

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GUSTAVO DE CASTRO GUIMARÃES BARBOSA
Orientador: Prof. Dr. Carlos Frederico Marcelo da Cunha Cavalcanti

**AUTENTICAÇÃO E AUTORIZAÇÃO EM MICRO SERVIÇOS:
ASPECTOS DE SEGURANÇA DE SISTEMAS**

Ouro Preto, MG
2024

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GUSTAVO DE CASTRO GUIMARÃES BARBOSA

**AUTENTICAÇÃO E AUTORIZAÇÃO EM MICRO SERVIÇOS: ASPECTOS DE
SEGURANÇA DE SISTEMAS**

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Carlos Frederico Marcelo da Cunha Cavalcanti

Ouro Preto, MG
2024



FOLHA DE APROVAÇÃO

Gustavo de Castro Guimarães Barbosa

AUTENTICAÇÃO E AUTORIZAÇÃO EM MICRO SERVIÇOS ASPECTOS DE SEGURANÇA DE SISTEMAS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 21 de Outubro de 2024.

Membros da banca

Carlos Frederico M. da Cunha Cavalcanti (Orientador) - Doutor - Universidade Federal de Ouro Preto
Fernando Cortez Sica (Examinador) - Doutor - Universidade Federal de Ouro Preto
Ricardo Augusto Rabelo Oliveira (Examinador) - Doutor - Universidade Federal de Ouro Preto

Carlos Frederico M. da Cunha Cavalcanti, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 12/11/2024.



Documento assinado eletronicamente por **Carlos Frederico Marcelo da Cunha Cavalcanti**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 13/11/2024, às 11:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0795434** e o código CRC **B1DFACC3**.

Resumo

Com o avanço dos microsserviços na arquitetura de sistemas distribuídos, surge a necessidade de garantir segurança, especialmente nas áreas de autenticação e autorização, para confirmar a identidade dos usuários. Essa preocupação se estende aos princípios fundamentais de segurança da informação, como confidencialidade, integridade e disponibilidade, que são mantidos por meio de técnicas criptográficas, como chaves simétricas e assimétricas, e o uso de funções hash. O padrão JSON Web Tokens (JWT) se destaca como uma ferramenta crucial para assegurar a integridade das informações sensíveis e garantir a autenticidade das comunicações. Em ambientes distribuídos, a adoção de práticas como o OAuth 2.0 e o Single Sign-On (SSO) são destacadas para garantir uma comunicação segura e persistir a sessão do usuário. No entanto, a natureza descentralizada dos microsserviços apresenta desafios únicos, exigindo abordagens inovadoras e contínuas para estabelecer confiança entre serviços distribuídos, autenticar usuários e mitigar ameaças. Neste contexto, este trabalho propõe a implementação de uma arquitetura de microsserviços com autenticação e autorização baseada em token e realização de teste de estresse, buscando compreender os requisitos de segurança em ambientes distribuídos.

Palavras-chave: Autenticação. Autorização. Micro serviços. API. REST. Chave simétrica. Chave assimétrica. Função hash. Segurança de sistema. Token. JWT. SSO. OWASP. FUZZING. OAuth 2.0. Teste de estresse.

Abstract

With the advancement of microservices in the architecture of distributed systems, the need to guarantee security arises, especially in the areas of authentication and authorization, to confirm the identity of users. This concern extends to the fundamental principles of information security, such as confidentiality, integrity and availability, which are maintained through cryptographic techniques, such as symmetric and asymmetric keys, and the use of hash functions. The JSON Web Tokens (JWT) standard stands out as a crucial tool for ensuring the integrity of sensitive information and guaranteeing the authenticity of communications. In distributed environments, the adoption of practices such as OAuth 2.0 and Single Sign-On (SSO) are highlighted to ensure secure communication and persist the user session. However, the decentralized nature of microservices presents unique challenges, requiring innovative and ongoing approaches to establishing trust between distributed services, authenticating users, and mitigating threats. In this context, this work proposes the implementation of a microservices architecture with token-based authentication and authorization and stress testing, seeking to understand the security requirements in distributed environments.

Keywords: Authentication. Authorization. Microservices. API. REST. Symmetrical key. Asymmetric key. Hash function. System security. Token. JWT. SSO. OWASP. FUZZING. OAuth 2.0. Stress testing.

Lista de Ilustrações

Figura 2.1 – Arquitetura monolítica e microsserviço	3
Figura 2.2 – Web Api	4
Figura 2.3 – criptografia simétrica	8
Figura 2.4 – criptografia assimétrica	10
Figura 2.5 – Estrutura jwt	13
Figura 2.6 – Exemplo token secret key jwt	14
Figura 2.7 – Fluxo OAuth 2.0	15
Figura 2.8 – Exemplo SSO	16
Figura 2.9 – Arquitetura baseada em API Gateway	19
Figura 2.10–Arquitetura baseada em Token	20
Figura 2.11–Arquitetura baseada em mTLS	21
Figura 2.12–Tabela comparativa entre os métodos	22
Figura 2.13–Modelo centralizado	23
Figura 2.14–modelo descentralizado	24
Figura 3.1 – Arquitetura baseada em token	27
Figura 3.2 – Estrutura de pastas da API de autenticação e autorização.	31
Figura 3.3 – Endpoints da API de autenticação e autorização.	31
Figura 3.4 – Middleware da API de autenticação e autorização.	32
Figura 3.5 – Função de autenticação da API de autenticação e autorização.	33
Figura 3.6 – Funções de autorizações da API de autenticação e autorização.	33
Figura 3.7 – Refresh token da API de autenticação e autorização.	34
Figura 3.8 – Estrutura de pastas da API produtos.	35
Figura 3.9 – Endpoints da API de produtos.	35
Figura 3.10–Middleware da API de produtos.	36
Figura 3.11–Funções de autorização da API de produtos.	37
Figura 4.1 – Login.	38
Figura 4.2 – Refresh token.	39
Figura 4.3 – Endpoint para recuperar produto com sucesso.	40
Figura 4.4 – Endpoint para recuperar produto não autorizado.	41
Figura 4.5 – Ambientes	42
Figura 4.6 – Teste 1	43
Figura 4.7 – Teste 2	44
Figura 4.8 – Teste 3	45
Figura 4.9 – Comportamento durante o erro do terceiro teste	46

Lista de Tabelas

Tabela 3.1 – Endpoints do sistema. 28

Lista de Abreviaturas e Siglas

API	Application Programming Interface
REST	Representational State Transfer
PIN	Personal Identification Number
AES	Advanced Encryption Standard
DES	Data Encryption Standard
SHA	Secure Hash Algorithm
RSA	Rivest-Shamir-Adleman
JSON	JavaScript Object Notation
JWT	JSON Web Token
ECDSA	Elliptic Curve Digital Signature Algorithm
HMAC	Hash-based Message Authentication Code
ASCII	American Standard Code for Information Interchange
.NET	.network
PHP	Hypertext Preprocessor
OAuth	Open-Authorization
WEB	world wide web
HTTP	Hypertext Transfer Protocol
SSO	Single Sign-on
CRUD	Create Read Update Delete
URI	Uniform Resource Identifier
OWASP	Open Web Application Security Project
OSI	Open Systems Interconnection
SAST	Security Analysis Security Testing
CI/CD	Integration/Continuous Delivery
CPU	Central Processing Unit
DVWA	Damn Vulnerable Web Application

ZAP	Zed Attack Proxy
PKI	Public Key Infrastructure
MTLS	Mutual Transport Layer Security
RBAC	Role-Based Access Control
ABAC	Attribute-Based Access Control
OIDC	OpenID Connect
MFA	Multifactor Authentication

Sumário

1	Introdução	1
1.1	Justificativa	1
1.2	Objetivos Gerais e Específicos	2
1.3	Organização do Trabalho	2
2	Revisão Bibliográfica	3
2.1	Fundamentação Teórica	3
2.1.1	Microserviços	3
2.1.2	API REST	4
2.1.3	Autenticação e Autorização	5
2.1.4	Segurança de sistema	6
2.1.5	Criptografia	7
2.1.5.1	Chave simétrica	8
2.1.5.2	Chave assimétrica	9
2.1.5.3	Função hash	10
2.1.6	OWASP	11
2.1.7	JWT	12
2.1.8	OAuth 2.0	14
2.1.9	SSO	16
2.1.10	Token de atualização	16
2.1.11	Fuzzing	17
2.1.12	Teste de estresse	17
2.2	Trabalhos Relacionados	18
2.2.1	Uma análise geral da autenticação Mecanismo em software baseado em microserviços Arquitetura: um documento de discussão	18
2.2.2	Aprimorando a segurança de microserviços com acesso baseado em token Método de controle	22
2.2.3	Superando desafios de segurança em arquiteturas de microserviços	25
2.2.4	Implementação de Controle de Acesso Baseado em Funções no OAuth 2.0 como Sistema de Autenticação	25
2.2.5	Projetando serviços seguros para autenticação, Autorização e Contabilização de Usuários	26
3	Desenvolvimento	27
3.1	Arquitetura	27
3.2	Endpoints	28
3.3	Implementação	28
3.3.1	Cliente	28
3.3.2	Banco de dados	29
3.3.2.1	Usuário	29
3.3.2.2	Produtos	29
3.3.3	Api	29
3.3.3.1	Autenticação e autorização	30
3.3.3.2	Produtos	34

4	Resultados	38
4.1	Login	38
4.2	Refresh token	38
4.3	Produtos	39
4.4	Teste de estresse	41
5	Considerações Finais	47
5.1	Conclusão	47
5.2	Trabalhos Futuros	47
	Referências	48

1 Introdução

Com o avanço dos microsserviços, a arquitetura de sistemas distribuídos com a ideia de desenvolver um sistema como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando com mecanismos leves (FOWLER M.; LEWIS, 2014). Os pequenos serviços geralmente são Interfaces de Programação de Aplicativos (APIs), que expõe um conjunto de dados e funções para facilitar as interações entre programas de computador e permitir-lhes trocar Informação (MASSE, 2011). Nesse cenário dinâmico, segundo (ERLICH; ZVIRAN, 2015) a questão da segurança ganha destaque, com ênfase especial nas áreas cruciais de autenticação e autorização, caracterizada como o procedimento de confirmação da identidade.

No contexto da segurança de sistemas, (GHOSH, 2006) destaca os princípios fundamentais de confidencialidade, integridade e disponibilidade. Aliado a isso, diante (STALLINGS, 2014) a criptográfica assegura os princípios mencionados ao dados abordando chaves simétricas, chaves assimétricas, funções hash. O padrão JSON Web Tokens (JWT), definido por (JONES; BRADLEY; SAKIMURA, 2015), delinea a sustentação técnica essencial para a integridade de informações sensíveis e garantia de autenticidade nas comunicações, essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente.

O modelo Open Authorization (OAuth 2.0), é um protocolo de autorização, seguro e aberto, amplamente utilizado na WEB, e é concebido para permitir que aplicativos acessem recursos protegidos em nome de usuários (HAMMER-LAHAV, 2010). E o Single Sign-On (SSO), é um conceito e uma técnica de autenticação que viabiliza a um usuário o acesso a diversos sistemas ou aplicativos mediante uma única identificação e autenticação (LI et al., 2004) acrescentam como certos padrões, das práticas de segurança, nesse processo de autenticação e autorização em ambientes de microsserviços dinâmicos e descentralizados.

A exploração dos riscos e ameaças potenciais, conforme delineados pela Fundação Open Web Application Security Project (OWASP), é discutido por (IDRIS; SYARIF; WINARNO, 2022) e oferece um alicerce sólido para a elaboração de estratégias de proteção e mitigação. Dentro desse panorama, a investigação se dá com métodos de avaliação de segurança, com destaque para a técnica de Fuzzing, consiste em submeter uma aplicação a uma variedade de entradas, incluindo algumas que são atípicas e aleatórias, com o objetivo de monitorar seu comportamento (MILLER; FREDRIKSEN; SO, 1990). Além disso, embora os testes de estresse não sejam especificamente testes de segurança, eles desempenham um papel importante na avaliação do desempenho do software. Conforme (MYERS; SANDLER; BADGETT, 2012), essas metodologias avaliam a resposta de um sistema a cargas abruptas e intensas, simulando altos níveis de atividade em períodos curtos.

1.1 Justificativa

O avanço dos micros serviços na esfera tecnológica contemporânea apresenta desafios significativos em relação à segurança, autenticação, autorização e vulnerabilidade. A natureza descentralizada dos micros serviços exige abordagens inovadoras para estabelecer confiança entre serviços distribuídos, autenticar usuários e mitigar possíveis ameaças. A proposta de arquiteturas

específicas torna-se essencial, contemplando modelos de autenticação baseados em tokens, práticas como a verificação contínua de vulnerabilidades. Investigações aprofundadas e análises críticas são imperativas para o desenvolvimento de abordagens seguras, adaptadas aos requisitos contemporâneos de segurança em ambientes distribuídos.

1.2 Objetivos Gerais e Específicos

O objetivo geral deste trabalho é implementar uma arquitetura em micro serviços com autenticação e autorização baseada em token. Com isso, para atingir o objetivo geral, os seguintes objetivos específicos terão de ser alcançados:

- Realizar pesquisas para geração de embasamento teórico e revisão bibliográfica sobre o tema;
- Determinar quais ferramentas e tecnologias serão utilizadas;
- Implementar a arquitetura proposta, contendo a API de autenticação e autorização com gestão de usuários e API de algum recurso específico.
- Realizar testes no fluxo proposto pela arquitetura.

1.3 Organização do Trabalho

Este trabalho está dividido em 5 capítulos:

Capítulo 1: Introdução.

Capítulo 2: Revisão Bibliográfica/ Embasamento Teórico (com o referencial teórico e trabalhos relacionados).

Capítulo 3: Desenvolvimento, material e métodos.

Capítulo 4: Resultados e Discussões.

Capítulo 5: Conclusão e trabalhos futuros.

2 Revisão Bibliográfica

Neste capítulo será apresentado uma revisão bibliográfica e as seções de fundamentação teórica e trabalhos relacionados.

2.1 Fundamentação Teórica

2.1.1 Microsserviços

Para compreender o contexto dos microsserviços, é importante conhecer a arquitetura monolítica que se antecede. A arquitetura monolítica na engenharia de software refere-se é uma aplicação construída como uma unidade única (FOWLER M.; LEWIS, 2014). Em sistemas monolíticos, figura 2.12, todos os componentes e funcionalidades estão interligados e são desenvolvidos, implantados e escalados como uma única entidade, ou seja, fortemente acoplados. Isso significa que a lógica de negócios e a interface do usuário estão todos contidos em um único monólito.

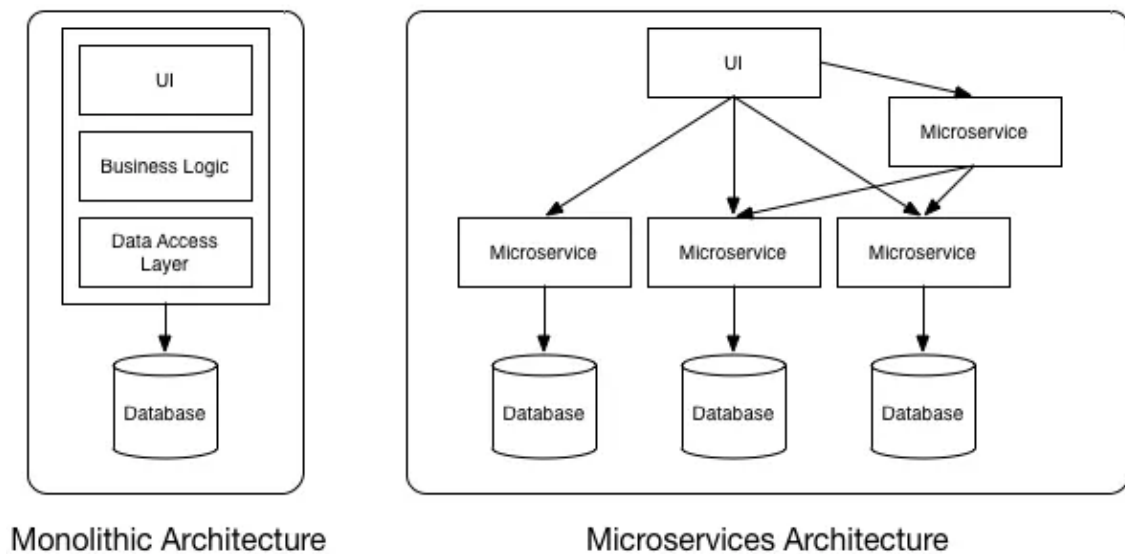


Figura 2.1 – Arquitetura monolítica e microsserviço

Passos (2019)

Microsserviços é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando com mecanismos leves (FOWLER M.; LEWIS, 2014). Em sistemas de microsserviços, a arquitetura de software é estruturada em torno de serviços independentes e autônomos, figura 2.12. Cada microsserviço é dedicado a uma função específica da aplicação, operando de maneira descentralizada. A comunicação entre esses serviços ocorre por meio de API, permitindo flexibilidade no desenvolvimento, implantação e escalabilidade. Essa abordagem modular facilita a manutenção

e atualização, uma vez que alterações em um microsserviço não afetam diretamente outros, proporcionando uma resposta ágil às demandas do ambiente.

2.1.2 API REST

Os programas clientes usam interfaces de programação de aplicativos (APIs) para comunicar com serviços web. De modo geral, uma API expõe um conjunto de dados e funções para facilitar as interações entre programas de computador e permitir-lhes trocar Informação (MASSE, 2011). Então, uma API Web é a face de um serviço Web conforme figura 2.2, diretamente a ouvir e responder às solicitações dos clientes.

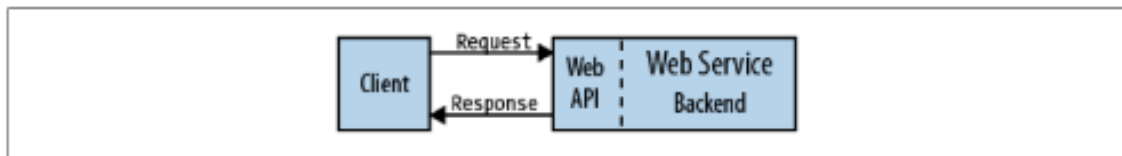


Figura 2.2 – Web Api

Masse (2011)

O estilo arquitetônico de software, Transferência Representacional de Estado (REST) foi proposta por (FIELDING, 2000) em sua tese de doutorado para definir uma interface genérica para interações entre componentes na Web, em sistemas distribuídos, incluindo o uso do Hypertext Transfer Protocol (HTTP) e Uniform Resource Identifiers (URI). Com isso, O REST é composto pelas seguintes restrições arquitetônicas:

- Cliente-servidor: O sistema é dividido em componentes de cliente e servidor, permitindo a separação de interesses e a evolução independente.
- Sem estado: cada solicitação do cliente ao servidor deve conter todas as informações necessárias para que o servidor entenda e processe a solicitação, sem depender de nenhum contexto anterior.
- Armazenável em cache: as respostas do servidor podem ser armazenadas em cache pelo cliente, melhorando a eficiência e reduzindo a carga no servidor.
- Interface uniforme: A interface entre o cliente e o servidor é padronizada, permitindo uma maneira uniforme e consistente de acessar e manipular recursos. Isso inclui o uso do protocolo HTTP, com URIs para identificar recursos, e operações de CRUD.
- Sistema em camadas: A arquitetura pode ser composta por várias camadas, com cada camada fornecendo um conjunto específico de funcionalidades. Isso permite escalabilidade, flexibilidade e facilidade de modificação.

Desse modo, uma API REST torna um serviço web “RESTful”. Uma API REST consiste em um assembly de recursos interligados. Este conjunto de recursos é conhecido como recurso da API REST modelo (MASSE, 2011). A aplicação desses requisitos é apropriada para reduzir a complexidade na implementação, uma vez que aqueles que seguem o padrão de uma API

RESTful encontram maior facilidade em compreender outras APIs que adotam o mesmo design. Essa facilidade é atribuída, em grande parte, à presença do requisito de uma interface uniforme. Adicionalmente, há vantagens em termos de escalabilidade, pois o desenvolvimento de cada parte é suficientemente desacoplado das demais. Essa independência facilita o progresso individual de diferentes partes do sistema, contribuindo para a expansão e manutenção eficiente do sistema como um todo.

2.1.3 Autenticação e Autorização

Segundo o Moderno Dicionário da Língua Portuguesa Michaelis, em sua edição de 2024, o termo **AUTENTICAÇÃO**, (2024) apresenta as seguintes definições:

- Ato ou efeito de autenticar, de tornar autêntico.
- Selo, chancela ou abonação dados por notório ou tabelião, pelos quais se reconhece um documento como verdadeiro.
- Atestado de veracidade de algo.

Aliado a isso, (ERLICH; ZVIRAN, 2015) define autenticação como o ato de verificar a identidade do usuário, e aponta três abordagens principais para autenticação de usuários:

- Baseada em conhecimento, algo de conhecimento do usuário. Uma senha, código de acesso ou PIN, por exemplo.
- Baseada em posse, algo de posse do usuário. Um cartão inteligente, cartão de memória ou tokens de acesso, são exemplos.
- Baseada em biometria, diz respeito a características fisiológicas do usuário. Impressões digitais, reconhecimento de íris ou comportamentais, como por exemplo, a digitação do usuário ou dinâmica de toque.

Dessa maneira, a autenticação é caracterizada como o procedimento de confirmação da identidade. Configurando-se como o procedimento pelo qual uma entidade valida sua identidade, com o intuito de atestar a veracidade por meio de elementos específicos, tais como credenciais, exemplificadas por usuário e senha. A fim de, efetuar a confirmação associada à troca de informações que atestem a legitimidade como, por exemplo, um Token.

Com tudo, (**AUTORIZAÇÃO**, 2024), apresenta as seguintes definições:

- Ato ou efeito de autorizar; consentimento, permissão.
- Ordem ou determinação pela qual se autoriza ou se concede poder ou licença.
- Permissão oficial para que um indivíduo execute uma ação ou pratique determinado ato legal.
- Poder conquistado com essa permissão.
- Documento legal que registra por escrito essa permissão.

Portanto, as ações do usuário devem ser consistentes com o que ele está autorizado a fazer. Ações não autorizadas não devem ser permitidas (GHOSH, 2006). Aliado a isso, a autorização, refere-se ao ato ou efeito de conceder consentimento ou permissão específica. Sendo assim, com posse de um token, o usuário solicita autorização a uma entidade específica. A concessão dessa autorização se manifesta por meio da atribuição de permissões específicas vinculadas ao token, permitindo que o detentor do mesmo execute ações previamente autorizadas, de acordo com as definições estabelecidas.

2.1.4 Segurança de sistema

Segurança de sistemas é um campo essencial no domínio da tecnologia da informação, abordando estratégias e práticas com o objetivo de garantir a integridade, confidencialidade e disponibilidade de sistemas computacionais. Este conceito visa mitigar riscos associados a potenciais ameaças, incluindo acessos não autorizados e ataques cibernéticos, que poderiam comprometer a funcionalidade eficaz e a proteção adequada das informações em sistemas computacionais. Dentre os componentes fundamentais estudados e implementados nesse contexto, destacam-se a aplicação de técnicas de criptografia para garantir a confidencialidade dos dados, a adoção de controles de acesso para regulamentar a autorização de usuários, a implementação de sistemas de detecção de intrusões para identificar atividades suspeitas, o gerenciamento proativo de vulnerabilidades para minimizar possíveis falhas de segurança, e a definição de políticas de segurança para estabelecer diretrizes normativas. O conjunto dessas práticas converge para assegurar a confiabilidade dos sistemas de informação em diversos contextos, incluindo ambientes corporativos, servidores online e infraestruturas computacionais.

Com isso, (STALLINGS, 2014) define os três objetivos principais mencionados, como:

- Confidencialidade, esse termo cobre dois conceitos relacionados:
 - Confidencialidade de dados, assegura que informações privadas e confidenciais não estejam disponíveis nem sejam reveladas para indivíduos não autorizados.
 - Privacidade, assegura que os indivíduos controlem ou influenciem quais informações relacionadas a eles podem ser obtidas e armazenadas, da mesma forma que como, por quem e para quem essas informações são passíveis de ser reveladas.
- Integridade, esse termo cobre dois conceitos relacionados:
 - Integridade de dados, assegura que as informações e os programas sejam modificados somente de uma maneira especificada e autorizada.
 - Integridade do sistema, assegura que um sistema execute as suas funcionalidades de forma íntegra, livre de manipulações deliberadas ou inadvertidas do sistema.
- Disponibilidade, assegura que os sistemas operem prontamente e seus serviços não fiquem indisponíveis para usuários autorizados.

A definição estruturada de serviços e mecanismos de segurança assume importância fundamental para orientar a escolha adequada de medidas de segurança para produtos e serviços. Nesse contexto, a arquitetura de segurança OSI oferece uma visão geral de diversos conceitos como ataque à segurança, mecanismos de segurança e serviço de segurança. A análise e

compreensão desses elementos são fundamentais para a formulação de estratégias eficazes de segurança em ambientes tecnológicos cada vez mais complexos e interconectados. Sendo assim, (STALLINGS, 2014) define:

- Ataque à segurança, qualquer ação que comprometa a segurança da informação pertencida a uma organização.
- Mecanismo de segurança, um processo (ou um dispositivo incorporando tal processo) que é projetado para detectar, impedir ou recuperar-se de um ataque à segurança.
- Serviço de segurança, um serviço de processamento ou comunicação que aumenta a segurança dos sistemas de processamento de dados e das transferências de informação de uma organização. Os serviços servem para frustrar ataques à segurança, e utilizam um ou mais mecanismos para isso.

2.1.5 Criptografia

No contexto da segurança da informação, a adoção da criptografia se destaca como uma necessidade essencial com o objetivo de assegurar a integridade e confidencialidade dos dados. A etimologia da palavra criptografia (do grego *kryptós*, que significa oculto ou escondido; e *gráphein*, que significa escrita) é tão antiga quanto o esforço do homem em preservar ou destruir o sigilo de uma mensagem (ANDRADE; LUNARDI; RAMOS, 2021). Dentro dessa perspectiva, a criptografia exerce uma função crucial ao codificar as informações de forma cifrada, tornando-as inacessíveis a qualquer indivíduo não autorizado. Essa metodologia proporciona exclusivamente ao destinatário final a capacidade de decifrar e compreender a mensagem verdadeira, conferindo, dessa maneira, uma camada essencial de proteção à comunicação e ao armazenamento de dados sensíveis.

A confidencialidade, como serviço essencial de segurança, assegura que informações não sejam acessíveis ou divulgadas a entidades não autorizadas. As cifras, algoritmos de cifragem, são cruciais para implementar esse serviço. Dois requisitos fundamentais são a utilização de algoritmos robustos e a manutenção de chaves seguras pelo emissor e receptor. Apesar de não ser necessário manter em segredo o algoritmo, o desafio principal continua sendo preservar a confidencialidade da chave. Diante disso, segue os conceitos necessários de uma encriptação (STALLINGS, 2014):

- Texto claro, essa é a mensagem ou dados originais, inteligíveis, que servem como entrada do algoritmo de encriptação.
- Algoritmo de encriptação, realiza várias transformações no texto claro.
- Chave secreta, no contexto simétrico, é uma entrada para o algoritmo de encriptação. A chave é um valor independente do texto claro e do algoritmo. O algoritmo produzirá uma saída diferente, dependendo da chave usada no momento. As substituições e transformações exatas realizadas pelo algoritmo dependem da chave. Aliado a isso, no contexto assimétrico tem a chave pública e privada, esse é um par de chaves que foi selecionado de modo que, se uma for usada para encriptação, a outra é usada para decifração. As transformações exatas realizadas pelo algoritmo dependem da chave pública ou privada que é fornecida como entrada.

- Texto cifrado, essa é a mensagem embaralhada produzida como saída. Ela depende do texto claro e da chave. Para determinada mensagem, duas chaves diferentes produzirão dois textos cifrados diferentes.
- Algoritmo de decifração, aceita o texto cifrado e a chave correspondente e produz o texto claro original.

Com isso, em criptografia existem três principais divisões (STALLINGS, 2014):

- Criptografia de chave secreta ou simétrica;
- Criptografia de chave pública ou assimétrica;
- Funções de hash.

2.1.5.1 Chave simétrica

Levando em conta os conceitos necessários de uma encriptação, mencionados por (STALLINGS, 2014), segue, na figura 2.3 um exemplo de fluxo com a chave simétrica. Neste, constitui o modelo tradicional de criptografia, no qual a chave, responsável pelo acesso à mensagem oculta compartilhada entre duas partes, é idêntica (simétrica) para ambas e requer sigilo absoluto. Geralmente, essa chave é representada por uma senha, servindo tanto ao remetente para cifrar a mensagem em uma extremidade quanto ao destinatário para decifrá-la na outra. Existem diversos algoritmos utilizado nesse processo, AES e DES, são alguns dos principais, por exemplo. Com tudo, de acordo com (GHOSH, 2006) é necessário que as duas partes cheguem a um acordo sobre uma chave secreta antes da comunicação começar. Existem numerosos exemplos de criptossistemas de chave secreta. Nós os classificamos em dois tipos diferentes:

- Cifras de bloco, os dados são primeiro divididos em blocos de tamanho fixo antes da criptografia. Isto é aplicável quando os dados a serem enviados estão disponíveis em sua totalidade antes do início da comunicação.
- Cifras de fluxo, são usadas para transmitir dados em tempo real que são gerados espontaneamente, por exemplo, dados de voz.

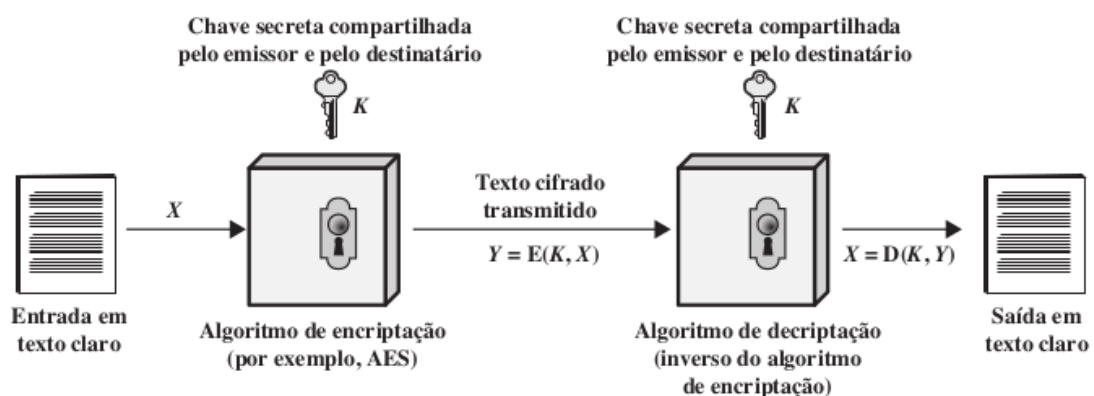


Figura 2.3 – criptografia simétrica

Stallings (2014)

2.1.5.2 Chave assimétrica

A chave assimétrica envolve a utilização de pares de chaves distintas: uma pública e outra privada. Todos os participantes do sistema possuem acesso às chaves públicas, enquanto as chaves privadas são geradas localmente por cada entidade, eliminando a necessidade de distribuição. Sob essa abordagem, a segurança da comunicação reside na proteção e confidencialidade da chave privada de cada usuário. Enquanto a chave privada permanecer resguardada, a comunicação é mantida segura. Notavelmente, é possível que um sistema modifique sua chave privada a qualquer momento, substituindo-a e divulgando uma nova chave pública correspondente, garantindo a continuidade da segurança. Essa flexibilidade permite uma gestão dinâmica das chaves, fortalecendo a robustez do sistema de criptografia assimétrica. Para o processo de encriptar e descriptografar a chave assimétrica existem alguns algoritmos, o RSA é um dos principais, por exemplo. Sendo assim, segue na figura 2.4 um exemplo de fluxo com a chave assimétrica descrito por (STALLINGS, 2014):

1. Cada usuário gera um par de chaves a ser usado para a encriptação e a decriptação das mensagens.
2. Cada usuário coloca uma das duas chaves em um registrador público ou em outro arquivo acessível. Essa é a chave pública. A chave acompanhante permanece privada. Como a Figura 9.1a sugere, cada usuário mantém uma coleção de chaves públicas obtidas de outros.
3. Se Bob deseja enviar uma mensagem confidencial para Alice, ele a encripta usando a chave pública de Alice.
4. Quando Alice recebe a mensagem, ela a decripta usando sua chave privada. Nenhum outro destinatário pode decriptar a mensagem, pois somente Alice conhece a chave privada de Alice.

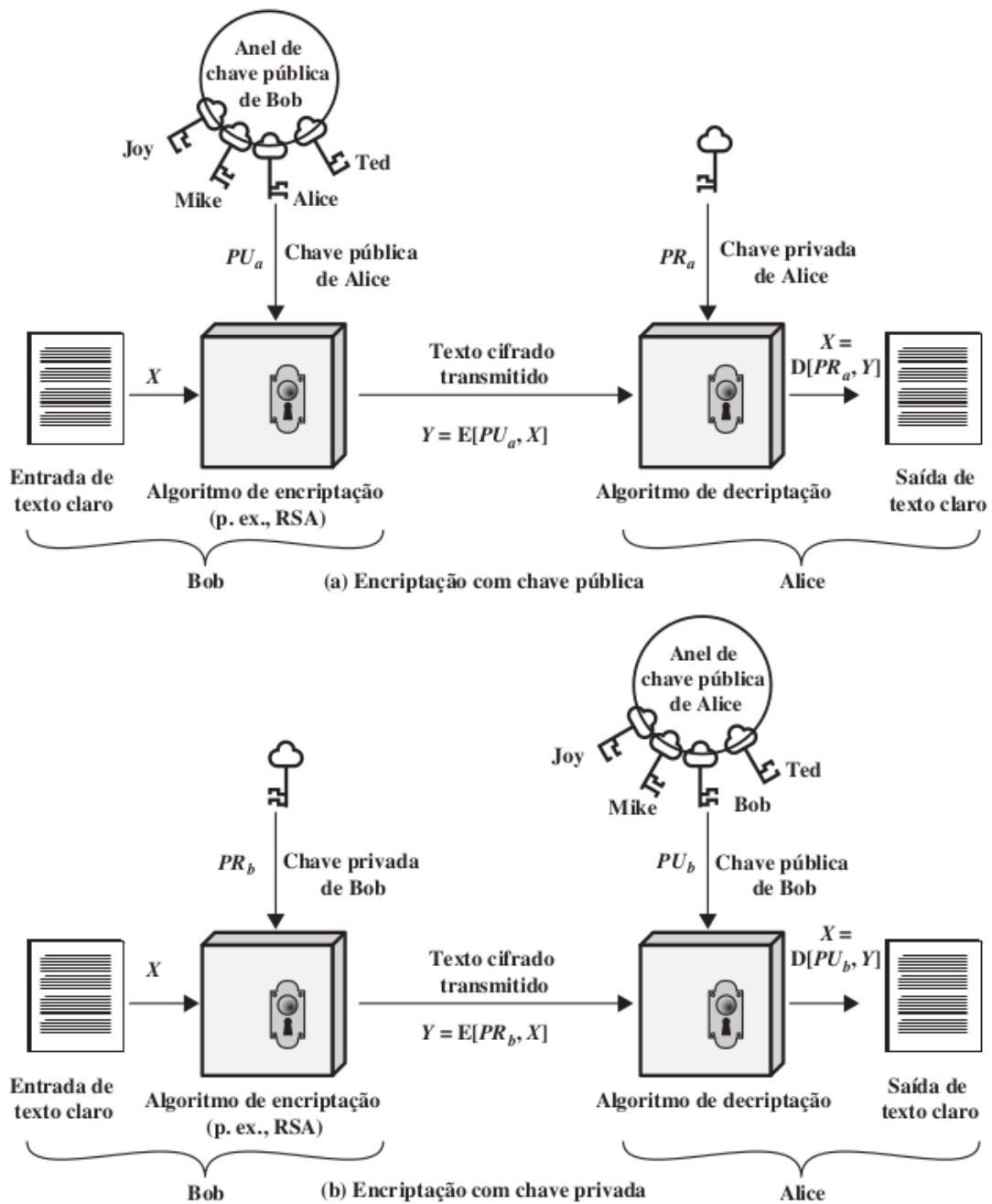


Figura 2.4 – criptografia assimétrica

Stallings (2014)

2.1.5.3 Função hash

Uma função de hash de acordo com (STALLINGS, 2014) aceita uma mensagem de tamanho variável M como entrada e produz um valor de hash de tamanho fixo $h = H(M)$. Uma “boa” função de hash tem a propriedade de que os resultados da aplicação da função a um grande conjunto de entradas produzirá saídas que são distribuídas por igual e aparentemente de modo aleatório. Em termos gerais, o objeto principal de uma função de hash é a integridade de dados. Uma mudança em qualquer bit ou bits em M resulta, com alta probabilidade, em uma mudança no código de hash. Com isso, a função hash, como no exemplo de fluxo da figura ??, opera

mediante o preenchimento da entrada até alcançar um múltiplo inteiro de um tamanho fixo, como 1024 bits. Esse preenchimento incorpora o valor do tamanho da mensagem original em bits. O campo de tamanho atua como uma medida de segurança, visando aumentar a complexidade para um invasor na produção de uma mensagem alternativa com o mesmo valor de hash. Esse procedimento contribui para a integridade e autenticidade da mensagem ao tornar desafiador o comprometimento da relação entre a mensagem original e seu valor de hash. A função hash criptográfica mais amplamente utilizada é a SHA.

2.1.6 OWASP

O Open Worldwide Application Security Project (OWASP) é uma comunidade aberta que visa facilitar o desenvolvimento, aquisição e manutenção de aplicativos e APIs seguros por organizações. A OWASP oferece gratuitamente ferramentas, padrões, livros, apresentações, vídeos, folhas de dicas, controles e bibliotecas de segurança, bem como mantém capítulos locais em todo o mundo. A organização adota uma abordagem aberta e imparcial, não sendo afiliada a empresas de tecnologia. A ênfase da OWASP reside na segurança de aplicações, considerando-a um desafio que envolve pessoas, processos e tecnologia. A organização é gerida pela Fundação OWASP, uma entidade sem fins lucrativos, e é composta principalmente por voluntários que contribuem para pesquisas inovadoras em segurança por meio de subsídios e infraestrutura. O OWASP defende a colaboração transparente e aberta na produção de diversos materiais para aprimorar a segurança de aplicações (FOUNDATION, 2023).

O (FOUNDATION, 2023), aborda as dez principais vulnerabilidades de segurança de sistemas associadas a APIs, do ano de 2023, englobando diversas fragilidades que podem comprometer a integridade, confidencialidade e disponibilidade dos dados. Essas vulnerabilidades incluem não apenas falhas em mecanismos de autenticação e autorização, mas também exposição inadequada de dados sensíveis, manipulação inadequada de entradas, falta de controle de acesso, execução inadequada de lógica de negócios, entre outras. Aliado a isso, oferece uma análise abrangente desses pontos críticos, visando fornecer orientações eficazes para fortalecer a segurança de sistemas que fazem uso de APIs. Segue as dez principais vulnerabilidades, de acordo com (IDRIS; SYARIF; WINARNO, 2022):

- Autorização em nível de objeto quebrado, refere-se a cenários nos quais os endpoints de API podem ser vulneráveis à manipulação do identificador (ID) de um objeto enviado na requisição. Isso possibilita que atacantes obtenham acesso ou realizem modificações em objetos aos quais, em princípio, não deveriam ter permissão de acesso.
- Autenticação quebrada, vulnerabilidades se manifestam quando os mecanismos de autenticação de uma API são inadequados ou implementados de maneira inadequada, tornando-os suscetíveis a ataques e representando alvos acessíveis para potenciais invasores.
- Autorização de nível de propriedade de objeto quebrado, destaca-se pela divulgação inadequada de características de um objeto por meio dos endpoints de API, especialmente em APIs REST e GraphQL.
- Consumo irrestrito de recursos, aparece quando uma API não estabelece restrições apropriadas no uso de recursos, como largura de banda, CPU, memória e armazenamento.

- Autorização de nível de função quebrada, refere-se a cenários nos quais uma API possibilita que usuários não autorizados acessem funções ou recursos específicos.
- Acesso irrestrito a fluxos de negócios confidenciais, Diz respeito à revelação de fluxos de negócios essenciais sem a implementação de restrições de acesso apropriadas.
- Falsificação de solicitação do lado do servidor, torna-se evidente quando um endpoint da API busca um recurso remoto usando URLs fornecidas pelo usuário, sem realizar a devida validação.
- Configuração incorreta de segurança, ocorre quando uma API opera com configurações de segurança inadequadas, expondo os dados e sistemas a riscos. Configurações deficientes, como práticas de autenticação frágeis e falta de criptografia, representam potenciais pontos de exploração para ameaças de segurança. A correção rigorosa dessas configurações é essencial para mitigar esses riscos.
- Gerenciamento de estoque inadequado, decorre da falta de controle e documentação adequada sobre as versões e endpoints de uma API, comprometendo a transparência e a segurança, além de dificultar a implementação eficaz por parte dos desenvolvedores.
- Consumo inseguro de APIs, ocorre quando interage com outras APIs sem criptografia, não valida e limpa dados adequadamente, segue redirecionamentos sem restrições, não limita recursos disponíveis para processar respostas e não implementa timeouts em interações com serviços externos.

(IDRIS; SYARIF; WINARNO, 2022) destaca a plataforma DVWA, uma aplicação web intencionalmente vulnerável, criada para fins educacionais e de treinamento em segurança cibernética. Também, integração de testes contínuos de segurança em CI/CD utilizando SAST, uma metodologia de teste é empregada para analisar o código-fonte em busca de vulnerabilidades de segurança. Aliado a isso, menciona também JMeter e Selênio, ferramentas para realização de testes automatizados. Por fim, OWASP ZAP, uma ferramenta de testes de penetração dinâmicos.

2.1.7 JWT

O JSON Web Token (JWT) é definido por (JONES, 2024), como sendo um padrão aberto ((JONES; BRADLEY; SAKIMURA, 2015)) que define uma maneira compacta e independente de transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente. Os JWTs podem ser assinados usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA. Aliado a isso, é possível ser utilizado com chave simétrica e assimétrica.

Com isso, de acordo com (MONTANHEIRO; CARVALHO; RODRIGUES, 2017), Várias bibliotecas que implementam o padrão JWT estão disponíveis para as mais variadas linguagens de programação, dentre elas: .NET, Python, NodeJs, Java, JavaScript, Perl, Ruby e PHP. As bibliotecas podem ser usadas tanto para autenticação de usuários, que é o cenário mais comum, sendo que a cada solicitação de uma rota, serviço ou recurso o usuário envia junto o token, permitindo seu acesso ou não com base nesse token, quanto para transmitir informações de forma segura, uma vez que com base em sua assinatura criptografada é possível verificar se o conteúdo não foi adulterado.

(JONES; BRADLEY; SAKIMURA, 2015) divide a estrutura do JWT em três partes, como segue na figura 2.5:

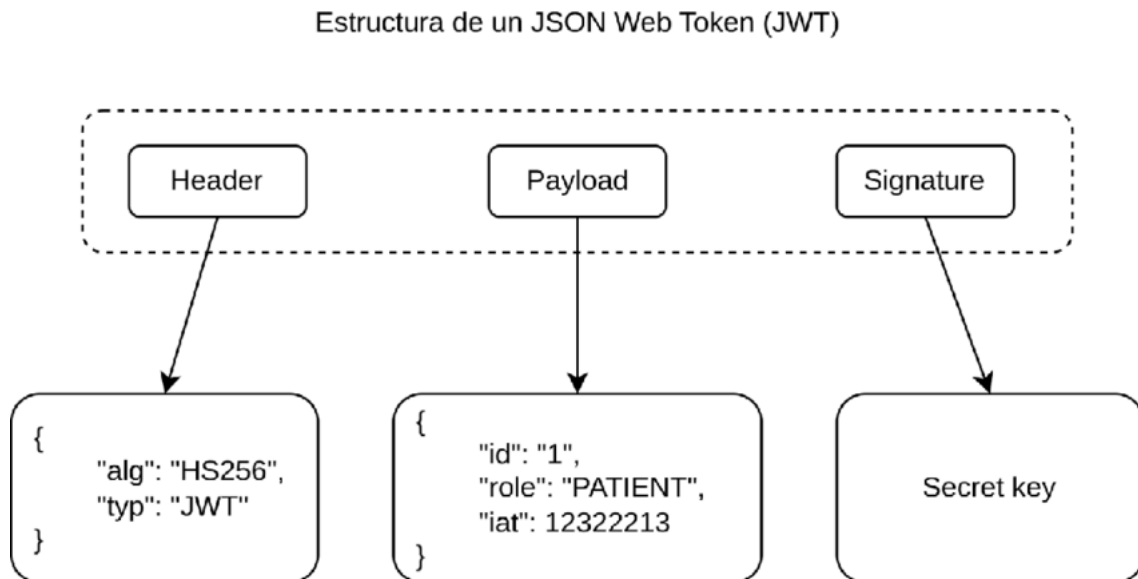


Figura 2.5 – Estrutura jwt

Gómez, Cárdenas e Ruiz (2023)

- Cabeçalho, normalmente consiste em duas partes: o tipo de token, que é JWT, e o algoritmo de assinatura usado, como HMAC, SHA256 ou RSA.
- Carga útil, contém declarações sobre uma entidade (normalmente, o usuário) e dados adicionais.
- Assinatura, é usada para verificar se a mensagem não foi alterada ao longo do caminho e, no caso de tokens assinados com chave privada, também pode verificar se o remetente do JWT é quem diz ser.

Sendo assim, cada parte, da estrutura mencionada, é codificada para formar uma sequência de caracteres que representa o token ¹. A codificação utilizada, frequentemente, é o Base64, um esquema de codificação binária para texto ASCII. Este método de codificação converte dados binários em uma representação textual usando um conjunto de caracteres específicos.

O processo inicia-se com a criação do cabeçalho, contendo informações sobre o tipo do token e o algoritmo de assinatura. Em seguida, a carga útil é adicionada, incluindo as informações desejadas. Ambas as partes são convertidas para Base64 individualmente. O token resultante, segue na figura 2.6, é então formado pela concatenação do cabeçalho codificado em Base64, um ponto ('.'), a carga útil codificada em Base64 e outro ponto ('.'), seguido pela assinatura. A assinatura é gerada utilizando a chave secreta do emissor, garantindo a integridade do token. Quando o token é recebido por um receptor, este pode verificar a autenticidade da assinatura ao recalculá-la com a chave correspondente. Portanto, o fluxo da utilização do JWT envolve a

¹ "Token", em criptografia, refere-se frequentemente a um dispositivo de segurança físico ou virtual utilizado para autenticação de usuários.

codificação de um cabeçalho e uma carga útil em Base64, a concatenação dessas partes com uma assinatura, a transmissão do token resultante e, finalmente, a verificação da autenticidade no receptor por meio da validação da assinatura.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG91IiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezDI1AVTmud2fU4
```

Figura 2.6 – Exemplo token secret key jwt

Jones (2024)

2.1.8 OAuth 2.0

O Open Authorization (OAuth 2.0) é um protocolo de autorização, seguro e aberto, amplamente utilizado na WEB, e é concebido para permitir que aplicativos acessem recursos protegidos em nome de usuários. O protocolo estabelece um processo flexível de autorização, onde usuários concedem permissões a aplicativos para acessar recursos em um servidor específico. Sua modularidade suporta diversos fluxos de concessão, adaptando-se a cenários como autenticação de usuário único, acesso de cliente para servidor e autorização por terceiros. Além disso, a estrutura de autorização do OAuth 2.0 permite que aplicativos de terceiros obtenham acesso limitado a um serviço HTTP, agindo em representação de um detentor de recursos, mediante coordenação de uma interação de aprovação. Notavelmente, esta especificação substitui e torna obsoleto o protocolo OAuth 1.0, conforme indicado na (HAMMER-LAHAV, 2010), (HARDT, 2012).

Sendo assim, em seu contexto o (HARDT, 2012) define quatro funções no protocolo, segue na figura 2.7:

- Proprietário do recurso, uma entidade capaz de conceder acesso a um recurso protegido. Quando o proprietário do recurso é uma pessoa, ele é chamado de usuário final.
- Servidor de recursos, o servidor que hospeda os recursos protegidos, capaz de aceitar e responder a solicitações de recursos protegidos usando tokens de acesso.
- Cliente², um aplicativo que faz solicitações de recursos protegidos em nome do proprietário do recurso e com sua autorização.
- Servidor de autorização, o servidor emitindo tokens de acesso para o cliente após sucesso autenticar o proprietário do recurso e obter autorização.

² “Cliente”, o termo não implica quaisquer características de implementação específicas (por exemplo, se o aplicativo é executado em um servidor, desktop ou outro dispositivos).

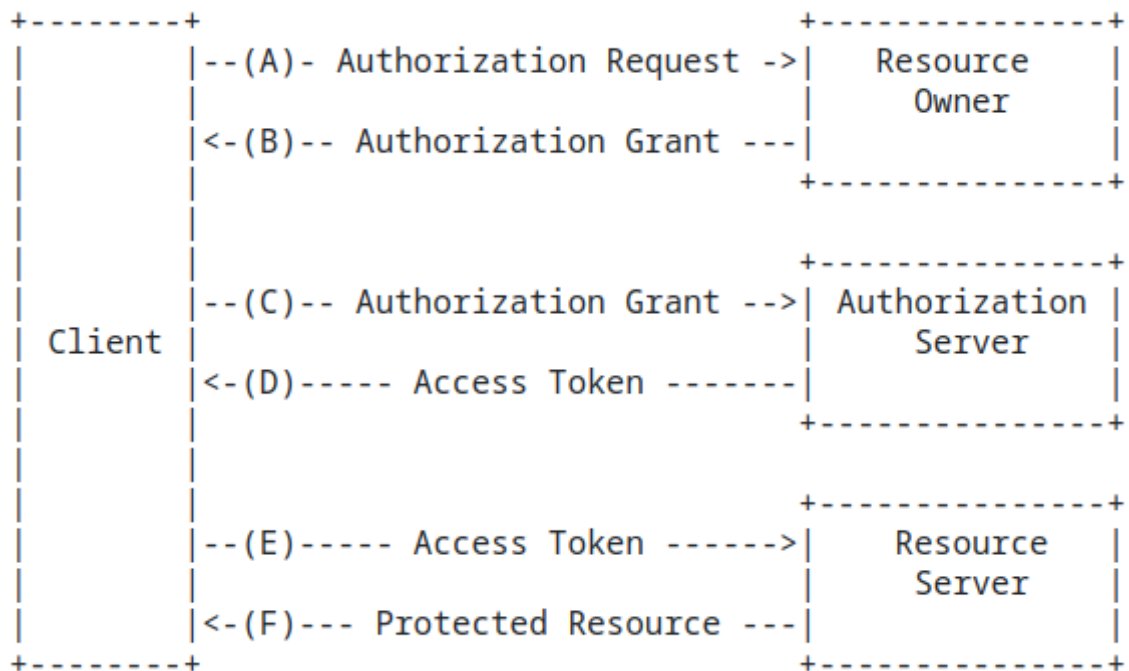


Figura 2.7 – Fluxo OAuth 2.0

Hardt (2012)

Portanto, (HARDT, 2012) propõe o seguinte fluxo no emprego do protocolo, como exemplificado na figura 2.7:

- A. O cliente solicita autorização do proprietário do recurso. O a solicitação de autorização pode ser feita diretamente ao proprietário do recurso (conforme mostrado), ou de preferência indiretamente através da autorização servidor como intermediário.
- B. cliente recebe uma concessão de autorização, que é uma credencial que representa a autorização do proprietário do recurso. O o tipo de concessão de autorização depende do método usado pelo cliente para solicitar autorização e os tipos suportados pelo servidor de autorização.
- C. O cliente solicita um token de acesso autenticando-se com o servidor de autorização e apresentação da concessão de autorização.
- D. O servidor de autorização autentica o cliente e valida a concessão de autorização e, se válida, emite um token de acesso.
- E. O cliente solicita o recurso protegido do recurso servidor e autentica apresentando o token de acesso.
- F. O servidor de recursos valida o token de acesso e, se válido, atende ao pedido.

Ressalta-se que a abordagem preferencial para que o cliente adquira uma concessão de autorização do detentor de recursos, conforme delineado nas etapas (A) e (B), consiste em empregar o servidor de autorização como intermediário. É relevante observar que um único servidor de autorização tem a capacidade de emitir tokens de acesso que são reconhecidos por diversos servidores de recursos.

2.1.9 SSO

A independência entre as informações do usuário e as políticas de segurança complica e compromete a gestão do usuário. Devido à heterogeneidade dos mecanismos de segurança, a transmissão de resultados de autenticação para concluir uma tarefa cooperativa é inviável. Portanto, a integração de informações do usuário, a padronização do processo de autorização e o estabelecimento de um sistema de Entrada Única (LI et al., 2004). Sendo assim, o Single Sign-On (SSO)³, é um conceito e uma técnica de autenticação que viabiliza a um usuário o acesso a diversos sistemas ou aplicativos mediante uma única identificação e autenticação.

Em sistemas baseado em token, figura 2.8, um usuário envia as credenciais para a autoridade de autenticação baseada em token, em quais as credenciais foram verificadas com seu banco de dados de credenciais. Se as credenciais do usuário corresponderem, então o usuário é retornado com um token. Quando o usuário deseja acessar um servidor de aplicação governado pela segunda autoridade de autenticação, o mesmo token é entregue para obter um ticket de acesso ao aplicativo servidor. O sucesso deste processo depende da confiança que as autoridades de autenticação têm entre si (RADHA; REDDY, 2012).

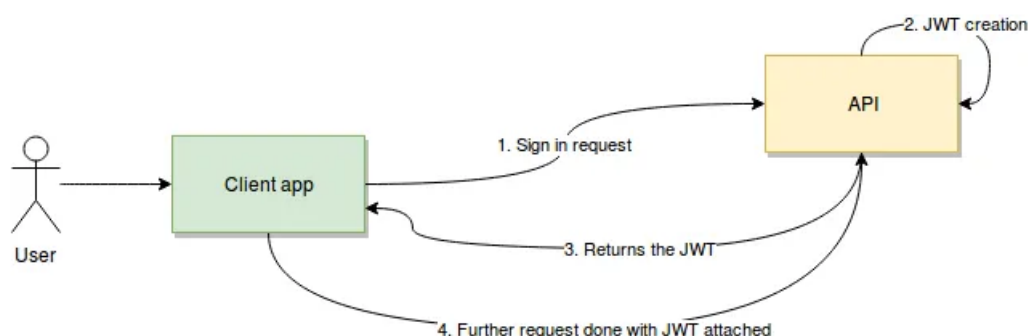


Figura 2.8 – Exemplo SSO

Adelaide (2023)

2.1.10 Token de atualização

Conforme descrito (HARDT, 2012) tokens de atualização são credenciais usadas para obter novos tokens de acesso quando os tokens atuais expiram ou se tornam inválidos. O servidor de autorização os emite, permitindo que o cliente continue usando os recursos sem exigir reautenticação total. Os tokens de atualização exigem que o cliente se autentique e forneça o token para receber um novo token de acesso ao usá-los. Esses tokens não são usados como servidores de recursos, eles são usados apenas com servidores de autorização. As etapas fundamentais envolvem solicitar primeiro um token de acesso, usá-lo para acessar recursos e, em seguida, usar o token de atualização para obter um novo quando o atual expirar. As seguintes precauções de segurança citadas por (LODDERSTEDT et al., 2019) são recomendadas para proteger e utilizar tokens de atualização de forma adequada:

- Transmitir e armazenar tokens de atualização com segurança em conexões protegidas por TLS é essencial para manter a confidencialidade.

³ "Single Sign-on", o termo traduzido para o português é entendido como "Entrada Única".

- Vinculação de cliente (clientId): Para garantir que apenas um determinado cliente possa utilizar tokens de atualização, o servidor de autorização deve vinculá-los a esse cliente.
- Autenticação do cliente: Sempre que possível, o token de atualização deve ser utilizado para autenticar o cliente durante o procedimento de renovação do token de acesso.
- Geração de token e prevenção de modificação: os invasores não conseguem gerar, alterar ou adivinhar tokens de atualização.
- Rotação de token: toda vez que um token de atualização é usado, ele precisa ser substituído por um novo, tornando o antigo inválido. Isso torna mais fácil detectar e impedir tentativas de uso de tokens hackeados.
- Restrição do remetente: use técnicas como vinculação de token ou autenticação mTLS (TLS mútuo) para vincular criptograficamente o token de atualização a uma instância de cliente específica.
- Revogação de token: Os tokens de atualização devem ser revogados automaticamente em caso de eventos de segurança, como alteração de senha ou logout.
- Expiração de tokens de inatividade: para evitar abusos após longos períodos de inatividade, os tokens de atualização devem expirar.

Ao garantir que os tokens de atualização sejam usados de forma segura, essas precauções reduzem a possibilidade de ataques.

2.1.11 Fuzzing

O Fuzzing, é uma ideia proposta por (MILLER; FREDRIKSEN; SO, 1990) consiste em submeter uma aplicação a uma variedade de entradas, incluindo algumas que são atípicas e aleatórias, com o objetivo de monitorar seu comportamento. Com isso, no domínio da segurança de sistemas, denota uma técnica automatizada de teste de segurança que incorpora a integração de API REST com cobertura de testes. Essa abordagem é empregada com o propósito de detectar vulnerabilidades e comportamentos imprevistos. Aliado a isso, (LIANG et al., 2018) explica que a ideia chave por trás fuzzing é gerar e alimentar o programa alvo com bastante casos de teste que podem desencadear erros de software. A maioria A parte criativa do fuzzing é a maneira de gerar casos de teste adequados, e os métodos populares atuais incluem estratégias guiadas por cobertura, algoritmo genético, execução simbólica, análise de contaminação, etc. Com base nesses métodos, uma técnica moderna de difusão é muito inteligente para revelar bugs ocultos.

2.1.12 Teste de estresse

Conforme (MYERS; SANDLER; BADGETT, 2012), teste de estresse é uma metodologia de teste que envolve submeter software a cargas abruptas e intensas, que replicam altos níveis de atividade em um breve período de tempo. O objetivo é avaliar a resposta do sistema em cenários que excedem as suas capacidades previstas. Os testes de estresse examinam como o programa responde a uma breve sobrecarga, em contraste com os testes de volume, que avaliam o software usando grandes volumes de dados ao longo do tempo.

Sistemas que funcionam sob condições variáveis, como programas interativos, em tempo real ou de controle de processos, são os principais alvos dos testes de estresse. Ao simular cenários como um elevado número de utilizadores simultâneos num website ou controlo de tráfego aéreo, visa detectar potenciais falhas ou comportamentos inesperados quando o sistema é levado ao seu limite ou além com mais aviões do que o previsto.

Esses testes ainda são úteis porque auxiliam na detecção de erros que podem surgir em cenários menos extremos do mundo real, embora tenham a capacidade de simular condições improváveis que o sistema raramente encontrará.

2.2 Trabalhos Relacionados

Neste capítulo, é apresentado os trabalhos correlatos relacionados ao estado da arte dos principais mecanismos de autenticação e autorização em micro serviços, sistemas distribuídos. É destacado a utilização tokens na garantia da integridade e confidencialidade das transmissões de dados e aspectos de vulnerabilidade e segurança associadas.

2.2.1 Uma análise geral da autenticação Mecanismo em software baseado em microsserviços Arquitetura: um documento de discussão

(JACK et al., 2023) em sua Revisão sistemática da literatura realiza uma pesquisa bibliográfica em várias bases de dados, incluindo IEEE Xplore, ACM Digital Library, ScienceDirect e SpringerLink, usando palavras-chave relacionadas ao tema de autenticação em arquitetura de software baseada em microservices. Eles selecionaram artigos relevantes e realizaram uma análise crítica dos mesmos, identificando os principais desafios e soluções relacionados à autenticação em microservices. Discute o mecanismo de autenticação e autorização em arquitetura de software baseada em microservices fornecendo uma análise geral dos desafios e soluções destacando a necessidade de comunicação segura entre serviços interconectados. Também, fornece uma visão geral e pontos fortes das abordagens de autenticação e autorização em arquitetura de microservices, no ambiente distribuído, tais como: API gateway, autenticação baseada em token e mTLS. Por fim, discute as vantagens e desvantagens de cada método.

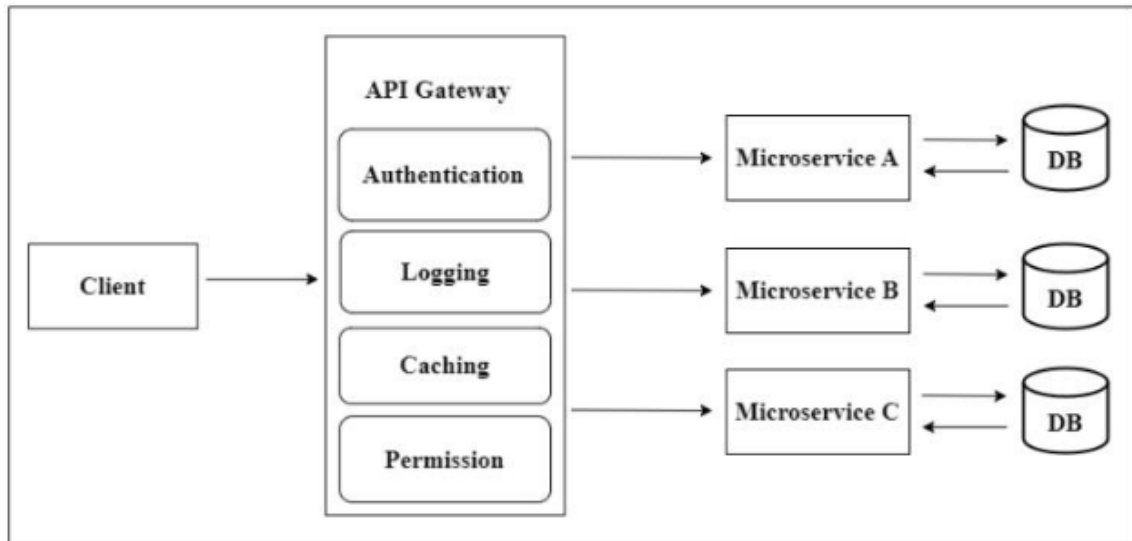


Figura 2.9 – Arquitetura baseada em API Gateway

Jack et al. (2023)

O API Gateway desempenha um papel centralizado ao lidar com todas as solicitações e oferece uma entrada unificada para clientes que interagem com microsserviços. Além da autenticação, fornece recursos como gerenciamento de tráfego, balanceamento de carga, armazenamento em cache e concessão de permissões. A autenticação no API Gateway assegura a segurança na comunicação entre clientes e microsserviços, utilizando uma lógica centralizada para identificar solicitações de usuários.

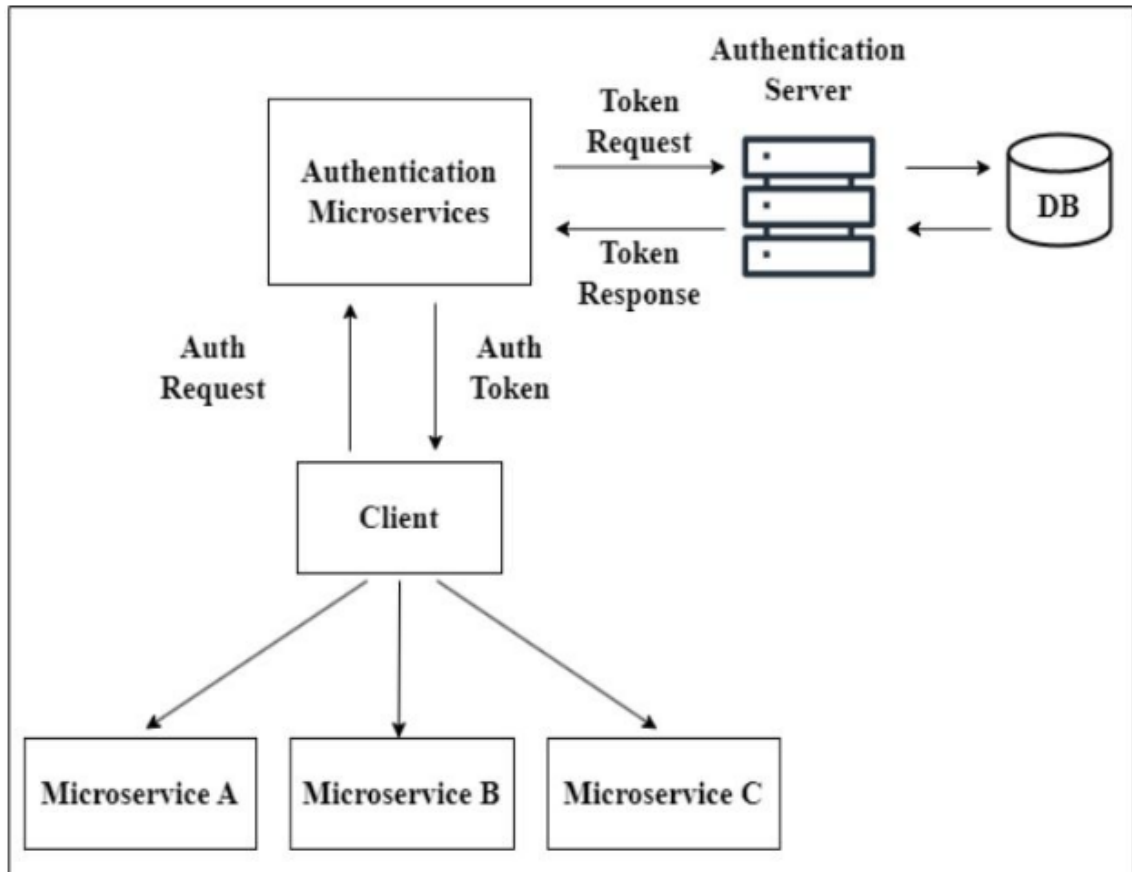


Figura 2.10 – Arquitetura baseada em Token

Jack et al. (2023)

A autenticação baseada em token gera tokens exclusivos para autenticar clientes e microsserviços após verificar a identidade do cliente pelo servidor de autenticação. O cliente inclui o token no cabeçalho de suas transações, e o microsserviço valida a assinatura e as informações contidas, como ID do usuário, endereço de e-mail, funções e permissões. Essa abordagem oferece segurança com tokens de vida útil curta, assinados e criptografados para evitar interceptações. Além disso, é flexível, integrando-se a provedores externos e aceitando vários tipos de tokens, como JWTs e tokens de acesso OAuth.

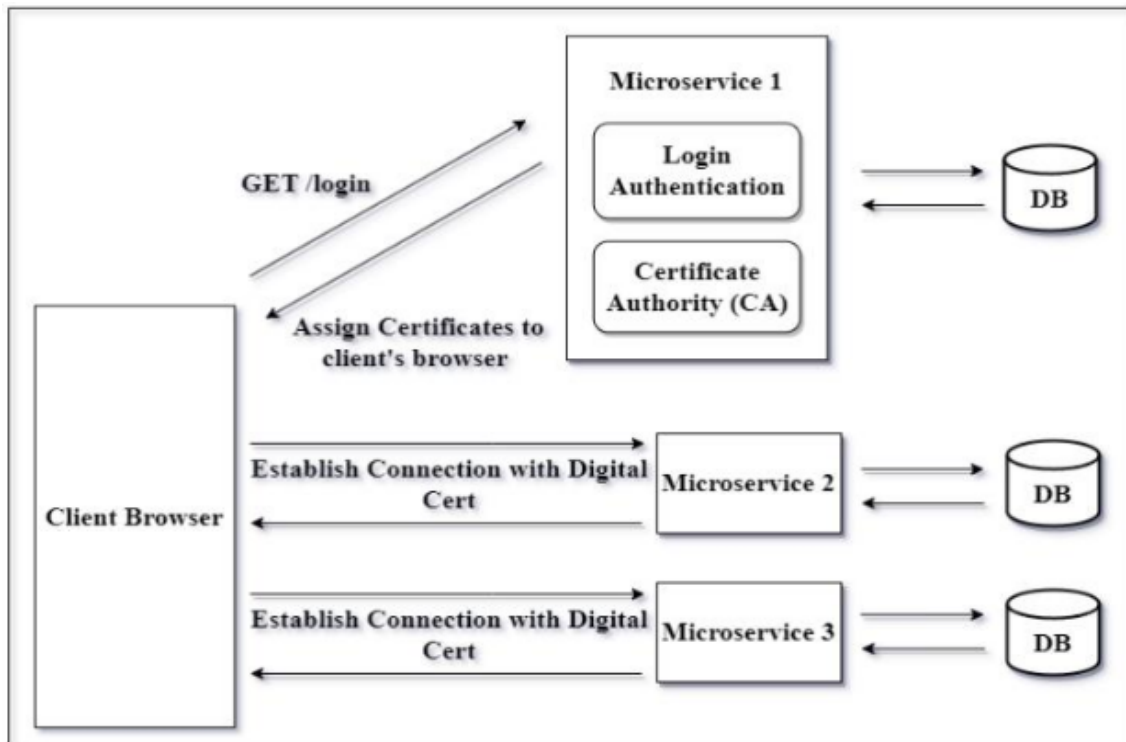


Figura 2.11 – Arquitetura baseada em mTLS

Jack et al. (2023)

O mTLS (Mutual Transport Layer Security) é um método de autenticação que garante a comunicação segura entre microsserviços em sistemas distribuídos, utilizando o protocolo TLS. Cada microsserviço adquire um certificado digital, que inclui uma chave assimétrica composta por uma chave pública e uma chave privada. Essa chave desempenha um papel vital na autenticação mTLS, onde o cliente solicita ao servidor de autenticação para obter um certificado. A chave pública valida assinaturas digitais, enquanto a chave privada é mantida confidencial.

Ela é a autenticação baseada em certificado, oferecendo uma abordagem segura e descentralizada. Essa metodologia proporciona criptografia robusta de ponta a ponta, assegurando a integridade e confidencialidade das mensagens trocadas entre microsserviços, sem depender de um sistema centralizado para autenticação.

Mecanismo de Autenticação	Forças	Limitações
Gateway de API	<ol style="list-style-type: none"> 1. Autenticação e autorização centralizadas 2. Fornece monitoramento e registro 3. Implementações de recursos de segurança adicionais 	<ol style="list-style-type: none"> 1. Despesas gerais de desempenho 2. Preocupações de segurança 3. Ponto único de falha 4. Reduza a flexibilidade
Segurança da camada de transporte mútuo (mTLS)	<ol style="list-style-type: none"> 1. Mecanismo de autenticação descentralizada 2. Fornece autenticação criptografada forte de ponta a ponta 3. Suporta integridade e confidencialidade de mensagens 	<ol style="list-style-type: none"> 1. Impacto no desempenho 2. Complexidade na implementação e configuração 3. Manutenção de certificados e chaves
Autenticação baseada em token	<ol style="list-style-type: none"> 1. Flexibilidade nas integrações 2. Melhoria na experiência de autenticação 3. Configuração de segurança em tokens 	<ol style="list-style-type: none"> 1. Ponto Único de Falha 2. Despesas gerais de desempenho 3. Considerações de segurança

Figura 2.12 – Tabela comparativa entre os métodos

Jack et al. (2023)

O texto destaca um estudo sobre mecanismos de autenticação em arquiteturas de microserviços. A figura 2.12 resume os pontos fortes e limitações dos mecanismos discutidos, focando em autenticação centralizada, autenticação baseada em token e mTLS (Mutual Transport Layer Security). O estudo destaca a possível similaridade de limitações entre autenticação centralizada e baseada em token, enquanto o mTLS se destaca por não ter um ponto único de falha. Conclui-se com a sugestão de que mecanismos de autenticação descentralizados, como o mTLS, podem ser mais adequados para arquiteturas de microserviços. O texto destaca a necessidade de pesquisas futuras para avaliar a segurança do mTLS em comparação com o gateway de API e a autenticação baseada em token.

2.2.2 Aprimorando a segurança de microserviços com acesso baseado em token Método de controle

(VENČKAUSKAS et al., 2023) destaca a importância dos microserviços, que são serviços independentes colaborativos para realizar funções específicas em uma aplicação. A característica única desses microserviços permite alterações em um serviço sem afetar os demais, oferecendo flexibilidade na gestão de aplicações distribuídas. No entanto, essa abordagem traz consigo desafios de segurança não presentes em aplicações monolíticas tradicionais.

Diante desse contexto, o autor propõe um método de controle de acesso focado em fortalecer a segurança dos microserviços. Esse método foi submetido a testes experimentais e validado em comparação com arquiteturas, tanto centralizadas quanto descentralizadas, de microserviços.

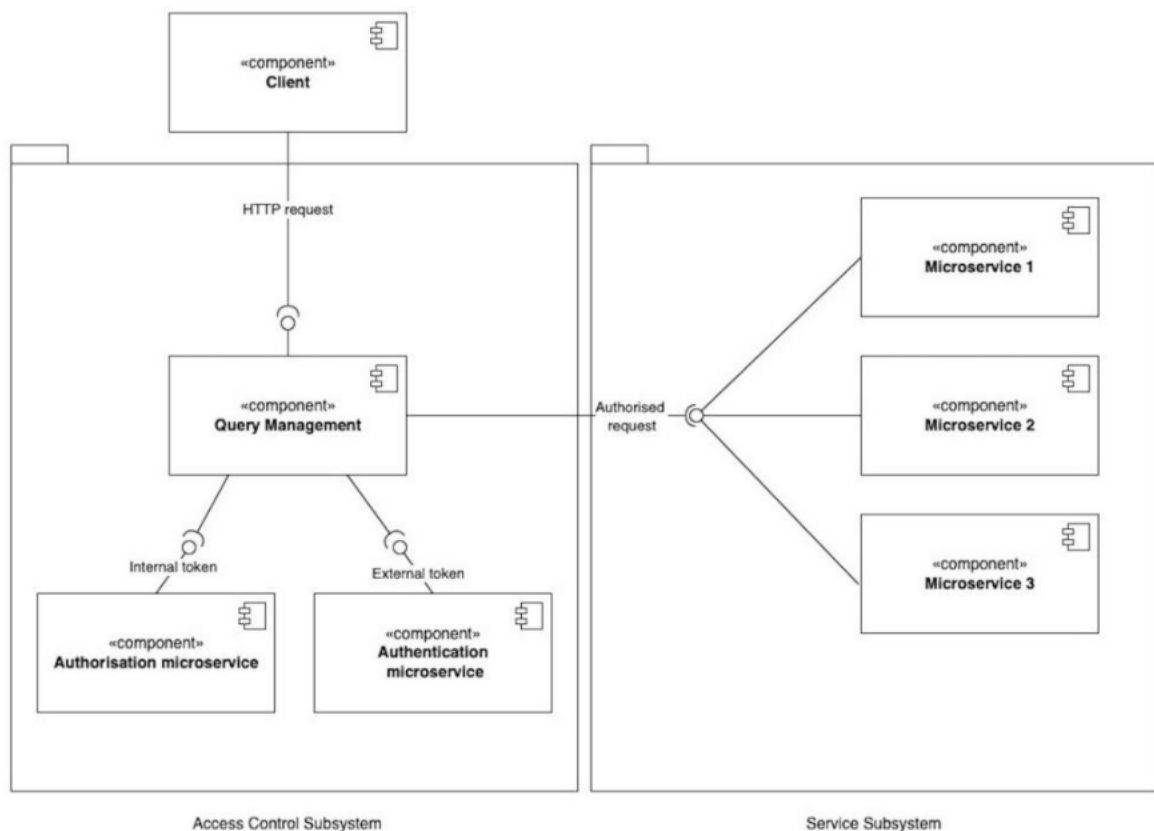


Figura 2.13 – Modelo centralizado

Venčkauskas et al. (2023)

No modelo centralizado, figura 2.13, além do controle e autoridade de tomada de decisões concentrados em uma única entidade ou servidor central, é importante destacar que a autenticação e autorização também são realizadas exclusivamente por esse ponto central. A chave de autenticação é mantida unicamente nesse servidor, o que significa que todo o processo de verificação de identidade e permissões é centralizado nessa entidade. Essa característica contribui para a simplicidade do gerenciamento, mas também intensifica a dependência e a vulnerabilidade do sistema em relação a esse ponto central, tornando-o suscetível a gargalos.

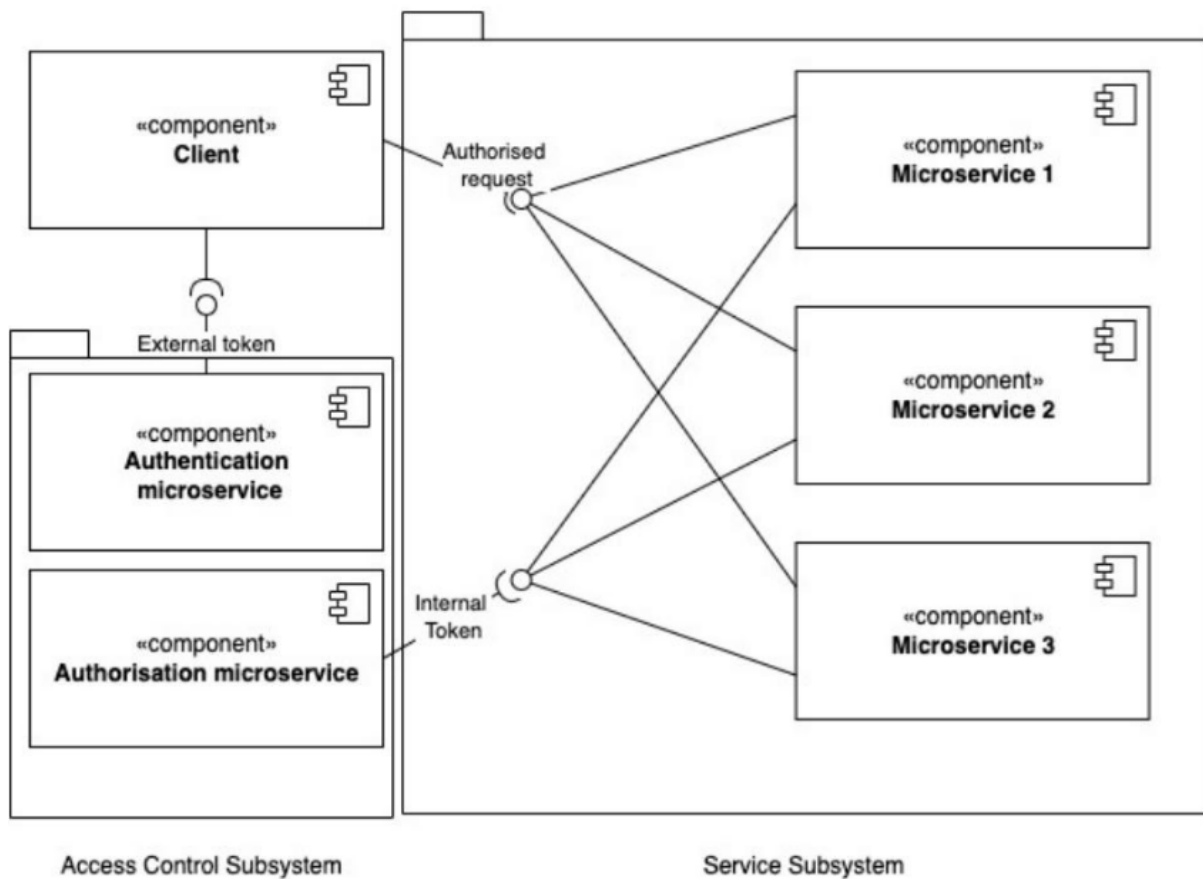


Figura 2.14 – modelo descentralizado

Venčkauskas et al. (2023)

No processo descentralizado, figura 2.14, cada microsserviço autoriza solicitações independentemente, eliminando a necessidade de um gerenciador de consultas central. Isso resulta em autorizações múltiplas para uma solicitação de cliente, dependendo das solicitações internas geradas. Os três microsserviços principais envolvidos são o gerenciador de consultas (encaminha solicitações), o microsserviço de autenticação (emite tokens externos) e o microsserviço de autorização (emite tokens de acesso internos). O cliente interage principalmente com o gerenciador de consultas, simplificando a implementação do controle de acesso descentralizado no sistema. O token interno é gerado durante a autenticação e é incluído em cada solicitação subsequente. Sua validação pelos microsserviços ocorre utilizando a mesma chave do procedimento de emissão.

Sendo assim, o método descentralizado, caracterizado pelo aumento no número de requisições internas, apresentou uma eficiência inferior em comparação com o modo centralizado de controle de acesso. Este último foi considerado duas vezes mais eficiente ao levar em consideração critérios como recursos de hardware, solicitações processadas e duração das solicitações. Embora a abordagem descentralizada permita maior flexibilidade e escalabilidade, ela também traz consigo desafios de segurança devido à natureza dispersa dos microsserviços. Para abordar essas preocupações, o método proposto no texto visa reforçar a segurança dos microsserviços descentralizados por meio da implementação de acesso baseado em tokens, token internos. Essa estratégia assegura um gerenciamento fácil de permissões e preserva a integridade das transações, contribuindo para a robustez e confiabilidade do sistema descentralizado.

2.2.3 Superando desafios de segurança em arquiteturas de microsserviços

O uso de Mutual Transport Layer Security (MTLS) em conjunto com uma Public Key Infrastructure (PKI) auto-hospedada é sugerido, por (YARYGINA; BAGGE, 2018) em seu artigo, como uma forma de proteger a intercomunicação final entre serviços neste artigo, que examina problemas de segurança e soluções em arquiteturas de microsserviços. Para fornecer um grau extra de segurança, essa abordagem busca garantir que cada microsserviço autentique seus pares antes de iniciar comunicações seguras. Junto com essas práticas comuns de autenticação e autorização, de como usar certificados para autenticação mútua e emitir tokens para autorização, o artigo também aborda o uso de tokens e autenticação local como tendências emergentes na segurança de microsserviços.

O estudo enfatiza que, embora os microsserviços tenham vantagens como flexibilidade e escalabilidade, eles também trazem consigo novos riscos de segurança que não estavam presentes nas arquiteturas monolíticas convencionais. Preocupações importantes são o gerenciamento de segredos distribuídos, a construção de confiança entre microsserviços individuais e a atual falta de uma solução de segurança em camadas.

Por fim, o artigo argumenta que, devido às características de acoplamento solto, isolamento e falha rápida dos microsserviços, essas arquiteturas, uma vez superados os desafios de segurança, podem não apenas aumentar a segurança geral de um sistema, mas também melhorar sua resiliência e capacidade de adaptação às exigências modernas de segurança.

2.2.4 Implementação de Controle de Acesso Baseado em Funções no OAuth 2.0 como Sistema de Autenticação

A mudança de arquiteturas de software monolíticas para microsserviços é abordada por (TRIARTONO; NEGARA et al., 2019) neste artigo, juntamente com a forma como os sistemas de autorização e autenticação são afetados. O controle de acesso baseado em função (RBAC) é uma abordagem tradicional usada por sistemas monolíticos, na qual as permissões dos usuários são atribuídas com base em suas funções dentro do sistema. Como esses sistemas são distribuídos, a mudança para uma arquitetura de microsserviços traz consigo novas dificuldades, particularmente com relação ao gerenciamento de autorização e autenticação.

O protocolo OAuth 2.0, que é frequentemente usado para conceder permissão aos usuários para acessar recursos específicos, geralmente é implementado por microsserviços. Os escopos, que especificam os graus de acesso que um cliente pode ter a recursos protegidos, são a base da operação deste protocolo. No entanto, um método que possa combinar a flexibilidade do escopo com a integração do OAuth 2.0 com o modelo RBAC convencional é necessário com a estrutura hierárquica de funções.

O framework Laravel e sua biblioteca Laravel Passport, que já suporta OAuth 2.0, são usados na solução proposta do artigo para integrar RBAC com OAuth 2.0. A ideia principal é construir um sistema de autorização e autenticação onde os escopos atribuídos a usuários dentro do contexto do OAuth 2.0 são determinados por suas funções, conforme definido pelo modelo RBAC. Como resultado, as permissões de acesso podem ser verificadas de forma confiável por microsserviços com base em funções e escopos relacionados.

O modelo é implementado por meio de uma API REST, e tokens de acesso contendo detalhes sobre as funções e escopos dos usuários são gerados pelo Laravel Passport. Os micros-

serviços usam esses tokens para restringir o acesso a recursos específicos, garantindo que apenas usuários autorizados possam alcançar endpoints específicos.

A ferramenta Siege foi usada para avaliar o desempenho do sistema simulando múltiplos usuários acessando a API ao mesmo tempo. Os resultados dos testes demonstraram que, mesmo com uma carga pesada, o sistema continua a operar bem e responde rápido o suficiente. No entanto, foi descoberto que o desempenho começa a cair drasticamente em 250 usuários simultâneos, o que provavelmente se deve às restrições de recursos do servidor durante o teste.

Para resumir, o método sugerido neste artigo fornece uma maneira útil de combinar OAuth 2.0 e RBAC em configurações de microsserviços. A proposta integra os benefícios de ambos os modelos para simplificar os processos de autorização e autenticação, preservando altos níveis de desempenho, mesmo em cenários com alta demanda. Para empresas que desejam garantir segurança e eficiência em seus sistemas de controle de acesso durante a transição de arquiteturas monolíticas para microsserviços, isso a torna uma solução viável.

2.2.5 Projetando serviços seguros para autenticação, Autorização e Contabilização de Usuários

(SHEVCHUK et al., 2023) aborda a criação de um serviço de autenticação e autorização para aumentar a segurança em aplicativos web e móveis. Ele destaca a importância de decidir entre implementar o serviço como um servidor separado ou integrá-lo na infraestrutura de rede local, com a integração local geralmente oferecendo maior controle e proteção de dados.

Para autenticação, o OpenID Connect (OIDC), baseado em OAuth 2.0, é recomendado por permitir a verificação segura da identidade dos usuários sem a necessidade de criar novas credenciais. O OAuth 2.0 também é utilizado para autorização, fornecendo um meio seguro para que aplicativos acessem recursos em nome dos usuários por meio de tokens de acesso, sendo essencial para fluxos de Single Sign-On (SSO) e integração com plataformas externas.

Para reforçar a segurança contra ataques de força bruta e outras ameaças, hashes seguros, "salt"⁴ exclusivo para cada usuário com a atualização periódica quando senhas são alteradas e autenticação multifator (MFA) são sugeridos. gerenciamento seguro de senhas é destacado. Essas técnicas garantem que as credenciais de login sejam adequadamente protegidas.

As duas técnicas primárias para controle de acesso que são examinadas neste trabalho são Role-Based Access Control (RBAC) e Attribute-Based Access Control (ABAC). O ABAC fornece mais flexibilidade ao habilitar direitos com base em certas qualidades do usuário e responder dinamicamente à situação do sistema, enquanto o RBAC simplifica o gerenciamento de permissões com base em funções.

Por fim, o artigo sugere que a escolha dos protocolos e métodos de segurança deve ser guiada pelas necessidades específicas do sistema e pela análise de riscos. A integração de OAuth 2.0 e OpenID Connect é recomendada para sistemas que necessitam de alta segurança e interação com serviços de terceiros, enquanto a implementação de práticas seguras, como MFA e gestão de sessões, é crucial para proteger contra vulnerabilidades comuns.

⁴ "Salt" é um termo usado em criptografia para se referir a um valor aleatório que é adicionado a uma senha antes que ela seja processada por uma função de hash.

3 Desenvolvimento

Neste capítulo, será apresentado a proposta da arquitetura baseada em token, descrevendo seu fluxo de trabalho, e explorando o desenvolvimento de seus componentes, aplicações, seguido das ferramentas e tecnologias utilizadas.

3.1 Arquitetura

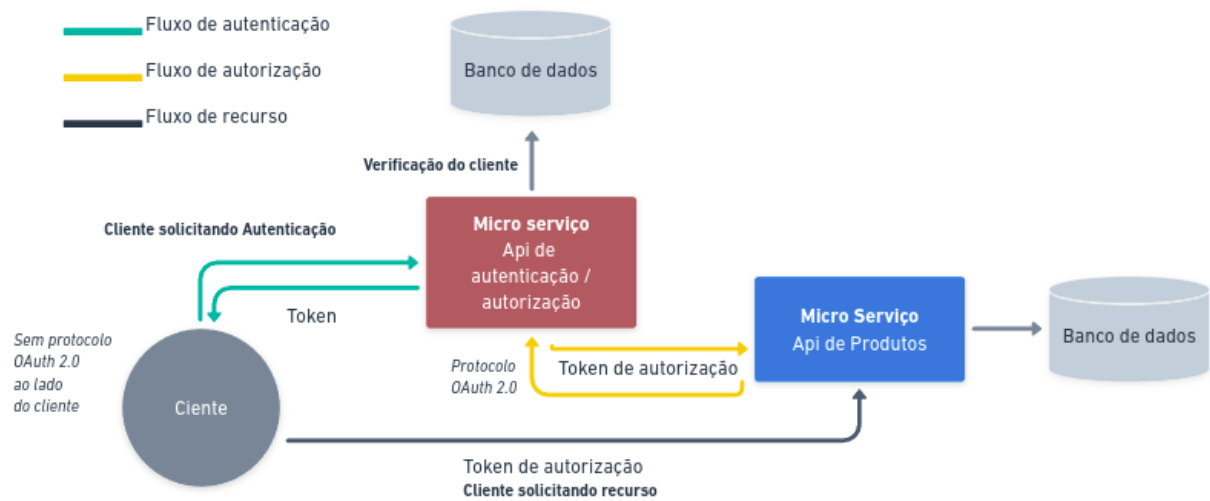


Figura 3.1 – Arquitetura baseada em token

Conforme na figura 3.1, arquitetura proposta baseada em token, apresenta três fluxos principais: autenticação, autorização e solicitação de recurso. As requisições feitas nos fluxos mencionados seguem o protocolo HTTP REST.

No primeiro fluxo, o cliente solicita um token de acesso ao micro serviço, API de autenticação e autorização, apresentando-se para validação das credenciais. Em caso de êxito, o token é fornecido ao cliente. Importa destacar que, neste contexto, não se emprega o protocolo OAuth 2.0, dado a ausência de uma etapa prévia de validação ou permissão, tal como observado em recaptcha ou autorização de domínio externo. A premissa adotada é que o usuário pertence ao domínio em questão.

Prosseguindo para o fluxo de solicitação de recurso, o cliente solicita um recurso ao micro serviço API de produtos. Este, ao receber a requisição, ingressa no fluxo de autorização entre micro serviços. Na instância desse processo, a API de produtos, munida do token do cliente, submete uma requisição à API de autenticação e autorização, validando as credenciais do usuário. Com a validação bem-sucedida, a API de produtos está habilitada a dar continuidade à solicitação de recursos, acessando sua base de dados e proporcionando ao cliente uma resposta satisfatória. Neste cenário, o protocolo OAuth 2.0 é empregado, pois o micro serviço API de produtos valida o

usuário junto à API de autenticação e autorização antes de prosseguir com o fluxo de solicitação de recursos, garantindo um ambiente seguro e autorizado.

O conceito de Single Sign-On (SSO) é integrado ao processo, permitindo ao cliente, por meio de um único token, requisitar recursos a diversos micro serviços. Essa abordagem simplifica a interação do usuário, eliminando a necessidade de múltiplos tokens para acessar diferentes serviços. Notavelmente, a parte de micro serviços poderá projetada com alta escalabilidade, assegurando uma expansão, sem o conhecimento do cliente.

No âmbito da segurança, é adotada uma chave simétrica para a geração do token. O segredo dessa chave é mantido como propriedade do micro serviço e da API de autenticação e autorização, sendo implementado com o JWT. Essa escolha contribui para a confidencialidade e integridade do processo, garantindo a autenticidade do token e pelos participantes envolvidos no fluxo transacional.

3.2 Endpoints

De acordo com as diretrizes estabelecidas para o desenho de uma Interface de Programação de Aplicações (API) Representacional de Estado Transferido (REST), foram empregados métodos HTTP GET para a recuperação de recursos, POST para a criação de novos dados, PUT para a modificação de dados existentes, e DELETE para a remoção de dados específicos do banco de dados. O quadro 3.1 ilustra algumas das rotas implementadas neste projeto.

Métodos	Permissões	Rota	Descrição
POST	-	oauth/login	Log in de usuário com retorno do <i>accessToken</i> e <i>refreshToken</i> .
POST	-	oauth/refresh	Gerar <i>accessToken</i> .
GET	-	oauth/logout	Log out do usuário.
GET	-	oauth	Autorização de micro serviços.
POST	MANAGER	user	Cadastrar um usuário.
PUT, DELETE	MANAGER	user/ <i>userId</i>	Atualizar e deletar um usuário.
GET	MANAGER, USER	user/ <i>userId</i>	Recuperar um usuário.
POST	MANAGER	product	Cadastrar um produto.
PUT, DELETE	MANAGER	product/ <i>productId</i>	Atualizar e deletar um produto.
GET	MANAGER, USER	product/ <i>productId</i>	Recuperar um produto.

Tabela 3.1 – Endpoints do sistema.

3.3 Implementação

3.3.1 Cliente

No intuito de simular as requisições efetuadas pelo cliente, empregou-se a ferramenta Postman, reconhecida no âmbito da engenharia de software pela sua eficácia nos testes de APIs. A seleção desta ferramenta fundamentou-se em sua capacidade de proporcionar uma interface intuitiva e funcionalidades abrangentes para a execução de requisições HTTP REST, facilitando a reprodução de cenários variados. Adicionalmente, a integração do token no cabeçalho durante a simulação reveste-se de importância, representando um mecanismo essencial de autenticação.

3.3.2 Banco de dados

No contexto do gerenciamento de dados, o banco de dados NoSQL MongoDB foi selecionado para a criação de dois distintos repositórios de informações: um destinado ao armazenamento de dados associados aos usuários e outro voltado aos produtos. A escolha do MongoDB fundamentou-se em suas características e na conformidade com padrões NoSQL, atendendo às exigências de integridade e consistência de dados. A implementação de dois bancos de dados separados, um para usuários e outro para produtos, foi orientada pela concepção de micro serviços. Essa abordagem visa promover a modularização e independência entre os serviços, fomentando a escalabilidade e a manutenção eficiente do sistema. A segregação em dois bancos de dados distintos proporciona uma estrutura flexível, alinhada com os princípios fundamentais da arquitetura de microserviços, permitindo uma gestão mais eficaz e adaptável às dinâmicas variáveis de dados."

3.3.2.1 Usuário

```
1 {
2   _id: String,
3   name: String,
4   email: String,
5   password: String,
6   role: String,
7   accessToken: String | null,
8   refreshToken: String | null
9 }
```

3.3.2.2 Produtos

```
1 {
2   _id: String,
3   name: String,
4   cod: String,
5   photo: String,
6 }
```

3.3.3 Api

No contexto da implementação de APIs, Node.js, empregando a linguagem JavaScript e adotando a arquitetura Model-View-Controller (MVC) sem a camada de visualização. O Node.js, construído com o motor V8 do Google Chrome. Ele permite desenvolver aplicativos do lado do servidor, proporcionando uma abordagem eficiente e escalável para a construção de aplicações web. Uma das suas principais características é a natureza assíncrona e orientada a eventos, o que significa que é especialmente adequado para operações de entrada e saída intensivas, como interações com bancos de dados ou chamadas de API. Na arquitetura MVC a ausência da camada de visualização neste contexto específico derivou da dispensabilidade de elementos de interface gráfica, concentrando-se primariamente na lógica do modelo e na manipulação dos dados. A

estruturação das APIs baseou-se na subdivisão em duas categorias distintas: Autenticação e Autorização, e Produtos. A primeira API, dedicada à gestão de autenticação e autorização, estabelece as medidas necessárias para a validação e concessão de acesso, enquanto a segunda, voltada para produtos, facilita a interação e manipulação de informações pertinentes a esta categoria específica. Ambas as APIs compartilham uma abordagem uniforme para o fluxo de autenticação e autorização, utilizando um middleware. Essa abordagem reforça a segurança, garantindo que apenas usuários autenticados e autorizados possam acessar os recursos protegidos da aplicação. O middleware, camada de software, atua como intermediário nas requisições, verificando as credenciais do usuário antes de conceder acesso a recursos protegidos. O middleware utilizando o JWT valida e decodifica o token, assegurando a identidade do usuário e autorizando o acesso com base nas informações do token.

3.3.3.1 Autenticação e autorização

A figura a seguir 3.2 ilustra a estrutura de diretórios da API de autenticação e autorização. O diretório "controllers" contém as funções relacionadas aos endpoints, enquanto "middlewares", figura 3.4, abriga a função de verificação de tokens. No diretório "models", encontramos o esquema do banco de dados, e em "routes", figura 3.3, estão os endpoints com suas respectivas funções. O diretório "service" contém a função de conexão com o banco de dados, e em "utils" está a função relacionada às roles.

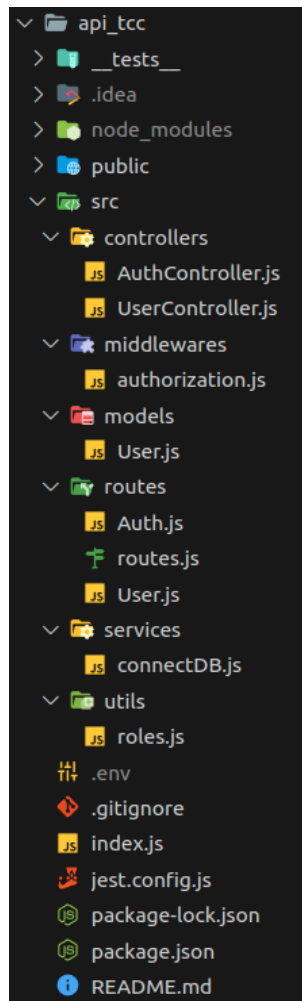


Figura 3.2 – Estrutura de pastas da API de autenticação e autorização.

```
Authjs x
api_tcc > src > routes > Authjs > ...
1  const AuthController = require('../controllers/AuthController');
2  const { verifyAccessToken, verifyRefreshToken } = require("../middlewares/authorization");
3
4  module.exports = {
5    auth(routes) {
6      routes.post('/oauth/login', AuthController.authentication);
7      routes.post('/oauth/refresh', verifyRefreshToken, AuthController.refresh);
8      routes.get('/oauth/logout', verifyAccessToken, AuthController.logout);
9      routes.get('/oauth', verifyAccessToken, AuthController.authorizationMicroService);
10   }
11 }
12

Userjs x
api_tcc > src > routes > Userjs > ...
1  const UserController = require('../controllers/UserController');
2  const AuthController = require('../controllers/AuthController');
3  const role = require("../utils/roles");
4  const { verifyAccessToken } = require("../middlewares/authorization");
5
6  module.exports = {
7    user(routes) {
8      routes.get('/user/:userId', verifyAccessToken, role(['MANAGER', 'USER']), AuthController.authorization, UserController.index);
9      routes.post('/user', verifyAccessToken, role(['MANAGER']), AuthController.authorization, UserController.store);
10     routes.put('/user/:userId', verifyAccessToken, role(['MANAGER']), AuthController.authorization, UserController.update);
11     routes.delete('/user/:userId', verifyAccessToken, role(['MANAGER']), AuthController.authorization, UserController.destroy);
12   }
13 }
```

Figura 3.3 – Endpoints da API de autenticação e autorização.

```
api_tcc > src > middlewares > authorization.js > ...
1  const Auth = require('../controllers/AuthController');
2  const jwt = require('jsonwebtoken')
3  require('dotenv').config()
4
5  async function verifyAccessToken(req, res, next) {
6    const token = (req.headers.authorization && req.headers.authorization.split(' ')[1]) || null;
7    jwt.verify(token, process.env.SECRETE, (error, verify) => {
8      if (error) {
9        return res.status(403).json({
10         code: 403,
11         message: 'Unauthorized',
12       })
13     }
14     req.user = verify.user
15     next()
16   });
17 }
18
19 async function verifyRefreshToken(req, res, next) {
20   const token = (req.headers.authorization && req.headers.authorization.split(' ')[1]) || null;
21   jwt.verify(token, process.env.SECRETE_REFRESH, async (error, verify) => {
22     if (error) {
23       if(jwt.decode(token).hasOwnProperty('mail')){
24         req.user = { email: jwt.decode(token).email }
25         await Auth.logoutRefreshToken(req, res)
26       }
27       return res.status(403).json({
28         code: 403,
29         message: 'Unauthorized',
30       })
31     }
32     req.user = { email: verify.email }
33     next()
34   });
35 }
36
37 module.exports = { verifyAccessToken, verifyRefreshToken }
```

Figura 3.4 – Middleware da API de autenticação e autorização.

As figuras a seguir 3.5, 3.6 e 3.7, estão na camada "controllers" e são responsáveis pela autenticação, autorização e atualização do token de acesso. Na função de autenticação foi adotado o *accessToken* com duração de duas horas e *refreshToken* de um dia. A função 'verifySession' valida a sessão do usuário, comparando o token enviado no cabeçalho da requisição com o token do usuário recuperado da base de dados.

```
AuthController.js X
api_tcc > src > controllers > AuthController.js > authorization
1  const User = require('./UserController');
2  const jwt = require('jsonwebtoken')
3  require('dotenv').config()
4
5  async function authentication(req, res) {
6    try {
7      const { email, password } = req.body;
8
9      const user = await User.getAuth(email, password)
10     const accessToken = jwt.sign({ user }, process.env.SECRETE, { expiresIn: '2h' })
11     const refreshToken = jwt.sign({ token: accessToken, email: user.email }, process.env.SECRETE_REFRESH, { expiresIn: '1d' })
12     await User.updateToken(user.email, accessToken, refreshToken)
13     return res.json({ accessToken: accessToken, refreshToken: refreshToken })
14   } catch (error) {
15     return res.json({ error: error.message });
16   }
17 }
18 }
```

Figura 3.5 – Função de autenticação da API de autenticação e autorização.

```
AuthController.js M X
api_tcc > src > controllers > AuthController.js > ...
21  async function verifySession(req) {
22    const token = (req.headers.authorization && req.headers.authorization.split(' ')[1]) || null;
23    const user = await User.getByEmail(req.user.email)
24    if (user.accessToken !== token) throw new Error()
25    return
26  }
27  async function authorization(req, res, next) {
28    try {
29      await verifySession(req)
30      next()
31    } catch (error) {
32      return res.status(403).json({
33        code: 403,
34        message: 'Unauthorized',
35      })
36    }
37  }
38  async function authorizationMicroService(req, res) {
39    try {
40      await verifySession(req)
41      return res.status(200).json({
42        code: 200,
43        message: 'ok',
44      })
45    } catch (error) {
46      return res.status(403).json({
47        code: 403,
48        message: 'Unauthorized',
49      })
50    }
51  }
```

Figura 3.6 – Funções de autorizações da API de autenticação e autorização.

```
AuthController.js M X
api_tcc > src > controllers > AuthController.js > verifySession > token
78 async function refresh(req, res) {
79   try{
80     const refreshToken = (req.headers.authorization && req.headers.authorization.split(' ')[1]) || null;
81     const user = await User.getByEmail(req.user.email)
82     if (user.refreshToken !== refreshToken) return res.status(403).json({
83       code: 403,
84       message: 'Unauthorized',
85     })
86
87     const accessToken = jwt.sign({ user: {
88       _id: user._id,
89       name: user.name,
90       email: user.email,
91       role: user.role,
92     }}, process.env.SECRETE, { expiresIn: '2h' })
93
94     await User.updateToken(user.email, accessToken, user.refreshToken)
95     return res.json({ accessToken: accessToken, refreshToken: refreshToken })
96   } catch(e){
97     return res.status(403).json({
98       code: 403,
99       message: 'Unauthorized',
100     })
101   }
102 }
```

Figura 3.7 – Refresh token da API de autenticação e autorização.

3.3.3.2 Produtos

A figura a seguir 3.8 ilustra a estrutura de diretórios da API de produtos. O diretório "controllers" contém as funções relacionadas aos endpoints, enquanto "middlewares", figura 3.10, abriga a função de verificação de token. No diretório "models", encontramos o esquema do banco de dados, e em "routes", figura 3.9, estão os endpoints com suas respectivas funções. O diretório "service" contém a função de conexão com o banco de dados e a função que faz chamadas na api de autenticação e autirzação para fins de autorização, e em "utils" está a função relacionada às roles.

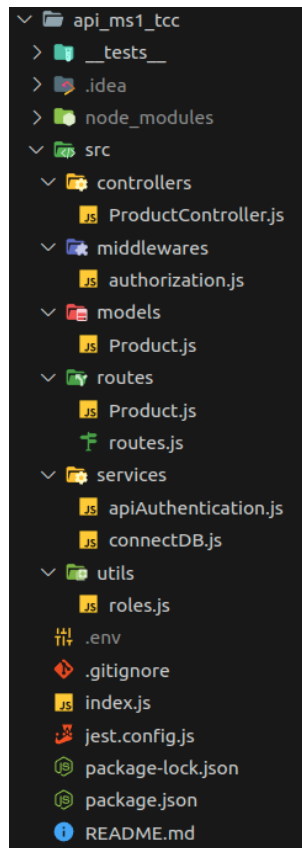


Figura 3.8 – Estrutura de pastas da API produtos.

```
Product.js M X
api_ms1_tcc > src > routes > Product.js > ...
1  const ProductController = require('../controllers/ProductController');
2  const role = require("../utils/roles");
3  const { authorization } = require("../middlewares/authorization");
4
5  module.exports = {
6    user(routes) {
7      routes.get('/product/:productId', authorization, role(['MANAGER', 'USER']), ProductController.index);
8      routes.post('/product', authorization, role(['MANAGER']), ProductController.store);
9      routes.put('/product/:productId', authorization, role(['MANAGER']), ProductController.update);
10     routes.delete('/product/:productId', authorization, role(['MANAGER']), ProductController.destroy);
11   }
12 }
```

Figura 3.9 – Endpoints da API de produtos.

```
api_ms1_tcc > src > middlewares > authorization.js > ...
1  const apiAuthentication = require('../services/apiAuthentication')
2  const jwt = require('jsonwebtoken')
3
4  async function authorization(req, res, next) {
5    try {
6      const token = (req.headers.authorization && req.headers.authorization.split(' ')[1]) || null;
7
8      if (token == null) {
9        return res.status(401).json({
10         code: 401,
11         message: 'not a found',
12       });
13     }
14
15     if(jwt.decode(token).hasOwnProperty('user')){
16       await apiAuthentication.oauth(token)
17       req.user = jwt.decode(token).user
18       next()
19     } else {
20       throw new Error()
21     }
22   } catch (e) {
23     return res.status(403).json({
24       code: 403,
25       message: 'unauthorized',
26     });
27   }
28 }
29
30 module.exports = { authorization }
```

Figura 3.10 – Middleware da API de produtos.

A figura a seguir 3.11 é responsável pela chamada na API de autenticação e autorização, passando no cabeçalho o *accessToken*.

```
api_ms1_tcc > src > services > apiAuthentication.js > ...
1  const fetch = require('node-fetch')
2  require('dotenv').config()
3
4  async function oauth (token) {
5    try{
6      const res = await fetch(process.env.API_AUTHETICATION, {
7        method: 'GET',
8        headers: {
9          'Authorization': 'Bearer ' + token
10       }
11     })
12     const respJson = await res.json()
13
14     if(respJson.code === 403) throw new Error()
15   }catch (e) {
16     console.log(e);
17     throw new Error()
18   }
19 }
20
21 module.exports = { oauth }
```

Figura 3.11 – Funções de autorização da API de produtos.

token do usuário, e a confirmação da autorização, é gerado um novo *accessToken* como resposta, permitindo a continuação das interações. Segue a figura 4.2 a abaixo.

The screenshot shows a REST client interface for a POST request to `http://localhost:9000/oauth/refresh`. The request is configured with a Bearer Token. The response status is 200 OK, with a response time of 9 ms and a size of 1.1 KB. The response body is displayed in a pretty-printed JSON format:

```

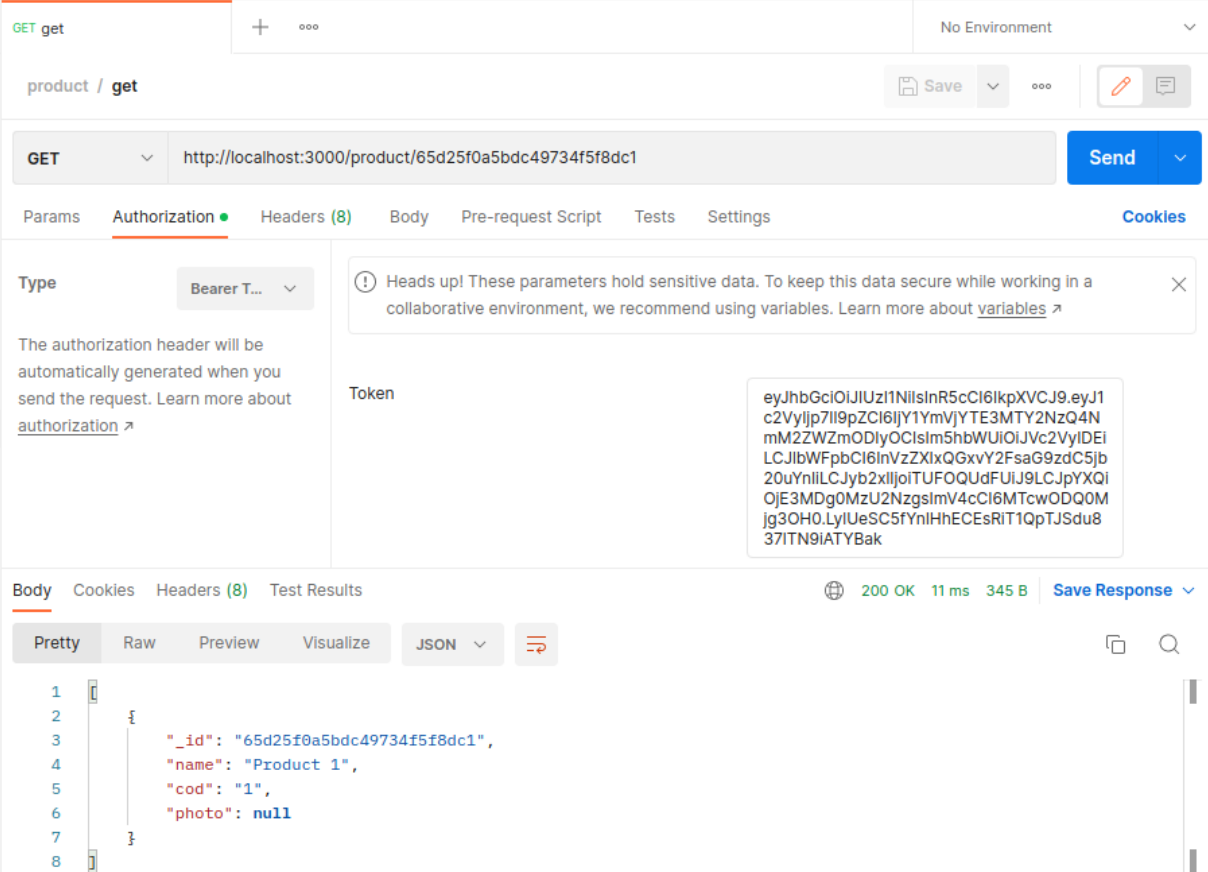
1  {
2    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjI1NiIsInR5cCI6IkpXVCJ9",
3    "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjI1NiIsInR5cCI6IkpXVCJ9"
4  }

```

Figura 4.2 – Refresh token.

4.3 Produtos

No processo de recuperação de recursos na API de produtos, o *accessToken* é empregado no cabeçalho da requisição para a rota `'product/productId'`. Posteriormente, o microserviço valida a sessão na API de autenticação e autorização. Após, ser validado temos como resultado a figura 4.3 a abaixo.



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/product/65d25f0a5bdc49734f5f8dc1
- Authorization:** Bearer Token
- Response Status:** 200 OK, 11 ms, 345 B
- Response Body (JSON):**

```
1 {
2   "id": "65d25f0a5bdc49734f5f8dc1",
3   "name": "Product 1",
4   "cod": "1",
5   "photo": null
6 }
7
8
```

Figura 4.3 – Endpoint para recuperar produto com sucesso.

Ainda sobre a recuperação de recursos na API de produtos, temos um caso de falha, segue a figura 4.4, quando passado um *accessToken* inválido. Essa resposta é da API de autenticação e autorização informando um acesso proibido.

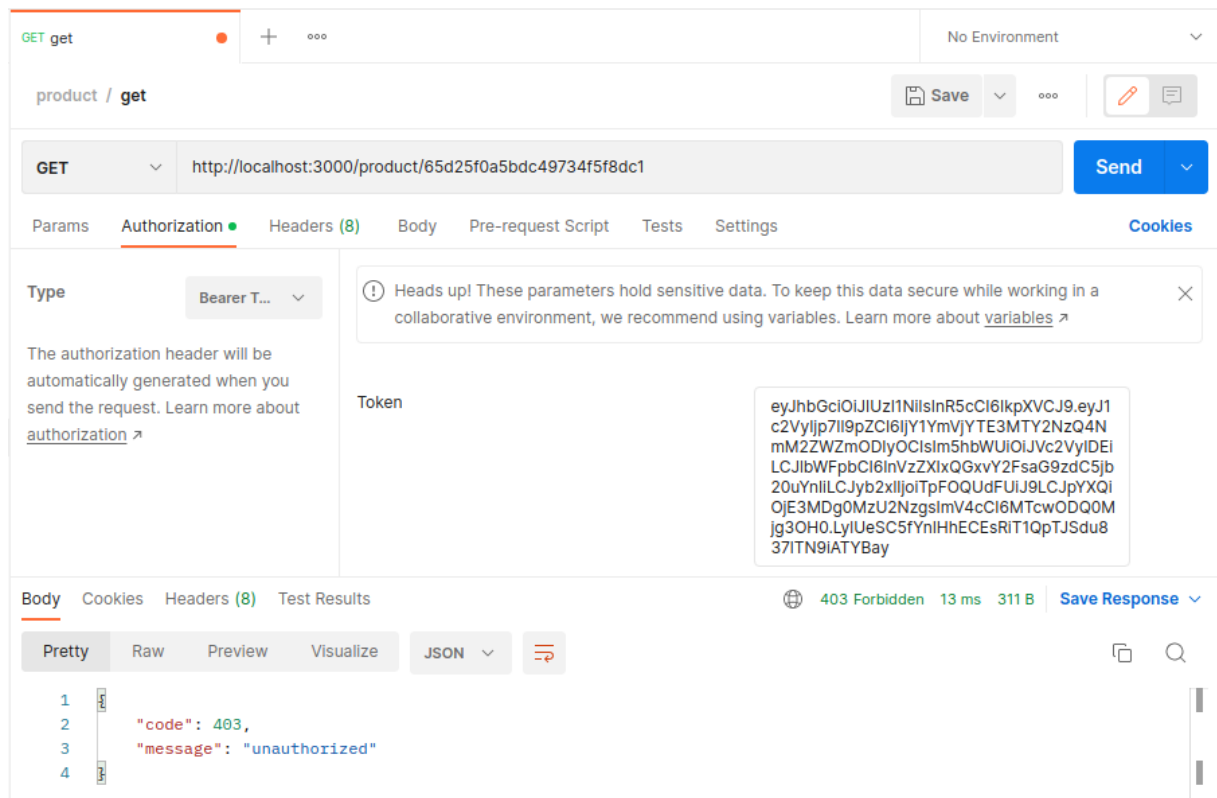


Figura 4.4 – Endpoint para recuperar produto não autorizado.

4.4 Teste de estresse

Embora o teste de estresse não seja uma característica diretamente relacionada à segurança, ele pode servir para um fator decisivo na aceitação ou rejeição do sistema. O principal objetivo desse teste é avaliar o comportamento do sistema quando ele atinge seu limite de capacidade em um determinado ambiente. Esse ambiente serve de base para analisar a necessidade de escalabilidade, conforme a uma determinada demanda.

Neste contexto, foi considerado um cenário similar ao de um serviço em nuvem AWS, utilizando uma instância EC2 t2.micro (1 vCPU, 1 GB de memória RAM, 5 GB de EBS, rodando Ubuntu). O teste foi realizado localmente em uma máquina com processador Core i7 de 1,8 GHz e 16 GB de memória RAM, em um ambiente virtualizado via Docker, configurado com recursos computacionais equivalentes aos da instância t2.micro.

As API's de autenticação (usuários) e recursos (produtos) e o banco de dados foram alocadas em containers separados, simulando a execução em máquinas distintas. As ferramentas utilizadas para o teste foram o JMeter, para simular requisições simultâneas de múltiplos usuários; o Prometheus e Cadvisor, para coletar métricas da máquina e do ambiente virtual; e o Grafana, para visualização e análise dessas métricas. Na figura 4.5 é possível observar os ambientes montados para a realização dos testes, onde: "api_tcc" é referente a API de autenticação e autorização; "api_ms1_tcc" é a referente a API produtos.

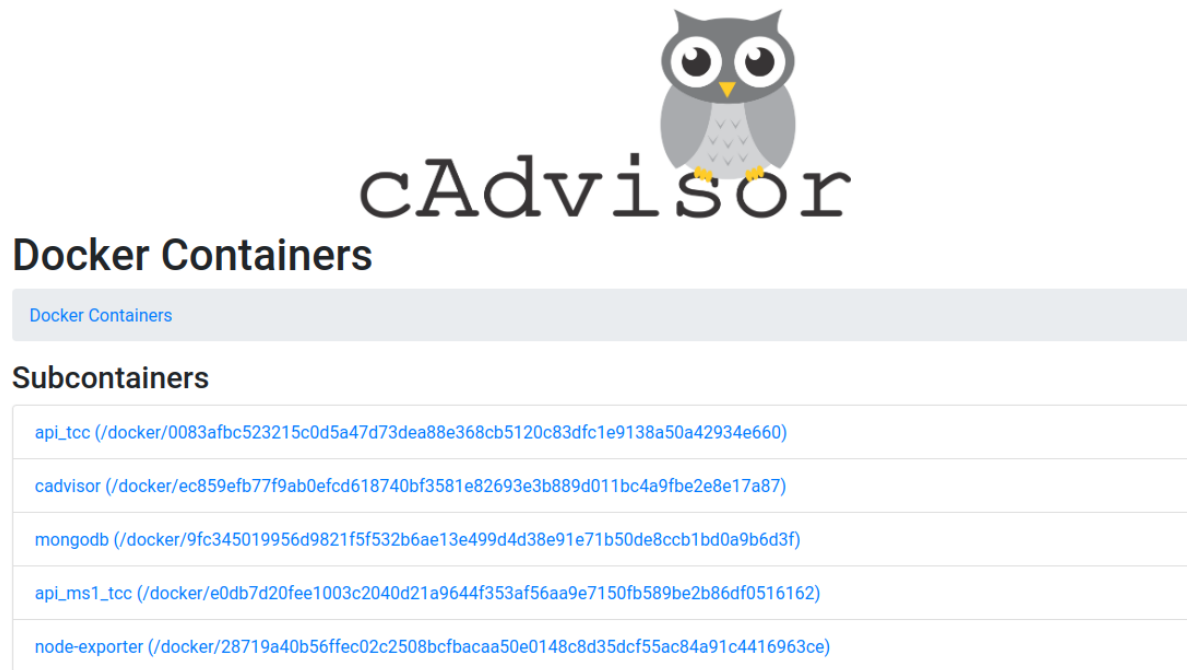


Figura 4.5 – Ambientes

Três cenários compuseram os testes: estresse na API de autenticação e autorização, na API de produtos e simultânea nas duas APIs. O objetivo foi simular o comportamento da autenticação e autorização de forma isolada e, posteriormente, em concorrência, tendo em conta que ambas têm acesso à base de dados. Lembrando que o teste isolado na API de autenticação e autorização é somente de autenticação e, quando é feito na API de produtos, utiliza o recurso de autorização, e consequentemente o teste simultâneo testa autenticação e autorização. O uso da CPU e o consumo de memória foram as principais métricas medidas durante o teste.

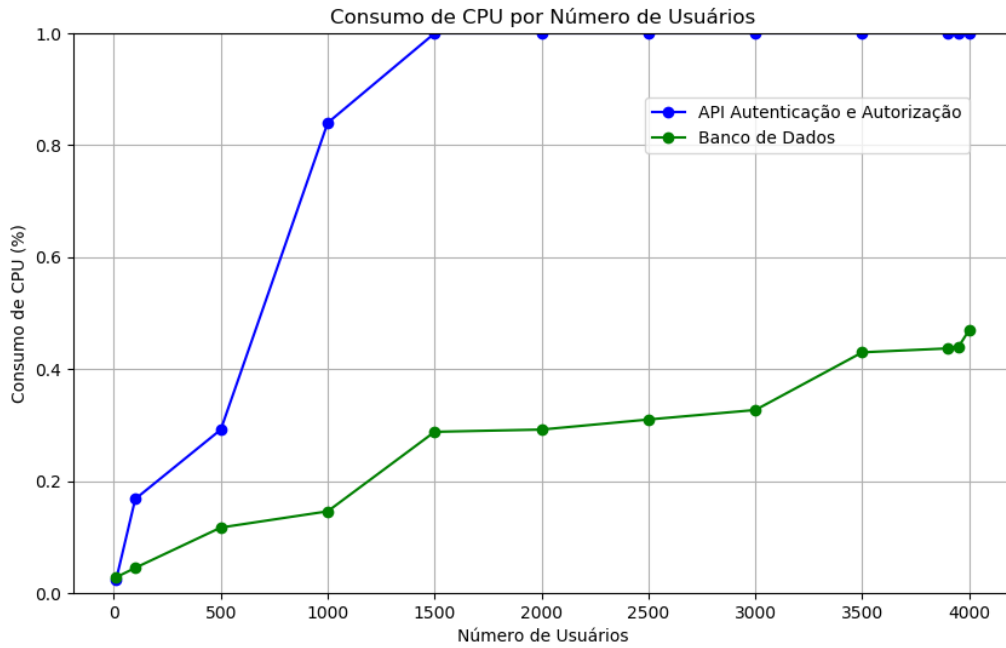


Figura 4.6 – Teste 1

Neste primeiro teste 4.6, o ambiente que hospeda a API de autenticação e autorização atingiu o seu máximo desempenho de CPU a partir de 1500 usuários simultâneos. No entanto, a API não apresentou falhas. Quando foi testada com 4000 usuários simultâneos, a API apresentou uma taxa de erro de 0,7%, e o consumo de memória desse ambiente chegou a 185 MB, enquanto a máquina que hospeda o banco de dados registrou um consumo de 184,3 MB.

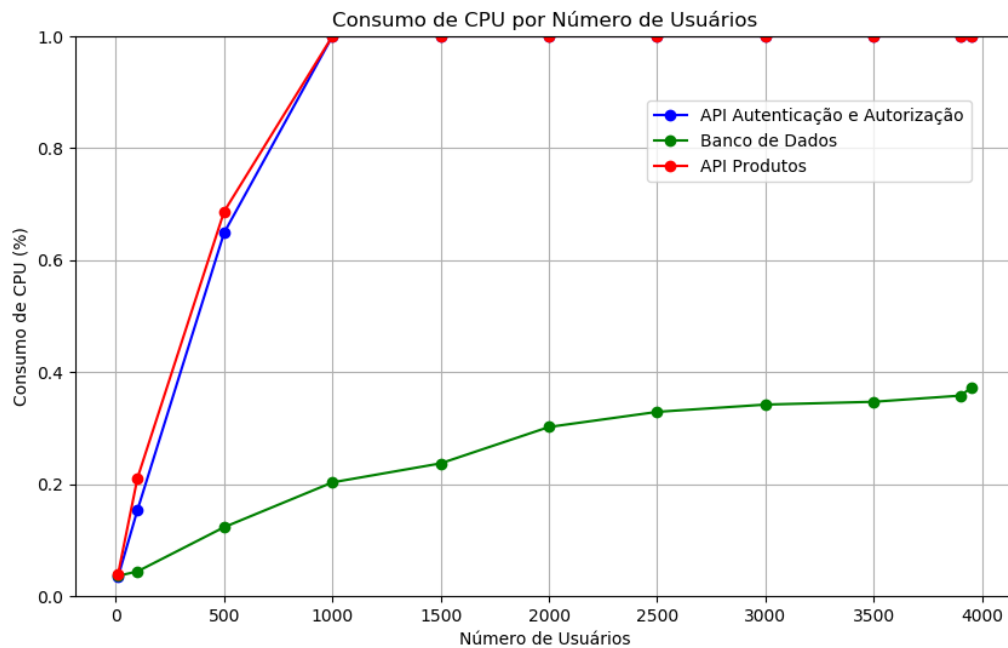


Figura 4.7 – Teste 2

No segundo teste 4.7, o ambiente que hospeda a API de autenticação e autorização, assim como a máquina com a API de produtos, atingiram o seu máximo desempenho de CPU a partir de 1000 usuários simultâneos. No entanto, as APIs não apresentaram falhas. Quando testadas com 3950 usuários simultâneos, a API de produtos apresentou uma taxa de erro de 0,05%. O consumo de memória desse ambiente que hospeda a API de autenticação e autorização chegou a 155 MB, 209 MB para o ambiente que hospeda a API de produtos, e 184 MB para o ambiente com o banco de dados.

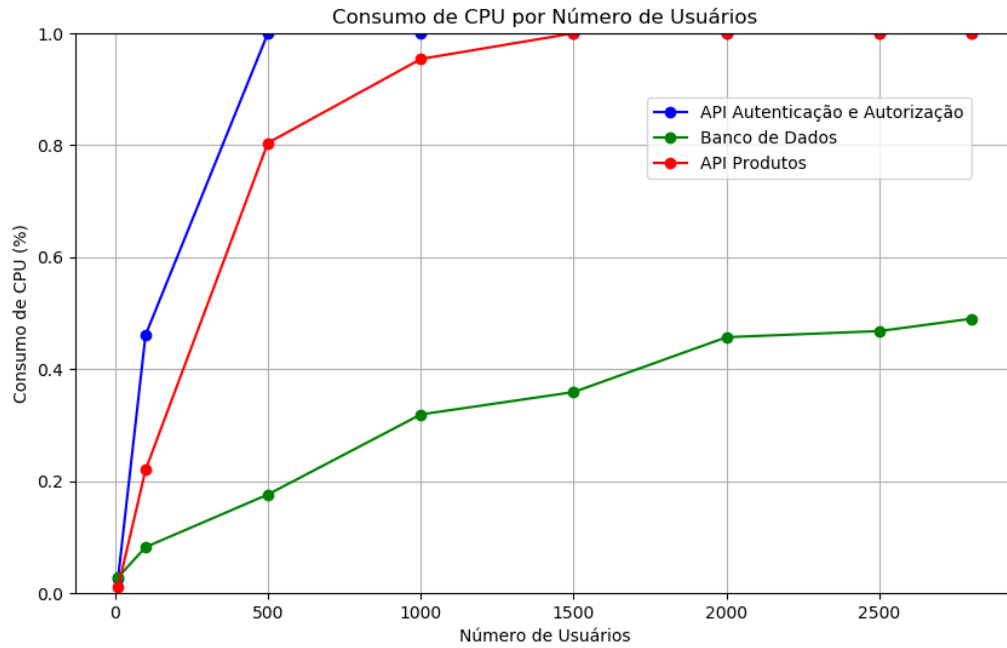


Figura 4.8 – Teste 3

No terceiro teste 4.8, o ambiente que hospeda a API de autenticação e autorização atingiu o seu máximo desempenho de CPU a partir de 500 usuários simultâneos, enquanto a máquina com a API de produtos chegou ao seu máximo desempenho de CPU a partir de 1500 usuários. No entanto, as APIs não apresentaram falhas. Quando testadas com 2800 usuários simultâneos, a API de autenticação e autorização apresentou uma taxa de erro de 1,79%, com o consumo de memória desse ambiente atingindo 154 MB, 177 MB para o ambiente que hospeda a API de produtos, e 184 MB para o ambiente com o banco de dados.

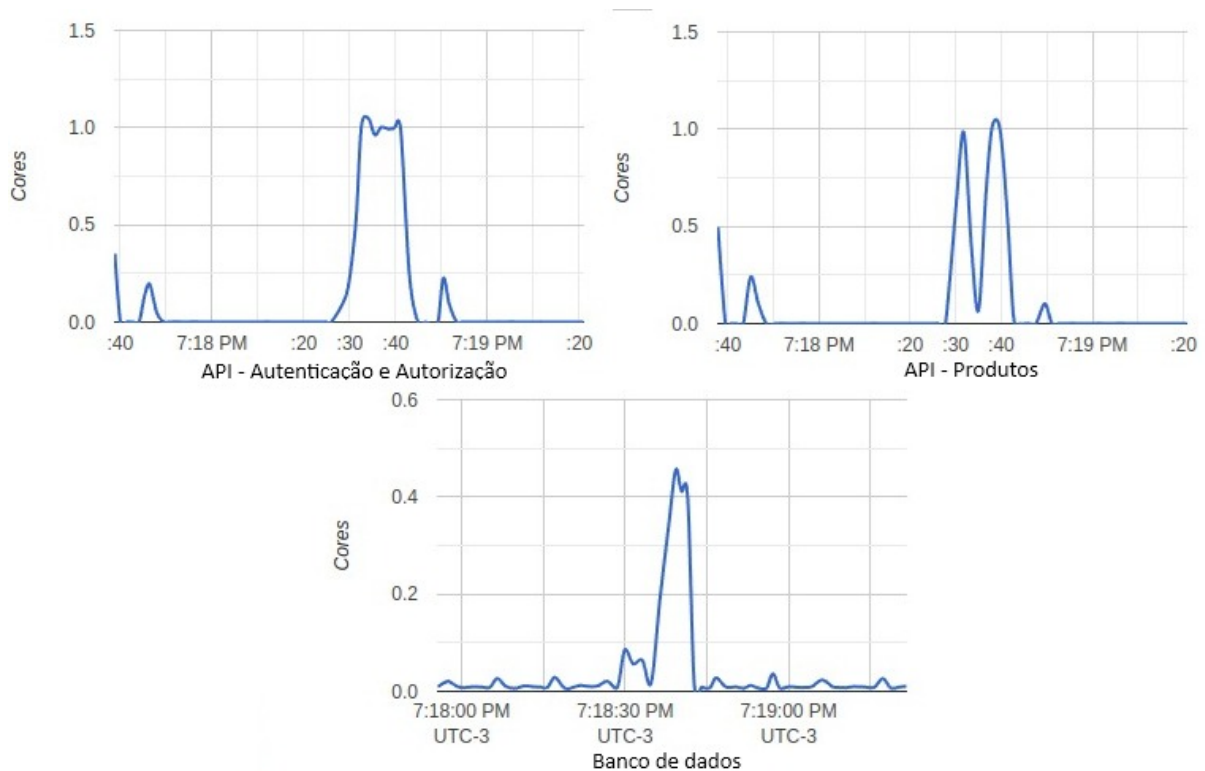


Figura 4.9 – Comportamento durante o erro do terceiro teste

Na imagem 4.9, é apresentado o comportamento dos ambientes quando ocorre o erro no terceiro teste da API de autenticação e autorização, com 2800 usuários simultâneos. É possível perceber que, quando ela falha, o ambiente da API de produtos apresenta uma queda no processamento, assim como o ambiente do banco de dados também sofre uma pequena queda. Essa degradação no desempenho pode ser atribuída à sobrecarga resultante do aumento no tráfego de requisições que ocorreu em decorrência da falha na API de autenticação. Quando ela não conseguiu processar as autenticações de maneira eficaz, muitas requisições ficaram pendentes, levando a um acúmulo de solicitações. Essa pressão adicional resultou em uma latência significativamente maior na comunicação entre os serviços, uma vez que cada requisição que passava pela rede demandava mais tempo para ser processada e respondida. Esse cenário ressalta a interdependência crítica entre os componentes do sistema, onde a falha em um serviço essencial pode comprometer a eficiência e a eficácia de todo o ecossistema de aplicações.

5 Considerações Finais

5.1 Conclusão

Este estudo propôs o desenvolvimento de uma arquitetura baseada em microserviços, com um sistema de autenticação e autorização baseado em token, utilizando JSON Web Tokens (JWT) e o protocolo OAuth 2.0, visando assegurar a confidencialidade dos dados e o controle das sessões de usuários. A interação do usuário com o sistema ocorre por meio de um token de acesso, caracterizado por um tempo de vida reduzido, e um token de sessão, com duração correspondente à sessão do usuário, destinado a atualizar o token de acesso durante a sessão, com o propósito de mitigar potenciais vulnerabilidades.

A funcionalidade central de autenticação e autorização foi implementada, alcançando seu objetivo de proporcionar um acesso seguro, conforme as devidas permissões de privilégios.

As interfaces de programação de aplicações (APIs) foram desenvolvidas em conformidade com o padrão arquitetural (REST), em JavaScript com Node.js e MongoDB para a persistência dos dados. A API de autenticação e autorização assume a responsabilidade pelo gerenciamento de usuários e permissões, utilizando do conceito de chave simétrica. Por sua vez, a API de produtos é responsável pelo gerenciamento dos recursos relacionados a produtos.

A solução proposta passou por testes de estresse que demonstraram sua estabilidade e bom desempenho em alguns cenários de carga controlada. Esses testes forneceram um cenário base para a avaliação do funcionamento da arquitetura implementada. Com tudo, este estudo se apresenta como uma base para possíveis trabalhos futuros.

5.2 Trabalhos Futuros

Como continuidade deste estudo, recomenda-se a realização de testes de penetração (pentests) para identificar possíveis vulnerabilidades. Adicionalmente, sugere-se a adoção do mTLS (Mutual Transport Layer Security) como uma camada extra de autenticação e segurança. A comparação entre o desempenho e a segurança proporcionados pelo mTLS e pela arquitetura baseada em JWT permitirá uma análise mais detalhada das melhores práticas de segurança para diferentes cenários. Para ampliar o escopo da pesquisa, seria relevante explorar a separação entre os processos de autenticação e autorização, avaliando os potenciais benefícios dessa abordagem. Isso permitirá identificar qual solução oferece o melhor equilíbrio entre segurança, escalabilidade e simplicidade, ajustando-se às necessidades específicas de cada aplicação.

Referências

- ADELAIDE. *Ultimate Guide: Safeguarding RESTful APIs with JSON Web Tokens (JWT) in JavaScript for Enhanced Security*. 2023. Disponível em: <ln.run/VdrXi>.
- ANDRADE, E.; LUNARDI, R.; RAMOS, N. *Conceitos básicos de Criptografia*. [S.l.: s.n.], 2021. 91 p. ISBN 978-65-86471-17-5.
- AUTENTICAÇÃO. In: *MICHAELIS: moderno dicionário da língua portuguesa*. São Paulo: Editora Melhoramentos. 2024. Disponível em: <<https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/autentica>>.
- AUTORIZAÇÃO. In: *MICHAELIS: moderno dicionário da língua portuguesa*. São Paulo: Editora Melhoramentos. 2024. Disponível em: <<https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/autoriza>>.
- ERLICH, Z.; ZVIRAN, M. Authentication practices from passwords to biometrics. *Encyclopedia of Information Science and Technology, Third Edition*, IGI Global, p. 4248–4257, 2015.
- FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine, 2000.
- FOUNDATION, O. *OWASP API Security Top 10*. 2023. Disponível em: <<https://owasp.org/API-Security/editions/2023/en/0x00-notice/>>.
- FOWLER M.; LEWIS, J. *Microservices*. [S.l.], 2014. Disponível em: <<https://martinfowler.com/articles/microservices.html>>.
- GHOSH, S. *Distributed systems: an algorithmic approach*. [S.l.]: Chapman and Hall/CRC, 2006.
- GÓMEZ, J.; CÁRDENAS, S.; RUIZ, F. Patient identification system based on nfc and blockchain technology. *Investigación e Innovación en Ingenierías*, v. 11, p. 1–15, 07 2023.
- HAMMER-LAHAV, E. *The OAuth 1.0 Protocol*. RFC Editor, 2010. RFC 5849. (Request for Comments, 5849). Disponível em: <<https://www.rfc-editor.org/info/rfc5849>>.
- HARDT, D. *The OAuth 2.0 Authorization Framework*. RFC Editor, 2012. RFC 6749. (Request for Comments, 6749). Disponível em: <<https://www.rfc-editor.org/info/rfc6749>>.
- IDRIS, M.; SYARIF, I.; WINARNO, I. Web application security education platform based on owasp api security project. *EMITTER International Journal of Engineering Technology*, p. 246–261, 2022.
- JACK, C. H.; TECK, S. K.; MING, L. T.; HONG, D. Y. An overview analysis of authentication mechanism in microservices-based software architecture: A discussion paper. In: *IEEE. 2023 IEEE 8th International Conference On Software Engineering and Computer Systems (ICSECS)*. [S.l.], 2023. p. 1–6.
- JONES, M. *JWT*. 2024. Disponível em: <<https://jwt.io/introduction/>>.

- JONES, M. B.; BRADLEY, J.; SAKIMURA, N. *JSON Web Token (JWT)*. RFC Editor, 2015. RFC 7519. (Request for Comments, 7519). Disponível em: <<https://www.rfc-editor.org/info/rfc7519>>.
- LI, B.; GE, S.; WO, T.-y.; MA, D.-f. Research and implementation of single sign-on mechanism for asp pattern. In: SPRINGER. *International Conference on Grid and Cooperative Computing*. [S.l.], 2004. p. 161–166.
- LIANG, H.; PEI, X.; JIA, X.; SHEN, W.; ZHANG, J. Fuzzing: State of the art. *IEEE Transactions on Reliability*, IEEE, v. 67, n. 3, p. 1199–1218, 2018.
- LODDERSTEDT, T.; BRADLEY, J.; LABUNETS, A.; FETT, D. *OAuth 2.0 Security Best Current Practice*. [S.l.], 2019. Work in Progress. Disponível em: <<https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/13/>>.
- MASSE, M. *REST API design rulebook: designing consistent RESTful web service interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011.
- MILLER, B. P.; FREDRIKSEN, L.; SO, B. An empirical study of the reliability of unix utilities. *Communications of the ACM*, ACM New York, NY, USA, v. 33, n. 12, p. 32–44, 1990.
- MONTANHEIRO, L. S.; CARVALHO, A. M. M.; RODRIGUES, J. A. Utilização de json web token na autenticação de usuários em apis rest. *XIII Encontro Anual de Computação. Anais do XIII Encontro Anual de Computação EnAComp*, p. 186–193, 2017.
- MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. 3rd ed. ed. Hoboken and N.J: John Wiley & Sons, 2012.
- PASSOS, C. *Padrões de Microserviços*. 2019. Disponível em: <<https://medium.com/codigorefinado/padr%C3%B5es-de-microservi%C3%A7os-86da0a88c135>>.
- RADHA, V.; REDDY, D. H. A survey on single sign-on techniques. *Procedia Technology*, Elsevier, v. 4, p. 134–139, 2012.
- SHEVCHUK, D.; HARASYMCHUK, O.; PARTYKA, A.; KORSHUN, N. Designing secured services for authentication, authorization, and accounting of users. *Cybersecurity Providing in Information and Telecommunication Systems*, Germany, v. 3550, p. 207–225, 2023.
- STALLINGS, W. *Criptografia E Segurança De Redes - 6ª Ed. 2014*. [S.l.]: Pearson Education - Br, 2014. ISBN 9788543005898.
- TRIARTONO, Z.; NEGARA, R. M. et al. Implementation of role-based access control on oauth 2.0 as authentication and authorization system. In: IEEE. *2019 6th international conference on electrical engineering, computer science and informatics (EECSI)*. [S.l.], 2019. p. 259–263.
- VENČKAUSKAS, A.; KUKTA, D.; GRIGALIŪNAS, Š.; BRŪZGIENĖ, R. Enhancing microservices security with token-based access control method. *Sensors*, MDPI, v. 23, n. 6, p. 3363, 2023.
- YARYGINA, T.; BAGGE, A. H. Overcoming security challenges in microservice architectures. In: IEEE. *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. [S.l.], 2018. p. 11–20.