. CDU 342.7(81).

SCHIRMER D. L. E THAINES, A. H. **A implementação da lei geral de proteção de dados nas rotinas dos profissionais da área contábil: percepções dos contabilistas associados à associação dos contabilistas do vale do paranhana/rs**. FACCAT, 2021. Accesso em: 17 abr. 2025. Disponível em: <https://repositorio.ucs.br/xmlui/handle/11338/11597>.

Sonic Wall. **SonicWall 2024 Mid-Year Threat Report**. [S.l.], 2024. Accesso em: 14 abr. 2025. Disponível em: <https://www.sonicwall.com/resources/white-papers/mid-year-2024-sonicwall-cyber-threat-report/gated/thank-you/asset>.

VIANNA, E. W.; FERNANDES, J. H. C. **O gestor da segurança da informação no espaço cibernético governamental: grandes desafios, novos perfis e procedimentos**. Brazilian Journal of Information Science: research trends, 2015, v. 9, n. 1. Accesso em: 14 abr. 2025. Disponível em: <https://revistas.marilia.unesp.br/index.php/bjis/article/view/5216>.