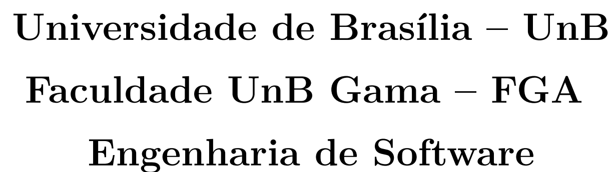


Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Desenvolvimento seguro de software utilizando boas práticas de segurança para microsserviços: um estudo de caso

Autores: Adrian Soares Lopes

Orientadora: Profª. Drª. Elaine Venson

Brasília, DF

2024

Adrian Soares Lopes

Desenvolvimento seguro de software utilizando boas práticas de segurança para microsserviços: um estudo de caso

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Profª. Drª. Elaine Venson

Brasília, DF

2024

Adrian Soares Lopes
Desenvolvimento seguro de software utilizando boas práticas de segurança para microsserviços: um estudo de caso/ Adrian Soares Lopes. – Brasília, DF, 2024-
76 p. : il. (algumas color.) ; 30 cm.

Orientador: Profª. Drª. Elaine Venson
Coorientador:

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2024.

1. Desenvolvimento Seguro de Software. 2. Microsserviço. 3. Arquitetura Distribuída. 4. Máquina Virtual. 5. Teste unitário. I. Profª. Drª. Elaine Venson. II. III. Universidade de Brasília. IV. Faculdade UnB Gama. V. Desenvolvimento seguro de software utilizando boas práticas de segurança para microsserviços: um estudo de caso

CDU

Adrian Soares Lopes

Desenvolvimento seguro de software utilizando boas práticas de segurança para microsserviços: um estudo de caso

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Profª. Drª. Elaine Venson
Orientador

Profª. Drª. Milene Serrano
Convidado 1

Prof. Dr. Tiago Alves da Fonseca
Convidado 2

Brasília, DF
2024

Agradecimentos

Agradeço a todos que me apoiaram no processo de vencer esse desafio que finaliza uma importante etapa da minha vida. Especialmente a Deus, pois sem ele nada disso seria possível. Também à minha esposa, Jussara Alves, o amor da minha vida, que esteve comigo em todos os momentos e nunca deixou de acreditar em mim, mesmo quando eu mesmo desacreditava. A minha mãe, Telma dos Santos, que sempre investiu em minha educação, demonstrando a importância dos estudos e da dedicação. Ao meu pai, que Deus o tenha, Hemiliano Lopes, que sempre me amou e fez despertar em mim o interesse pelo software e pela programação. Também ao meu sogro, Pedro Lopes e a minha sogra, Patrícia Alves, que foram como pais para mim, segurando as pontas em todos os momentos de dificuldade que tive durante essa trajetória e acreditando no meu sucesso. Por fim, a minha querida orientadora Prof.ª Dr.ª Elaine Venson que, mesmo tendo diversos orientandos, topou me ajudar com esse desafio e disponibilizou todo o seu conhecimento e experiência nos meus momentos de dificuldades e dúvidas, sendo sempre paciente e um amor de pessoa.

Resumo

Para diferentes problemas, múltiplos paradigmas e arquiteturas foram criados, propondo novas formas de visualizar e resolver esses problemas. Desde a popularização das conexões de rede entre computadores, o mercado tem adotado sistemas distribuídos como soluções em diferentes contextos. Uma das arquiteturas que tem sido amplamente adotada é a arquitetura de Microsserviços. Essa arquitetura implementa serviços com funções de negócios específicas e independentes. No entanto, um dos desafios é a segurança dos microsserviços, dada a capilaridade de diversos pontos de comunicação entre serviços, que criam uma complexidade de observabilidade, mitigação e correção de falhas de segurança. Este trabalho teve como objetivo analisar o esforço e os desafios encontrados no desenvolvimento seguro de microsserviços. Para tanto, com base em uma pesquisa descritiva, foi proposto um conjunto de requisitos focados em segurança para a construção de microsserviços que formam um MVP de uma aplicação. Em seguida, o MVP foi desenvolvido e analisado, com foco no esforço (em horas) necessário para implementar os principais requisitos de segurança e nas vantagens e dificuldades encontradas.

Palavras-chave: segurança de software; microsserviços; desenvolvimento de software; estudo de caso.

Abstract

For different problems, multiple paradigms and architectures have been created, proposing new ways to visualize and solve these problems. Since the popularization of network connections between computers, the market has adopted distributed systems as solutions in different contexts. One of the architectures that has been widely adopted is the Microservices architecture. This architecture implements services with specific and independent business functions. However, one of the challenges is the security of microservices, given the numerous communication points between services, which create complexity in terms of observability, mitigation, and correction of security failures. This study aimed to analyze the effort and challenges encountered in the secure development of microservices. To that end, based on descriptive research, a set of security-focused requirements was proposed for building microservices that form an MVP of an application. Next, the MVP was developed and analyzed, focusing on the effort (in hours) needed to implement the main security requirements and on the advantages and difficulties encountered.

Keywords: software security; microservices; software development; case study.

Lista de ilustrações

Figura 1 – Segurança em profundidade na visão dos microsserviços. 16
Figura 2 – Esquema da metodologia proposta para o trabalho. 18
Figura 3 – 21 áreas de conhecimento da cibersegurança. 21
Figura 4 – Exemplo de análise de cobertura do repositório. 45
Figura 5 – Exemplo de análise de segurança do repositório. 46
Figura 6 – Exemplo da análise de *contêiner* do repositório. 46
Figura 7 – Processo geral de desenvolvimento proposto. 50
Figura 8 – Etapas da engenharia de requisitos. 52
Figura 9 – Visão do subprocesso de planejamento 52
Figura 10 – Subprocesso de desenvolvimento 52
Figura 11 – Etapas do subprocesso de verificação. 53
Figura 12 – Arquitetura dos serviços. Componentes básicos são o *Kubernetes* para gerenciamento dos microsserviços e um serviço de mensageria para a comunicação entre os serviços. 55
Figura 13 – Análise de ameaças na arquitetura de alto nível 58
Figura 14 – Visão dos *deployments* executados no Cluster no *Dashboard*. Mostra quais possuem *sidecars* do *Linkerd* na coluna *Meshed*. 64
Figura 15 – Visão dos *Deployments e Pods específicos do OpenFERP*. 64
Figura 16 – Visualização do serviço de *Broker* e suas conexões em um dado momento pelo *Dashboard*. 65

Lista de tabelas

Tabela 1 – Exemplos de Histórias do *SAFECode* - Contém as histórias 3 e 8, com as respectivas tarefas e indicação das práticas do *SAFECode*. 30
Tabela 2 – Histórias de usuário do projeto *Open FERP*. Objetivam a descrição de requisitos para o desenvolvimento de Software em um projeto Ágil. . . 48
Tabela 3 – Histórias de segurança para o contexto do desenvolvimento do projeto. 49
Tabela 4 – Histórias de segurança para o contexto de infraestrutura/arquitetura do projeto. 50
Tabela 5 – Especificação das possíveis ameaças, considerando o modelo de comunicação da Figura 13 59
Tabela 5 – Especificação das possíveis ameaças, considerando o modelo de comunicação da Figura 13 (continuação) 60
Tabela 6 – Checklist de definição de concluído com requisitos de segurança para a **US14**. Esses critérios podem se relacionar com um ou mais requisitos de segurança. 61
Tabela 7 – Tempos de Implementação dos Componentes de Segurança 66
Tabela 8 – Sumarização da definição de concluído dos requisitos de segurança para o microserviço de RH e a respectiva duração para implementação. . . . 67
Tabela 9 – Sumarização da definição de concluído dos requisitos de segurança para o microsserviço de vendas e a respectiva duração para implementação. 68
Tabela 10 – Sumarização da definição de concluído dos requisitos de segurança para o microsserviço de estoque e a respectiva duração para implementação. 70

Lista de abreviaturas e siglas

AMQP *Advanced Message Queuing Protocol*

API *Application Programming Interface*

DNS *Domain Name System*

ERP *Enterprise Resource Planning*

HTML *HyperText Markup Language*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

JWT *JSON Web Token*

MQTT *Message Queuing Telemetry Transport*

MVP *Minimum Viable Product*

RH Recursos Humanos

SOA *Service-Oriented Architecture*

SQL *Structured Query Language*

SSD *Secure Software Development*

TCP *Transmission Control Protocol*

TLS *Transport Layer Security*

UnB Universidade de Brasília

US *User Story*

Sumário

1 INTRODUÇÃO . . . **13**
1.1 Contexto . . . **13**
1.2 Problema . . . **15**
1.3 Objetivos . . . **16**
1.3.1 Objetivo Geral . . . 16
1.3.2 Objetivos Específicos . . . 16
1.4 Metodologia . . . **17**
1.4.1 Classificação Metodológica . . . 17
1.4.2 Plano Metodológico . . . 17
1.5 Organização do Trabalho . . . **19**

2 REVISÃO BIBLIOGRÁFICA . . . **20**
2.1 Considerações Iniciais do Capítulo . . . **20**
2.2 Cibersegurança . . . **20**
2.2.1 *CyBOK* . . . 20
2.2.2 Desenvolvimento Seguro de *Software* . . . 24
2.3 Segurança de Microsserviços . . . **31**
2.3.1 Modelagem de Ameaças . . . 32
2.3.2 Estratégias de Segurança para implementações e Ameças específicas . . . 34
2.3.2.1 Estratégias para identificação e gerência de acesso . . . 34
2.3.2.2 Estratégias para o registro de microsserviços . . . 36
2.3.2.3 Estratégias para Protocolos de Comunicação Segura . . . 36
2.3.2.4 Estratégias para o monitoramento de segurança . . . 37
2.3.2.5 Estratégias para disponibilidade / resiliência . . . 37
2.3.2.6 Estratégias para garantia da integridade . . . 38
2.3.2.7 Estratégias para *framework* de arquitetura . . . 38
2.4 *Kubernetes* . . . **39**
2.4.1 Visão Geral . . . 39
2.4.1.1 Configuração de Aplicação . . . 40
2.4.1.2 Pods no Kubernetes . . . 40
2.4.2 *Service Mesh* . . . 41
2.4.2.1 *Linkerd* . . . 42
2.5 *Mensageria* . . . **42**
2.5.1 Visão geral . . . 42
2.5.2 Protocolos de Mensagens . . . 43

2.5.2.1 *Advanced Message Queuing Protocol (AMQP)* . . . 43
2.5.2.2 *Message Queuing Telemetry Transport (MQTT)* . . . 43
2.5.2.3 Protocolo Apache Kafka . . . 43
2.5.3 *Broker Rabbit MQ* . . . 44

3 PROJETO *OPEN FERP* . . . 45
3.1 Produto *Open FERP* . . . 46
3.1.1 Visão geral . . . 46
3.1.2 *Backlog do Produto* . . . 47
3.2 Processo de desenvolvimento . . . 50

4 INICIAÇÃO . . . 54
4.1 Arquitetura e Comunicação . . . 54
4.1.1 Componentes e Fluxo de Comunicação . . . 54
4.1.1.1 Usuário . . . 54
4.1.1.2 *API Gateway* . . . 54
4.1.1.3 Serviços . . . 54
4.1.1.3.1 Funcionários . . . 54
4.1.1.3.2 Vendas . . . 54
4.1.1.3.3 Estoque . . . 55
4.1.1.4 Stream de Mensagem . . . 55
4.1.2 Fluxos de Operações . . . 56
4.1.2.1 Criação de Funcionário . . . 56
4.1.2.2 Criação e Atualização de Produto . . . 56
4.1.2.3 Atualização de Produto . . . 56
4.1.3 Ambiente Kubernetes . . . 56
4.1.4 Mensageria . . . 56
4.2 Modelagem de Segurança . . . 57
4.3 Backlog das *Sprints* . . . 60

5 IMPLEMENTAÇÃO DOS REQUISITOS DE SEGURANÇA . . . 62
5.1 Desenvolvimento seguro a nível Arquitetural . . . 62
5.2 Análise da Implementação dos Requisitos de Segurança . . . 62
5.2.1 Service Mesh Linkerd . . . 63
5.2.2 Observabilidade com Dashboards do Linkerd . . . 63
5.2.3 Ingress Nginx . . . 65
5.2.4 Resumo dos Tempos de Implementação . . . 65
5.3 Desenvolvimento seguro no Serviço de RH . . . 66
5.4 Segurança no Serviço de Vendas . . . 67
5.5 Segurança no Serviço de Estoque . . . 69

5.6 Considerações sobre o Desenvolvimento 70
6 CONCLUSÃO 72
REFERÊNCIAS 73

1 Introdução

1.1 Contexto

A arquitetura de um *software* tem importância na interligação entre suas funcionalidades e seus requisitos não funcionais (DRAGONI et al., 2017). As arquiteturas são formas de compor sistemas para satisfazer diferentes requisitos. À medida que os sistemas crescem em complexidade, surgem, também, modelos de arquitetura que buscam solucionar os problemas relacionados.

O mercado da tecnologia evolui com a popularização da conexão via internet, que cria novas demandas de *softwares* cada vez mais robustos e complexos. A internet permite a criação de Sistemas Distribuídos (TANENBAUM; STEEN, 2016) para atender às novas necessidades. O objetivo desse tipo de sistema é facilitar o compartilhamento e acesso dos recursos de um sistema aos usuários, livremente e de forma **escalável**.

Um exemplo de sistema distribuído é a (*World Wide Web* - WWW) (TANENBAUM; STEEN, 2016), que se comporta como uma coleção de documentos acessíveis por um endereço, que se conectam reciprocamente por meio de *hiperlinks*, parecendo um único servidor. Ela é composta, porém, de um conjunto de servidores que disponibilizam seus documentos separadamente.

Os sistemas distribuídos atuam como um único sistema aos olhos dos usuários, mas são uma composição de sistemas interagindo e se comunicando. Essa é a chamada **Transparência** do sistema (TANENBAUM; STEEN, 2016). A transparência de um sistema distribuído pode ser garantida em diferentes níveis, como revela Tanenbaum e Steen (2016), considerando o que o sistema fonte esconde do usuário:

- **Acesso**: Escondem diferenças na representação dos dados;
- **Localização:** Esconde onde está localizado;
- **Migração:** Esconde a possibilidade de mobilidade de um recurso;
- **Realocação:** Esconde que um recurso foi movido para outra localização;
- **Replicação:** Esconder que está replicado;
- **Concorrência:** Esconder que um recurso pode ser compartilhado por vários usuários;
- **Falha:** Esconder que etapas de recuperação e falhas de um recurso;

Como um tipo de sistema distribuído temos as Arquiteturas Orientadas a Serviço (*Service Oriented Architectures* - SOA). Podem ser definidas como um paradigma para lidar com processos rodando em diferentes sistemas com tecnologias heterogêneas, sendo integrados mediante o uso de mensagens (DRAGONI et al., 2017).

Os principais atributos de tal arquitetura podem ser enumerados, segundo Yarygina e Bagge (2018):

- **Serviços:** Representam uma capacidade da empresa, autocontida e independente;
- **Interoperabilidade:** Os serviços se comunicam através da estrutura da empresa;
- **Baixo acoplamento:** O baixo acoplamento é necessário para possibilitar atributos de escalabilidade, disponibilidade e flexibilidade necessários aos serviços.

Pode-se perceber que a SOA é uma forma de granular os conceitos de Sistemas Distribuídos, para implementar funções específicas do negócio da organização.

Uma evolução do conceito de SOA, que é presente no mercado e estudada pela academia (BOGNER et al., 2019) são os Microsserviços.

Alguns autores consideram as definições quase idênticas (ZIMMERMANN, 2017), sendo os microsserviços apenas diferentes em conceitos adicionais que não envolvem a arquitetura em si, mas sim processos e a cultura de desenvolvimento. Um exemplo de prática relativa à cultura organizacional de microsserviços seria o conceito de priorizar produtos e não projetos. Outros autores já tentam distinguir os microsserviços da SOA (DRAGONI et al., 2017).

Fato é que casos de adoção da arquitetura de microsserviços na indústria, como nas empresas *Netflix* e *Amazon*, chamaram a atenção e motivaram sua popularização (ZIMMERMANN, 2017).

O termo Microsserviços surgiu em 2011 (DRAGONI et al., 2017) e desde então foi utilizado pelas empresas, ou ao menos conceitos muito semelhantes, como *Fine Grained SOA* (SOA Granular, em tradução livre) pela *Netflix*. A academia tem se dedicado a estudar os benefícios e desafios dessa arquitetura, refletindo-se no crescimento da publicação de artigos sobre o tema em 2015 (FRANCESCO; LAGO; MALAVOLTA, 2019).

Benefícios como escalabilidade e baixo-acoplamento são percebidos com a utilização de microsserviços, no entanto, desafios também surgem, como questões de desempenho e segurança dos sistemas.

A segurança no desenvolvimento de *Software* e o desenvolvimento de *Softwares* seguros são uma preocupação maior dos desenvolvedores, no momento em que a conexão remota entre sistemas se torna possível, tendo sido ampliada com o advento da internet e com uma maior conectividade e criticidade dos sistemas (SHUKLA et al., 2022).

A segurança é um atributo fundamental da qualidade de *Software* (NIAZI et al., 2020), tendo crescido em importância nas últimas décadas, em que se percebe o surgimento de sistemas com operações críticas de segurança, como a realização de operações bancárias pela internet.

O ambiente cada vez mais complexo também facilita o surgimento de problemas (FUJDIAK et al., 2019), inserção de erros que causam mau funcionamento e exposição de vulnerabilidades. Toda a extensibilidade do *software* facilita sua exposição a ataques maliciosos.

Para endereçar esses problemas, a segurança deve ser considerada em todo o ciclo de vida do *software*, o conceito metodológico chamado Ciclo de Vida de Desenvolvimento de *Software* Seguro (*Secure Software Development Life-Cycle - SSDL*) (FUJDIAK et al., 2019). Esse conceito faz frente à percepção geral das organizações de que a segurança é algo a ser atacado ao final do ciclo de desenvolvimento de um produto de *software* (NIAZI et al., 2020), através de versões de *patch*.

Pode-se perceber a importância do Desenvolvimento Seguro já que grandes organizações propõem Ciclos de Vida de Desenvolvimento Seguro. Um exemplo é a organização *SAFECode*, que propõe guias como o Fundamentos de Desenvolvimento de *Software* Seguro (SAFECODE, 2018). Possui, inclusive, guias práticos voltados para o Desenvolvimento Ágil de *Software* (ASHTANA et al., 2012), adaptando os conceitos para Histórias de Usuário em processos iterativos.

Essas preocupações se perpetuam em outras áreas do desenvolvimento de *Software* e são influenciadas pelas escolhas durante o desenvolvimento. Não difere para as escolhas de arquitetura. Diferentes arquiteturas trazem diferentes preocupações de segurança.

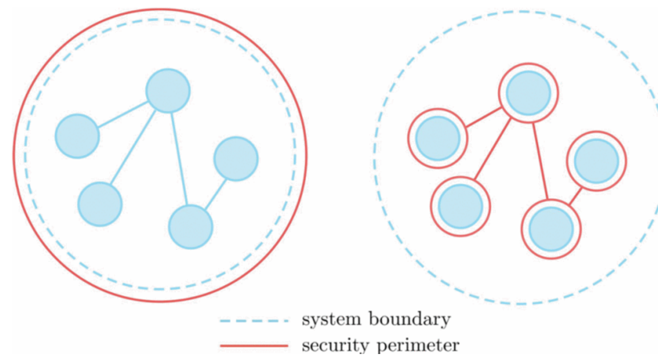
1.2 Problema

O interesse da academia pela temática da segurança em microsserviços tem crescido (Pereira-Vale et al., 2019), já que a arquitetura traz novas maneiras de lidar com os desafios específicos, bem como novos problemas de segurança e vulnerabilidades.

Os microsserviços possuem especificidades em como lidar com a segurança, dada a sua capilaridade e alto grau de comunicação entre os serviços, já que cada microsserviço cuida de uma área de negócio específica.

Uma visão de segurança em profundidade deve ser aplicada para que cada microsserviço tenha a sua segurança e sua independência garantida, ou seja, para que um microsserviço comprometido não consiga afetar os outros no sistema (YARYGINA; BAGGE, 2018). A visão difere da segurança de perímetro mais comum para aplicações monolíticas. A Figura 1 demonstra essa visão representando os limites de segurança mais próximos de

Figura 1 – Segurança em profundidade na visão dos microsserviços.



Fonte: (YARYGINA; BAGGE, 2018)

cada módulo do sistema, que na arquitetura de microsserviços seriam os próprios microsserviços.

Na visão à esquerda, existe uma abrangência geral, sem profundidade, ou seja, a segurança dos serviços de forma independente não é garantida. Já à direita, um sistema com segurança em profundidade é representado. Cada serviço e sua comunicação tem um perímetro de segurança.

Com isso em mente, as perguntas levantadas são:

Como aplicar as práticas de desenvolvimento seguro para aumentar a segurança de aplicações de microsserviços? Quais são os conhecimentos e nível de esforço necessários?

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo deste trabalho é utilizar práticas de desenvolvimento seguro de software, no contexto de um sistema de microsserviços, mapeando as principais vulnerabilidades, ferramentas e métodos específicos aplicáveis aos microsserviços e, com isso, analisar o esforço empregado e os conhecimentos aplicados ao processo de desenvolvimento.

1.3.2 Objetivos Específicos

Para o alcance do objetivo geral, foram definidos os seguintes objetivos específicos:

- Propor um produto de Software para ser construído utilizando a arquitetura de microsserviços;
- Utilizar uma metodologia ágil para desenvolvimento de software;

- Integrar boas práticas para desenvolvimento de Software Seguro mais adequados para a arquitetura de microsserviços;
- Avaliar o esforço e os conhecimentos necessários para a aplicação dessas práticas no escopo dos microsserviços.

1.4 Metodologia

A metodologia de desenvolvimento deste trabalho foi definida e classificada e um plano metodológico foi estruturado.

1.4.1 Classificação Metodológica

A pesquisa é de natureza aplicada, pois foi proposta uma solução prática para um problema existente, utilizando metodologias e ferramentas já propostas pela literatura.

A abordagem é qualitativa, utilizando uma análise de percepção dos processos e do produto entregue.

A tipologia é descritiva, pois foi fundamentada na descrição e caracterização de metodologias e sistemas.

A técnica empregada será o estudo de caso, através do desenvolvimento de um sistema, apoiado pela pesquisa bibliográfica.

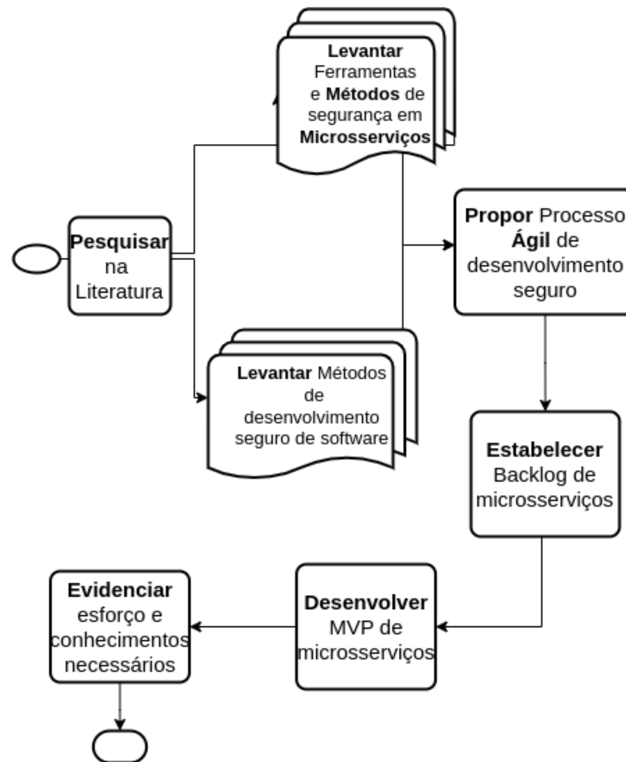
1.4.2 Plano Metodológico

A metodologia geral do trabalho está representada na Figura 2.

Para a técnica do Estudo de Caso, foi utilizado um protocolo baseado em Brereton et al. (2008). O protocolo define itens a serem desenvolvidos em quatro etapas base:

- **Planejamento de Pesquisa:** Nesta etapa se expõe os objetivos do trabalho, a revisão bibliográfica, a pergunta de pesquisa e o plano de pesquisa.
- **Coleta dos Dados:** Utilizam-se técnicas de revisão documental e bibliográfica, além do estudo de caso para adquirir dados.
- **Análise dos Dados:** Os resultados são analisados quantitativamente e qualitativamente, em conjunto com a validade do trabalho.
- **Relatório:** No contexto deste trabalho, é a própria redação da monografia, contemplando os resultados alcançados.

Figura 2 – Esquema da metodologia proposta para o trabalho.



Fonte: Autor

Para cumprimento desses itens, no presente trabalho, são apresentadas 9 etapas das 11 apresentadas por Brereton et al. (2008) :

1. **Background:** Realizado nos capítulos 1 - Introdução e 2 - Referencial teórico.
2. **Design:** O tipo do estudo é um estudo de caso único. O objeto de estudo é o referido nos objetivos gerais e específicos já relatados, ou seja, o desenvolvimento de um sistema de microsserviços com requisitos de segurança.
3. **Seleção de Caso:** A especificação do caso advém dos objetivos relatados anteriormente nos objetivos, com a implementação de um sistema de microsserviços. A especificação do produto desenvolvido está na Seção 3.1.
4. **Procedimentos e papéis:** O autor do presente trabalho é o desenvolvedor dos microsserviços e da arquitetura, que servem como o estudo de caso. Foi escolhida uma metodologia ágil de desenvolvimento, baseada em histórias de usuário. A especificação do processo de desenvolvimento se encontra no Seção 3.2.
5. **Coleta de dados:** Realizada durante o desenvolvimento dos serviços propostos, computando os requisitos de segurança de cada microsserviço e o esforço de implementação medido em horas.

6. **Análise:** Compreende a comparação dos dados de esforço, em horas, coletados na etapa anterior.

7. **Validade do Plano:** São expostas no Capítulo 3.

8. **Limitações do Estudo:** São expostas no Capítulo 6.

9. **Relatórios:** Compreendem os relatos nos Capítulos 4 e 5.

1.5 Organização do Trabalho

Este trabalho possui seis capítulos. Este, Capítulo 1 - Introdução, compreende a introdução do trabalho, com a contextualização junto ao problema, objetivos de pesquisa, a metodologia e o plano metodológico. O Capítulo 2, aborda o Referencial Teórico com a exposição de conceitos utilizados para o desenvolvimento deste trabalho. O Capítulo 3 descreve o produto a ser desenvolvido e analisado durante o estudo de caso. O Capítulo 4, apresenta a Iniciação do projeto e as etapas de modelagem de segurança e de desenho da arquitetura da solução. O Capítulo 5 relata da implementação dos requisitos de segurança e do esforço exigido. Por fim, o Capítulo 6 é a conclusão deste trabalho, com considerações sobre a validade do estudo de caso e sobre as limitações do estudo.

2 Revisão Bibliográfica

2.1 Considerações Iniciais do Capítulo

Este capítulo contém as bases conceituais utilizadas neste trabalho. A pesquisa bibliográfica foi centrada em princípios de Cibersegurança e Segurança de Sistemas, além de conceitos de desenvolvimento seguro de *software* e boas práticas de desenvolvimento para a segurança. Em seguida, conceitos de microsserviços e suas especificidades frente a outras arquiteturas, bem como problemas de segurança associados a essa arquitetura. Finalizando, apresenta-se uma metodologia de desenvolvimento para um único desenvolvedor.

2.2 Cibersegurança

A Cibersegurança é uma área abrangente para a segurança de sistemas da informação. Pode ser definida como a proteção de sistemas informacionais (abrangendo software e hardware) e seus dados e serviços, contra mau-uso, ameaças e acesso não-autorizado (RASHID et al., 2021).

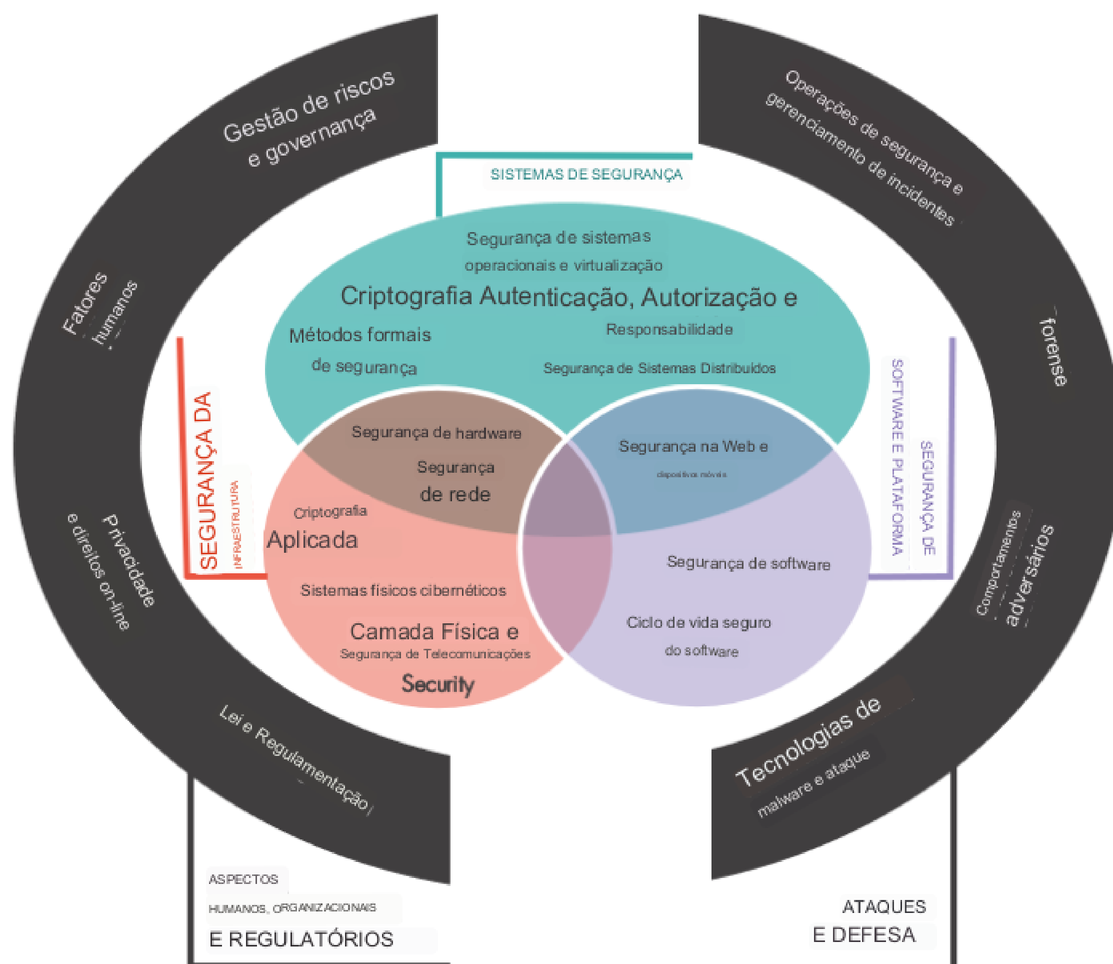
2.2.1 *CyBOK*

O Corpo de Conhecimento da Cibersegurança (RASHID et al., 2021)(*Cybersecurity Body of Knowledge* - CyBOK) é um guia que visa definir um corpo de conhecimento sobre o tema de cibersegurança, utilizando os principais trabalhos acadêmicos e do mercado.

- **Áreas de Conhecimento:**

  O CyBOK define 21 áreas de conhecimentos relacionadas a cibersegurança, para lidar com a multitude de aspectos que afetam a segurança, que vão desde questões humanas e de processos de desenvolvimento passando por ferramentas de software e questões técnicas dos sistemas. A Figura 3 representa a relação dessas áreas com a cibersegurança e suas intersecções.

Figura 3 – 21 áreas de conhecimento da cibersegurança.



Fonte: Adaptado de Rashid et al. (2021)

- **Princípios:**

  Um conceito relacionado com a cibersegurança, e que é um de seus aspectos, é o de Segurança da Informação (ISO, 2018 apud RASHID et al., 2021). A segurança da informação pode ser definida como a preservação da confidencialidade, disponibilidade e integridade da informação. Segundo Khan et al. (2022), confidencialidade é disponibilizar informação para os que possuem autorização para o acesso a essa informação. Integridade significa que o sistema realiza suas funções de maneira não prejudicada. Disponibilidade significa que sistema se mantém funcionando e não é negado aos usuários.

  Saltzer e Schroeder (1975) enumeram oito princípios para a engenharia de controle de segurança. São diretrizes para guiar o projeto e desenvolvimento de sistemas seguros e melhor garantir a Confidencialidade, Integridade e Disponibilidade dos sistemas. São eles:

1. **Princípio do menor privilégio:** Os programas devem operar com os privilégios mais baixos necessários para realizar suas funções, a fim de limitar possíveis danos causados por erros ou ataques.
2. **Princípio da separação de privilégios:** O acesso a recursos sensíveis deve ser controlado e limitado apenas a entidades autorizadas, evitando que uma única entidade tenha privilégios excessivos.
3. **Princípio da defesa em profundidade:** Várias camadas de proteção devem ser implementadas para resguardar os sistemas, com mecanismos de segurança em diferentes níveis, desde a rede até o aplicativo.
4. **Princípio da simplicidade:** Os sistemas de segurança devem ser projetados de maneira simples e compreensível, facilitando a detecção de falhas e reduzindo a probabilidade de introdução de vulnerabilidades.
5. **Princípio da abertura:** O design de segurança deve ser aberto à avaliação pública, permitindo que especialistas identifiquem falhas e contribuam para melhorias.
6. **Princípio do controle completo:** O controle de acesso a recursos sensíveis deve ser abrangente e detalhado, baseado em políticas de segurança claramente definidas.
7. **Princípio da menor surpresa:** As medidas de segurança devem ser previsíveis e consistentes, evitando surpresas para os usuários e tornando as políticas de segurança compreensíveis.
8. **Princípio da recuperação:** Os sistemas devem ser projetados para permitir a rápida e eficiente recuperação após uma violação de segurança, com capacidade de restaurar o estado anterior e minimizar os danos.

- **Fatores humanos:**

  Outro ponto de destaque dentre as áreas atreladas à cibersegurança são os humanos que interagem direta ou indiretamente com esses sistemas.

  Tanto fatores individuais e de comportamento quanto a interação dos indivíduos em um contexto social, são significantes e devem ser considerados nos modelos e designs de segurança (RASHID et al., 2021). Há também a influência de aspectos gerais das capacidades intelectuais e físicas dos seres-humanos que devem guiar sistemas seguros e utilizáveis.

  Saltzer e Schroeder (1975) estabelecem, dentre seus princípios para o design de segurança, que os mecanismos de segurança devem ser psicologicamente aceitáveis pelas pessoas. De nada adianta um sistema seguro se as pessoas não desejam ou não são aptas a usá-los. Cada indivíduo, bem como cada organização, em um contexto de mercado, deve ser responsável pela segurança.

Esses problemas não ocorrem somente com os usuários. Os próprios desenvolvedores não se preocupam naturalmente com aspectos de segurança e precisam ser rememorados destes e, mesmo quando o são, não costumam utilizar as melhores práticas e ferramentas *CyBOK* (NAIAKSHINA et al., 2017 apud RASHID et al., 2021). Pode-se perceber a relevância de uma cultura de desenvolvimento seguro no decorrer de projetos de desenvolvimento.

- ***Malwares* e Tecnologias de Ataque:**

  Malwares são softwares e códigos criados com propósitos de ataques e outras operações de ciber-ataques (RASHID et al., 2021). *Malware* é abreviação para *Malicious Software* (Software Malicioso) e pode ser tanto Software quanto código malicioso. Deve-se conhecer as formas de ataque para se preparar corretamente para as formas de ataque, que vem se sofisticando à medida em que as tecnologias e métodos de segurança também evoluem (RASHID et al., 2021).

  Os ataques maliciosos podem executar diversos tipos de atividades (RASHID et al., 2021) que prejudicam diferentes aspectos da cibersegurança.

  1. **Confidencialidade:** os programas maliciosos podem roubar dados valiosos como dados de autenticação ou dados financeiros.
  2. **Integridade:** Modificação de dados ou injeção de dados falsos.
  3. **Disponibilidade:** podem tornar programas indisponíveis através do desvio do uso dos recursos dos computadores para outras atividades, congestionamento de tráfego ou serviços, ou criptografar dados tornando-os inacessíveis.

  Existem vários tipos de *Malwares*. Segundo Rashid et al. (2021), eles podem ser classificados em 6 diferentes dimensões.

  A **primeira** dimensão diferencia entre *malwares* executados de forma independente e os que são injeção de instruções ou códigos em aplicações terceiras.

  A **segunda** diferencia os que persistem na memória secundária dos que somente existem em tempo de execução das aplicações ou do sistema operacional (e são perdidos ao parar ou desligar processos ou dispositivos).

  A **terceira** considera *malwares* persistentes de acordo com o nível do sistema em que ele se instala, como no firmware, ou no *kernel* ou em aplicações no espaço de usuário.

  A **quarta** diferencia os que são executados automaticamente dos que necessitam de alguma ação do usuário.

  A **quinta** separa em *malwares* estáticos e atualizáveis. Podem ser atualizados por meio de um servidor e se tornam mais difíceis de combater.

A **sexta** e última dimensão contrasta entre os que agem individualmente e os que fazem parte de uma rede coordenada de ataques.

O conhecimento sobre os *Malwares* tem sua relevância para que se possa estabelecer técnicas de identificação de atividades maliciosas e formas de se responder a elas (RASHID et al., 2021).

- **Segurança de Sistemas Distribuídos:**

  A segurança no contexto de sistemas distribuídos relaciona-se principalmente à forma como a superfície de ataque dos serviços é extensa (RASHID et al., 2021), considerando os fluxos de dados e comunicação, o controle de acessos, os mecanismos de transporte dos dados entre os serviços, os mecanismos internos de controle e coordenação entre os serviços e as próprias aplicações construídas consumindo os serviços.

  O contexto específico dos sistemas distribuídos enseja em ameaças e vulnerabilidades específicas. Vulnerabilidades são características que tornam possível a exposição ou comprometimento de um sistema e ameaças revelam a probabilidade de um sistema ser comprometido.

  Maiores noções sobre segurança da arquitetura de sistemas distribuídos de microsserviços foram expandidas na seção 2.3.

2.2.2 Desenvolvimento Seguro de *Software*

Historicamente, os aspectos de segurança dos softwares são verificados de forma reativa, ou seja, o desenvolvimento do sistema e as falhas são descobertas após a ocorrência de ataques (RASHID et al., 2021).

Essa estratégia é ineficiente (MCGRAW, 1998) já que a ocorrência de falhas de segurança é custosa para as organizações e a utilização de atualizações de segurança pode inserir novos erros nas aplicações. Além disso, muitas vezes, as atualizações para maior segurança não são utilizadas pelos usuários.

Para tanto, deve-se entender a diferença de segurança de aplicação e segurança de software: segurança de software é construir software seguro, garantir sua segurança e educar sobre a incorporação de segurança. Segurança de aplicação protege o software e sistemas pós-desenvolvimento (MCGRAW, 1998).

Na última década, práticas de Ciclo de Vida de Desenvolvimento de Segurança (*Security Development Lifecycle* - SDL) tem mudado o foco da segurança de aplicação para a segurança de software (RANSOME et al., 2014). Empresas afetadas por defeitos de segurança de software, como a Microsoft, passaram a se preocupar com a incorporação de

segurança no código, melhorando as práticas de desenvolvimento de software. Surgiram então práticas recomendadas de SDL provenientes da academia e dessas empresas.

- **Ciclo de Vida Seguro de Software:**

  Existem diferentes processos que estabelecem etapas e boas práticas do desenvolvimento seguro de software. O restante da seção 2.2.2 abordará dois métodos de SDL bem estabelecidos. O *Microsoft Security Development Lifecycle* (SDL) e o *SAFECode Fundamental Practices for Secure Software Development*.

  São processos prescritivos, que recomendam práticas e cobrem variadas etapas do ciclo de vida do *software* (RASHID et al., 2021).

- ***Microsoft Security Development Lifecycle - SDL*:**

  O SDL (MICROSOFT, 2023) da Microsoft inclui 12 práticas enumeradas. O processo define um limite mínimo para a conformidade com o SDL. As equipes de desenvolvimento devem adaptar o SDL de acordo com os recursos disponíveis, sem comprometer a segurança organizacional.

  Desde 2004, o SDL é uma política obrigatória na Microsoft, visando incorporar segurança e privacidade no software. Tem em vista reduzir vulnerabilidades ao longo do processo de desenvolvimento e introduzir a segurança nas fases do processo de desenvolvimento.

  As fases do desenvolvimento de software especificadas são Pré-SDL, Requisitos, Design, Implementação, Verificação e Lançamento (*Release*). Para cada fase, existem atividades que precisam ser completadas segundo MICROSOFT (2023):

  1. **Estabelecer requisitos de Treinamento:** Os treinamentos devem envolver os conceitos de design seguro, modelagem de ameaças, código seguro, teste de segurança e privacidade.
  2. **Estabelecer requisitos de segurança:** A análise de requisitos de segurança e privacidade é realizada no início do projeto e inclui a especificação dos requisitos mínimos de segurança para a aplicação, considerando o ambiente operacional planejado, além da especificação e implementação de um sistema de rastreamento de vulnerabilidades de segurança.
  3. **Definir métricas e relatório de conformidade:** Manter Níveis mínimos de qualidade de segurança. As métricas devem ser observadas pelas equipes de engenharia. Ajuda a compreender riscos associados a problemas de segurança, identificar e corrigir defeitos de segurança durante o desenvolvimento e aplicar os padrões ao longo do projeto. marcados com a devida gravidade. Isso possibilita o acompanhamento e relatório precisos do trabalho de segurança.

4. **Realizar a modelagem de Ameaças:** É um processo iterativo que, a partir dos requisitos de segurança, cria o diagrama de ameaças, identifica as ameaças, mitiga-as e valida a sua mitigação. Existem técnicas como o STRIDE (Acrônimo para *Spoofing*, *Tampering*, *Info Disclosure*, *Repudiation*, *Denial of Service* e *Elevation of Privilege*) que é explorado mais à frente neste trabalho.
5. **Estabelecer requisitos de Design:** Deve-se ponderar a seleção de ferramentas e implementações de segurança, tendo noção da segurança que elas fornecem.
6. **Definir e usar padrões de criptografia:** Utilizar, por norma, os padrões de criptografia estabelecidos pelos especialistas na área (já que possuem complexidade) e da maneira recomendada. São importantes para a segurança da comunicação.
7. **Gerenciar o risco do uso de componentes de terceiros:** Para selecionar componentes de terceiros, deve-se entender o impacto que uma vulnerabilidade de segurança advinda pode ter para a segurança do sistema como um todo. Ter um inventário dos componentes de terceiros e um plano de resposta quando novas vulnerabilidades forem descobertas é fundamental.
8. **Usar ferramentas aprovadas:** Utilizar ferramentas (compiladores, analisadores de código etc) bem estabelecidas em suas versões atualizadas.
9. **Realizar testes de análise estática de segurança:** Método flexível e escalável de análise do código antes da compilação. Existem ferramentas automatizadas que podem ser integradas à esteira de ações (*pipeline*) dos projetos.
10. **Realizar testes de análise dinâmica de segurança:** Existem aspectos do sistema que vêm à tona quando todos os componentes estão devidamente rodando e integrados. As ferramentas dinâmicas observam a interação com o sistema operacional (memória, privilégios e afins).
11. **Realizar testes de penetração:** Os testes de penetração são simulações da atuação de entidades maliciosas realizadas por especialistas. O objetivo e encontrar e explorar vulnerabilidades para mapeá-las e mitigá-las posteriormente. Deve ser integrado aos outros métodos de análise.
12. **Estabelecer processo padrão para resposta aos incidentes:** Deve ser criado um plano em conjunto com uma área da empresa dedicada ao gerenciamento de riscos de incidentes. O plano deve contemplar a cadeia de comunicação que deve ocorrer e protocolos de ações de segurança a serem tomadas. Deve considerar tanto ações para códigos-fonte internos a organização quanto para códigos de terceiros. O plano deve ser testado.

- ***SAFECode*:**

  Os guias SAFECode, sigla para *Software Assurance Forum for Excellence in Code* (Fórum de Garantia de Segurança para excelência em código), são criados pela organização homônima. SAFECode é uma organização, sem fins lucrativos, criada em 2007 como uma coordenação entre líderes globais da indústria para remediar a falta de lideranças na indústria sobre a implementação de métodos de desenvolvimento e entrega de software seguro (LICATA, 2023). Um de seus principais guias é o Fundamentos de Desenvolvimento de Software Seguro (SAFECODE, 2018), reconhecido por organismos internacionais de Segurança e pelo próprio CyBOK.

  É um guia para processos de desenvolvimento de *software* seguros, ou seja, no estabelecimento de um ciclo de vida de desenvolvimento seguro (*Secure Development Life Cycle* - SDLC). Contém práticas basilares para o estabelecimento dos ciclo, sendo não exaustivo. São práticas utilizadas na indústria, com eficácia. Estabelece 8 áreas essenciais do ciclo de desenvolvimento seguro (SAFECODE, 2018):

  1. **Definição do controle de segurança da aplicação:** Se relaciona com os requisitos de segurança. As práticas de segurança visam cumprir os controles de segurança definidos, portanto, os controles de segurança são a base do ciclo de vida, iniciando desde o design da aplicação e sendo gerenciado durante as demais etapas. O definição básica de requisitos inclui as seguintes etapas:
     - Identificar ameaças, riscos e requisitos de conformidade enfrentados.
     - Identificar os requisitos de segurança adequados para lidar com essas ameaças e riscos.
     - Comunicar os requisitos de segurança às equipes de apropriadas.
     - Validar a implementação de cada requisito de segurança.
     - Realizar auditorias, se necessário, para demonstrar a conformidade com políticas ou regulamentos aplicáveis (políticas de *compliance*).
  2. **Design:** Deve seguir os princípios de design citados na seção 2.2.1. Para o design, deve ser utilizada a modelagem de ameaças, que pode ter técnicas variadas, como apresentado na metodologia SDL (seção 2.2.2). É importante estabelecer mecanismos de cifração, controle de acesso e identificação, visibilidade e auditoria.
  3. **Práticas de Código Seguro:** A partir das tecnologias e ferramentas definidas, deve-se estabelecer padrões e boas práticas, utilizando as próprias plataformas de desenvolvimento e ferramentas já integradas às tecnologias utilizadas, quando possível. Deve-se utilizar somente bibliotecas e funções sem vulnerabilidades conhecidas. Deve-se aproveitar de analisadores de código automatizados e de revisões de código. É essencial considerar que nenhuma entrada de dados

é naturalmente segura e cada componente da aplicação deve estabelecer mecanismos para se proteger de entradas incorretas ou maliciosas. Alguma das estratégias para isso incluem utilizar codificação específica para os inputs e especificar tipos de dados para as entradas serem interpretadas como tal, dentro do contexto do tipo especificado. O gerenciamento de erros específicos ou não mapeados deve ser realizado, estabelecendo quando a aplicação deve parar, por se encontrar em um estado desconhecido, e com mensagens de erros e registros precisos para a organização e genéricos para o usuário, focando em aspectos de usabilidade.

4. **Gerenciamento de risco de segurança inerente ao uso de componentes de terceiros:** Componentes de terceiros são tratados de forma opaca, com menos controle em comparação com componentes internos. Isso introduz riscos que podem afetar a segurança. Ao selecionar componentes, deve-se dar prioridade a estruturas e bibliotecas comprovadas que fornecem a segurança da qual se necessita. Não se deve tentar reimplementar recursos de segurança já fornecidos, pois há um gasto de recursos e um aumento do risco de introdução de vulnerabilidades no *software*.
5. **Teste e Validação:** Podem ser manuais ou automatizados. Para as ferramentas automatizadas tem-se ferramentas de análise estática e dinâmica do código, além de análise de rede, utilização de testes unitários sobre os requisitos de segurança e a configuração dos modos de análise de segurança de compiladores, interpretadores e plataformas de desenvolvimento. Para os testes manuais tem-se verificações de segurança (como em verificações manuais da qualidade do software) e os testes de penetração.
6. **Gerenciar descobertas de segurança:** Gerenciar, de forma organizada, os riscos e vulnerabilidades encontradas, as falhas de segurança ocorridas e ações de mitigação, remediação ou aceitação dos riscos, durante todo o ciclo de vida do produto. Estabelecer os graus de severidade de problemas de segurança faz-se importante para que todos entendam as prioridades de segurança da organização. Para tanto, o Sistema Comum de Pontuação de Vulnerabilidade (*Common Vulnerability Scoring System* - CVSS) separados em alto, médio e baixo.
7. **Resposta e Divulgação de Vulnerabilidade:** Inclui comunicar-se efetivamente com os *stakeholders* e usuários para auxiliá-los ante a descoberta de vulnerabilidades e formas de ação para mitigação das vulnerabilidades. Necessita-se de políticas internas (para vulnerabilidades divulgadas privadamente) e externas (para vulnerabilidades já publicizadas). Todos na organização devem saber os papéis e responsabilidades durante crises. A severidade da vulnerabilidade e sua importância devem determinar a urgência do processo de conserto.

Possibilidades de mitigação e contorno devem ser verificadas. A ocorrência de problemas deve retroalimentar o SDL através da análise de causa raíz.

8. **Planejando a Implementação e Implantação de Práticas de Desenvolvimento Seguro:** Deve-se considerar os seguintes fatores:
   a) Cultura da organização;
   b) Expertise e nível de habilidade da organização;
   c) Modelo de desenvolvimento e ciclo de vida do produto;
   d) Escopo da implantação inicial;
   e) Gerenciamento de partes interessadas e comunicação;
   f) Medição da eficiência;
   g) Saúde do processo de SDL vigente;
   h) Proposta de valor para justificar as práticas de desenvolvimento seguro.

- **Guia prático *SAFECode* para ambientes Ágeis:**

  A organização *SAFECode* também propõe um guia para equipes e grupos em um contexto ágil (ASHTANA et al., 2012). O trabalho adapta os conceitos de Desenvolvimento Seguro com *SAFECode* para o contexto de Histórias de Usuário e de ciclos curtos de planejamento e implementação.

  Para tanto, um conjunto de 36 Histórias de Usuário é proposto, para os principais requisitos de segurança, com exemplos de tarefas associadas. Os usuários são atores comuns às equipes e organizações ágeis. Especificamente, Arquitetos, Desenvolvedores e *Quality Assurance*.

  O guia também associa um ID de *Common Weakness Enumeration* (CWE), para que se possa ter os problemas de segurança associados com riqueza de detalhes.

  A Tabela 2.2.2 é um exemplo retirado do guia. Pode-se observar as História de Usuário, as tarefas indicadas para cada escopo de usuário, bem como a identificação das práticas do *SAFECode* associadas à história e o ID CWE. Para as tarefas, a letra **A** indica o Arquiteto de *Softwate*; **D** indica o Desenvolvedor e **T** indica o profissional da Área de Qualidade e Testes.

Tabela 1 – Exemplos de Histórias do *SAFECode* - Contém as histórias 3 e 8, com as respectivas tarefas e indicação das práticas do *SAFECode*.

No.	**História focada em segurança**	**Tarefa(s) no Backlog**	**Prática(s) Fundamental(is) do SAFECode**	**CWE-ID**
3	Como arquiteto(a)/desenvolvedor(a), quero garantir e como QA, quero verificar a aplicação apropriada de ou acesso dentro dos limites de índices de buffers e arrays.	**[A/D]** Defina onde ocorrem operações de buffer (em buffers dinâmicos). Defina tipos de dados e limites para operações de buffer. **[D]** Adote as Práticas Fundamentais do SAFECode para Desenvolvimento Seguro de Software para prevenção de estouros de buffer. **[D]** Faça a varredura do código fonte para tais violações usando ferramentas de análise de código estático, por exemplo, Coverity. **[A/D]** Conduza análise de falso positivo de problemas sinalizados. **[D]** Corrija problemas de estouro de buffer analisados e confirmados. **[T]** Use testes de fuzzing para verificar se processos/sistemas travam ou param. Se isso ocorrer, corrija-os e reexecute a ferramenta.	Minimize o Uso de Strings Inseguras e Funções de Buffer Use um Conjunto Atual de Ferramentas do Compilador Use Ferramentas de Análise Estática	CWE-120 CWE-131 CWE-805
8	Como arquiteto(a)/desenvolvedor(a), quero garantir e como QA, quero verificar que os usuários tenham acesso aos recursos específicos que requerem e que estão autorizados a usar.	**[A]** Crie uma matriz de autorização detalhada que especifique quais grupos de usuários/usuários têm acesso a quais recursos (pastas, arquivos, UI, etc.). **[D]** Garanta que o mecanismo de autorização da sua aplicação esteja em conformidade com a matriz criada no passo anterior (por exemplo, se o controle de acesso baseado em função [RBAC] for usado, garanta que ele corresponda à matriz de autorização criada). **[T]** Teste a eficácia utilizando uma combinação de meios manuais e automatizados.	Use Privilégio Mínimo	CWE-862 CWE-863

Fonte: Adaptado de (ASHTANA et al., 2012).

2.3 Segurança de Microsserviços

A arquitetura de Microsserviços tem relação com as SOAs, porém, possuindo todos os seus componentes como microsserviços (DRAGONI et al., 2017). E o microsserviço é, segundo os mesmos autores, um processo enxuto e independente, que utiliza mensagens para se comunicar com outros processos.

Os microsserviços se tornam atrativos para o mercado por conta de certos atributos que dão alternativas a problemas de aplicações monolíticas (DRAGONI et al., 2017). Eles facilitam a Manutenibilidade por terem um escopo menor (portanto, menor chances de inserção de erros e maior isolamento e testabilidade). Também são escaláveis, pois não há necessidade de parada ou reinício completo de um sistema para modificar um conjunto de seus módulos. Isso, aliado à questão dos microsserviços serem naturalmente passíveis de criação de contêineres, faz com que os microsserviços facilitem a integração contínua e a implantação contínua dos serviços (DRAGONI et al., 2017).

Apesar das facilidades, as arquiteturas em microsserviços também trazem novos desafios. Os sistemas acabam por aumentar o tamanho de sua arquitetura por meio de seus componentes e, consequentemente, sua complexidade (SOLDANI; TAMBURRI; HEUVEL, 2018).

A consistência dos dados também é uma preocupação considerando a interação com múltiplas bases de dados e a maior quantidade de mensagens entre os processos, causando maior probabilidade de inconsistência com a transmissão dos dados. Preocupações sobre o desempenho, devido ao gargalo da quantidade de comunicações e requisições, muitas vezes feitas utilizando a internet, também são constantes na indústria e da academia (SOLDANI; TAMBURRI; HEUVEL, 2018).

Há também uma dificuldade de monitoramento dos serviços (CHANDRAMOULI, 2019), que exige um sistema de monitoramento distribuído, mas que necessita de visualização centralizada.

Testes de integração se tornam mais complicados, por ser necessário configurar um ambiente de teste no qual os microsserviços estão presentes e consigam se comunicar (CHANDRAMOULI, 2019).

Outra questão recorrente vista como uma dificuldade na arquitetura de microsserviços é a garantia da segurança do sistema e de seus componentes (BERARDI et al., 2022).

A complexidade inserida pelos sofisticados sistemas de implantação e integração contínuas e a maior superfície de ataque trazem diferentes dificuldades a serem superadas nos requisitos de segurança. Ao se utilizar a comunicação via chamadas de API, os processos e ferramentas para comunicação segura para APIs devem ser utilizados (CHAN-

DRAMOULI, 2019). Os microsserviços tendem a ser mais expostos do que um modelo de servidor monolítico, pois o que antes era uma camada mais robusta entre a camada do usuário e os dados no banco de dados, se transforma em serviços granularizados (CHANDRAMOULI, 2019). Com isso, tem-se a importância de desenhar e implementar cada microsserviço de maneira segura.

Pode-se analisar a abordagem de segurança em microsserviços por alguns aspectos mostrados em seguida.

2.3.1 Modelagem de Ameaças

Há uma falta de estudos relacionados a modelagem de ameaças em microsserviços. Uma hipótese levantada por Berardi et al. (2022) é pela pesquisa de segurança em microsserviços ser, em sua maioria, realizada por pesquisadores da área de Engenharia de Software, que não possuem, em sua maior parte, o foco da segurança pelo design e requisitos que os especialistas em segurança possuem. Os modelos de ameaças para microsserviços tendem a ser específicos para os casos analisados nos estudos e são menos generalizáveis.

Contudo, ameaças comuns aos serviços web continuam sendo preocupações para os microsserviços (CHANDRAMOULI, 2019) como ataques de injeção, codificação e serialização. Dentre elas, pode-se mencionar (OWASP, 2021c):

- **Injeção:** Falhas de injeção ocorrem quando um aplicativo envia dados não confiáveis para um interpretador (OWASP, 2021b). São comuns em códigos legados. Exemplos de operações passíveis de injeção são consultas SQL[1](*Structured Query Language* – Linguagem de Busca Estruturada), consultas XPath[2] (*XML Path Language*) comandos de sistema operacional e argumentos de programa.
  - **Diagnóstico:** A forma mais simples de descobrir falhas é ao examinar o código e fazer revisões. *Scanners* podem ajudar os atacantes a encontrá-las. Pode-se, também, testar a validação de inputs e parâmetros de url para verificar a execução de scripts.
  - **Proteção:** A melhor abordagem é corrigir o problema no código-fonte ou redesenhar o sistema. Pode-se utilizar plataformas de desenvolvimento com *Statements* (Declarações) preparadas para validar entradas para as buscas ou o uso de procedimentos do próprio banco de dados (*Stored Procedures*). Em último caso, deve-se escapar (tornar literal) todos os caracteres especiais de um input e validá-lo.

[1] Para mais informações sobre o SQL e XPath: https://www.w3schools.com/sql/
[2] Cf. Nota de Rodapé 1.

- **Desserialização:** A serialização consiste em codificar dados da aplicação em um outro formato para ser salvo ou enviado para outro sistema ou para alguma base de dados (OWASP, 2021a). Desserialização envolve a reconstrução dos dados em um objeto. O formato mais utilizado para a serialização de dados é o JSON (*Javascript Object Notation*). Os mecanismos de desserialização nativos das linguagens podem ser explorados ao lidar com dados não confiáveis, o que acarreta riscos de segurança. Ataques a desserializadores podem resultar em negação de serviço, acesso não autorizado e execução remota de código (RCE).
  - **Diagnóstico:** Verificar as bibliotecas de desserialização sendo utilizados e vulnerabilidades conhecidas. Utilizar análise estática de código para mal-cheiros das bibliotecas.
  - **Proteção:** Utilizar formatos padrões de arquivos para serialização, ou seja, evitar serialização customizada em favor do uso de JSON, XML e similares. Ferramentas de análise estática para detectar uso de APIs de desserialização não confiáveis.
- **Injeção de Script Entre Sites:** XSS (*Cross Site Scripting*) consiste em inserir código malicioso em páginas que são confiáveis (KIRSTENS et al., 2018 2022). Geralmente é utilizado um *Browser* para executar o script. O atacante passa a ter acesso aos dados dos usuários da página pois acredita-se que vem de uma fonte confiável.
  - **Diagnóstico:** Deve-se investigar locais em que há requisições HTTP[3] (*Hypertext Transfer Protocol* – Protocolo de Transferência de Hipertexto) que podem ser inseridas diretamente no código HTML. Existem ferramentas para análise automática, porém uma análise mais detalhista precisa ser feita em conjunto.
  - **Proteção:** As plataformas de desenvolvimento *web* modernas costumam ter mecanismos de prevenção de XSS implementados por padrão. É importante saber as limitações da plataforma de desenvolvimento e utilizar atributos e métodos que façam *encoding* para texto dentro do HTML[4] (Hypertext Markup Language - Linguagem de Marcação de Hipertexto) e não injeção de código HTML. Também deve-se realizar o *encoding* de urls antes de utilizá-las nos atributos HTML.
- **Falsificação de Solicitação entre Sites:** *Cross-Site Request Forgery* (CSRF) ocorre quando um site malicioso ou programa faz com que o navegador de um usuário execute uma ação indesejada em um site confiável enquanto o usuário está

[3] Para detalhes sobre HTML, HTTP e Web: https://developer.mozilla.org/pt-BR/docs/Web
[4] Cf. Nota de rodapé 3

autenticado (OWASP, 2018). Já que as solicitações do navegador incluem automaticamente todos os cookies, se o usuário estiver autenticado no site, o site considera uma solicitação legítima.

- **Diagnóstico:** Deve-se averiguar os mecanismos de autorização e identificação dentre todas as requisições. Os impactos desse ataque dependem do quanto dos mecanismos expostos que a aplicação cliente tem acesso e das autorizações do usuário.
- **Proteção:** Utilizar mecanismos padrões contra CSRF das plataformas de desenvolvimento utilizadas. Caso a plataforma não forneça soluções, deve-se utilizar *tokens* de requisição para requisições que mudem o estado da aplicação. Outras formas de mitigação incluem o uso de atributos que restringem a atuação de cookies e a verificação dos *headers* de origem e destino das requisições.

Existem, porém, ameaças e questões de segurança mais específicas para os microsserviços (CHANDRAMOULI, 2019).

Os **mecanismos de descoberta** dos microsserviços (registro, retirada e descobrimento de microsserviços) estão sujeitos ao registro de serviços maliciosos ou ao redirecionamento, tornando o serviço indisponível ou executando serviços de terceiros (CHANDRAMOULI, 2019).

Os **ataques de internet** tem atuação na comunicação entre os microsserviços (CHANDRAMOULI, 2019). Controles de segurança de uma cadeia de microsserviços podem ser ignorados ao se comunicar diretamente com o microsserviço. Verificações no código se tornam mais complexas, por existirem múltiplos contextos de uso de um microsserviço.

A **falha em cascata** pode ocorrer caso uma transação dependa da ação de algum microsserviço que não está disponível. Isso pode acarretar a falha de vários microsserviços dependentes que, apesar do favorecimento ao baixo acoplamento, um design de sistema dificilmente terá isso de forma absoluta.

2.3.2 Estratégias de Segurança para implementações e Ameças específicas

Algumas estratégias para a implementação segura de requisitos comuns a maioria dos microsserviços e estratégias de combate à ameças específicas são sugeridas por Chandramouli (2019).

2.3.2.1 Estratégias para identificação e gerência de acesso

Os microsserviços costumam estar em formato de API (DRAGONI et al., 2017) e a autenticação, portanto, envolve, geralmente, o uso de chaves de API. Tokens de autentica-

ção codificados em *Security Assertion Markup Language* (SAML) ou através do *OpenID Connect* sob a plataforma OAuth 2.0 costumam ser boas opções (CHANDRAMOULI, 2019).

A autorização necessita da centralização de serviço para a provisão e aplicação de políticas de acesso, devido a maior complexidade de microsserviços, à utilização de APIs e à necessidade de combinar serviços para executar as regras de negócio.

Um método padronizado de transmissão de decisões de autorização através de um token padronizado (por exemplo, *JSON Web Tokens* (JWT) é necessário, pois ele independe de plataforma, o que se adequa a heterogeneidade dos microsserviços.

Uma dificuldade é criada caso se implemente políticas de acesso individualmente em cada microsserviço pois facilita discrepâncias entre as políticas que impactam todo o sistema.

Políticas específicas são implementadas em cada microsserviço ou próximas a ele, o que pode resultar em problemas de desempenho desses nós. Esses problemas se propagam devido a dependência da comunicação em rede entre os nós.

Estratégias de autorização incluem (CHANDRAMOULI, 2019):

1. Uso de *tokens* (chaves de API) com tempo de vida limitado e assinados digitalmente ou credenciados para dados muito sensíveis.
2. Restringir chaves de API para aplicações e APIs específicas.
3. Proporção entre restrição de acesso e garantia de confiança de identidade.
4. *Tokens* com o menor tempo de validade possível e chaves de segurança fora do código fonte.

Já para a gerência de acesso (CHANDRAMOULI, 2019):

1. *Cache* de dados para acesso deve ser usado somente quando o servidor de acesso está inacessível e de expirar.
2. Servidor de acesso deve conseguir granularizar suas políticas.
3. Decisões de acesso codificadas em *tokens*.
4. O escopo de tokens deve incluir somente os endereços necessários para a requisição específica.
5. Deve-se tomar cuidado ao utilizar a autenticação centralizada no gateway. Mecanismos de autenticação mútua e outros mecanismos devem ser utilizados.

2.3.2.2 Estratégias para o registro de microsserviços

Os mecanismos de registro facilitam a descoberta de novos microsserviços e são implementados, geralmente, em um serviço de registro (CHANDRAMOULI, 2019). Para tanto, o microsserviço deve conseguir se registrar e buscar outros nós. Os princípios de confiabilidade, integridade e disponibilidade devem ser seguidos para a descoberta de microsserviços.

A implementação do registro pode ser tanto do lado do cliente, consultando o registro e se comunicando diretamente com os serviços registrados, quanto dentre os microsserviços, utilizando um serviço roteador centralizado ou um servidor resolvedor de *Domain Name System* (Sistema de Nome de Domínio - DNS) implementado para cada microsserviço, sendo o servidor DNS o responsável por buscar no registro e atualizar as rotas para os nós (CHANDRAMOULI, 2019). Ambas as técnicas podem ser utilizadas (para os clientes e entre os microsserviços).

Existem estratégias de segurança comuns a todo o tipo de implementação de registro (CHANDRAMOULI, 2019).

Deve-se garantir a disponibilidade do servidor por meio de uma boa qualidade de serviço de rede.

O servidor deve utilizar protocolos seguros de comunicação segura com a internet (HTTPS ou TLS) e deve validar se os serviços que interagem com ele são legítimos (CHANDRAMOULI, 2019). O registro dos nós deve ser terceirizado, pois caso um nó caia, sua inaptidão para se remover do registro pode afetar todo o sistema.

Arquiteturas de microsserviços grandes devem buscar o uso de servidores de registros distribuídos, tomando cuidado para manter a consistência da comunicação e dos dados do sistema.

2.3.2.3 Estratégias para Protocolos de Comunicação Segura

Os serviços devem utilizar protocolos como SSL (*Secure Socket Layer*) ou TLS (*Transport Layer Security*) (CHANDRAMOULI, 2019). Esses protocolos podem trazer gargalos de desempenho ao sistema que, porém, podem ser minimizados utilizando conexões virtuais que se mantém após serem estabelecidas (conexões *keep-alive*) (FIELDING et al., 1997) e podem ser reutilizadas milhares de vezes para novas requisições entre os mesmos serviços.

Os clientes do serviço devem ter seu acesso aos microsserviços centralizado por meio de *gateways* e devem ser mutuamente autenticadas, assim como conexões entre os serviços.

2.3.2.4 Estratégias para o monitoramento de segurança

O monitoramento deve ser performado no nível do *gateway* e no nível de cada microsserviço, por exemplo, para perceber requisições maliciosas a nível de clientes, validação de entradas, injeção de código e outros (CHANDRAMOULI, 2019). O sistema deve gerir uma riqueza de *logs* para microsserviço. Para lidar com essa complexidade, existem plataformas de desenvolvimento como o *AppSensor* [5].

Painéis de monitoramento centralizado podem ser estabelecidos para monitorar diversos microsserviços, seus comportamentos e suas interrelações.

O monitoramento deve ser guiado por dados base sobre fluxos normais de acesso, repostas normais da aplicação e questões relacionadas para que anomalias sejam percebidas e comparadas aos comportamentos padrões (CHANDRAMOULI, 2019).

2.3.2.5 Estratégias para disponibilidade / resiliência

A resiliência é um dos objetivos da cibersegurança (Seção 2.2.1) sendo especialmente crítico para os microsserviços que dependem de comunicação constante via APIs (CHANDRAMOULI, 2019). Para tanto, estratégias devem ser adotadas para o cumprimento desses objetivos.

Existem perigos relacionados ao comportamento de nós não-confiáveis e o princípio de falha rápida pode ser utilizado para a prevenção de maiores danos (CHANDRAMOULI, 2019) utilizando o mecanismo de *Circuit Breaker*. Esse mecanismo permite que o serviço se comportando de forma inesperada possa ser desligado rapidamente, para que não haja falhas de requisição e inserção de erros em outras partes do sistema. Eles podem ser implementados tanto do lado das aplicações clientes quanto dos microsserviços ou como *proxies* entre microsserviços e aplicações clientes. Outras estratégias podem ser implementadas para a garantia da disponibilidade dos microsserviços (CHANDRAMOULI, 2019):

1. **Estratégias para balanceamento de carga:** O balanceador de carga é um módulo importante nas implementações de microsserviços. Permite um equilíbrio de carga dos serviços usando *namespaces* para identificar os serviços e utiliza algoritmos de escalonamento para acesso aos serviços. O balanceador de carga deve ser desacoplado dos outros serviços do sistema.

2. **Limitação de Taxa:** Visa limitar o número de requisições que podem ser feitas de um mesmo cliente em um período. Deve ser implementado em acordo com a robustez da infraestrutura e da aplicação e das APIs. A limitação de taxa também é uma forma de combate aos ataques de Negação de Serviço Distribuídos (*Distributed Denial of Service*.

[5] Mais sobre: https://owasp.org/www-project-appsensor/

2.3.2.6 Estratégias para garantia da integridade

A integridade precisa ser garantida, principalmente, durante a implantação de novas versões dos microsserviços e na interação com banco de dados durante transações (CHANDRAMOULI, 2019).

A implantação deve ser realizada através de um *gateway*, mantendo a versão nova e a versão antiga e, aos poucos, migrar o tráfego de comunicações para a nova versão, com o processo sendo detalhadamente monitorado. Essa técnica é chamada de Lançamento Canário (*Canary Release*).

A integridade com relação a base de dados e a persistência pode ser garantida através do armazenamento seguro das informações dos serviços clientes, além da separação entre os tokens de acesso ao *gateway* e os tokens de autorização interna entre os microsserviços.

2.3.2.7 Estratégias para *framework* de arquitetura

Geralmente, as arquiteturas de microsserviços são implementadas em *gateways* de API ou um malha de serviços (menos centralizada) (CHANDRAMOULI, 2019).

Para implementar um API gateway de forma segura, deve-se integrá-lo com um gerenciador de identidade para fornecer credenciais antes de ativar a API e fornecer interfaces para o uso de *tokens* de acesso. O tráfego do gateway deve ser monitorado para detectar possíveis anomalias e ataques. No caso de *gateways* distribuídos, deve haver uma granularidade de tokens para os gateways mais próximos dos microsserviços sendo acessados e *tokens* com escopo maior para o *gateway* principal (CHANDRAMOULI, 2019).

Para a malha de serviços deve-se fornecer política de protocolo de comunicação específico entre pares de serviços e especificar a carga de tráfego para os mesmos. Além disso, todos os serviços devem ter políticas de acesso habilitadas.

Deve-se evitar configurações com escalonamento de privilégios, os privilégios devem ser bem estabelecidos e constantes sempre que possível.

Implantações de malha de serviço devem definir limites de uso de recursos para seus componentes. Sem isso exite o potencial de que esses componentes afetem a resiliência e a disponibilidade do aplicativo de microsserviços como um todo, criando um efeito em cascata.

A malha também deve ter recursos de configuração para coletar e enviar métricas de ambiente, incluindo métricas de solicitação, para um serviço centralizado de monitoramento.

2.4 *Kubernetes*

2.4.1 Visão Geral

O *Kubernetes* é uma plataforma *Open-Source* de gerenciamento de *Contêineres* em um ambiente de máquinas distribuídas (KUBERNETES, 2023). Os *contêineres* são uma forma flexível de virtualização de máquina que utiliza os recursos do próprio Sistema Operacional em que está sendo executado, porém, permitindo o isolamento de programas e bibliotecas, criando um ambiente estável de execução (KUBERNETES, 2023).

A plataforma abstrai atributos para um gerenciamento eficiente e seguro dos *contêineres*. Com ela é possível utilizar as máquinas de forma distribuída com uma configuração simplificada das máquinas, utilizando nós de controle (*control-node*), responsáveis pelo servidor de API do *Kubernetes* e por se comunicar com e controlar os outros nós.

A forma distribuída de execução permite, dentre outros atributos, o balanceamento de carga de um serviço, que pode rodar em diferentes máquinas, reduzindo o estresse de uma máquina, além de facilitar a disponibilidade do serviço, pois, caso uma das máquinas venha a falhar, o serviço replicado em outra máquina assume imediatamente a competência de rodar o serviço (KUBERNETES, 2023).

Os *contêineres* também podem ser rapidamente escalonados com um comando ou de forma automática, através de métricas de rede e de uso de recursos. Os *deployments* também conseguem reiniciar os serviços (*self-heal*) em caso de falhas através de checagens pré-definidas (como rotas com respostas específicas para indicar serviço saudável).

Além dessas vantangens, o kubernetes também permite (KUBERNETES, 2023):

- Expor contêineres através de um nome DNS ou IP estático, distribuindo o tráfego entre eles para garantir alta disponibilidade;
- Automatizar a montagem de volumes de armazenamento, tanto locais quanto em nuvem;
- Facilitar a gestão de atualizações de aplicações, alterando somente uma parte da carga para uma nova versão ou possibilitando a reversão em casos de erro.
- Armazenar e gerenciar informações sensíveis, como credenciais de acesso.
- Suporte nativo aos protocolos IPv4 e IPv6.
- Acesso à API do *Kubernetes* com autenticação e autorização utilizando controle de acesso baseado em papéis (Role Based Access Control).

Todas essas configurações são facilitadas através de comandos por uma interface CLI (*Command Line Interface*) ou utilizando arquivos no formato *yaml*, que facilitam a

replicação e reutilização de configurações. Eles interagem com servidor de API do *Kubernetes*.

2.4.1.1 Configuração de Aplicação

Na arquitetura do *Kubernetes* existe o conceito de *Workload*. *Workloads* são quaisquer aplicações que estejam executando em um *cluster Kubernetes*. Essas aplicações são compostas de um ou múltiplos *pods* que interagem para a execução da aplicação.

2.4.1.2 Pods no Kubernetes

Um *pod* no Kubernetes é a unidade básica de computação. É um grupo de um ou mais *contêineres* que compartilham recursos como rede e armazenamento. Os *contêineres* dentro de um *pod* são fortemente acoplados e frequentemente colaboram para executar uma aplicação específica (KUBERNETES, 2024b).

O ciclo de vida de um *pod* passa por diferentes fases:

- **Pending:** O *pod* foi criado, mas ainda não está em execução.
- **Running:** Pelo menos um *contêiner* do *pod* está em execução.
- **Succeeded:** Todos os *contêineres* do *pod* terminaram com sucesso.
- **Failed:** Um ou mais *contêineres* falharam.
- **Unknown:** O estado do *pod* é desconhecido, geralmente devido a problemas de comunicação.

A natureza efêmera dos *pods* é uma característica fundamental do *Kubernetes*. Quando um *pod* falha ou é deletado, o sistema *Kubernetes* é responsável por criar um novo *pod* para garantir a alta disponibilidade das aplicações.

Um mecanismo para gerenciamento dos *pods* é o *deployment*. Eles gerenciam o ciclo de vida dos *pods*. Permitem atualizações declarativas para as aplicações, possibilitando que os usuários definam o estado desejado do sistema (KUBERNETES, 2024a). Entre as principais funcionalidades estão:

- Atualizações contínuas e reversões;
- Capacidade de escalonamento;
- Funcionalidades de pausa e retomada;

Para exposição dos *pods* são utilizados os *services* (AUTHORS, 2024). Eles abstraem os serviços de rede permitindo o uso de um DNS padrão para os serviços. Oferece:

- Balanceamento de carga entre os pods;
- Pontos de acesso de rede estáveis;
- Mecanismos de descoberta de serviços;

2.4.2 *Service Mesh*

O *Service Mesh* constitui uma infraestrutura especializada que otimiza a comunicação entre serviços em arquiteturas de microsserviços (CHANDRAMOULI; BUTCHER, 2020). Esta camada proporciona funcionalidades essenciais, como descoberta de serviços, balanceamento de carga e configuração de tráfego, além de aspectos de segurança e monitoramento. Sua operação ocorre em um nível de abstração superior à camada de transporte do modelo OSI, focando-se nas preocupações da camada de sessão.

A implementação do *Service Mesh* envolve a interceptação e direcionamento do tráfego do *cluster*, possibilitando o monitoramento e a proteção das comunicações entre serviços (CHANDRAMOULI; BUTCHER, 2020). Essa abordagem viabiliza recursos avançados, como testes A/B e implantações canário, fundamentais para a manutenção e evolução de sistemas complexos.

As vantagens do *Service Mesh* tornam-se evidentes em aplicações com um número elevado de microsserviços, proporcionando uma infraestrutura robusta para garantir disponibilidade, escalabilidade e segurança. No entanto, é importante considerar algumas limitações, como o aumento do número de instâncias em execução e a potencial expansão da superfície de ataque da aplicação por conta da maior quantidade de serviços e componentes que são criados para suportar as funcionalidades do *Service Mesh* (CHANDRAMOULI; BUTCHER, 2020).

A arquitetura do *Service Mesh* compreende dois componentes principais: o plano de dados e o plano de controle. O plano de dados consiste em *proxies* interconectados que gerenciam a comunicação entre serviços, enquanto o plano de controle é responsável por configurar e controlar o comportamento do plano de dados.

As funções suportadas pelo *Service Mesh* abrangem autenticação, autorização, descoberta segura de serviços, comunicação segura com suporte a TLS mútuo, e recursos de resiliência. Adicionalmente, oferece funcionalidades de observabilidade, como registro de *logs* e rastreamento distribuído, essenciais para o monitoramento e manutenção de sistemas distribuídos complexos (CHANDRAMOULI; BUTCHER, 2020).

Com o TLS mútuo($mTLS$), os serviços se comunicam utilizando os proxies, com mensagens encriptadas, sem que os serviços e a lógica de negócio tenha que ser modificada e sem que certificados de encriptação tenham que ser gerenciados pelas aplicações, para cada comunicação entre serviços.

2.4.2.1 *Linkerd*

O *Linkerd* é uma *service mesh* desenvolvida para Kubernetes, cujo objetivo é facilitar a execução de serviços com maior segurança e eficiência (LINKERD, 2024b).

Seu plano de dados é formado por *micro-proxies* transparentes, que operam como *sidecars* nos *pods*.

O funcionamento do *Linkerd* baseia-se na instalação desses *micro-proxies* leves e transparentes junto a cada instância de serviço. Desenvolvidos em *Rust*, esses *proxies* são otimizados para proporcionar alto desempenho e segurança, gerenciando automaticamente todo o tráfego de entrada e saída dos serviços, funcionando como pilhas de rede altamente instrumentadas (LINKERD, 2024b).

O plano de controle inclui componentes como o serviço de destino, o serviço de identidade e o injetor de *proxy* (LINKERD, 2024a). O serviço de destino fornece informações sobre a descoberta de serviços e políticas para os *proxies* do plano de dados. O serviço de identidade, por sua vez, atua como uma Autoridade Certificadora *TLS*, emitindo certificados para assegurar comunicações seguras entre os *proxies*. O injetor de *proxy* é um controlador de admissão do Kubernetes que adiciona contêineres *proxy* aos *pods* conforme necessário.

No plano de dados, o *Linkerd2-proxy*, um *micro-proxy*, oferece funcionalidades como *proxying* transparente para protocolos *HTTP* e *TCP*, exportação automática de métricas para *Prometheus*, balanceamento de carga sensível à latência e *TLS* automático.

A implementação do *Linkerd* em um *cluster* Kubernetes envolve a instalação do plano de controle, seguida pela injeção do plano de dados nos *workloads*, em um processo denominado "*meshing*". Esta abordagem permite que o *Linkerd* forneça suas funcionalidades de *service mesh* sem impactar significativamente o desempenho das aplicações (LINKERD, 2024a).

Além dessas funcionalidades básicas, o *Linkerd* também oferece vantagens adicionais, como uma interface de usuário amigável e suporte para integração com outras ferramentas de observabilidade e monitoramento. No *Linkerd*, é possível a injeção dos *proxies* com simples comandos *CLI* ou anotações em arquivos de configuração de *deploy* do *Kubernetes*.

2.5 *Mensageria*

2.5.1 Visão geral

As filas de mensagens são um método de *Inter-Process Communication* (IPC), possibilitando a comunicação assíncrona entre processos. Nessa abordagem, múltiplos cli-

entes podem se conectar a uma fila de mensagens para publicar ou consumir mensagens (SCHUMETH, 2024). A fila funciona como um *buffer*, permitindo que os processos enviem e recebam mensagens em momentos distintos. Essa característica proporciona maior flexibilidade e robustez ao sistema, especialmente em ambientes distribuídos, onde os processos operam de forma independente.

2.5.2 Protocolos de Mensagens

Os protocolos de mensagens são conjuntos de regras que estabelecem a troca de mensagens entre sistemas ou aplicações distintos. Esses protocolos garantem a transmissão de mensagens, possibilitando uma comunicação eficiente entre diferentes componentes (SCHUMETH, 2024). Compreender esses protocolos permite entender o comportamento dos *message brokers* que os utilizam.

2.5.2.1 *Advanced Message Queuing Protocol (AMQP)*

O *Advanced Message Queuing Protocol* (AMQP) é um padrão de envio de mensagens entre aplicações ou sistemas. Sua arquitetura inclui um *AMQP broker*, representado por um servidor que gerencia e roteia mensagens (SCHUMETH, 2024). Os clientes publicam mensagens para o *broker*, que as encaminha para as filas por meio de *exchanges*. As filas armazenam as mensagens até que sejam coletadas pelos clientes assinantes.

O protocolo suporta comunicação segura através da criptografia de mensagens e permite que um único cliente atue tanto como publicador quanto como assinante (SCHUMETH, 2024).

2.5.2.2 *Message Queuing Telemetry Transport (MQTT)*

O *Message Queuing Telemetry Transport* (MQTT) é um protocolo de mensagens leve, desenvolvido especificamente para dispositivos móveis e sensores de pequeno porte (SCHUMETH, 2024). O *MQTT broker* é representado por um servidor que gerencia a distribuição de mensagens. Neste modelo, os clientes podem atuar como publicadores ou assinantes, permitindo uma comunicação desacoplada.

O MQTT utiliza tópicos para o roteamento de mensagens, os quais são *strings* hierárquicas que funcionam como canais para a distribuição das mensagens (SCHUMETH, 2024). A segurança no MQTT é assegurada pelo *Transport Layer Security* (TLS), que criptografa os dados transmitidos entre os clientes e o *broker*.

2.5.2.3 Protocolo Apache Kafka

O Apache Kafka é um protocolo binário que opera sobre *TCP*. No núcleo do Kafka está o *broker*, um servidor responsável por gerenciar o armazenamento e a transmissão

de mensagens(SCHUMETH, 2024) . O Kafka funciona em um modelo de publicação-assinatura, onde os produtores publicam mensagens em tópicos e os consumidores se inscrevem nesses tópicos para receber as mensagens.

Os tópicos no Kafka são divididos em partições, o que permite que múltiplos consumidores leiam de diferentes partições simultaneamente, aumentando a eficiência e a taxa de transferência (*throughput*). O Kafka assegura a entrega confiável de mensagens por meio da replicação das mensagens em múltiplos *brokers*.

2.5.3 *Broker Rabbit MQ*

O RabbitMQ funciona como um *broker de mensagens* que facilita a comunicação entre sistemas e aplicações. Sua função principal é gerenciar o fluxo de mensagens, direcionando-as e armazenando-as até o consumo ou entrega imediata, quando há consumidores disponíveis (BROADCOM, 2024).

O RabbitMQ suporta diversos protocolos, como AMQP 0-9-1, AMQP 1.0, MQTT e STOMP, cada um com características específicas de destinação e encaminhamento de mensagens.

No AMQP 0-9-1, as mensagens são enviadas para *exchanges*, enquanto no MQTT, são publicadas em tópicos. O STOMP suporta diferentes tipos de destinos, incluindo tópicos e filas.

O roteamento direciona uma mensagem publicada para uma fila ou tópico com consumidores potenciais. A entrega ocorre quando a mensagem é roteada para uma fila com um consumidor disponível. Os publicadores no RabbitMQ podem manter conexões de longa duração, estabelecidas na inicialização da aplicação, ou operar dinamicamente, iniciando e encerrando atividades conforme necessário, como em clientes WebSocket ou dispositivos móveis (BROADCOM, 2024).

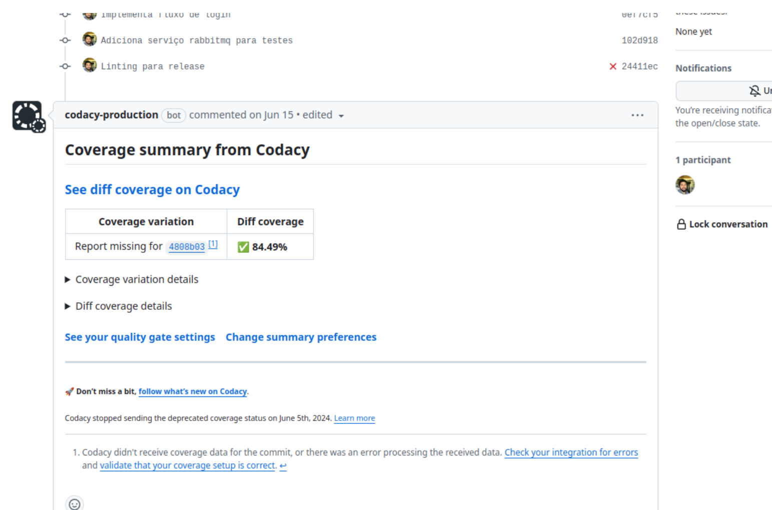
3 Projeto *Open FERP*

Neste capítulo é dada uma visão do produto Open FERP de microsserviços a ser desenvolvido como estudo de caso e exemplo para a aplicação de práticas de segurança. Estão explicitados o *backlog* do produto, bem como o processo de desenvolvimento a ser utilizado para esse produto. Todo o código desenvolvido pode ser visualizado na organização *SecMicroDev* do *Github*[1]. Nessa organização estão todos os repositórios criados para o presente trabalho, bem como todos os códigos fonte e arquivos de configuração do *Kubernetes*.

Na organização é possível acessar cada um dos microsserviços. A organização contém as variáveis e segredos comuns a todos os repositórios. Em cada repositório, é possível verificar os *Pull Requests*, com o histórico de análises de imagens *Docker*, de código e cobertura de testes. As análises das ferramentas foram postadas como comentários em cada *Pull Request*.

A Figura 4 mostra um exemplo de comentário de cobertura de testes. A análise dos testes e a análise estática de Código foi realizada utilizando a ferramenta *Codacy*[2]

Figura 4 – Exemplo de análise de cobertura do repositório.



Fonte: Autor.

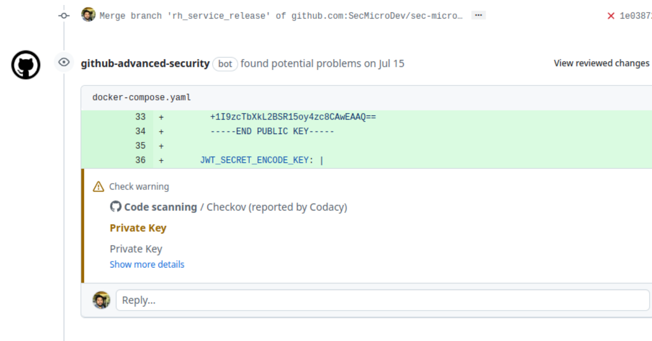
As Figuras 5 e 6 demonstram, respectivamente, a análise de segurança da ferramenta *Codacy* e a análise de *Contêiner* utilizando o plugin *Docker Scout*.[3]

São exemplos de recomendações comentados durante as atualizações de um *Pull Request* dentro do repositório. Na análise de segurança, é possível observar a atenção para

[1] Endereço do repositório: https://github.com/SecMicroDev.
[2] Informações sobre a ferramenta: https://www.codacy.com/quality.
[3] Mais informações sobre o *Docker Scout*: https://docs.docker.com/scout/explore/analysis/.

uma chave privada que foi identificada como exposta. Nesse caso é um falso positivo pois se encontrava disponível em um ambiente utilizado para o fluxo de teste, somente.

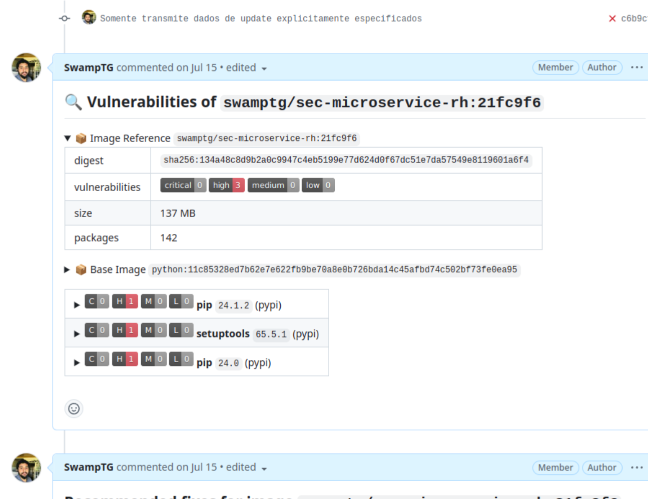
Figura 5 – Exemplo de análise de segurança do repositório.



Fonte: Autor.

A análise do *contêiner*, nesse exemplo, avisou a utilização de bibliotecas desatualizadas ou que possuíam vulnerabilidades. Essas análises foram consideradas e as bibliotecas, substituídas e atualizadas.

Figura 6 – Exemplo da análise de *contêiner* do repositório.



Fonte: Autor.

3.1 Produto *Open FERP*

3.1.1 Visão geral

Open FERP (*Open Free ERP*, ou ERP Livre e Aberto) é o nome do produto desenvolvido como estudo de caso para esse trabalho. Objetiva atender funcionalidades mínimas de um ERP (*Enterprise Resource Planning* - Planejamento de Recursos da Empresa), sendo de código aberto e de acesso livre e possuindo escalabilidade e modularização para a inserção de novos módulos e novos pacotes de funcionalidades.

Um ERP é um sistema integrado de gestão de organizações, preocupado em gerenciar os principais processos internos da organização em seus diversos setores, como a área de Recursos Humanos, Gerência de produção, estoque, relações comerciais e quaisquer outras áreas passíveis de informatização do seu gerenciamento (SAP, 2019).

O *Open FERP* atende as áreas de Gerenciamento de Recursos Humanos, Gerenciamento de Estoque, Gerenciamento de Vendas e relações comerciais.

O módulo de Gerenciamento de Recursos Humanos trata dos quadros internos da empresa, ou seja, funcionários relacionados diretamente a ela, prestando serviços ou sendo contratados de forma efetiva pela empresa.

O módulo de gerenciamento de estoque trata dos estoques de toda a empresa, tendo tanto os estoques produtivos, que servem de matéria prima, como os estoques auxiliares com materiais de escritório.

O módulo de gerenciamento de vendas e relações comerciais trata do ciclo que vai desde o contato inicial com um possível cliente, as propostas comerciais, até a concretização da relação comercial e o pós-venda, como *repostas* dos clientes.

3.1.2 *Backlog do Produto*

O Backlog utiliza o modelo de Histórias de Usuário, comum na elicitação de requisitos (LUCASSEN et al., 2016). O formato das histórias de usuário utilizado é *Eu*, como *<Identificação do tipo de usuário>*, quero *<objetivo>*, para que eu possa *<motivação>*.

As histórias são identificadas por um código para que possam ser referenciadas e buscadas ao longo do projeto.

A Tabela 2 representa as histórias de usuário contendo os requisitos planejados que precisam ser atendidos pelo produto para formar o MVP (*Minimum Viable Product* - Produto Mínimo Viável).

O MVP proposto para o estudo de caso se restringe aos requisitos US1, US3, US4, US5, US6, US7, US8, US9, US10 e US11, suficientes para o escopo proposto neste trabalho, para cumprimento dos objetivos de análise de esforço de Desenvolvimento Seguro.

Dentre as histórias de usuário, como proposto pela plataforma SAFe Ágil (ASHTANA et al., 2012), serão inseridas histórias de usuários que evocam necessidades de segurança do sistema, já que requisitos de segurança costumam ser negligenciados no contexto do desenvolvimento ágil de *software* (ASHTANA et al., 2012). A Tabela 3 contém as histórias de segurança do SAFe Ágil, selecionadas para o contexto do projeto proposto.

Além desses requisitos de segurança, para explorar as indicações do NIST para desenvolvimento seguro de microsserviços (CHANDRAMOULI; BUTCHER, 2020), alguns

Tabela 2 – Histórias de usuário do projeto *Open FERP*. Objetivam a descrição de requisitos para o desenvolvimento de Software em um projeto Ágil.

Código	Descrição da história
US1	Eu, como Gerente de RH, quero poder consultar o quadro de funcionários para ver e atualizar informações ou cadastrar novos funcionários.
US2	Eu, como Gerente de RH, quero ter informações sobre processos relacionados a cada funcionário, para acompanhar a situação e as necessidades de cada funcionário.
US3	Eu, como Gerente Comercial, quero ter um histórico de clientes com descrição de suas informações, pessoas importantes e interesses, para aumentar o sucesso de vendas através de boas relações com os cliente.
US4	Eu, como Gerente Comercial, quero acessar dados de vendas para saber os maiores clientes, os produtos mais vendidos e as datas das vendas, para ter foco e vendas mais eficientes, baseadas em dados.
US5	Eu, como Vendedor(a), desejo acessar meu histórico de vendas para gerenciar meus contatos e novas oportunidades de vendas e verificar minhas metas.
US6	Eu, como Vendedor(a), desejo cadastrar novos clientes, com informações sobre pessoas importantes, interesses e anotações de gerais para que eu não esqueça os contatos e possa realizar novas vendas.
US7	Eu, como Vendedor(a), desejo cadastrar novas vendas efetivadas, para reportar ao Gerente sobre a venda e acompanhar minhas metas comerciais.
US8	Eu, como Gerente Comercial, desejo cadastrar, alterar e remover dados sobre produtos para que meus funcionários tenham acesso aos dados atualizados e possam realizar suas vendas.
US9	Eu, como Operador(a) de estoque, gostaria de movimentar os ativos no sistema para que eles possam ser usados em diversos departamentos da empresa.
US10	Eu, como Operador(a) de estoque, gostaria de cadastrar novos itens de estoque adquiridos pela empresa, com descrições e códigos identificadores para melhor gerência do estoque.
US11	Eu, como Gerente de Estoque, desejo saber os responsáveis por cada movimentação dentro do estoque para ter o controle total dos funcionários e dirimir eventuais dúvidas sobre as movimentações.
US12	Eu, como Gerente de Estoque, desejo saber a situação de cada item de estoque para reduzir a chance perdas, desperdícios e avarias dos itens do estoque.
US13	Eu, como Funcionário(a), quero ter acesso aos meus dados para saber minha situação na empresa e corrigir eventuais erros de cadastro.

Fonte: Autor.

requisitos de segurança para infra-estrutura/arquiteturais foram propostos. A Tabela 4 relata quais são esses requisitos.

Os requisitos baseados no SAFe Ágil tem em seu identificador as siglas S-US (de *Security User Stories*, e os demais tem somente a sigla US (*User Story*).

Seguindo também as práticas do SAFe Ágil, ao lado das histórias de segurança, as práticas fundamentais relacionadas aos problemas de segurança endereçados pela S-US são relacionadas.

Os tipos de usuários considerados para o sistema são a(o) Gerente de RH, a(o) Vendedor, a(o) Gerente Comercial, a(o) Operadora(o) do estoque e a(o) Gerente de estoque e a(o) Funcionária(o). Para as histórias de segurança, os stakeholders considerados foram o Desenvolvedor e o Arquiteto do projeto.

Tabela 3 – Histórias de segurança para o contexto do desenvolvimento do projeto.

Código	Descrição da história
S-US1	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que a alocação de recursos esteja dentro dos limites ou controle de velocidade.
S-US2	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir a aplicação da codificação apropriada para o contexto de saída.
S-US3	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir a aplicação ou o acesso dentro dos limites de índices de *buffers e arrays*.
S-US4	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir o tratamento adequado de todas as exceções.
S-US5	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir o uso de formatos de string controlados.
S-US6	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir o uso de limites de inteiros controlados.
S-US7	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os usuários tenham acesso aos recursos específicos que eles requerem e que estão autorizados a utilizar.
S-US8	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir a conversão correta entre tipos numéricos.
S-US9	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir a atribuição e manutenção correta de permissões para todos os recursos críticos.
S-US10	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os dados sensíveis sejam mantidos restritos aos atores autorizados a acessá-los.
S-US11	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os danos causados ao sistema e seus dados sejam limitados caso um ator não autorizado consiga assumir o controle de um processo ou influenciar seu comportamento de maneiras imprevisíveis.
S-US12	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir a limitação de um caminho de acesso a um diretório restrito.
S-US13	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os ataques de *Cross-Site Scripting* sejam evitados.
S-US15	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os ataques de *Cross-Site Request Forgery* sejam evitados.
S-US16	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que as consultas ao banco de dados funcionem conforme o esperado, separando os dados da consulta.
S-US17	Como um(a) arquiteto(a)/desenvolvedor(a), não desejo armazenar informações sensíveis codificadas no sistema e desejo verificar que o sistema não armazena informações sensíveis codificadas.
S-US18	Como um(a) arquiteto(a)/desenvolvedor(a), desejo prevenir a exposição de informações através de mensagens de erro.
S-US19	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que o redirecionamento de URLs para sites não confiáveis não seja possível.
S-US20	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que os recursos sejam inicializados onde necessário.
S-US21	Como um(a) arquiteto(a)/desenvolvedor(a), desejo prevenir o acesso não autorizado a contas de usuário por meio de tentativas de adivinhação de senha.
S-US22	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que o usuário esteja protegido por autenticação robusta e gerenciamento de sessão.
S-US23	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que exista proteção suficiente na camada de transporte.
S-US24	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que o algoritmo criptográfico utilizado não esteja quebrado ou seja arriscado.
S-US25	Como um(a) arquiteto(a)/desenvolvedor(a), desejo garantir que o sistema não permita o uso de funções potencialmente perigosas.

Fonte: Adaptado de (ASHTANA et al., 2012)

Tabela 4 – Histórias de segurança para o contexto de infraestrutura/arquitetura do projeto.

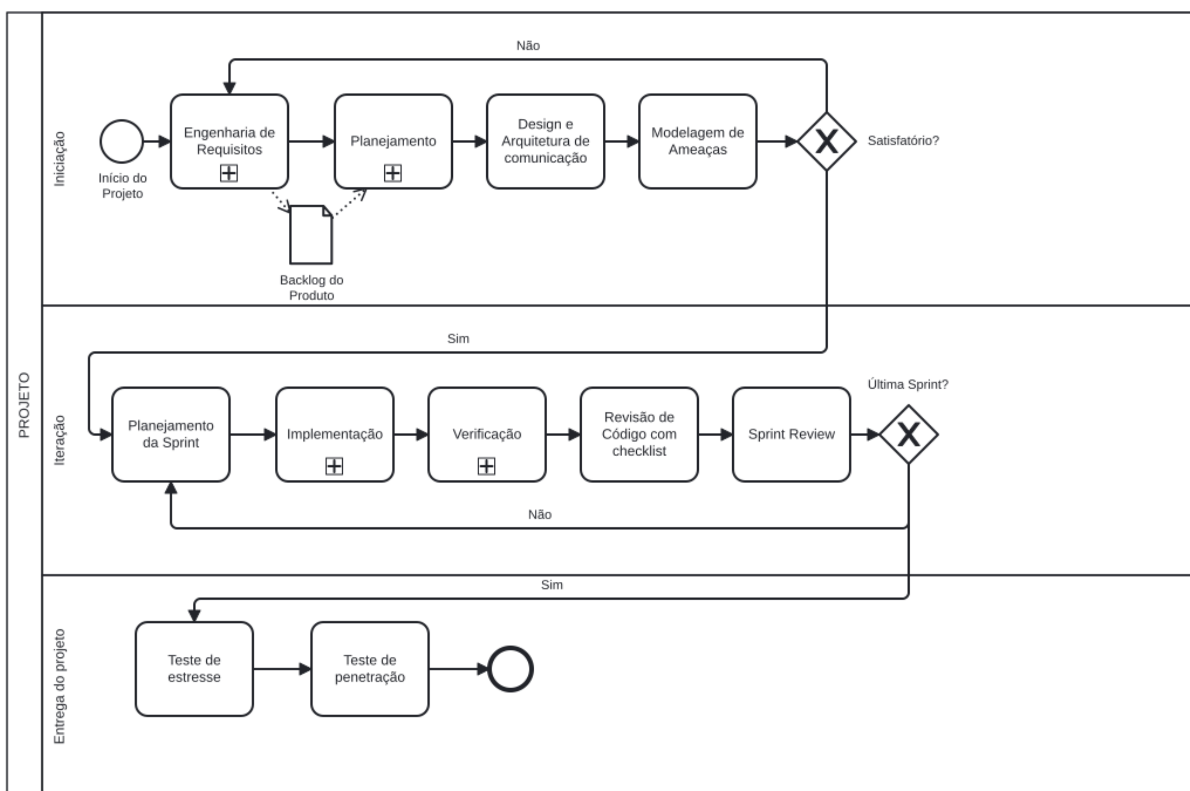
Código	Descrição do requisito
S-IA1	Utilizar um *service mesh* para fornecer mTLS na comunicação entre os serviços expostos.
S-IA2	Acessos externos ao *API Gateway* devem ser feitos via HTTPS.
S-IA3	Implementar um *Ingress* para *proxies* de todos os serviços e páginas de administração utilizando subdomínios
S-IA4	Deve ser possível a observabilidade dos serviços, para verificação da saúde e do status de cada serviço OpenFERP.

Fonte: Adaptado de (CHANDRAMOULI, 2019)

3.2 Processo de desenvolvimento

O processo foi proposto considerando as fases de planejamento, execução e entrega do projeto. A Figura 7 evidencia esse processo. Nas fase de iniciação tem-se o levantamento de requisitos, bem como o planejamento do projeto e da arquitetura do sistema. Após a ideação do produto, a iniciação também compreende a modelagem de ameaças para identificar as principais preocupações quanto a segurança.

Figura 7 – Processo geral de desenvolvimento proposto.



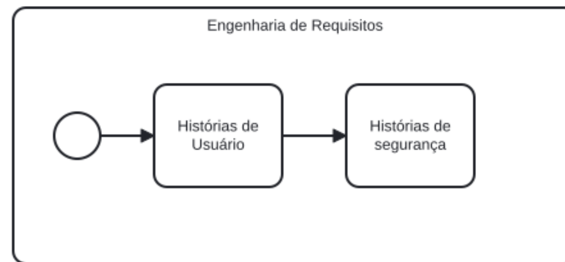
Fonte: Autor.

Uma das saídas importantes da Iniciação, no Processo de Modelagem de Ameaças, é um Manual de Desenvolvimento para o projeto, listando as boas práticas e os cuidados necessários para o desenvolvimento seguro.

A fase de iteração compreende a execução do desenvolvimento do produto, separado em um processo iterativo, com etapas de implementação, verificação e revisão.

As etapas internas dos subprocessos podem ser visualizadas nas figuras a seguir.

Figura 8 – Etapas da engenharia de requisitos.

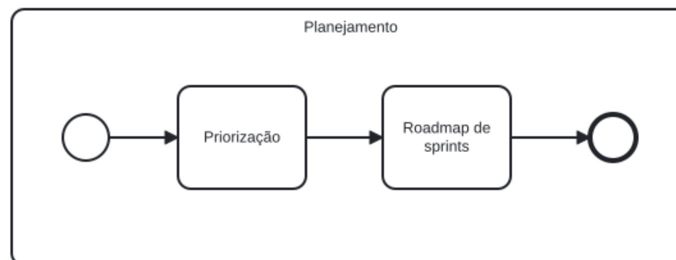


Fonte: Autor.

A etapa de requisitos, presente na Figura 8 compreende a instanciação das histórias de usuário e das de segurança. Para as histórias de segurança, o foco se torna a equipe de desenvolvimento, Garantia da Qualidade e os arquitetos de Software.

O planejamento, na Figura 9, abrange a priorização e um *Roadmap* geral do desenvolvimento.

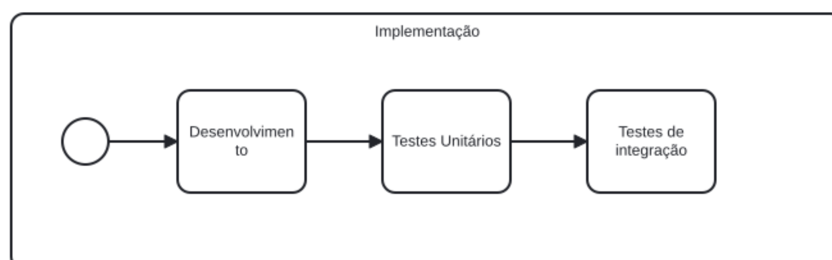
Figura 9 – Visão do subprocesso de planejamento



Fonte: Autor.

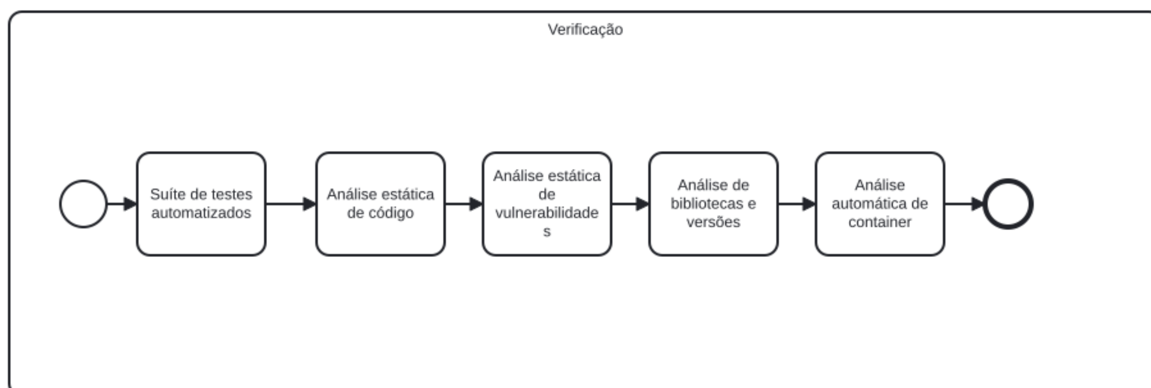
Dentro da iteração, como exposto nas Figuras 10 e 11, tem-se as etapas o subprocessos de implementação e de verificação. Para essas fases, tem-se, além da codificação e dos testes, a verificação automática de código por algumas ferramentas para análise estática e afins.

Figura 10 – Subprocesso de desenvolvimento



Fonte: Autor.

Figura 11 – Etapas do subprocesso de verificação.



Fonte: Autor.

Ao final do projeto, pretende-se realizar testes de penetração e de estresse para verificar a robustez do sistema e para verificar de forma mais concreta a eficácia dos mecanismos de segurança.

4 Iniciação

4.1 Arquitetura e Comunicação

Faz-se necessária a discussão da modelagem da comunicação entre os principais componentes da aplicação para visualização de decisões sobre as tecnologias utilizadas e suas justificativas. A Figura 12 exemplifica a comunicação entre esses componentes. A comunicação entre os serviços é feita de forma assíncrona, com a utilização de um *Broker* para *Stream* de mensagens (forma cilíndrica do diagrama). O *broker* permite um maior desacoplamento entre os serviços, como menciona o Capítulo 2.5, além de outros atributos vantajosos para arquiteturas de microsserviços.

4.1.1 Componentes e Fluxo de Comunicação

4.1.1.1 Usuário

O usuário interage com a aplicação através do *API Gateway*.

4.1.1.2 *API Gateway*

Recebe as requisições dos usuários e as encaminha para os serviços apropriados dentro da infraestrutura *Kubernetes.*

4.1.1.3 Serviços

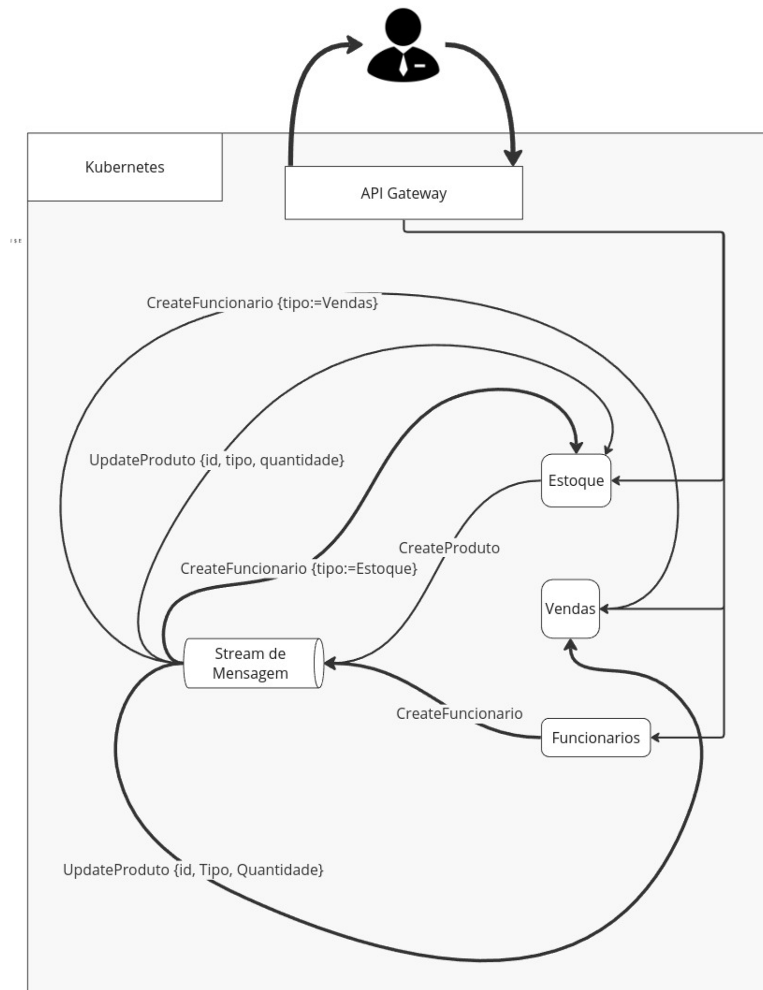
4.1.1.3.1 Funcionários

- Gerencia as operações relacionadas a funcionários, como a criação de novos funcionários.
- Recebe mensagens do *Stream* de Mensagem para operações como `CreateFuncionario`.

4.1.1.3.2 Vendas

- Trata operações de vendas, incluindo a criação de funcionários do tipo vendas e atualização de produtos.
- Recebe requisições do *API Gateway* e envia mensagens para o *Stream* de Mensagem.

Figura 12 – Arquitetura dos serviços. Componentes básicos são o *Kubernetes* para gerenciamento dos microsserviços e um serviço de mensageria para a comunicação entre os serviços.



Fonte: Autor.

4.1.1.3.3 Estoque

- Gerencia o inventário de produtos, incluindo a criação e atualização de informações de produtos.
- Interage com o serviço de Vendas e o Stream de Mensagem para manter os dados atualizados.

4.1.1.4 Stream de Mensagem

- Atua como um sistema intermediário para a troca de mensagens entre os serviços.

- Gerencia eventos como `CreateFuncionario` e `CreateProduto`, garantindo que todos os serviços relevantes sejam notificados das mudanças.

4.1.2 Fluxos de Operações

4.1.2.1 Criação de Funcionário

Quando um novo funcionário é criado com o tipo "Vendas"ou "Estoque", a operação é gerenciada pelo serviço de Funcionários e a mensagem correspondente é publicada no *Stream* de Mensagem.

4.1.2.2 Criação e Atualização de Produto

- O serviço de Vendas ou Estoque pode solicitar a criação ou atualização de um produto, enviando as informações através do API Gateway.
- O serviço de Estoque processa as requisições de criação de produtos e envia atualizações de estoque para o Stream de Mensagem.

4.1.2.3 Atualização de Produto

- Tanto o serviço de Vendas quanto o de Estoque podem solicitar atualizações de produtos.
- As atualizações são publicadas no Stream de Mensagem e processadas pelo serviço de Estoque para garantir que todas as informações estejam sincronizadas.

4.1.3 Ambiente Kubernetes

Todos os serviços e componentes estão implantados em um *cluster Kubernetes*, que orquestra a execução, escalabilidade e resiliência dos micro serviços, garantindo alta disponibilidade e gestão eficiente de recursos, como exposto na Seção 2.4.

4.1.4 Mensageria

A interação via mensageria visa ao menor acoplamento entre os serviços, já que, tanto o produtor quanto o consumidor da mensagem somente precisam da interface da mensagem e do tópico em que ela é publicada, como exposto na Seção 2.5.

O *broker* selecionado foi o *Rabbit MQ*, por ser um produto já estabelecido e que permite fácil integração com o *Kubernetes* por meio de recursos de configuração customizados e desenvolvidos pela própria comunidade do *Rabbit MQ*.

4.2 Modelagem de Segurança

Utilizou-se a técnica *STRIDE* (*Spoofing, Tampering, Info Disclosure, Repudiation, Denial of Service e Elevation of Privilege*) para elencar as possíveis ameaças.

Esse processo de modelagem faz parte do *Microsoft Security Development Lifecycle* (como dito na Seção 2.2.2). O STRIDE é um método para auxiliar os *stakeholders* a pensarem como os atacantes (pessoas com intenções maliciosas). Objetiva o auxílio da modelagem de ameaças para identificá-las durante o desenvolvimento, antes que sejam enviadas para produção (MICROSOFT, 2023).

É geralmente indicado para equipes menos experientes em segurança, para formar um escopo de ameaças comuns sem que se esqueça de ameaças importantes (MICROSOFT, 2023). Porém, é importante que pessoas com experiência em ameaças de segurança possam avaliar a modelagem e pensar como um atacante, pois o STRIDE não é uma lista exaustiva de ameaças e detalhes podem ser perdidos.

Para cada letra da sigla, tem-se os seguintes conceitos (MICROSOFT, 2023):

- ***Spoofing:*** refere-se à falsificação de identidade de um componente dentro de um sistema, o que compromete os objetivos de autenticação. Isso acontece, por exemplo, em comunicações onde remetentes se passam por outros componentes, alegando uma identidade falsa.

- ***Tampering:*** envolve a alteração não autorizada de dados, comprometendo a integridade do sistema. Nas comunicações, isso pode se manifestar através da modificação de mensagens enquanto estão em trânsito.

- ***Repudiation:*** ocorre quando um componente nega ter realizado uma ação, mesmo que de fato a tenha executado, o que geralmente é resultado da ausência de mecanismos de auditabilidade no sistema.

- ***Information Disclosure:*** refere-se à exposição não autorizada de informações, comprometendo a confidencialidade do sistema. Em termos de comunicação, isso ocorre quando componentes não autorizados obtêm acesso a mensagens.

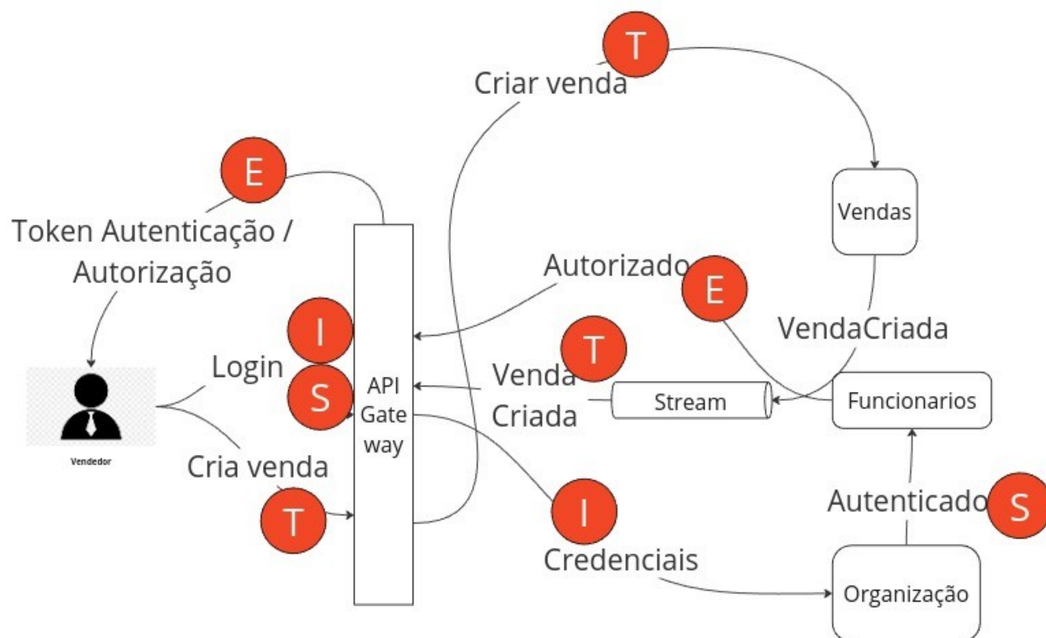
- ***Denial of Service:*** envolve a interrupção ou atraso não autorizado de um serviço, afetando a disponibilidade do sistema. Em comunicações, pode se manifestar por meio do bloqueio ou atraso na entrega de mensagens.

- ***Elevation:*** ocorre quando um componente adquire permissões ou capacidades sem a devida autorização, comprometendo os objetivos de autorização do sistema. Este cenário geralmente envolve a execução de ações que não são permitidas pela política de controle de acesso do sistema.

A Figura 13 demonstra as ameaças no fluxo de comunicação dentro da arquitetura de alto nível da solução. Na figura tem-se a interação de um usuário com os serviços por meio de um *Gateway*. Essa interação representa a criação de uma venda. Cada letra representa uma ameça identificada segundo a metodologia STRIDE. Por exemplo, falhas nas verificações de credenciais podem facilitar o roubo de identificação de um usuário por um usuário malicioso.

A Tabela 5 contém o relato de cada uma das ameaças identificadas.

Figura 13 – Análise de ameaças na arquitetura de alto nível



Fonte: Autor

Tabela 5 – Especificação das possíveis ameaças, considerando o modelo de comunicação da Figura 13

Categoria	Ameaça
S	• Os serviços estão sujeitos a roubo de identidade caso não valide corretamente as credenciais • Spoofing também pode ocorrer caso um atacante consiga interceptar as requisições com as credenciais do usuário • Um atacante pode injetar dados via SQL e alterar as credenciais de um usuário, conseguindo se autenticar como ele • Um atacante pode se autenticar como outro usuário caso consiga acesso ao token JWT utilizado pelo usuário após se autenticar
T	• Um usuário com acessos indevidos pode cadastrar dados novos e incorretos • Um usuário com acessos indevidos pode modificar dados existentes caso consiga acessos de gerente • Um atacante pode conseguir alterar dados por meio de injeção no banco de dados caso os inputs não sejam tratados • Um atacante pode conseguir alterar dados por meio de injeção no banco de dados caso parâmetros de url não sejam tratados ou sejam usados para execução de comandos
R	• Caso um atacante invada ou injete dados no sistema, ele poderá alterar dados dos responsáveis por ações que seriam visualizados pelos gerentes de departamento

Tabela 5 – Especificação das possíveis ameaças, considerando o modelo de comunicação da Figura 13 (continuação)

Categoria	Ameaça
I	• Os dados de credenciais podem ser interceptados caso não se use https • Tokens com informações do usuário podem ser interceptados caso sejam armazenados no cliente de forma insegura • Atacantes com acesso indevido poderão ver informações sensíveis das empresas
D	• Caso os servidores de API não tenham limite de requisições, poderão ficar indisponíveis com muitos acessos • Os serviços podem ficar indisponíveis caso se permita queries custosas ou respostas muito grandes
E	• Falhas de autorização ao retornar um token de acesso podem permitir a visualização de dados sensíveis

Fonte: Autor

A elaboração dos critérios de aceitação das histórias de usuário implementadas, portanto, considera essas ameaças para a criação dos itens de *checklist*, explicados na Seção 4.3

4.3 Backlog das *Sprints*

Para cada história de usuário, foi incluída uma definição de concluído, que especifica tanto critérios funcionais quando uma lista de critérios relacionados aos requisitos de segurança. Os critérios foram selecionados com base nos requisitos de segurança aplicáveis a cada história de usuário. Um critério pode estar associado a mais de um requisito de segurança.

A Tabela 6 contém um exemplo de itens do *check-list*, relacionando-os com requisitos de segurança levandados para a definição de concluído.

Tabela 6 – Checklist de definição de concluído com requisitos de segurança para a **US14**. Esses critérios podem se relacionar com um ou mais requisitos de segurança.

Item do Checklist	**Requisitos de Segurança**
Saídas de texto possuem a codificação correta	S-US2, S-US13
Os acessos a índices de buffers e arrays foram verificados e testados	S-US3
Nenhuma variável possivelmente nula é acessada antes de ser inicializada	S-US20
Casos de exceção foram estabelecidos e a resposta de exceção foi verificada por testes	S-US4
Os modelos ORM possuem limites de tamanho de string	S-US5
Os modelos ORM possuem valores numéricos com limites máximo e mínimo	S-US6
As variáveis numéricas possuem formato suficiente que impede overflow ao realizar cálculos	S-US8
Código não possui segredos expostos	S-US10, S-US17
Informações sensíveis como senhas são armazenadas de forma encriptada	S-US10, S-US24
Os *logs* de erro são tratados e nenhuma informação sensível é exposta por meio deles	S-US18
Os níveis de *logs* de erro e alertas são gerenciados	S-US18, S-US25
Bibliotecas seguras de criptografia, com versão atualizada, estão sendo utilizadas	S-US24, S-US23
O projeto possui cobertura de testes de no mínimo 60%	S-US4, S-US18
O usuário é identificado antes de acessar qualquer recurso	S-US7, S-US22
O nível de acesso do usuário é verificado antes de acessar qualquer recurso	S-US7, S-US9
Requisições de *login* são limitadas para impedir tentativas de adivinhação	S-US21
Token de sessão e de autenticação foram implementados	S-US22, S-US23

5 Implementação dos Requisitos de Segurança

5.1 Desenvolvimento seguro a nível Arquitetural

Durante o desenvolvimento do OpenFERP utilizando microsserviços, houve dificuldades na implementação dos requisitos de segurança. As implementações no escopo da arquitetura de microsserviços são importantes para garantir alguma robustez na segurança dos microsserviços e de sua comunicação. A maior dificuldade foi a implementação do *Ingress* e do *Service Mesh* em um ambiente gerenciado manualmente, diferente dos ambientes de nuvem, que automatizam a gerência do *cluster*.

Durante o desenvolvimento dos serviços, também houveram dificuldades para implementação do consumo de mensagens do *broker* e da autenticação usando *tokens JWT* com criptografia assimétrica (RSA256).

Esse Capítulo relata o esforço de implementação, utilizando a métrica de horas gastas para os requisitos de segurança e para os testes e configuração do cluster.

Essa Seção relata as escolhas dos componentes e as respectivas dificuldades de implementação a nível arquitetural.

5.2 Análise da Implementação dos Requisitos de Segurança

A segurança dos microsserviços, em grande parte, passa por seus detalhes de arquitetura e comunicação entre seus componentes para que seja reforçada, como evidenciado no Capítulo 2.

A escolha e implementação dos componentes da arquitetura, portanto, foi uma etapa crítica.

A flexibilidade do desenvolvimento utilizando microsserviços, porém, possibilitou a implementação paralela ao desenvolvimento das histórias de usuário. Testes Unitários das funcionalidades permitiram o teste dos microsserviços fora da arquitetura *Kubernetes*, em conjunto com testes locais utilizando a composição dos contêineres de microsserviços para comunicação entre os serviços.

Para um desenvolvimento seguro mais robusto, foi decidida a implementação de recomendações de segurança específicas para arquiteturas de microsserviços, como consta na Seção 2.3 o que incluiu a adoção de um *service mesh* (SIA-1). Optou-se pelo *Linkerd*[1].

Também fez-se necessária a implementação do *Ingress Nginx* e ferramentas de observabilidade com os *dashboards* instalados junto ao próprio *Linkerd*.

5.2.1 Service Mesh Linkerd

A implementação do *service mesh Linkerd* foi um dos pontos mais desafiadores do projeto. Este ajuda a garantir a comunicação segura entre os micro serviços, gerenciamento de tráfego e políticas de segurança. A configuração e integração do *Linkerd* com o ambiente levou cerca de um mês. Entre as principais dificuldades encontradas, destacam-se:

- Configuração inicial complexa e ajuste fino para garantir compatibilidade com todos os serviços.
- Necessidade de entender a documentação e as melhores práticas de uso do *Linkerd*.
- Integração com outros componentes do sistema, como o *Ingress Nginx* e o *Rabbit MQ*.

5.2.2 Observabilidade com Dashboards do Linkerd

Para garantir a observabilidade do sistema (SIA-4) e monitorar a segurança e desempenho dos microsserviços, foram utilizados os *dashboards* fornecidos pelo *Linkerd*.

O *Dashboard* mostra métricas de acesso aos serviços e quais que estão utilizando os *sidecars*, bem como percentuais de falhas e as conexões internas entre os serviços. As Figuras 14, 15 e 16 mostram os *dashboards* gerados pelo *Linkerd* para a aplicação OpenFERP.

Observa-se as taxas de requisição, quais *deployments* estão *meshed* (termo usado para os *Pods* que possuem os *proxies* do *Linkerd* como explicado na Seção 2.4.2).

Na Figura 14, é possível observar todos os *namespaces* do *cluster*, contendo os diferentes *deployments*. Dentre os *namespaces*, os do *linkerd* e o *tcc* estão gerenciados pelo *service mesh*.

Na Figura 15, é possível ver todos os *deployments* do *namespace tcc*, junto aos seus *pods*, com métricas de conexão, latência e perda de pacotes. Os valores na imagem indicam uma boa saúde dos serviços, com valores baixos de latência e sem perdas de pacotes.

[1] Para mais informações, veja a Seção 2.4.2

Figura 14 – Visão dos *deployments* executados no Cluster no *Dashboard*. Mostra quais possuem *sidecars* do *Linkerd* na coluna *Meshed*.

Namespaces

HTTP Metrics

Namespace	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency
argocd	0/7	—	—	—	—	—
cert-manager	0/0	—	—	—	—	—
default	0/1	—	—	—	—	—
ingress-nginx	0/1	—	—	—	—	—
kube-flannel	0/2	—	—	—	—	—
kube-node-lease	0/0	—	—	—	—	—
kube-public	0/0	—	—	—	—	—
kube-system	0/8	—	—	—	—	—
linkerd	3/3	100.00%	2.42	1 ms	3 ms	4 ms
linkerd-viz	5/5	100.00%	4.67	2 ms	50 ms	90 ms
metallb-system	0/3	—	—	—	—	—
rabbitmq	0/0	—	—	—	—	—
tcc	7/7	100.00%	2.13	1 ms	1 ms	1 ms
test	0/0	—	—	—	—	—

Fonte: Autor

Figura 15 – Visão dos *Deployments e Pods específicos do OpenFERP.*

Namespace > tcc

Deployments

Deployment	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency
demo	1/1	100.00%	0.3	1 ms	1 ms	1 ms
postgres-rh-dev	1/1	100.00%	0.3	1 ms	1 ms	1 ms
postgres-spt-dev	1/1	100.00%	0.3	1 ms	1 ms	1 ms
ptservice-dev	1/1	100.00%	0.3	1 ms	1 ms	1 ms
rabbit-exchange	1/1	100.00%	0.3	1 ms	1 ms	1 ms
rhservice-dev	1/1	100.00%	0.3	1 ms	1 ms	1 ms
sellservice-dev	1/1	100.00%	0.3	1 ms	1 ms	1 ms

Pods

Pod	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency
demo-7fff9cff84-slbqm	1/1	100.00%	0.3	1 ms	1 ms	1 ms
postgres-rh-dev-79dbddb6b4-b7mmd	1/1	100.00%	0.3	1 ms	1 ms	1 ms
postgres-spt-dev-7497b49dcb-f27hx	1/1	100.00%	0.3	1 ms	1 ms	1 ms
ptservice-dev-5945c6c694-8fnjd	1/1	100.00%	0.3	1 ms	1 ms	1 ms

Fonte: Autor

Na Figura 16, é possível ver detalhes do *deployment* do serviço de *broker* e serviços que estavam conectados a ele em um dado momento.

Figura 16 – Visualização do serviço de *Broker* e suas conexões em um dado momento pelo *Dashboard.*

Namespace > tcc > deployment/rabbit-exchange

Inbound

Resource	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency
deploy/prometheus	1/1	100.00%	0.1	3 ms	3 ms	3 ms

Pods

Pod	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency
rabbit-exchange-68f6cb5cbb-v2kz4	1/1	100.00%	0.3	1 ms	1 ms	1 ms

TCP

Pod	Meshed	Connections	Read Bytes / sec	Write Bytes / sec
rabbit-exchange-68f6cb5cbb-v2kz4	1/1	5	24.78B/s	685.8B/s

Edges (Identity: default.tcc)

	Namespace	Name	Identity	Secured
FROM	linkerd-viz	prometheus	prometheus.linkerd-viz	✓
FROM	tcc	ptservice-dev	default.tcc	✓
FROM	tcc	sellservice-dev	default.tcc	✓

Fonte: Autor

A implementação desta ferramenta foi relativamente rápida, levando aproximadamente uma semana. As dificuldades incluíram:

- Configuração dos *dashboards* para fornecer insights úteis sem sobrecarregar o sistema.

5.2.3 Ingress Nginx

O *Ingress Nginx* foi implementado (SIA-2 e SIA-3) para gerenciar o tráfego de entrada e aplicar políticas de segurança a nível de rede, como a conexão HTTPS. Esta etapa levou cerca de duas semanas e incluiu desafios como:

- Configuração das regras de *Ingress* para garantir segurança e alta disponibilidade.
- Integração com o *service mesh Linkerd* e outros componentes do sistema.
- Testes para validar a segurança e desempenho das configurações.

5.2.4 Resumo dos Tempos de Implementação

A Tabela 7 resume os tempos de implementação dos principais componentes de segurança adotados no projeto.

Tabela 7 – Tempos de Implementação dos Componentes de Segurança

Componente	**Tempo de Implementação**
Service Mesh Linkerd	30h
Observabilidade (*Dashboards* do *Linkerd*)	8h
Ingress Nginx	16h

A adoção dessas ferramentas e práticas ajuda a garantir a segurança dos microsserviços desenvolvidos, apesar dos desafios e do tempo investido em cada etapa. A implementação dessas soluções permitiu criar uma arquitetura seguindo recomendações de segurança como relatado na Seção 2.3, alinhada à boas práticas do mercado.

5.3 Desenvolvimento seguro no Serviço de RH

O serviço de RH foi o primeiro a ser implementado, pois possibilita a gerência dos usuários pertencentes a uma empresa. Essa gerência foi utilizada para determinação de autorização e autenticação, utilizando papéis e escopos determinados para a empresa, com cada papel tendo um nível de gerência em seu determinado escopo.

É interessante notar que apesar de alguns gargalos para implementação de certos requisitos de segurança, eles foram mais rapidamente implementados nos outros micro serviços pois foi possível reutilizar as bibliotecas.

Um exemplo é o módulo de autenticação e autorização. Os outros serviços não necessitaram criar *Tokens* de autenticação e autorização, nem fazer o seu gerenciamento, simplesmente verificar a autenticidade do *Token* utilizando a chave e decodificar as informações para verificar a permissão de acesso.

A Tabela 8 mostra a duração para cada um dos critérios levantados para o microsserviço de RH.

Alguns dos critérios, como variáveis nulas e acesso a índices de vetores, apesar de possuírem recursos da linguagem de programação para sua garantia, levaram algum tempo maior por conta da implementação de testes para a garantia da verificação.

Outros recursos, como restrições de formatos e tamanhos no banco de dados foram implementados de forma rápida, utilizando configurações simples das próprias bibliotecas de integração com o banco de dados.

Tabela 8 – Sumarização da definição de concluído dos requisitos de segurança para o microserviço de RH e a respectiva duração para implementação.

Item do Checklist	**Requisitos de Segurança**	**Duração**
Saídas de texto possuem a codificação correta	S-US2, S-US13	2h
Os acessos a índices de buffers e arrays foram verificados e testados	S-US3	4h
Nenhuma variável possivelmente nula é acessada antes de ser inicializada	S-US20	8h
Casos de exceção foram estabelecidos e a resposta de exceção foi verificada por testes	S-US4	1h
Os modelos ORM possuem limites de tamanho de string	S-US5	1h
Os modelos ORM possuem valores numéricos com limites máximo e mínimo	S-US6	1h
As variáveis numéricas possuem formato suficiente que impede overflow ao realizar cálculos	S-US8	2h
Código não possui segredos expostos	S-US10, S-US17	1h
Informações sensíveis como senhas são armazenadas de forma encriptada	S-US10, S-US24	4h
Os *logs* de erro são tratados e nenhuma informação sensível é exposta por meio deles	S-US18	4h
Os níveis de *logs* de erro e alertas são gerenciados	S-US18, S-US25	Não implementado
Bibliotecas seguras de criptografia, com versão atualizada, estão sendo utilizadas	S-US24, S-US23	4h
O projeto possui cobertura de testes de no mínimo 60%	S-US4, S-US18	11h
O usuário é identificado antes de acessar qualquer recurso	S-US7, S-US22	4h
O nível de acesso do usuário é verificado antes de acessar qualquer recurso	S-US7, S-US9	8h
Requisições de *login* são limitadas para impedir tentativas de adivinhação	S-US21	Não implementado
Tokens autenticação foram implementados	S-US22, S-US23	11h
Token de sessão foram implementados	S-US22, S-US23	Não implementado
Total	67h	

Fonte: Autor

5.4 Segurança no Serviço de Vendas

Para esse serviço, após a implementação das primeiras histórias de usuário, as bibliotecas puderam ser reutilizadas.

Para certos requisitos, houve um reuso total do que já havia sido implementado. Para outros, apesar de haver implementação, o tempo foi reduzido pois foi possível reutilizar a lógica e os fluxos implementados anteriormente, conforme demonstrado na Tabela 9.

Ambos os serviços de Vendas e de Estoque reutilizaram, a biblioteca JWT, utilizando algoritmo de chave assimétrica para validação do *token* utilizado na requisição.

Tabela 9 – Sumarização da definição de concluído dos requisitos de segurança para o microsserviço de vendas e a respectiva duração para implementação.

Item do Checklist	**Requisitos de Segurança**	**Duração**
Saídas de texto possuem a codificação correta	S-US2, S-US13	2h
Os acessos a índices de buffers e arrays foram verificados e testados	S-US3	2h
Nenhuma variável possivelmente nula é acessada antes de ser inicializada	S-US20	4h
Casos de exceção foram estabelecidos e a resposta de exceção foi verificada por testes	S-US4	1h
Os modelos ORM possuem limites de tamanho de string	S-US5	1h
Os modelos ORM possuem valores numéricos com limites máximo e mínimo	S-US6	1h
As variáveis numéricas possuem formato suficiente que impede overflow ao realizar cálculos	S-US8	1h
Código não possui segredos expostos	S-US10, S-US17	1h
Os *logs* de erro são tratados e nenhuma informação sensível é exposta por meio deles	S-US18	2h
Os níveis de *logs* de erro e alertas são gerenciados	S-US18, S-US25	Não implementado
Bibliotecas seguras de criptografia, com versão atualizada, estão sendo utilizadas	S-US24, S-US23	1h
O projeto possui cobertura de testes de no mínimo 60%	S-US4, S-US18	6h
O usuário é identificado antes de acessar qualquer recurso	S-US7, S-US22	1h
O nível de acesso do usuário é verificado antes de acessar qualquer recurso	S-US7, S-US9	4h
Requisições de *login* são limitadas para impedir tentativas de adivinhação	S-US21	Não implementado
Tokens autenticação foram implementados	S-US22, S-US23	1h
Token de sessão foram implementados	S-US22, S-US23	Não implementado
Total	28h	

Fonte: Autor

Somente o serviço de RH tem acesso a chave privada para assinatura do token. Os outros utilizam a chave pública para verificação da assinatura. Essa solução, além de simplificar a implementação, torna a chave privada mais isolada, diminuindo possibilidades de vazamento e roubo.

As chaves são armazenadas como variáveis dentro da arquitetura *Kubernetes*, utilizando um *vault*.

Apesar das funções de autorização terem sido reutilizados, a lógica de autorização foi implementada para os casos específicos do serviço.

Testes unitários também foram implementados, porém, o tempo foi diminuido graças aos templates previamente estabelecidos e ao escopo menor do serviço.

5.5 Segurança no Serviço de Estoque

O desenvolvimento do estoque foi semelhante ao serviço de vendas. Ambos os serviços necessitaram de um subconjunto menor dos dados de usuário e empresa. Somente dos identificadores principais e de dados relevantes de escopo e hierarquia dos usuários.

Os dados do serviço de RH foram coletados utilizando a estrutura de mensageria.

Apesar de diferentes, os serviços de vendas e estoque compartilham o mesmo banco de dados, devido a restrições de sincronização de quantidade de estoques e produtos vendidos. Os bancos separados tornariam o gerenciamento da sincronização desses dados muito complexa e possibilitariam maiores chances de condições de corrida e inconsistência dos dados.

A flexibilidade durante o desenvolvimento dos microsserviços permitiu a configuração de comunicação com diferentes bancos de dados sem modificar o código, utilizando variáveis de ambiente e o *vault do Kubernetes*.

A Tabela 10 contém os tempos de implementação para os requisitos de segurança do microsserviço de estoque. O tempo de implementação foi significativamente menor que o primeiro microsserviço. A maior parte do tempo de implementação foi gasto na implementação de testes.

Tabela 10 – Sumarização da definição de concluído dos requisitos de segurança para o microsserviço de estoque e a respectiva duração para implementação.

Item do Checklist	**Requisitos de Segurança**	**Duração**
Saídas de texto possuem a codificação correta	S-US2, S-US13	1h
Os acessos a índices de buffers e arrays foram verificados e testados	S-US3	2h
Nenhuma variável possivelmente nula é acessada antes de ser inicializada	S-US20	2h
Casos de exceção foram estabelecidos e a resposta de exceção foi verificada por testes	S-US4	1h
Os modelos ORM possuem limites de tamanho de string	S-US5	1h
Os modelos ORM possuem valores numéricos com limites máximo e mínimo	S-US6	1h
As variáveis numéricas possuem formato suficiente que impede overflow ao realizar cálculos	S-US8	1h
Os *logs* de erro são tratados e nenhuma informação sensível é exposta por meio deles	S-US18	2h
Os níveis de *logs* de erro e alertas são gerenciados	S-US18, S-US25	Não implementado
Bibliotecas seguras de criptografia, com versão atualizada, estão sendo utilizadas	S-US24, S-US23	1h
O projeto possui cobertura de testes de no mínimo 60%	S-US4, S-US18	5h
O usuário é identificado antes de acessar qualquer recurso	S-US7, S-US22	1h
O nível de acesso do usuário é verificado antes de acessar qualquer recurso	S-US7, S-US9	3h
Requisições de *login* são limitadas para impedir tentativas de adivinhação	S-US21	Não implementado
Tokens autenticação foram implementados	S-US22, S-US23	1h
Token de sessão foram implementados	S-US22, S-US23	Não implementado
Total	22h	

Fonte: Autor

5.6 Considerações sobre o Desenvolvimento

O desenvolvimento de uma aplicação segura baseada em microsserviços apresenta desafios significativos, especialmente quando os requisitos de segurança são integrados desde as etapas iniciais do projeto.

Apesar do aumento de complexidade inicial, a utilização das práticas permitiu uma solução testada e com implementações preemptivas para possíveis ameaças ao sistema.

Quanto aos conhecimentos necessários para a aplicação do Desenvolvimento Seguro em Microsserviços, alguns pontos devem ser destacados. Primeiramente, a modelagem de ameaças e o planejamento da aplicação foram importantes para mitigar possíveis riscos. Além disso, o gerenciamento das necessidades de segurança teve de ocorrer em todas as etapas do desenvolvimento, garantindo uma abordagem abrangente.

Outro aspecto relevante foi o entendimento do funcionamento da comunicação entre os microsserviços dentro de um cluster, seja por meio de implementações síncronas ou assíncronas, que influenciam a eficiência e segurança do sistema. Nesse contexto, a implementação de políticas de autenticação e autorização foi fundamental, utilizando padrões como o JWT.

A segurança da comunicação entre os microsserviços também teve de ser considerada, com a implementação de HTTPS tanto para a comunicação externa quanto interna no cluster, levando em conta as configurações de cada ferramenta, como o Ingress Nginx e o service mesh Linkerd. Além disso, o gerenciamento seguro de segredos e chaves, utilizando soluções como Vaults, evitou a exposição de informações sensíveis.

A qualidade do código-fonte de cada microsserviço foi reforçada por meio de análises automatizadas, com pipelines que facilitaram essa verificação. Da mesma forma, a implementação de pipelines de testes para cada microsserviço assegurou a execução e validação contínua. O uso de ferramentas de observabilidade permitiu o monitoramento da saúde dos serviços e a identificação mais eficiente de erros.

Quanto ao esforço necessário, o reuso de componentes e uma arquitetura que fa- vorece o baixo acoplamento foram aspectos facilitadores do desenvolvimento. Outro esforço foi a implementação de testes unitários com mocks e configurações específicas. Porém, depois de estabelecidos, foram reutilizados como base para os outros serviços.

Os testes foram importantes para tratar desafios específicos, além de verificar as funcionalidades da aplicação, como a verificação de acessos a índices de buffers e arrays, a gestão de variáveis nulas, e a implementação de limites em modelos ORM.

O service mesh exigiu um esforço de aprendizado considerável, mas que compensa pela segurança de comunicação e observabilidade proporcionadas e pela facilidade de integrar novos microsserviços após sua configuração.

Dificuldades enfrentadas, como a configuração inicial complexa e a integração com outros componentes do sistema, apesar de custosas inicialmente, foram superadas com uma compreensão aprofundada das melhores práticas e uma configuração detalhada.

6 Conclusão

A Segurança é uma atributo essencial do *Software* e o Desenvolvimento Seguro é uma prática recomendada para lidar com a segurança. No desenvolvimento de microsserviços não se pode negligenciá-la e deve-se tratar da Segurança dentro de suas especificidades.

Com isso, este trabalho tinha como objetivo utilizar práticas de desenvolvimento seguro de *Software* em um sistema de microsserviços e analisar o esforço empregado e os conhecimentos aplicados.

Este estudo de caso permitiu perceber que a implementação de segurança em microsserviços requer uma abordagem estratégica, planejamento detalhado e a adoção de ferramentas e práticas que garantam a proteção dos dados e a integridade do sistema considerando detalhes aplicáveis aos microsserviços, como a densidade de comunicação e o desacoplamento entre os serviços. Dados os custos e a curva de aprendizado mais demorada, o desenvolvimento seguro de microsserviços deve ser reforçado e revisado com afinco durante o desenvolvimento dos sistemas, utilizando-se de ferramentas bem estabelecidas e automatizações.

Este estudo, entretanto, possui limitações inerentes a sua metodologia. Por ser um estudo de caso único, a generalização dos resultados para outras aplicações ou contextos pode ser limitada.

As conclusões tiradas deste projeto específico podem não ser aplicáveis a todas as arquiteturas de microsserviços ou a diferentes ambientes de desenvolvimento.

Para aumentar a validade e a confiabilidade das conclusões, futuros estudos poderiam incluir a implementação de aplicações semelhantes sem o uso de metodologias de desenvolvimento seguro para que se possa fazer uma comparação direta. Além disso, a inclusão de outros métodos de padrão de comparação e ferramentas de segurança poderia fornecer uma base mais robusta para comparação e avaliação dos resultados.

Outra área de melhoria potencial seria a realização de estudos comparativos utilizando diferentes plataformas de service mesh e ferramentas de observabilidade, para avaliar sua eficácia e impacto na segurança dos microsserviços. Implementar práticas de desenvolvimento seguro em diferentes contextos e ambientes também poderia oferecer insights valiosos sobre a adaptabilidade e a eficácia dessas práticas.

Referências

ASHTANA, V. et al. Practical Security Stories and Security Tasks. *SAFECode Releases Software Security Guidance for Agile Practitioners*, v. 1, n. 1, p. 34, jul. 2012. Citado 5 vezes nas páginas 15, 29, 30, 47 e 49.

AUTHORS, K. *Service*. 2024. Disponível em: <https://kubernetes.io/docs/concepts/services-networking/service/>. Citado na página 40.

BERARDI, D. et al. Microservice security: A systematic literature review. *PeerJ Computer Science*, PeerJ Inc., v. 8, p. e779, jan. 2022. ISSN 2376-5992. Disponível em: <https://peerj.com/articles/cs-779>. Citado 2 vezes nas páginas 31 e 32.

BOGNER, J. et al. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. *Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019*, Institute of Electrical and Electronics Engineers Inc., p. 187–195, maio 2019. Citado na página 14.

BRERETON, P. et al. Using a Protocol Template for Case Study Planning. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. [s.n.], 2008. Disponível em: <https://scienceopen.com/document?vid=47e2b89a-628d-4975-9908-b943cdc96258>. Citado 2 vezes nas páginas 17 e 18.

BROADCOM. *Publishers | RabbitMQ*. 2024. Disponível em: <https://www.rabbitmq.com/docs/publishers>. Acesso em: 20 de Fevereiro de 2024. Citado na página 44.

CHANDRAMOULI, R. *Security Strategies for Microservices-Based Application Systems*. Gaithersburg, MD, 2019. NIST SP 800-204 p. Disponível em: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf>. Citado 8 vezes nas páginas 31, 32, 34, 35, 36, 37, 38 e 50.

CHANDRAMOULI, R.; BUTCHER, Z. *Building Secure Microservices-Based Applications Using Service-Mesh Architecture*. Gaithersburg, MD, 2020. NIST SP 800-204A p. Disponível em: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204A.pdf>. Citado 2 vezes nas páginas 41 e 47.

DRAGONI, N. et al. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, Springer International Publishing, p. 195–216, nov. 2017. Citado 4 vezes nas páginas 13, 14, 31 e 34.

FIELDING, R. T. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1997. Disponível em: <https://datatracker.ietf.org/doc/rfc2068>. Citado na página 36.

FRANCESCO, P. D.; LAGO, P.; MALAVOLTA, I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, Elsevier Inc., v. 150, p. 77–97, abr. 2019. ISSN 01641212. Citado na página 14.

FUJDIAK, R. et al. Managing the Secure Software Development. In: *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. CANARY ISLANDS, Spain: IEEE, 2019. p. 1–4. ISBN 978-1-72811-542-9. Disponível em: <https://ieeexplore.ieee.org/document/8763845/>. Citado na página 15.

ISO. *ISO/IEC 27000:2018: Information Technology — Security Techniques — Information Security Management Systems — Overview and Vocabulary*. 2018. Citado na página 21.

KHAN, R. A. et al. Systematic Literature Review on Security Risks and its Practices in Secure Software Development. *IEEE Access*, v. 10, p. 5456–5481, 2022. ISSN 2169-3536. Disponível em: <https://ieeexplore.ieee.org/document/9669954/>. Citado na página 21.

KIRSTENS et al. *Cross Site Scripting (XSS) | OWASP Foundation*. 2018 2022. Disponível em: <https://owasp.org/www-community/attacks/xss/>. Acesso em: 25 de Maio de 2023. Citado na página 33.

KUBERNETES. *Kubernetes Overview*. 2023. Disponível em: <https://kubernetes.io/docs/concepts/overview/>. Acesso em: 13 de Maio de 2024. Citado na página 39.

KUBERNETES, A. *Deployments*. 2024. Disponível em: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. Citado na página 40.

KUBERNETES, A. *Pods*. 2024. Disponível em: <https://kubernetes.io/docs/concepts/workloads/pods/>. Citado na página 40.

LICATA, S. *Our Work*. 2023. Disponível em: <https://safecode.org/our-work/>. Acesso em: 01 de Maio de 2023. Citado na página 27.

LINKERD, A. *Linkerd Architecture*. 2024. Disponível em: <https://linkerd.io/2-edge/reference/architecture/>. Acesso em: 27 de Janeiro de 2024. Citado na página 42.

LINKERD, A. *Linkerd Overview*. 2024. Disponível em: <https://linkerd.io/2-edge/overview/>. Acesso em: 27 de Janeiro de 2024. Citado na página 42.

LUCASSEN, G. et al. Improving agile requirements: The Quality User Story framework and tool. *Requirements Engineering*, v. 21, n. 3, p. 383–403, set. 2016. ISSN 0947-3602, 1432-010X. Disponível em: <http://link.springer.com/10.1007/s00766-016-0250-x>. Citado na página 47.

MCGRAW, G. Testing for security during development: Why we should scrap penetrate-and-patch. *IEEE Aerospace and Electronic Systems Magazine*, v. 13, n. 4, p. 13–15, abr. 1998. ISSN 1557-959X. Citado na página 24.

MICROSOFT. *Microsoft Security Development Lifecycle*. 2023. Disponível em: <https://learn.microsoft.com/pt-br/windows/security/threat-protection/msft-security-dev-lifecycle>. Acesso em: 24 de Maio de 2023. Citado 2 vezes nas páginas 25 e 57.

NAIAKSHINA, A. et al. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, 2017. p. 311–328. ISBN 978-1-4503-4946-8. Disponível em: <https://dl.acm.org/doi/10.1145/3133956.3134082>. Citado na página 23.

NIAZI, M. et al. A maturity model for secure requirements engineering. *Computers & Security*, v. 95, p. 101852, ago. 2020. ISSN 01674048. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167404820301243>. Citado na página 15.

OWASP. *Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series*. 2018. Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#use-built-in-or-existing-csrf-implementations-for-csrf-protection>. Acesso em: 12 de Junho de 2023. Citado na página 34.

OWASP. *Deserialization - OWASP Cheat Sheet Series*. 2021. Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html>. Acesso em: 13 de Junho de 2023. Citado na página 33.

OWASP. *Injection Prevention - OWASP Cheat Sheet Series*. 2021. Disponível em: <https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html>. Acesso em: 01 de Junho de 2023. Citado na página 32.

OWASP. *Introduction - OWASP Cheat Sheet Series*. 2021. Disponível em: <https://cheatsheetseries.owasp.org/index.html>. Acesso em: 25 de Maio de 2023. Citado na página 32.

Pereira-Vale, A. et al. Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping. In: *2019 XLV Latin American Computing Conference (CLEI)*. [S.l.: s.n.], 2019. p. 01–10. Citado na página 15.

RANSOME, J. F. et al. *Core Software Security: Security at the Source*. Boca Raton London New York: CRC Press, an Auerbach book, 2014. ISBN 978-1-4665-6096-3 978-1-4665-6095-6. Citado na página 24.

RASHID, A. et al. The Cyber Security Body of Knowledge. v. 1.1.0, p. 1067, jul. 2021. Citado 6 vezes nas páginas 20, 21, 22, 23, 24 e 25.

SAFECODE. *Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program*. SAFECode, Tech. Rep., 2018. Disponível em: <https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf>. Citado 2 vezes nas páginas 15 e 27.

SALTZER, J.; SCHROEDER, M. The protection of information in computer systems. *Proceedings of the IEEE*, v. 63, n. 9, p. 1278–1308, 1975. ISSN 0018-9219. Disponível em: <http://ieeexplore.ieee.org/document/1451869/>. Citado 2 vezes nas páginas 21 e 22.

SAP. *O que é ERP | Definição de Planejamento de Recursos Empresariais | SAP Insights*. 2019. Disponível em: <https://www.sap.com/brazil/products/erp/what-is-erp.html>. Acesso em: 17 de Julho de 2023. Citado na página 47.

SCHUMETH, J. *Evaluation of Message Brokers for Interprocess Communication in a Microservice Architecture*. Tese (Doutorado) — FH Campus Wien, Viena, 2024. Disponível em: <https://permalink.obvsg.at/fcw/AC17236211>. Citado 2 vezes nas páginas 43 e 44.

SHUKLA, A. et al. System security assurance: A systematic literature review. *Computer Science Review*, v. 45, p. 100496, ago. 2022. ISSN 1574-0137. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1574013722000338>. Citado na página 14.

SOLDANI, J.; TAMBURRI, D. A.; HEUVEL, W. J. V. D. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, Elsevier Inc., v. 146, p. 215–232, dez. 2018. ISSN 01641212. Citado na página 31.

TANENBAUM, A. S.; STEEN, M. van. *Distributed Systems: Principles and Paradigms*. Second edition, adjusted for digital publishing. The Netherlands?: Maarten van Steen, 2016. ISBN 978-1-5302-8175-6. Citado na página 13.

YARYGINA, T.; BAGGE, A. H. Overcoming Security Challenges in Microservice Architectures. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. [S.l.: s.n.], 2018. p. 11–20. Citado 3 vezes nas páginas 14, 15 e 16.

ZIMMERMANN, O. Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development*, Springer Verlag, v. 32, n. 3-4, p. 301–310, jul. 2017. ISSN 18652042. Citado na página 14.