UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

JULIO CESAR GARCIA RIBEIRO

COMPARAÇÃO ENTRE *FRAMEWORKS* DE DESENVOLVIMENTO DE *SOFTWARE* COM FOCO NAS PRINCIPAIS *FEATURES* DE SEGURANÇA EM UM SISTEMA *WEB* BANCÁRIO

APUCARANA

2023

JULIO CESAR GARCIA RIBEIRO

COMPARAÇÃO ENTRE *FRAMEWORKS* DE DESENVOLVIMENTO DE *SOFTWARE* COM FOCO NAS PRINCIPAIS *FEATURES* DE SEGURANÇA EM UM SISTEMA *WEB* BANCÁRIO

Comparison between software development frameworks focused on the main security features in a web banking system

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação do Curso de Bacharelado em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Luiz Fernando Carvalho

APUCARANA

2023

JULIO CESAR GARCIA RIBEIRO

COMPARAÇÃO ENTRE *FRAMEWORKS* DE DESENVOLVIMENTO DE *SOFTWARE* COM FOCO NAS PRINCIPAIS *FEATURES* DE SEGURANÇA EM UM SISTEMA *WEB* BANCÁRIO

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação do Curso de Bacharelado em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 14/novembro/2023

Prof. Me. Muriel de Souza Godoi
Universidade Tecnológica Federal do Paraná

Prof. Dr. Fernando Barreto
Universidade Tecnológica Federal do Paraná

Prof. Dr. Ricardo Theis Geraldi
Externo

Prof. Dr. Luiz Fernando Carvalho
Orientador
Universidade Tecnológica Federal do Paraná

APUCARANA

2023

AGRADECIMENTOS

Ao Prof. Dr. Ricardo Theis Geraldi, pela amizade, orientações, ajuda e todos os ensinamentos.

Ao Prof. Dr. Luiz Fernando Carvalho por aceitar orientar este trabalho. Obrigado por todo o conhecimento compartilhado e todos os momentos descontraídos.

RESUMO

Há uma demanda em ascensão da indústria pela necessidade de sistemas *web* em diferentes domínios de aplicação (finanças, cidades inteligentes e agricultura de precisão). Neste contexto, aumenta a possibilidade de falhas e ataques de segurança desses sistemas, visto que podem estar acessíveis na *web* por qualquer pessoa. Existem dificuldades na indústria de *software* em garantir que sistemas *web* sejam efetivamente seguros. Testes de segurança são realizados durante e após o desenvolvimento do *software*, a fim de encontrar vulnerabilidades e falhas de segurança, sendo um processo caro e que exige esforço de times ágeis. Na tentativa de minimizar falhas e ataques, práticas de segurança têm sido adotadas desde a concepção do *software*. Essas práticas são descritas por *frameworks* de desenvolvimento seguro, que apresentam *features* de segurança do *software*. Este trabalho objetiva comparar e modelar as principais *features* de segurança em dois *frameworks* de desenvolvimento seguro: BSA *Framework for Secure Software* e NIST *Secure Software Development Framework*, com foco em melhorar segurança em sistemas *web*. Esta comparação será realizada por meio da representação das práticas de cada *framework* em *feature models* e pela definição de configurações de segurança de cada *framework* analisado. As práticas de segurança serão utilizadas no processo de desenvolvimento de dois módulos de um sistema *web* bancário: autenticação e transferência de dinheiro. Os módulos serão avaliados no contexto de falhas e ataques de segurança com base nas vulnerabilidades mais comuns identificadas em uma revisão da literatura para sistemas *web*. Com este trabalho, se espera: (i) criar representações das práticas de cada *framework* por meio de *feature models*; (ii) definir configurações de segurança a nível de projeto (*design*) para cada *framework*, selecionando as principais práticas relacionadas ao desenvolvimento seguro; (iii) adaptar os *frameworks* de desenvolvimento no contexto da segurança em um sistema *web*; e (iv) desenvolver dois módulos seguros de um sistema *web* bancário. Por fim, conceber um corpo de conhecimento preliminar das principais *features* de segurança por meio da comparação dos *frameworks* para minimizar falhas e ataques em sistemas *web*.

Palavras-chave: práticas de segurança; *frameworks* de desenvolvimento; *feature models*; desenvolvimento *web* seguro; engenharia de *software*.

ABSTRACT

There is a growing industry demand for web systems in various application domains (finance, smart cities, and precision agriculture). In this context, the possibility of security failures and attacks on these systems increases, as they may be accessible on the web by anyone. The software industry faces challenges in ensuring the effective security of web systems. Security testing is conducted during and after software development to identify vulnerabilities and security flaws, which is a costly process requiring agile team efforts. In an attempt to minimize failures and attacks, security practices have been adopted from the inception of software. These practices are described by secure development frameworks that offer software security features. This work aims to compare and model the main security features in two secure development frameworks: BSA Framework for Secure Software and NIST Secure Software Development Framework, with a focus on enhancing security in web systems. This comparison will be carried out by representing the practices of each framework in feature models and defining security configurations for each framework analyzed. The security practices will be applied in the development process of two modules of a banking web system: authentication and money transfer. The modules will be evaluated in the context of security flaws and attacks based on the most common vulnerabilities identified in a literature review for web systems. This work aims to: (i) create representations of the practices of each framework through feature models; (ii) define security configurations at the design level for each framework, selecting key practices related to secure development; (iii) adapt the development frameworks in the context of security in a web system; and (iv) develop two secure modules for a banking web system. Lastly, it aims to establish a preliminary body of knowledge on the main security features by comparing the frameworks to minimize failures and attacks in web systems.

Keywords: security practices; development frameworks; feature models; secure web development; software engineering.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Descrição do projeto

Transformação digital pode ser entendida como um paradigma impulsionado por novas tecnologias que gera disrupções nas industrias. As disrupções resultam em impactos positivos na melhoria de um negócio e o aprimoramento da experiência do cliente (FEROZ; ZO; CHIRAVURI, 2021).

Acelerado pela transformação digital, a presença de sistemas digitais em diversos setores da atualidade ajudou na evolução de diversos processos que antes eram demorados, complicados e tediosos. Se pode citar a evolução em setores como hotelaria, que se beneficiou do *Booking.com* [1] e *Airbnb* [2]; entretenimento, que foi revolucionado pelo *Spotify* [3]; e a digitalização de empresas tradicionais, como o Starbucks (FEROZ; ZO; CHIRAVURI, 2021).

Um setor que se beneficiou da transformação digital foi o setor bancário. O processo de transferência de dinheiro entre pessoas, que antes era lento e realizado de maneira física, hoje se tornou rápido e accessível de qualquer aparelho eletrônico. Se pode citar como exemplo o desenvolvimento do *PIX*, que tornou instantânea as transferências de dinheiro entre os brasileiros (SANTIAGO; ZANETONI; VITA, 2020).

Uma pesquisa realizada em 2022 pela Febraban atestou que cerca de 77% de todas as transações bancárias são realizadas por plataformas digitais, sendo estas por meio de *Mobile banking* (aplicativos móveis), *Internet banking* (navegadores de internet) e *WhatsApp* [4]. Desta forma, oito em cada dez transações são realizadas de maneira digital (Federação Brasileira de Bancos, 2023).

Essa digitalização gerou uma grande quantidade de dados, muito destes sensíveis e pessoais. A preocupação na defesa e manuseio correto destes dados aumenta à medida em que novos sistemas e tecnologias são desenvolvidos. Assim, podem surgir novas ameaças, que vão desde o vazamento de dados pessoais até o roubo de identidades (AJAYI *et al.*, 2022).

É estimado que cada evento de vazamento de dados custa para os Estados Unidos em média de quatro a oito milhões de dólares. O efeito anual de ataques cibernéticos na economia global é de cerca de 400 bilhões de dólares (GUEMBE *et al.*, 2022). A crescente sofisticação de ataques cibernéticos representam um risco inerente a empresas e serviços essenciais pela capacidade de interromper operações, destruir dados e diminuir a reputação de negócios (GUEMBE *et al.*, 2022).

O nível de sofisticação de controles de segurança deve depender do valor dos dados a serem protegidos. A presença de dados financeiros em sistemas bancários cria um potencial

[1] *Booking.com*: https://www.booking.com/.
[2] *Airbnb*: https://www.airbnb.com.br/.
[3] *Spotify*: https://spotify.com/.
[4] Pagamentos por *WhatsApp*: https://www.whatsapp.com/payments/.

imediato para ganhos ilícitos que podem ser conquistados a partir de um acesso indevido, possibilitando compras não solicitadas e venda de dados sensíveis no mercado negro (LIN, 2016).

Neste contexto, surge uma grande preocupação em garantir a confidencialidade, integridade e disponibilidade das informações gerenciadas por estes sistemas, visto que são elementos críticos para o sucesso do negócio. Todavia, garantir a segurança de sistemas é um desafio complexo que envolve o manuseio de uma quantidade gigantesca de dados confidenciais e o aumento da sofisticação das ameaças externas (ASLAN *et al.*, 2023).

Para enfrentar esse desafio, podem ser adotadas diretrizes, como o OWASP *Top Ten Project* (Open Worldwide Application Security Project, 2021), que descreve as dez principais vulnerabilidades de segurança em aplicações *web* e/ou normas de segurança como a ISO/IEC 27001 (International Organization for Standardization, 2005), que define os requisitos para um Sistema de Gestão de Segurança da Informação.

Mesmo com a adoção de diretrizes e normas, um *gap* é gerado em produtos de *softwares*, que não são desenvolvidos considerando práticas de segurança desde o início de seu ciclo de vida. Isso pode ocasionar na criação de vulnerabilidades difíceis e caras de serem corrigidas posteriormente. Um estudo realizado pela *International Business Machines Corporation* (IBM) aponta que o custo de correção de falhas é até 100 vezes maior em relação à prevenção na fase de desenvolvimento (DAWSON *et al.*, 2010).

Uma das formas mais eficientes de atacar este problema, é a adição de segurança durante todo o ciclo de desenvolvimento de *software*. Desta forma, é possível entender as ameaças de segurança de um sistema, o que permite a resolução destes problemas nos componentes antes da fase de desenvolvimento (DAWSON *et al.*, 2010).

Frameworks de desenvolvimento seguro podem ser utilizados para aumentar a segurança dos *softwares* desenvolvidos. Dentre estes *frameworks*, destacam-se o BSA *Framework for Secure Software* (Business Software Alliance, 2019) e o NIST *Secure Software Development Framework* (SOUPPAYA; SCARFONE; DODSON, 2022). Estes *frameworks* apresentam práticas de segurança estruturadas, que podem ser adicionadas durante todo o ciclo de desenvolvimento do *software*.

Apesar da existência desses *frameworks*, há poucos estudos comparativos que avaliam a sua aplicação práticas no desenvolvimento de *software* seguro em sistemas *web*. Assim, surge a necessidade de comparar esses *frameworks* na prática. Essa comparação será realizada por meio da representação das práticas de cada *frameworks* por meio de *feature models*. Estes *feature models* serão então utilizados para guiar o desenvolvimento de dois módulos de um sistema bancário *web*: autenticação e transferência de dinheiro.

É fundamental incorporar segurança no ciclo de vida de sistemas bancários para favorecer sua manutenção e evolução segura. Nesse sentido, a comparação entre os *frameworks* BSA e NIST pode contribuir para a avaliação da sua aplicação prática na segurança de sistemas. A investigação e modelagem de *feature models* pode ser um caminho útil para modelar *features*

de segurança e comparar *frameworks* de segurança com o intuito de orientar o desenvolvimento de *software* seguro.

Esta pesquisa contribui para a área de segurança da informação, apresentando práticas de desenvolvimento por meio da comparação entre os referidos *frameworks*. Portanto, este trabalho apresenta o desenvolvimento prático de módulos em um sistema bancário com a utilização de *feature models* para apoiar o processo de desenvolvimento e representação de *features* de segurança. A segurança é um requisito não-funcional primordial para proteger produtos ou sistemas de software desenvolvidos na indústria e no contexto da engenharia de *software* (REINEHR, 2020).

1.2 Objetivos

1.2.1 Objetivo geral

Esta pesquisa tem como objetivo principal comparar os *frameworks* de desenvolvimento seguro BSA *Framework for Secure Software* e NIST *Secure Software Development Framework* (SSDF) com representação das práticas por meio de *feature models*, definição de uma configuração de segurança com práticas relacionadas ao desenvolvimento de código seguro e do desenvolvimento de dois módulos de um sistema *web* bancário apoiado pela configuração de segurança obtida.

1.2.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

1. Criar representações das práticas de desenvolvimento seguro dos *frameworks* BSA e NIST por meio de *feature models*;
2. Definir configurações de segurança para cada *framework* por meio da seleção das principais práticas de segurança relacionadas ao desenvolvimento de código seguro;
3. Adaptar práticas de *frameworks* de desenvolvimento seguro no desenvolvimento dos módulos de um sistema *web* bancário;
4. Desenvolver dois módulos seguros de um sistema *web* bancário com base na configuração de segurança obtida.

1.3 Estrutura do trabalho

Este trabalho foi escrito com a seguinte estrutura:

- **Capítulo 1**: apresenta a introdução do trabalho e contém as seções: (i) Descrição do projeto, apresentando o contexto em que este trabalho está inserido, o problema abordado e uma possível solução; (ii) Objetivos, listando o objetivo geral e objetivos específicos deste trabalho; e (iii) Estrutura do trabalho, com um panorama geral dos capítulos deste trabalho;

- **Capítulo 2**: apresenta a estruturação desta pesquisa com a descrição de cada etapa de desenvolvimento;

- **Capítulo 3**: apresenta a revisão da literatura com os temas abordados neste trabalho, dividida pela seções: (i) Segurança da informação, com as definições relacionadas a segurança da informação e segurança em aplicações; (ii) *Features*, com definições sobre *features* e modelagem de *features*; (iii) Desenvolvimento seguro de software, abordando o ciclo de vida de desenvolvimento seguro; (iv) *Frameworks* para desenvolvimento seguro, apresentando os *frameworks* utilizados neste trabalho; e (v) Contendo alguns trabalhos relacionados sobre *features* em segurança da informação;

- **Capítulo 4**: apresenta os *feature models* desenvolvidos com base nas práticas dos *frameworks* apresentados, dividido em três sessões: (i) *Feature models*, descrevendo os *feature models* desenvolvidos; (ii) Critérios para seleção das práticas, descrevendo como foi realizada a escolha das práticas de segurança para definir as configurações de segurança; e (iii) Configurações de segurança, demonstrando as práticas escolhidas para o desenvolvimento dos módulos;

- **Capítulo 5**: apresenta um caso de estudo sobre a aplicação dos *feature models* desenvolvidos no processo de desenvolvimento de módulos de um sistema *web*, onde foram implementadas as práticas de segurança do *framework* BSA, abordado nesta pesquisa;

- **Capítulo 6**: Apresenta as principais discussões deste trabalho, dividida em três sessões: (i) *Frameworks* utilizados: BSA e NIST, demonstrando as diferenças entre os *frameworks* BSA e NIST e seus pontos fortes e fracos; (ii) *Feature models* desenvolvidos, demonstrando a eficácia da utilização de *feature models* no contexto deste trabalho; e (iii) Módulos implementados, comentando a segurança atingida nos módulos e o uso dos *frameworks* e *feature models* no desenvolvimento destes módulos;

- **Capítulo 7**: apresenta as principais conclusões, comentários e possíveis trabalhos futuros relacionados a esta pesquisa.

2 ESTRUTURAÇÃO DA PESQUISA

As etapas do desenvolvimento deste trabalho foram adaptadas de procedimentos do protocolo de engenharia de software experimental de Wohlin *et al.* (2012). As etapas são ilustradas na Figura 1 e descritas a seguir.

A **primeira etapa** do trabalho se baseou no planejamento da pesquisa. Nesta etapa foi realizada:

- Uma revisão da literatura com informações sobre segurança da informação, segurança em aplicações, conceitos básicos sobre *features* e modelagem de *features*. Na sequência, foi realizada uma introdução sobre desenvolvimento seguro de *software* e foram apresentados os *frameworks* BSA *Framework for Secure Software* e NIST *Secure Software Development Framework* (SSDF). Por fim, os poucos trabalhos relacionados a *feature models* e segurança da informação foram sintetizados;

- Criação dos *feature models*, contendo as práticas de segurança descritas pelos *frameworks* BSA *Framework for Secure Software* e NIST *Secure Software Development Framework*. Foram modelados dois *feature models* para representar e especificar as práticas de cada *framework* utilizando a ferramenta FeatureIDE[1]. Os *feature models* são uma maneira de visualizar as práticas sintetizadas que cada *framework* apresenta, facilitando a visualização destas práticas e apoiando o desenvolvimento de aplicações seguras. Nem todas as práticas foram utilizadas no desenvolvimento dos módulos, visto que foram geradas configurações de segurança personalizadas com o apoio da FeatureIDE para os módulos a serem desenvolvidos;

- Definição dos critérios para escolha das *features* de segurança. Levando em consideração as práticas aplicáveis no desenvolvimento do código-fonte, foram escolhidas as principais *features* de segurança, utilizadas no desenvolvimento dos módulos do sistema *web* bancário.

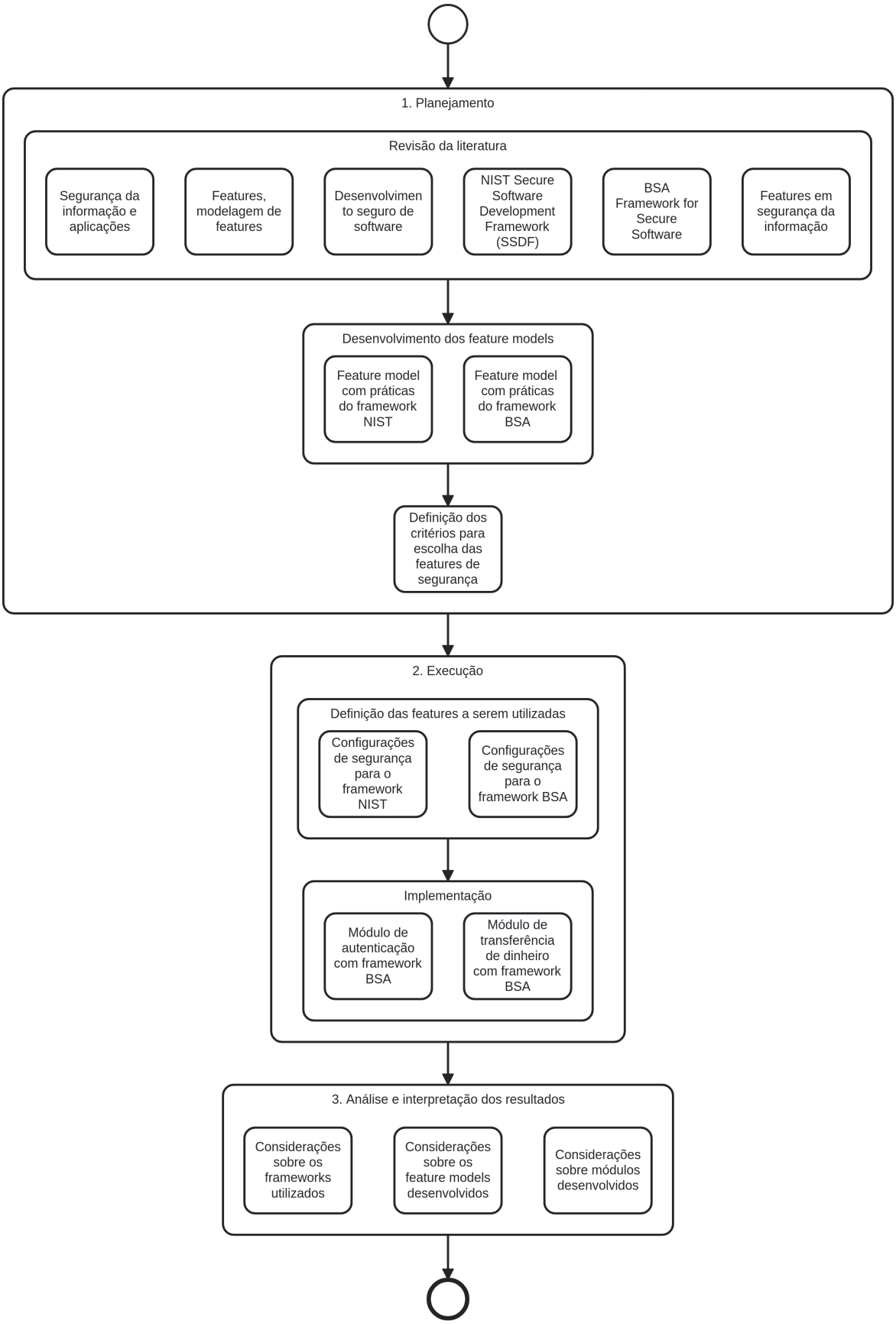
A **segunda etapa** é a fase de execução deste trabalho, na qual foram realizadas as seguintes tarefas:

- Definição das *features* a serem utilizadas. Com base nos *feature models* desenvolvidos e por meio do critério de seleção definido na etapa anterior, foram geradas duas configurações de segurança para cada módulo a ser implementado (autenticação e transferência de dinheiro). Essas configurações contêm as práticas de segurança a adotadas na implementação dos módulos;

- Com as configurações de segurança definidas, os módulos foram implementados seguindo as práticas de segurança apontadas nas configurações obtidas dos *feature mo-*

[1] FeatureIDE: https://www.featureide.de/.

Figura 1 – Etapas do Método de Pesquisa



Fonte: Autoria própria (2023).

dels desenvolvidos. Foram desenvolvidos dois módulos: (i) módulo de autenticação seguindo as práticas do *framework* BSA; (ii) módulo de transferência de dinheiro seguindo as práticas do *framework* BSA. A configuração de segurança do *framework* NIST não foi utilizada por resultar em um menor número de práticas aplicáveis, além de todas as suas práticas terem sido abordadas pelo *framework* BSA.

A **terceira etapa** deste trabalho se baseou na análise e interpretação dos resultados obtidos. Foram analisados: (i) os *frameworks* utilizados, suas diferenças, pontos fortes e pontos fracos; (ii) os *feature models* desenvolvidos, com base na capacidade de representação das práticas de segurança demonstradas pelos *frameworks*; e (iii) os módulos desenvolvidos, com base na sua segurança em relação a ameaças comuns. Após isso, foram realizadas discussões sobre a aplicação dos *frameworks* e dos *feature models* no desenvolvimento seguro de *software*.

3 REVISÃO DA LITERATURA

3.1 Segurança da informação

Em linhas gerais, segurança é definida como “a qualidade ou o estado de estar seguro - estar livre de perigo”. Podemos entender segurança como a proteção contra aqueles que querem fazer o mal, adversários, seja de maneira intencional ou não (MEERTS, 2018).

O Comitê de Sistemas de Segurança Nacional dos Estados Unidos (CNSS) define a segurança da informação como “a proteção da informação e de sistemas de informação contra o acesso, uso, divulgação, interrupção, modificação ou destruição não autorizada, com o objetivo de garantir a confidencialidade, integridade e disponibilidade” (Committee on National Security Systems, 2022).

Segundo a Associação de Auditoria e Controle de Sistemas de Informação (ISACA), a segurança da informação “garante que apenas usuários autorizados (confidencialidade) tenham acesso a informação precisa e completa (integridade) quando necessário (disponibilidade)"(Information Systems Audit and Control Association, 2015). Os três termos mencionados compõem a tríade CIA (*Confidentiality*, *Integrity* e *Availability*) e são definidos como:

- **Confidencialidade (*Confidentiality*)**: Uso de restrições para o acesso e divulgação de informações incluindo meios para a preservação da privacidade e dados pessoais; Prevenção ao acesso a dados por pessoas não autorizadas com técnicas como a criptografia (Committee on National Security Systems, 2022);
- **Integridade (*Integrity*)**: Proteção contra a modificação ou destruição não autorizada de informações (Information Systems Audit and Control Association, 2015); Regras que regem a modificação e destruição de elementos de um sistema (Committee on National Security Systems, 2022);
- **Disponibilidade (*Availability*)**: Garantia de acesso confiável e uso da informação (Committee on National Security Systems, 2022).

De acordo com a norma ISO/IEC 25010, a confidencialidade, a integridade e a disponibilidade são considerados requisitos não-funcionais de *software* (International Organization for Standardization, 2011). Requisitos não-funcionais se referem a maneira em que um software irá operar sob determinadas circunstancias e são relacionados ao desempenho, disponibilidade, usabilidade, etc. (POHL; RUPP, 2015), ou seja, são atributos de qualidade associados a um software (REINEHR, 2020).

Esses requisitos estão associados ao modelo de qualidade para produtos de software baseado na norma ISO/IEC 25010 (REINEHR, 2020). Esse modelo é ilustrado na Figura 2. O modelo define 8 características e 31 subcaracterísticas, sendo a confiabilidade e segurança uma de suas características.

Figura 2 – Modelo de qualidade para produtos de software da ISO/IEC 25010

Fonte: (REINEHR, 2020).

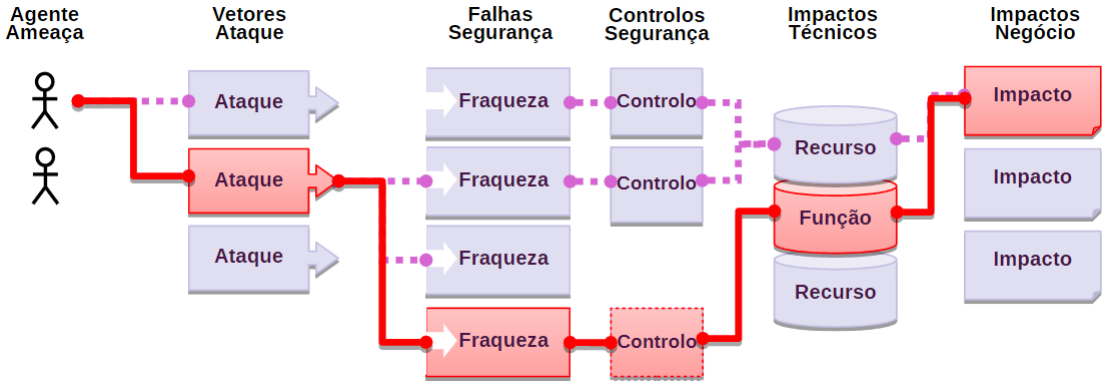
Na característica confiabilidade, o conceito de disponibilidade é definido como uma medida quantitativa da quantidade de tempo que o produto está disponível para acesso quando necessário. A característica de segurança define confidencialidade como a garantia de que os dados são acessados apenas por aqueles que devem ter acesso e integridade como a prevenção do acesso por pessoas não autorizadas a dados e programas (REINEHR, 2020).

Como ilustrado na Figura 3, atacantes (agentes de ameaça) podem utilizar de diferentes caminhos para afetar uma aplicação, causando a "quebra"de um ou mais dos princípios da tríade CIA (Open Worldwide Application Security Project, 2017). Estes caminhos utilizam de um vetor de ataque como vulnerabilidades em *softwares*. Assim, falhas de segurança são exploradas e permitem que um controle de segurança seja burlado para o acesso de um recurso ou função. Este acesso não autorizado gera um impacto para a organização (como impactos financeiros).

Uma vulnerabilidade é definida como uma fraqueza em um ativo ou grupo de ativos que pode ser explorada por uma ou mais ameaças. Um ativo é qualquer coisa que possua valor para uma organização, suas operações e a continuidade de suas operações (International Organization for Standardization, 2008). Vulnerabilidades não incluem apenas *bugs* gerados no código fonte, também fraquezas causadas por configurações de segurança, hipóteses de confiança incorretas e análises de risco desatualizadas (SOUPPAYA; SCARFONE; DODSON, 2022).

Os ataques emergem vulnerabilidades de um produto de *software* e são iniciados por ameaças que, por sua vez, são:

Figura 3 – Caminhos possíveis para um ataque



Fonte: (Open Worldwide Application Security Project, 2017).

> "agentes ou condições que causam incidentes que comprometem as informações e seus ativos por meio da exploração de vulnerabilidades, provocando perdas de confidencialidade, integridade e disponibilidade, e, consequentemente, causando impactos aos negócios de uma organização" (SEMOLA, 2013).

Ameaças são classificadas como: (i) naturais, causadas por eventos da natureza como desastres naturais; (ii) involuntárias, como acidentes, erros ou quedas de energia; e (iii) voluntárias, causadas por agentes humanos, como *hackers* ou *malwares* (SEMOLA, 2013).

Com base na Agência Europeia para a Segurança das Redes e da Informação (ENISA), "um vetor de ataque é um meio pelo qual um agente mal intencionado pode abusar de fraquezas e vulnerabilidades em ativos para alcançar um resultado específico" (European Union Agency for Cybersecurity, 2018). Configurações incorretas em componentes de *software* ou *hardware* são exemplos de vetores de ataques e servem como ponto de entrada para um atacante nestes sistemas (MARTINEZ; COSENTINO; CABOT, 2017).

Vetores de ataque levam a falhas de segurança. Uma falha é definida como o momento em que ocorre a incapacidade de um produto de realizar uma função obrigatória. Falhas de segurança ocorrem na incapacidade de realização de funções relacionadas à segurança (Institute of Electrical and Electronics Engineers, 2010).

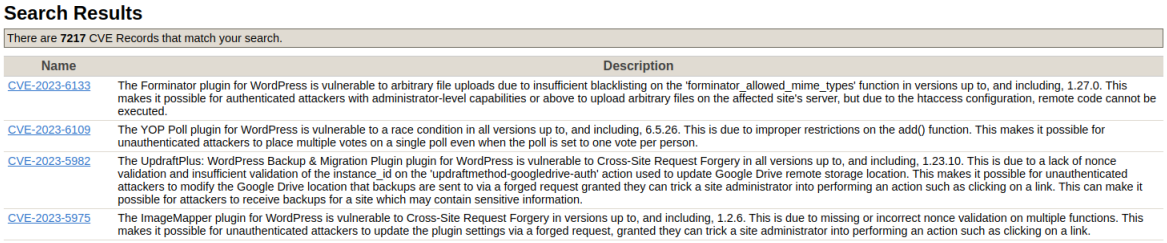
Agentes mal intencionados utilizam de vetores de ataques (como vulnerabilidades em sistemas) para atacar ativos. Estes ataques podem ser realizados com *exploits*. *Exploits* são fragmentos de softwares utilizados para atacar diferentes *hardwares* ou *softwares* tomando proveito de uma vulnerabilidade. *Exploits* são projetados para causar danos nos sistemas, alterando seu comportamento padrão e gerando algum benefício para o atacante (VARELA-VACA *et al.*, 2023).

Vulnerabilidades de sistemas são encontradas em repositórios de vulnerabilidades. Estes repositórios fornecem informações sobre como os vetores de ataque e as vulnerabilidades interagem em um sistema vulnerável. Se pode citar repositórios de vulnerabilidade como o

Common Vulnerabilities and Exposures (CVE)[1], o *National Vulnerability Database* (NVD)[2] e o Open Source Vulnerability Database (VulDB)[3]. As vulnerabilidades são organizadas em um modelo padrão (CVEs), contendo um identificador único, uma descrição e uma nota de severidade calculada a partir de um método padrão denominado *Common Vulnerability Scoring System* (CVSS)[4].

As Figuras 4, 5 e 6 ilustram, respectivamente, o resultado de uma pesquisa nos bancos de dados CVE, NVD e VulDB por vulnerabilidades recentes associadas ao sistema de gerenciamento de conteúdos *Wordpress*[5] (utilizado para desenvolver aplicativos web).

Figura 4 – Pesquisa por "*Wordpress*" no CVE



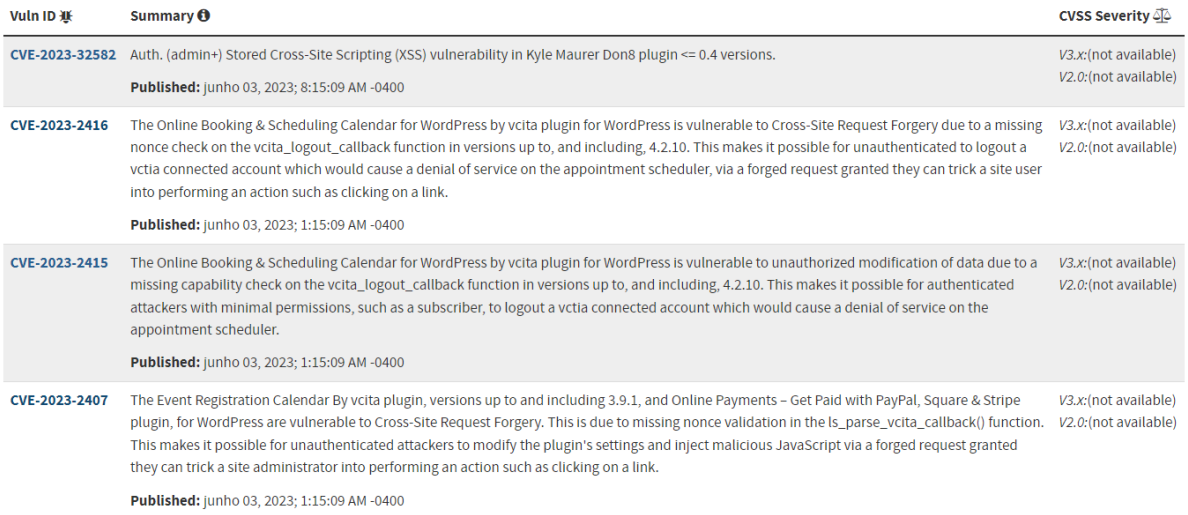
Search Results

There are **7217** CVE Records that match your search.

Name	Description
CVE-2023-6133	The Forminator plugin for WordPress is vulnerable to arbitrary file uploads due to insufficient blacklisting on the 'forminator_allowed_mime_types' function in versions up to, and including, 1.27.0. This makes it possible for authenticated attackers with administrator-level capabilities or above to upload arbitrary files on the affected site's server, but due to the htaccess configuration, remote code cannot be executed.
CVE-2023-6109	The YOP Poll plugin for WordPress is vulnerable to a race condition in all versions up to, and including, 6.5.26. This is due to improper restrictions on the add() function. This makes it possible for unauthenticated attackers to place multiple votes on a single poll even when the poll is set to one vote per person.
CVE-2023-5982	The UpdraftPlus: WordPress Backup & Migration Plugin plugin for WordPress is vulnerable to Cross-Site Request Forgery in all versions up to, and including, 1.23.10. This is due to a lack of nonce validation and insufficient validation of the instance_id on the 'updraftmethod-googledrive-auth' action used to update Google Drive remote storage location. This makes it possible for unauthenticated attackers to modify the Google Drive location that backups are sent to via a forged request granted they can trick a site administrator into performing an action such as clicking on a link. This can make it possible for attackers to receive backups for a site which may contain sensitive information.
CVE-2023-5975	The ImageMapper plugin for WordPress is vulnerable to Cross-Site Request Forgery in versions up to, and including, 1.2.6. This is due to missing or incorrect nonce validation on multiple functions. This makes it possible for unauthenticated attackers to update the plugin settings via a forged request, granted they can trick a site administrator into performing an action such as clicking on a link.

Fonte: Autoria própria (2023).

Figura 5 – Pesquisa por "*Wordpress*" no NVD



Vuln ID	Summary	CVSS Severity
CVE-2023-32582	Auth. (admin+) Stored Cross-Site Scripting (XSS) vulnerability in Kyle Maurer Don8 plugin <= 0.4 versions. **Published:** junho 03, 2023; 8:15:09 AM -0400	*V3.x:*(not available) *V2.0:*(not available)
CVE-2023-2416	The Online Booking & Scheduling Calendar for WordPress by vcita plugin for WordPress is vulnerable to Cross-Site Request Forgery due to a missing nonce check on the vcita_logout_callback function in versions up to, and including, 4.2.10. This makes it possible for unauthenticated to logout a vctia connected account which would cause a denial of service on the appointment scheduler, via a forged request granted they can trick a site user into performing an action such as clicking on a link. **Published:** junho 03, 2023; 1:15:09 AM -0400	*V3.x:*(not available) *V2.0:*(not available)
CVE-2023-2415	The Online Booking & Scheduling Calendar for WordPress by vcita plugin for WordPress is vulnerable to unauthorized modification of data due to a missing capability check on the vcita_logout_callback function in versions up to, and including, 4.2.10. This makes it possible for authenticated attackers with minimal permissions, such as a subscriber, to logout a vctia connected account which would cause a denial of service on the appointment scheduler. **Published:** junho 03, 2023; 1:15:09 AM -0400	*V3.x:*(not available) *V2.0:*(not available)
CVE-2023-2407	The Event Registration Calendar By vcita plugin, versions up to and including 3.9.1, and Online Payments – Get Paid with PayPal, Square & Stripe plugin, for WordPress are vulnerable to Cross-Site Request Forgery. This is due to missing nonce validation in the ls_parse_vcita_callback() function. This makes it possible for unauthenticated attackers to modify the plugin's settings and inject malicious JavaScript via a forged request granted they can trick a site administrator into performing an action such as clicking on a link. **Published:** junho 03, 2023; 1:15:09 AM -0400	*V3.x:*(not available) *V2.0:*(not available)

Fonte: Autoria própria (2023).

Como visto na Figuras 4, 5 e 6, os bancos de dados de vulnerabilidades demonstram as vulnerabilidades encontradas para um determinado produto. Na pesquisa podemos observar qual o CVE associado, uma descrição da vulnerabilidade, data de publicação (apenas no VulDB), e qual a severidade dessa vulnerabilidade (apenas no NVD).

1 *Common Vulnerabilities and Exposures*: https://cve.mitre.org/index.html.
2 *National Vulnerability Database*: https://nvd.nist.gov/.
3 *Open Source Vulnerability Database*: https://vuldb.com/.
4 *Common Vulnerability Scoring System*: https://nvd.nist.gov/vuln-metrics/cvss.
5 *Wordpress*: https://wordpress.com/.

Figura 6 – Pesquisa por "*Wordpress*" no VulDB

Publicado em	Base	Temp	Vulnerabilidade	Prod	Exp	Mas	CTI	EPSS	CVE
23/05/2023	6.3	6.1	WordPress Go Pricing Plugin direitos alargados	Content...	Not Def...	Not Def...	0.00	0.00050	CVE-2023-2494
17/05/2023	5.5	5.3	WordPress Directório Traversal	Content...	Not Def...	Official ...	0.53	0.00235	CVE-2023-2745
15/05/2023	5.7	5.6	Webcodin WCP Contact Form Plugin WordPress Roteiro Cruzado de Sítios	Deacon...	Not Def...	Not Def...	0.00	0.00046	CVE-2023-22703
08/05/2023	4.4	4.4	Branko Borilovic WSB Brands Plugin WordPress Roteiro Cruzado de Sítios	Deacon...	Not Def...	Not Def...	0.00	0.00045	CVE-2022-47437
23/03/2023	5.8	5.8	Blockonomics WordPress Bitcoin Payments Plugin Roteiro Cruzado de Sítios	Content...	Not Def...	Not Def...	0.03	0.00046	CVE-2022-47145

Fonte: Autoria própria (2023).

De maneira análoga à vulnerabilidades, existem repositórios de *exploits*. Se pode citar o *Exploit Database* (Exploit-DB)[6]. Os *exploits* são descritos com: (i) um identificador; (ii) uma lista de CVEs com suas vulnerabilidades associadas; (iii) data de publicação; (iv) tipo de *exploit* (aplicações *web*, códigos executáveis); (v) plataforma (linguagem de programação, sistema operacional...); (vi) autor do *exploit*; (vii) aplicativo alvo; e (viii) código-fonte (VARELA-VACA *et al.*, 2023).

3.1.1 Vulnerabilidades em aplicações web

O projeto OWASP *Top Ten* tem o foco voltado na conscientização de desenvolvedores de software e documenta as vulnerabilidades mais comuns em sistemas web, oferecendo técnicas para a proteção (Open Worldwide Application Security Project, 2017). O projeto recebe submissões de dados de empresas na área de segurança da informação sobre vulnerabilidades identificadas em aplicações e APIs. Essas vulnerabilidades são ordenadas de acordo com a sua prevalência e uma uma estimativa ponderada do potencial de abuso, deteção e impacto (Open Worldwide Application Security Project, 2017).

A última edição do OWASP *Top Ten* ocorreu em 2021. Algumas vulnerabilidades do *rank* são citadas abaixo (Open Worldwide Application Security Project, 2021):

- **Quebra de controle de acesso**: Essa vulnerabilidade ocorre quando um usuário não autorizado acessa uma aplicação ou um usuário com permissões restritas eleva o seu nível de acesso se aproveitando de: (i) validações inadequadas das credenciais fornecidas ou; (ii) divulgação de dados confidenciais ou; (iii) falta de gerenciamento de exceções ou; (iv) redirecionamentos descontrolados; entre outros (HASSAN *et al.*, 2018a);

- **Falhas criptográficas**: Essa vulnerabilidade ocorre por causa da falta de proteções em dados em trânsito e dados armazenados. A vulnerabilidade ocorre quando dados mais sensíveis, como senhas, dados financeiros, informações pessoais não são tratados de maneira correta e especificadas por padrões de mercado, como o *Payment Card*

[6] *Exploit Database*: https://www.exploit-db.com/.

Industry Data Security Standard (PCI DSS) para dados de cartões de crédito (Open Worldwide Application Security Project, 2021);

- **Injeção**: Ataques de injeção (*SQL injection*) são causados pela má validação de entradas fornecidas pelo usuário. Esses ataques são causados por uma má configuração nos servidores de banco de dados. A injeção é utilizada para recuperar informações do banco de dados, alterar informações sem autorização, estabelecer privilégios de acesso, o que pode levar a perda de controle total sobre o servidor (LU; FEI; LIU, 2023);

- **Projeto (*Design*) Inseguro**: Falhas por *design* inseguro surgem de uma má implementação em aplicações, que podem causar comportamento não intencional. Isso permite a manipulação de funcionalidades legítimas para atingir objetivos maliciosos (PortSwigger, 2023). Essas vulnerabilidades são diferentes de vulnerabilidades geradas por implementações inseguras e não podem ser corrigidas com uma implementação perfeita. Isso é corroborado pelo fato de não existirem controles de segurança que previnem falhas de design (Open Worldwide Application Security Project, 2021);

- **Configuração Incorreta de Segurança**: Essas vulnerabilidades são causadas pela falta de configuração dos sistemas, como a utilização de usuários e senhas padrões, e a utilização de configurações padrões nestes sistemas (VARELA-VACA *et al.*, 2019);

- **Componentes Vulneráveis e Desatualizados**: A utilização de bibliotecas de código desatualizadas, componentes do sistema com versões antigas e a falta de atualização nestes componentes, geram vulnerabilidades deste tipo (Open Worldwide Application Security Project, 2021);

- **Falhas de identificação e autenticação**: Ocorrem por má implementação de mecanismos de autenticação e gerenciamento de sessão em aplicativos. Estas falhas surgem pela falta de prevenção de ataques de força bruta em formulários de login, uso de más políticas de senha, uso de sessões muito longas, controle de acesso a funções restritas mal implementados, quebra de mecanismos de autenticação, identificadores de sessão enumeráveis, entre outras (HASSAN *et al.*, 2018b);

- **Falsificação de Solicitação do Lado do Servidor (SSRF)**: Essas vulnerabilidades acontecem quando uma aplicação *web* não filtra requisições realizadas que tentam acessar recursos internos ao servidor. Uma aplicação *web* vulnerável redireciona as requisições para a rede interna e expõe serviços locais a essa rede. Para que esta vulnerabilidade ocorra, o servidor web deve ter acesso a recursos internos de sua rede, o que geralmente é necessário para sua funcionalidade (JABIYEV *et al.*, 2021).

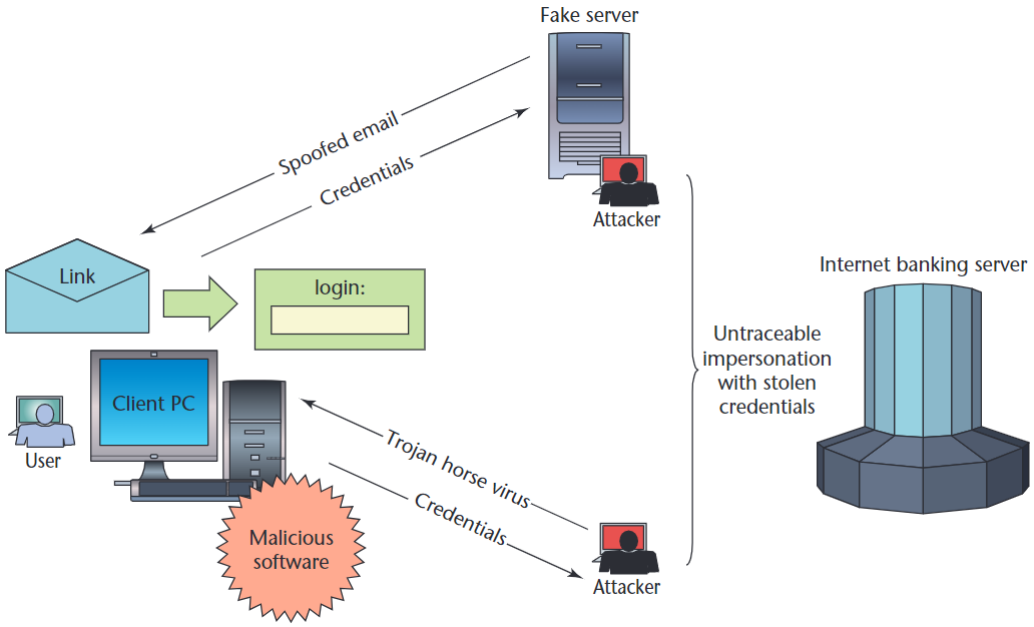
Em um sistema bancário, usuários precisam se autenticar para provar quem realmente são. A autenticação é realizada por um módulo de autenticação, que deve ser resistente ao roubo de credenciais (HILTGEN; KRAMP; WEIGOLD, 2006).

No roubo de credenciais, um atacante pode utilizar softwares mal intencionados (*malwares*) para infectar o computador de um usuário legítimo, o que possibilita o roubo de informações e espionagem deste usuário. Para que este ataque tenha sucesso, o computador da vítima deve estar vulnerável. O uso de *software* antivírus ou sistema operacional desatualizado, e configurações de segurança incorretas são motivos comuns de sistemas vulneráveis (HILTGEN; KRAMP; WEIGOLD, 2006).

Além de *malwares*, atacantes podem utilizar de técnicas de engenharia social para enganar um usuário a fornecer as suas credenciais voluntariamente. Uma das maneiras de realizar este ataque é usando formulários falsos hospedados na internet e divulgados utilizando *e-mail* ou SMS. Desta forma, o usuário insere suas credenciais em um site que pensa ser legítimo (HILTGEN; KRAMP; WEIGOLD, 2006).

A Figura 7 ilustra métodos de roubo de informação, por meio de *softwares* mal intencionados e utilização de engenharia social.

Figura 7 – Roubo de informações

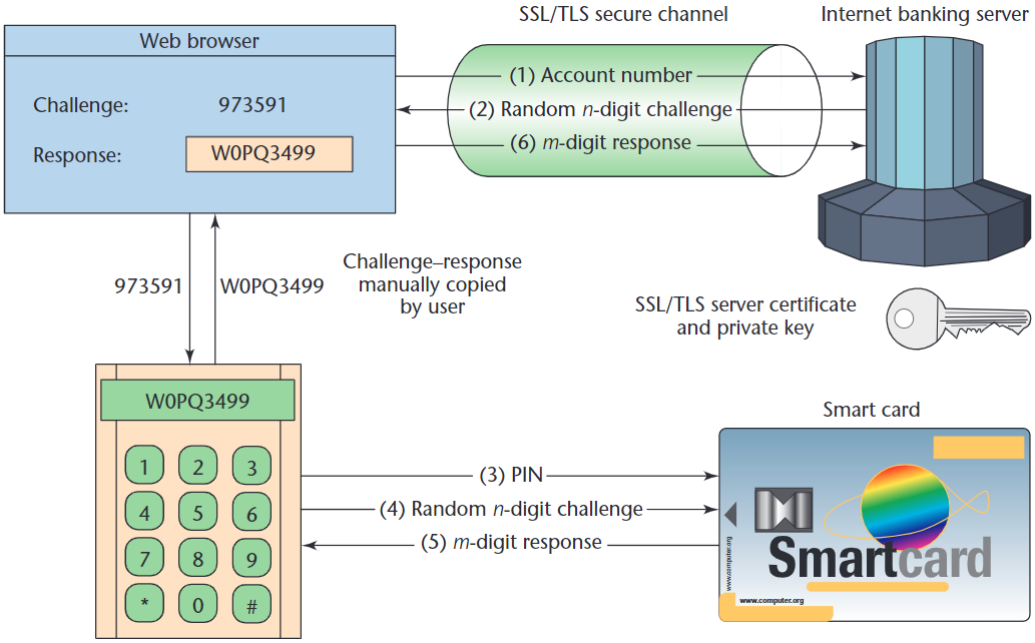


Fonte: (HILTGEN; KRAMP; WEIGOLD, 2006).

Para contornar este problema, Hiltgen, Kramp e Weigold (2006) propuseram um método de autenticação de múltiplos fatores utilizando a combinação de senhas de curta duração, um cartão inteligente e um leitor de cartões *offline*. A Figura 8 apresenta o funcionamento deste método de autenticação. O usuário se conecta com o servidor do banco, fornece o número de sua conta (1), o servidor fornece um código de *n* dígitos (2) e pede uma confirmação de *m* dígitos. No leitor de cartões, o usuário digita um número de identificação para desbloquear o cartão inteligente (3), fornece o código de *n* (4) dígitos que será encriptado com base na chave

criptográfica contida no cartão inteligente e devolve o código de *m* dígitos para o usuário (5). Finalmente, o usuário fornece esse código para o servidor (6) e tem acesso a sua conta.

Figura 8 – Solução de autenticação segura com *smart-card*



Fonte: (HILTGEN; KRAMP; WEIGOLD, 2006).

Métodos atuais de autenticação incluem: autenticação simples utilizando um *e-mail* e senha; autenticação com *e-mail* e senha e um fator adicional, como um código de segurança enviado como uma mensagem instantânea (SMS) (GHELANI; HUA; KODURU, 2022). O método proposto por Ghelani, Hua e Koduru (2022) inclui verificação biométrica em conjunto ao *e-mail* e senha fornecidos pelo cliente. Os dados biométricos do cliente são salvos no banco de dados e verificados ao realizar a tentativa de autenticação. A utilização de biometria para autenticação se prova mais segura que o uso de autenticação simples com *e-mail* e senha e códigos enviados por SMS. Fraudadores podem facilmente roubar as credenciais de um usuário, bem como seu aparelho de SMS, garantindo o acesso a conta. Fatores biométricos não podem ser roubados, e necessitam a presença do usuário legítimo de uma conta para serem utilizados (GHELANI; HUA; KODURU, 2022).

Um método de autenticação forte inclui mais de um tipo de credencial. Essas credenciais podem ser: (i) algo que sabemos (*something we know*), como uma senha decorada; (ii) algo que temos (*something we have*), como cartões inteligentes; ou (iii) algo que somos (*something we have*), como a biometria. Se forem combinados algo que sabemos (uma senha) com algo que temos (um cartão inteligente), um atacante deve aprender a senha e roubar o cartão inteligente para que tenha acesso a uma conta. Isso aumenta a segurança de um módulo de autenticação (KAMBOU; BOUABDALLAH, 2019).

A segurança é um dos principais requisitos não-funcionais na engenharia de software e pode ser representada por meio de modelos (REINEHR, 2020). Assim, os requisitos de se-

gurança de informação tem sido representados com *feature models* por meio de *features* em sistemas de *software* (VARELA-VACA *et al.*, 2021).

3.2 Features

A engenharia de Linhas de Produto de *Software* (LPS) utiliza conceitos de linhas de produção em massa (por exemplo, indústria automobilística (OLIINYK *et al.*, 2015)) e resulta em entregas de *softwares* a baixo custo e em um período curto de tempo (*just-in-time*) (BRITO, 2013; LINDEN; SCHMID; ROMMES, 2007). Dentre as definições de LPS, (POHL; BöCKLE; LINDEN, 2005) define LPS como um paradigma para desenvolvimento de sistemas utilizando plataformas e customização em massa.

A Engenharia de LPS é encontrada em diversas aplicações na indústria. Bosch e Phillips são exemplos de empresas que adotaram a engenharia de LPS[7]. A Bosch utiliza LPS na fabricação de sistemas para motores à gasolina (como bombas e injetores de combustível)[8] e a Phillips na fabricação de televisores[9]. LPS é escolhida pela sua capacidade de gerar produtos de maior qualidade em um menor tempo e gerando uma melhor manutenibilidade (OLIINYK *et al.*, 2015). A utilização adequada de LPS gera uma maior produtividade no ciclo de desenvolvimento de *software*, reduz custos, riscos e melhora a qualidade da entrega de um produto a um cliente de forma a garantir as suas necessidades e expectativas (GERALDI; REINEHR; MALUCELLI, 2020).

A engenharia de LPS diverge do processo de desenvolvimento de software tradicional. Os produtos são personalizados baseados em um núcleo de ativos e uma família de produtos (GERALDI, 2022). Para atingir a personalização requerida pelo cliente, a LPS promove a reutilização de uma família de produtos baseados em *features* em comum aos invés de produzi-los um a um individualmente (BENAVIDES; SEGURA; RUIZ-CORTES, 2010).

Segundo Kang *et al.* (1990), uma *feature* é "um aspecto proeminente ou distintivo visível ao usuário, com qualidade, ou uma característica de um sistema de software ou sistemas". Ainda, com base em Kang *et al.* (1998), *features* são classificadas como "abstrações funcionais distintamente identificáveis que devem ser implementadas, testadas, entregues e mantidas". Bosch, Capilla e Hilliard (2015) e Bosch e Lee (2010) definem uma *feature* como "uma unidade lógica de comportamento especificado por um conjunto de requisitos funcionais e não-funcionais". *Features* representam as características funcionais e não funcionais abstraídas em diferentes fases do desenvolvimento de software (GERALDI, 2022).

Features podem ser consideradas como as primeiras abstrações que necessitam ser compartilhadas entre desenvolvedores e clientes (APEL *et al.*, 2013; KANG *et al.*, 1998). Se-

[7] *Product Line Hall of Fame*: https://splc.net/fame.html.

[8] *Bosch: Gasoline Systems Engine Control Software*: https://splc.net/fame/bosch/.

[9] *Philips Product Line of Software for Television Sets*: https://splc.net/fame/philips-software-for-television-sets/.

gundo Apel *et al.* (2013), representa “uma característica ou comportamento visível ao usuário final de um sistema de *software*”. *Features* também são definidas como uma unidade lógica ou um comportamento especificados por requisitos funcionais e não funcionais, bem como uma característica distinguível e relevante de um sistema ou componente (BERGER *et al.*, 2015). *Features* podem ser típicas ou não típicas (*outliers*). Uma *feature* típica representa uma funcionalidade central e são pré requisitos de um negócio, essas *features* podem ser requisitos de um cliente ou demandadas do mercado. *Features* não típicas não são requisitos visíveis a um usuário e apenas existem na arquitetura de um produto (BERGER *et al.*, 2015).

Em adição, *features* podem ser classificadas como boas ou ruins. Boas *features* tem boa aceitação do usuário e proporcionam funcionalidades distintas em uma LPS. Além disso, boas *features* são bem implementadas e livres de erros. Já *features* ruins são o resultado de um desenvolvimento acelerado e descuidado. Por este motivo, *features* ruins são recebidas pelos usuários de maneira negativa (BERGER *et al.*, 2015).

Em termos gerais, *features* são entendidas como funcionalidades (ou preocupações) de um sistema (BERGER *et al.*, 2020; KRüGER *et al.*, 2018), sendo utilizadas para especificar, gerenciar e comunicar propriedades funcionais e não-funcionais de um sistema de *software*. *Features* ajudam desenvolvedores a compreender e adaptar estes sistemas (KRUGER *et al.*, 2019).

3.2.1 Modelagem de *features*

Segundo Lee, Kang e Lee (2002), a modelagem de *features* pode ser considerada uma “atividade de identificar externamente *features* visíveis de produtos em um domínio e organizá-las dentro de um modelo denominado *feature model*”. Este termo foi apresentado por Kang *et al.* (1990) no estudo *Feature-Oriented Domain Analysis* (FODA), que, segundo Benavides, Segura e Ruiz-Cortes (2010) “captura informações da linha de produtos de software sobre *features* comuns e variantes da linha de produtos de *software* em diferentes níveis de abstração”.

Features são representadas por meio de *feature models* (CZARNECKI; HELSEN; EISENECKER, 2004). Um *feature model* possui a descrição das *features* e suas associações e relacionamentos (GERALDI, 2022). Um diagrama de *features* pode ser definido como uma representação gráfica que possui uma hierarquia de *features* com relacionamentos e/ou outras (APEL *et al.*, 2013; LEE; KANG; LEE, 2002).

O diagrama de *features* é representado por uma árvore, A raiz da árvore representa um conceito (por exemplo, sistemas de software) e as folhas são as *features* (CZARNECKI; HELSEN; EISENECKER, 2004). A Figura 9 representa um *feature model* inspirado na indústria de telefones (BENAVIDES; SEGURA; RUIZ-CORTES, 2010). Este modelo determina uma família de softwares que têm características em comum (*features* suportadas por um telefone). De acordo com o *feature model* (BENAVIDES; SEGURA; RUIZ-CORTES, 2010), se percebe que:

1. Todos os softwares de celular devem oferecer suporte a chamadas e a tela, seja ela básica, colorida ou de alta resolução;

2. Os *softwares* para celulares podem conter suporte a mídias;

3. *Softwares* de celulares que possuem suporte a GPS excluem uma tela básica, e aqueles com suporte a câmera necessitam de uma tela de alta resolução.

As associações entre as *features* podem ser (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013):

- **Obrigatórios (*Mandatory*)**: uma *feature* filha é considerada obrigatória quando é incluída em todos os produtos que a *feature* mãe aparece (todos os *softwares* de celulares devem dar suporte a chamadas e a tela);

- **Opcionais (*Optional*)**: uma *feature* filha é considerada opcional quando pode ser incluída em todos os produtos que a *feature* mãe aparece (*softwares* de celulares podem oferecer suporte a GPS e mídia);

- **Alternativa (*Alternative*)**: representa uma relação exclusiva. Apenas uma das *features* filhas pode ser incluída nos produtos que uma *feature* pai aparece (o *software* de um celular pode dar suporte a apenas um tipo de tela: básica; colorida; de alta resolução);

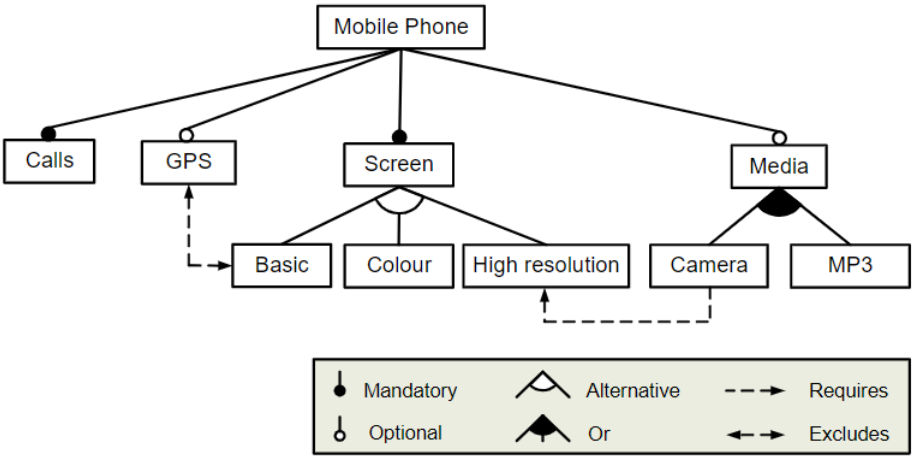
- **Ou (*OR*)**: representa uma relação inclusiva. Uma ou mais *features* filhas podem ser incluídas nos produtos que uma *feature* pai aparece (o *software* de celular pode ter suporte a câmera e/ou MP3).

Além das associações entre parentes, diagramas de *features* podem conter relações transversais entre diferentes subárvores (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013):

- **Requerida (*Requires*)**: Se uma *feature* A necessita de uma *feature* B então a inclusão de uma *feature* A em um produto implica na inclusão da *feature* B (para que o *software* de celular possua suporte a câmera este deve ter suporte a uma tela de alta resolução);

- **Excluída (*Excludes*)**: Se uma *feature* A excluir a *feature* B, ambas as *features* não podem fazer parte do mesmo produto (o *software* de celular não pode oferecer suporte a GPS se possuir uma tela básica).

Figura 9 – Exemplo de *feature model*

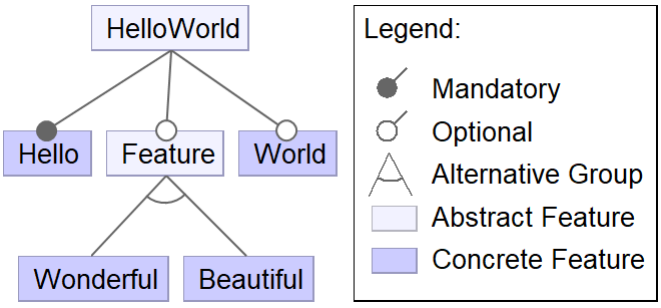


Fonte: (BENAVIDES; SEGURA; RUIZ-CORTES, 2010).

A modelagem de *feature models* pode ser realizada utilizando *softwares* como pure::variants[10], Gears[11], *Software Product Line Online Tools* (S.P.L.O.T.)[12], XFeature[13], But4Reuse[14] e FeatureIDE[15].

O FeatureIDE é um projeto de código aberto que funciona na IDE do Eclipse[16]. Esta ferramenta (*plug-in* interno para EclipseIDE) permite a representação de *feature models* e possui exemplos prévios de *feature models* em alguns domínios de aplicação. A Figura 10 demonstra o *feature model* de uma família de softwares de *Hello World*.

Figura 10 – *Feature model* da família de produtos *Hello World*



Fonte: (MEINICKE *et al.*, 2017).

O *feature model* do exemplo *Hello World* da Figura 10 demonstra uma família de produtos de um *software* que apresenta na tela a mensagem "*Hello World*". Esta família pode ter produtos que mostrem "*Hello World*", "*Hello Wonderful World*" ou "*Hello Beautiful World*". Isto é

[10] pure::variants: https://www.pure-systems.com/purevariants.
[11] Gears: https://biglever.com/solution/gears/.
[12] *Software Product Line Online Tools* (S.P.L.O.T.): http://www.splot-research.org/.
[13] XFeature: https://www.pnp-software.com/XFeature/Home.html.
[14] But4Reuse: https://but4reuse.github.io/.
[15] FeatureIDE: https://www.featureide.de/.
[16] Eclipse: https://www.eclipse.org/.

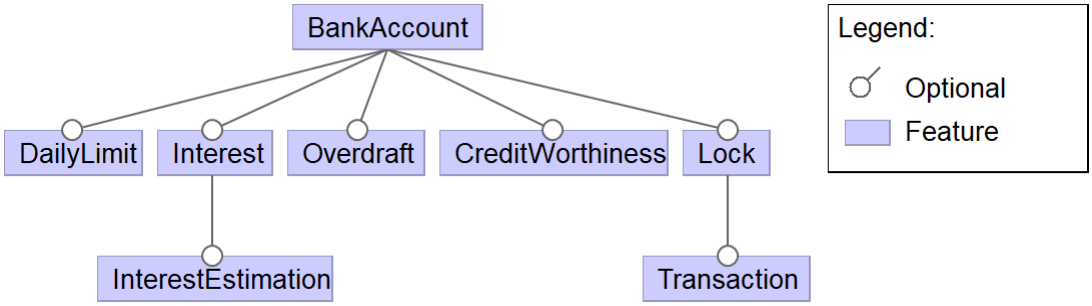
perceptível pois as *features Hello* e *World* são obrigatórias, enquanto as *features Wonderful* e *Beautiful* são opcionais e alternativas.

É possível observar na Figura 10 que o diagrama é composto por dois tipos de *features*: abstratas e concretas. *Features* abstratas são utilizadas apenas para estruturar o diagrama como um requisito não-funcional (por exemplo, segurança e usabilidade), enquanto *features* concretas são requisitos funcionais (por exemplo, o algoritmo MD5 ou SHA) (MEINICKE *et al.*, 2017).

Além de possibilitar representação de *feature models*, a ferramenta FeatureIDE suporta todo o processo de desenvolvimento orientado a *features*. A estrutura de um projeto contém: (i) *feature model* para definir as *features*; (ii) configurações de *features* que representam um produto; (iii) arquivos que implementam a funcionalidade das *features*; e (iv) código-fonte que representa o produto para as configurações selecionadas (MEINICKE *et al.*, 2017).

A Figura 11 ilustra o *feature model* de uma família de sistemas bancários. Neste exemplo, a *feature BankAccount* implementa uma conta bancária simples com uma função de atualizar o saldo, sem demais funções como limites de transferência diários (*feature DailyLimit*). As outras *features* são opcionais.

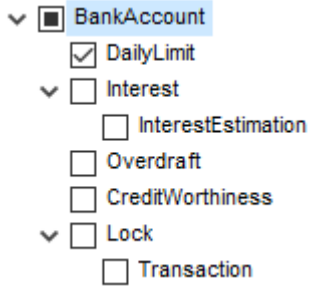
Figura 11 – *Feature model* de uma família de softwares bancário



Fonte: (THUM *et al.*, 2014).

Para gerar um produto específico a partir dessa LPS, se pode configurar quais *features* serão utilizadas. A Figura 12 demonstra a configuração de um produto que inclui a funcionalidade de limites de transferência, além da *feature* principal.

Figura 12 – Configuração de um software possível na LPS



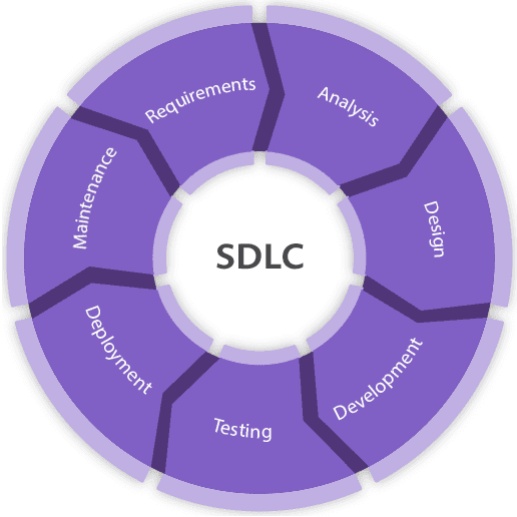
Fonte: (THUM *et al.*, 2014).

A adição de segurança durante todo o ciclo de desenvolvimento de *software*, possibilita entender as ameaças de segurança de um sistema, facilitando a resolução destes problemas no componente antes da fase de implementação (DAWSON *et al.*, 2010). Isso pode ser realizado por meio da adição de praticas de desenvolvimento seguro em cada fase do processo de desenvolvimento de software (SOUPPAYA; SCARFONE; DODSON, 2022).

3.3 Desenvolvimento seguro de software

Ciclos de vida de desenvolvimento de software (SDLC) descrevem como aplicações são construídas e geralmente são compostos de 7 fases distintas: Elicitação dos requisitos; análise dos requisitos; design de *features* baseadas nos requisitos; desenvolvimento das *features*; teste e verificação das *features* quantos aos requisitos coletados; implantação do produto; e manutenção e evolução do produto (Snyk, 2023). Essas fases são ilustradas pela Figura 13.

Figura 13 – Fases do ciclo de vida de desenvolvimento de *software*



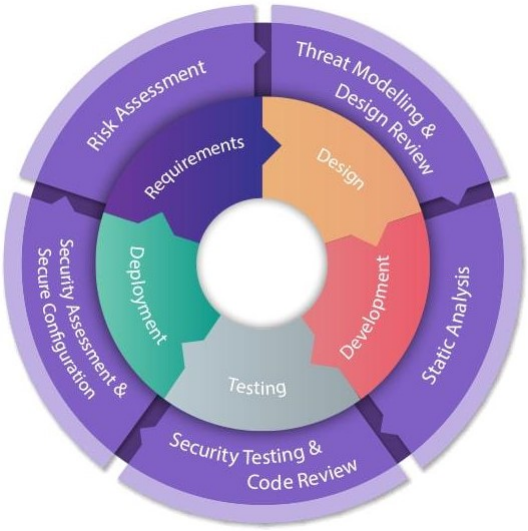
Fonte: (Snyk, 2023).

SDLC é uma metodologia repetitiva, e os desenvolvedores devem garantir a qualidade do código em cada etapa e eliminar trabalho redundante por meio do *feedback* de usuários e das partes interessadas (*stakeholders*) (KAMAL *et al.*, 2020).

O ciclo de vida de desenvolvimento de software seguro (SSDLC) integra verificações de segurança nas fases do ciclo de vida. Essas verificações permitem que um time identifique problemas de segurança durante todo o ciclo do desenvolvimento. A identificação e correção de problemas durante os estágios de desenvolvimento resulta em um *software* mais seguro e com uma menor quantidade de vulnerabilidades, diminuindo possíveis impactos a uma organização (KAMAL *et al.*, 2020).

A Figura 14 apresenta as verificações de segurança em cada fase de um SSDLC. As fases e suas verificações de segurança são:

Figura 14 – Ciclo de vida de desenvolvimento de *software* seguro



Fonte: (Snyk, 2023).

- **Requisitos**: se concentra na coleta dos detalhes do projeto fornecidos pelo cliente e na definição dos requisitos para o desenvolvimento do *software*. Nesta fase, são definidos os requisitos de segurança e é realizada uma avaliação de risco (*Risk Assessment*) (KAMAL *et al.*, 2020). A avaliação de risco foca na identificação, análise e priorização de riscos e é utilizado para determinar a extensão de uma possível ameaça e o risco associado a um sistema (STONEBURNER; GOGUEN; FERINGA, 2002);

- ***Design***: inclui atividades que ocorrem antes do desenvolvimento do código, como a definição da arquitetura do software e de seu fluxo de dados. A modelagem de ameaças (*Threat Modelling*) é realizada nessa fase e consiste na identificação de possíveis ameaças e nas técnicas que podem ser utilizadas para mitigar possíveis ataques (XIONG; LAGERSTROM, 2019);

- **Implementação**: corresponde ao desenvolvimento do código do sistema. Análise estática do código fonte é utilizada nesta fase. Este processo normalmente envolve a utilização de ferramentas automatizadas que analisam o código fonte sem executá-lo. A estrutura do código e sequência das declarações é analisada, além de como as variáveis são processadas em cada chamada de função (LI *et al.*, 2017);

- **Teste**: nesta fase são realizados testes de segurança e revisões no código. Revisões no código podem ser feitas com ferramentas automatizadas ou manualmente e tenta identificar fraquezas no código. Uso apenas de ferramentas automatizadas não é totalmente confiável e, idealmente, revisões no código devem ser feitas por pessoas de fora da equipe de desenvolvimento (BUTTNER *et al.*, 2020). Testes de segurança são realizados para confirmar se o sistema satisfaz requisitos relacionados às propriedades de segurança, como confidencialidade, integridade e disponibilidade e se estes requisitos

estão corretamente implementados. Estes testes são realizados tentando mostrar conformidade com as propriedades de segurança (considerando entradas bem definidas e esperadas pelo sistema) ou tentando encontrar vulnerabilidades comuns (com o uso de entradas maliciosas e inesperadas) (FELDERER *et al.*, 2016);

- **Implantação**: nesta fase é importante garantir configurações complacentes com segurança, como criptografia de rede e permissões corretas. A manutenção no *software* também deve ser feita, para melhorar a experiência do usuário e corrigir vulnerabilidades encontradas após a implantação do sistema por meio de *patches*. É importante a implantação de um plano de resposta a incidentes, para ser executado imediatamente na ocorrência de um ataque (KAMAL *et al.*, 2020);

Poucos modelos de ciclo de vida de desenvolvimento de software incluem segurança da informação de maneira detalhada, fazendo com que práticas de desenvolvimento seguro sejam adicionadas em cada ciclo do SDLC (SOUPPAYA; SCARFONE; DODSON, 2022).

Independente do modelo de ciclo de vida utilizado, práticas de desenvolvimento seguro devem ser integradas para: (i) reduzir o número de vulnerabilidades no software; (ii) reduzir o potencial de impacto devido a exploração de vulnerabilidades não conhecidas; e (iii) abordar as causas de vulnerabilidades e evitar novas ocorrências (SOUPPAYA; SCARFONE; DODSON, 2022).

Essas práticas de desenvolvimento seguro podem ser adicionadas por meio de *frameworks* específicos, como o BSA *Framework for Secure Software* e NIST *Secure Software Development Framework*.

3.4 *Frameworks* para desenvolvimento seguro

3.4.1 BSA *Framework for Secure Software*

O BSA *Framework for Secure Software* possui uma série de práticas para o desenvolvimento seguro de software, sendo o primeiro *framework* a consolidar melhores práticas para medir o nível de segurança independente do ambiente. Este *framework* tem como objetivo: (i) descrever o estado atual de segurança de *software* em produtos individuais; (ii) descrever o estado alvo para segurança de informação em produtos de software; (iii) identificar e priorizar oportunidades de melhorias nos processos de desenvolvimento e gerenciamento de ciclo de vida; e (iv) comunicar segurança de software e riscos entre as partes interessadas (*stakeholders*) (Business Software Alliance, 2019).

O *framework* é organizado em: funções, categorias, subcategorias, declarações de diagnóstico, notas de implementação e referências. Funções organizam as atividades de segurança de *software* no nível mais alto, sendo elas: (i) desenvolvimento seguro, aborda segurança durante toda a fase de desenvolvimento de um *software*; (ii) capacidades de segurança, identifica

características de segurança recomendadas para um produto de *software*; e (iii) ciclo de vida seguro, aborda como manter a segurança em um produto de *software* do seu desenvolvimento até o fim do seu ciclo de vida (Business Software Alliance, 2019).

Além das funções, o *framework* descreve (Business Software Alliance, 2019):

- **Categorias**: dividem uma função em práticas relevantes, como codificação segura e gerenciamento de vulnerabilidades;
- **Subcategorias**: dividem uma categoria em conceitos que expressam boas práticas de segurança;
- **Declarações de diagnósticos**: identificam os resultados esperados da implementação de uma prática
- **Notas de implementação**: fornecem informações adicionais (por exemplo, a maneira de como uma organização pode alcançar os resultados esperados descritos nas declarações de diagnósticos);
- **Referências**: apontam para outras fontes de informação sobre as práticas descritas

A função de desenvolvimento seguro deste *framework* possui a categoria (prática) de codificação segura, ilustrada na Figura 15. Uma das suas subcategorias indica que o *software* deve ser seguro contra vulnerabilidades conhecidas, funções e bibliotecas inseguras. Uma das declarações de diagnóstico relacionadas a essa subcategoria indica que o *software* evita ou possui documentação sobre como mitigar vulnerabilidades contidas em funções e bibliotecas utilizadas. O *framework* comenta que o software deve evitar vulnerabilidades conhecidas dentro do possível, mas que se for necessário a inclusão de bibliotecas ou funções vulneráveis, deve existir documentação de como mitigar essas vulnerabilidades, garantindo que não poderão ser exploradas no produto final (Business Software Alliance, 2019).

3.4.2 NIST *Secure Software Development Framework*

Assim como o BSA *Framework for Secure Software*, o NIST *Secure Software Development Framework* (SSDF) descreve uma série de boas práticas baseadas em padrões estabelecidos, orientações e documentos para o desenvolvimento de *software* seguro. Este *framework* pode ser implantado em conjunto com práticas pré-existentes no desenvolvimento de *software* seguro e utilizado como uma documentação de requisitos de segurança para guiar a aquisição de *softwares* complacentes às práticas abordadas (SOUPPAYA; SCARFONE; DODSON, 2022).

O SSDF não prescreve como implementar as práticas e foca nos resultados obtidos com base na implementação dessas práticas. As práticas são descritas em linguagem comum e faz com que este *framework* possa ser utilizado por diversas organizações, aplicado em diferentes

Figura 15 – Exemplo de prática do *framework* BSA *Framework for Secure Software*

Category	Subcategory	Diagnostic Statement	Comments on Implementation	Relevant Standards and Informative Resources
SECURE DEVELOPMENT				
Secure Coding (SC) *(continued)*	**SC.2.** Software is developed according to recognized, enforceable coding standards.	**SC.2-1.** Standards are formally identified and documented.		ISO/IEC TS 17961; SEI CERT C Coding Standard; SEI CERT C++ Coding Standard; SEI CERT Java Coding Standard; NCSC
		SC.2-2. Software uses canonical data formats.		SAFECode "Fundamental Practices"; CWE-21; CWE-22; CWE-35; CWE-36; CWE-37; CWE-38; CWE-39; CWE-40
	SC.3. The software is secure against known vulnerabilities, unsafe functions, and unsafe libraries.	**SC.3-1.** Software avoids, or includes documented mitigations for, known security vulnerabilities in included functions and libraries.	Software should avoid known vulnerabilities to the greatest extent possible. In some instances, there may be reasons for software to incorporate functions or libraries known to include vulnerabilities; such functions or libraries should only be incorporated when developers include documented mitigations that ensure the vulnerabilities are not exploitable.	NIST NVD; CWE/SANS Top 25 Most Dangerous Software Errors; OWASP Top 10; CWE-1006; CWE-242

Fonte: (Business Software Alliance, 2019).

nichos e integrado com fluxos de trabalho pré definidos (SOUPPAYA; SCARFONE; DODSON, 2022).

As práticas do *framework* SSDF são divididas em quatro grupos: (i) preparar a organização (PO), focado em preparar pessoas, processos e tecnologias para realizarem desenvolvimento de *software* seguro; (ii) proteger o *software* (PS), com práticas para proteger o código de acesso não autorizado; (iii) desenvolver *software* bem estruturado (PW), com práticas para implementar *softwares* seguros e com a menor quantidade de vulnerabilidades; e (iv) responder a vulnerabilidades (RV), para identificar e responder a vulnerabilidades residuais e prevení-las no futuro (SOUPPAYA; SCARFONE; DODSON, 2022).

O *framework* obedece uma estrutura padrão que inclui os seguintes elementos: (i) prática, contendo o nome da pratica, um identificador único e uma breve explicação; (ii) tarefas, com ações necessárias para executar a prática; (iii) exemplos de implementação, com exemplos conceituais de ferramentas, processos ou outros métodos que podem ser utilizados para implementar uma tarefa; e (iv) referências, contendo mais documentos sobre uma tarefa (SOUPPAYA; SCARFONE; DODSON, 2022).

Uma das práticas descritas por este *framework*, descreve maneiras de criar código-fonte que adere às práticas de desenvolvimento seguro. A prática é ilustrada na Figura 16. O *framework* alega que essa prática diminui o número de vulnerabilidades, gerando uma redução nos custos de manutenção do *software*. A tarefa relacionada a essa prática é de seguir todas as práticas de codificação segura apropriadas a linguagem de programação e ambiente de desenvolvimento utilizados. Exemplos de como implementar essa prática incluem: (i) validar todas

as entradas e saídas; (ii) evitar a utilização de funções não seguras; (iii) detectar e lidar com erros; (iv) implementar *log* de dados; (v) utilizar uma formatação padrão para o código-fonte; entre outras (SOUPPAYA; SCARFONE; DODSON, 2022).

Figura 16 – Exemplo de prática do *framework* NIST SSDF

	***PW.4.5**: Moved to PW.4.1 and PW.4.4*		
Create Source Code by Adhering to Secure Coding Practices (PW.5): Decrease the number of security vulnerabilities in the software, and reduce costs by minimizing vulnerabilities introduced during source code creation that meet or exceed organization-defined vulnerability severity criteria.	**PW.5.1**: Follow all secure coding practices that are appropriate to the development languages and environment to meet the organization's requirements.	**Example 1**: Validate all inputs, and validate and properly encode all outputs. **Example 2**: Avoid using unsafe functions and calls. **Example 3**: Detect errors, and handle them gracefully. **Example 4**: Provide logging and tracing capabilities. **Example 5**: Use development environments with automated features that encourage or require the use of secure coding practices with just-in-time training-in-place. **Example 6**: Follow procedures for manually ensuring compliance with secure coding practices when automated methods are insufficient or unavailable. **Example 7**: Use tools (e.g., linters, formatters) to standardize the style and formatting of the source code. **Example 8**: Check for other vulnerabilities that are common to the development languages and environment. **Example 9**: Have the developer review their own human-readable code to complement (not replace) code review performed by other people or tools. See PW.7.	**BSAFSS**: SC.2, SC.3, LO.1, EE.1 **BSIMM**: SR3.3, CR1.4, CR3.5 **EO14028**: 4e(iv), 4e(ix) **IDASOAR**: 2 **IEC62443**: SI-1, SI-2 **ISO27034**: 7.3.5 **MSSDL**: 9 **OWASPASVS**: 1.1.7, 1.5, 1.7, 5, 7 **OWASPMASVS**: 7.6 **SCFPSSD**: Establish Log Requirements and Audit Practices, Use Code Analysis Tools to Find Security Issues Early, Handle Data Safely, Handle Errors, Use Safe Functions Only **SP800181**: SP-DEV-001; T0013, T0077, T0176; K0009, K0016, K0039, K0070, K0140, K0624; S0019, S0060, S0149, S0172, S0266; A0036, A0047

Fonte: (SOUPPAYA; SCARFONE; DODSON, 2022).

3.5 *Features* em segurança da informação

Segurança da informação é uma área com poucos estudos na área de LPS. A maioria desses estudos foca na aplicação de requisitos de segurança em LPS (VARELA-VACA *et al.*, 2019; VARELA-VACA *et al.*, 2021). Exemplos citam o uso de *feature models* no contexto de LPS para apoiar o gerenciamento de configurações em sistemas, resultando em sistemas seguros por configuração.

Varela-Vaca *et al.* (2019) propuseram a utilização de *feature models* para a detecção de configurações não complacentes de políticas de segurança da informação na configuração de sistemas. A aplicação de *feature models* é proposta como um *baseline* para a validação de configurações corretas de *softwares* e produtos. Segundo Varela-Vaca *et al.* (2019), *feature models* podem ser considerados como um projeto baseado em modelo: "os produtos e sistemas de software são projetados de acordo com os requisitos das políticas de projeto de segurança cibernética".

Varela-Vaca *et al.* (2021) criaram o *framework* CARMEN, que utiliza *feature models* e um catálogo de configurações recomendadas para fazer o gerenciamento de configurações, que garantem a segurança de sistemas ciber-físicos. Este *framework* foca na fase de desenvolvimento de sistemas ciber-físicos "representando um sistema de suporte que guia todo o processo de levantamento de requisitos de segurança" (VARELA-VACA *et al.*, 2021).

Uma outra solução para automatizar a análise e teste de vulnerabilidades de sistemas foi proposto por Varela-Vaca *et al.* (2020). Um *framework* chamado AMADEUS foi proposto para guiar a criação e interpretação de *feature models* extraídos da infraestrutura de uma organização e de vulnerabilidades recuperadas de repositórios de vulnerabilidades. A partir da análise da infraestrutura utilizada, esse *framework* procura por vulnerabilidades relacionadas aos serviços da organização em repositórios de vulnerabilidades e gera *feature models* contendo as possíveis configurações que resultam em sistemas vulneráveis a ataques.

O *framework* AMADEUS foi melhorado em Varela-Vaca *et al.* (2023). Este trabalho descreve as adições e melhoramentos realizados no *framework*, que foi nomeado AMADEUS-Exploit. Foram adicionados mais repositórios de vulnerabilidades para recuperar os dados e foi levado em consideração os *exploits* de uma vulnerabilidade para montar os *feature models*. O trabalho apresenta a vantagem de se utilizar *feature models* para representar as informações coletadas, como a centralização de toda a informação em um modelo unificado e a possibilidade da utilização de sistemas de análise automatizados (VARELA-VACA *et al.*, 2023).

Feature models vêm sendo utilizados em poucos estudos para modelar e especificar vulnerabilidades em softwares específicos. Kenner *et al.* (2020) conduziram um estudo exploratório no navegador web *Mozilla Firefox*, extraindo informações sobre vulnerabilidades conhecidas para criar *feature models* de vulnerabilidades. Para validar as informações coletadas, foram criados ambientes para testá-las. Assim, as informações e o *feature model* foram validadas(o).

A utilização de *feature models* no campo da segurança da informação tem emergido como uma estratégia para gerenciar, especificar e detectar vulnerabilidades em sistemas. Abordagens como a detecção de configurações não complacentes de políticas de segurança (VARELA-VACA *et al.*, 2019) e o desenvolvimento de *frameworks* como AMADEUS (VARELA-VACA *et al.*, 2020) e CARMEN (VARELA-VACA *et al.*, 2021) aproveitam as vantagens dos *feature models* para garantir segurança em sistemas.

Estes avanços na literatura reforçam o potencial dos *feature models* como recursos úteis para impulsionar práticas de desenvolvimento seguro. Neste sentido, esta pesquisa corrobora na representação das práticas dos *frameworks* de desenvolvimento seguro BSA e NIST por meio de *feature models*. Os *feature models* criados são descritos no próximo capítulo.

4 *FEATURE MODELS*: BSA E NIST

Como mencionado na revisão da literatura, *frameworks* de segurança podem ser utilizados para guiar o desenvolvimento seguro de aplicações. Essas práticas podem ser organizadas por meio da abordagem de *feature models*, gerando um modelo unificado com todas as principais afirmações sobre as *features* de segurança de cada *framework*.

Esta seção apresenta o uso de *feature models* para a geração de um modelo unificado contendo as práticas dos *frameworks* abordados nesta pesquisa, sendo um possível caminho de apresentação das práticas de maneira facilitada.

4.1 *Feature models*

Dois *feature models* foram modelados utilizando da ferramenta *FeatureIDE*. Os *feature models* representam graficamente as principais práticas dos *frameworks* BSA *Framework for Secure Software* e NIST *Secure Software Development Framework*. Cada *feature model* representa e descreve as práticas de cada *framework*.

Para o *framework* BSA, as *features* principais foram definidas com as categorias das práticas. As categorias são representadas por *features* abstratas e as práticas de cada categoria são *features* concretas. As práticas de cada categoria foram colocadas como *child features*. Por exemplo, as práticas das funções de capacidades de segurança foram adicionadas como *child features* da *feature* “Proteger o software”, por se tratarem de funcionalidades de segurança do *software*, que o tornam mais seguro.

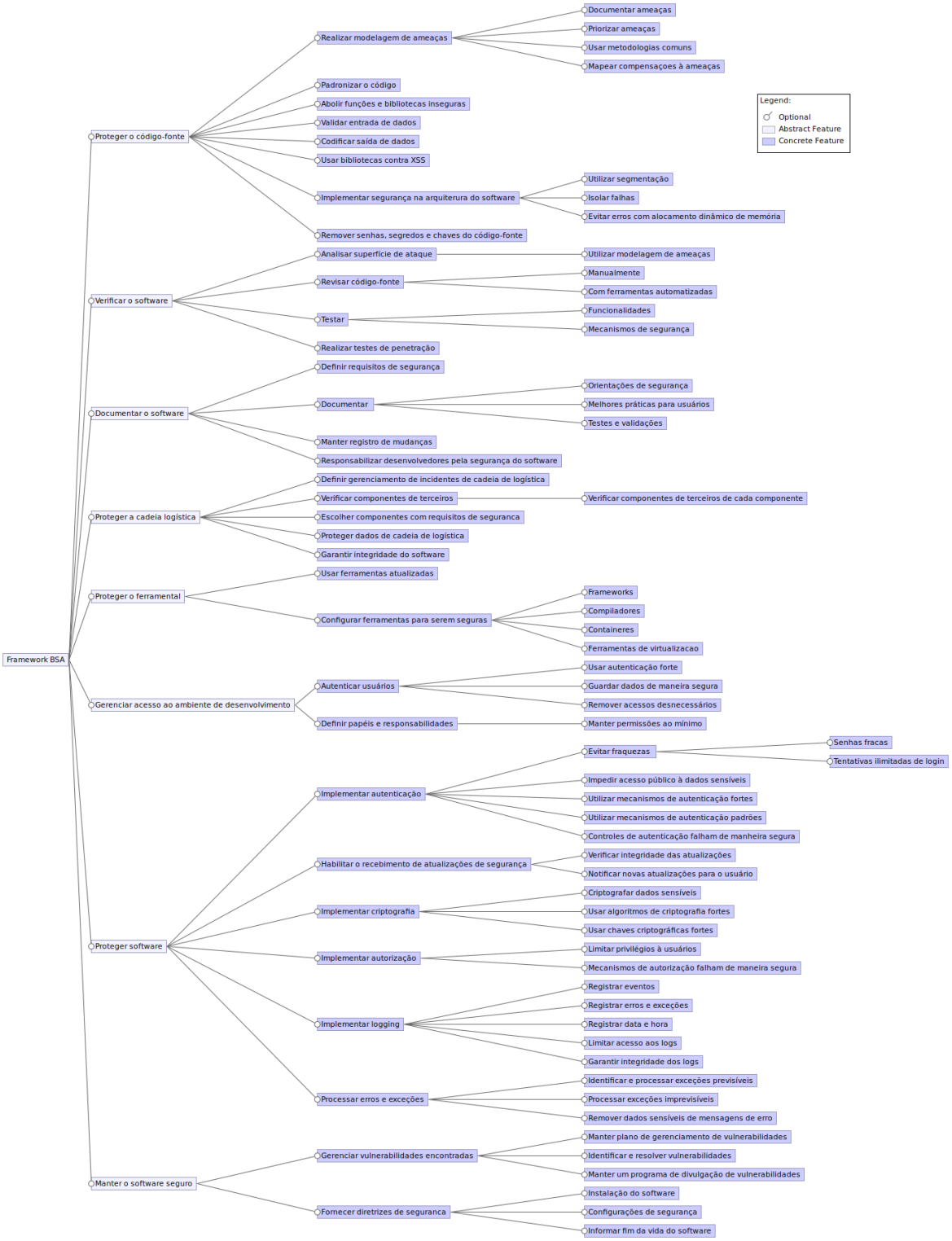
A estrutura do *feature model* do *framework* NIST apresenta as tarefas e exemplos de implementação de cada prática como *features* concretas (passíveis de implementação a nível de código-fonte), sendo estas *child features* das práticas, representadas por *features* abstratas. São *features* abstratas: (i) a *feature* principal “*Framework* NIST”, que tem todas as outras *features* como filhas (*child*); (ii) os grupos de práticas, como a *feature* “Preparar a organização”; e (iii) as práticas do *framework*, como a *feature* “Definir requisitos de segurança”.

Os *feature models* são apresentados nas Figuras 17 e 18.

4.2 Critérios para seleção das práticas

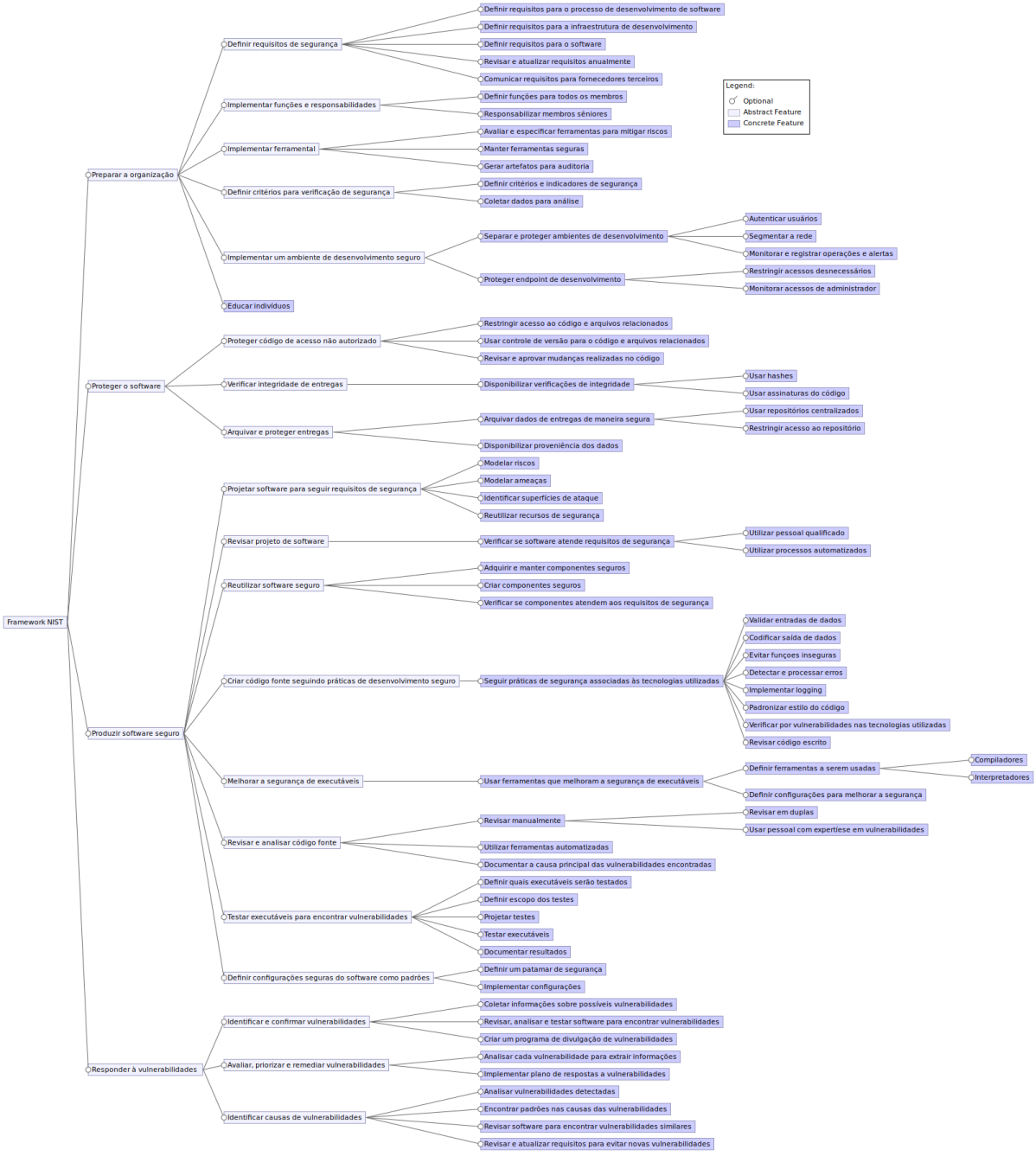
Os *feature models* representam o conjunto total de práticas apresentadas em cada *framework* de maneira sintetizada. Porém, não foram utilizadas todas as práticas no desenvolvimento dos módulos desenvolvidos neste trabalho. O *framework* BSA destaca que algumas considerações de segurança são mais importantes para um tipo de software do que para outros (Business Software Alliance, 2019). Por sua vez, o *framework* NIST comenta que algumas práticas não são aplicáveis em alguns casos e que diferentes organizações não possuem os

Figura 17 – *Feature model* desenvolvido para as práticas do *framework* BSA *Framework for Secure Software*



Fonte: Autoria própria (2023).

Figura 18 – *Feature model* desenvolvido para as práticas do *framework* NIST SSDF



Fonte: Autoria própria (2023).

mesmos objetivos e necessidades relacionados a segurança (SOUPPAYA; SCARFONE; DODSON, 2022).

Os *frameworks* enfatizam a necessidade de se conhecer o perfil de risco de cada organização e software: segundo Business Software Alliance (2019), "*com o entendimento de tolerância de risco, organizações podem priorizar atividades de segurança no seu processo de desenvolvimento de software*"; e segundo Souppaya, Scarfone e Dodson (2022), "*organizações devem adotar uma abordagem baseada em risco para determinar quais práticas são relevantes, apropriadas e efetivas para mitigar ameaças ao seu software*".

O *framework* NIST apresenta que todas as suas práticas são passíveis de implementação pelas organizações. As necessidades de segurança de cada organização devem definir o grau em que essas práticas são implementadas (SOUPPAYA; SCARFONE; DODSON, 2022). Risco, custo, viabilidade e aplicabilidade devem ser levados em consideração ao decidir quais práticas devem ser aplicadas no processo de desenvolvimento do *software* (SOUPPAYA; SCARFONE; DODSON, 2022).

A identificação, análise e avaliação de risco são tarefas do processo de avaliação de risco. O resultado deste processo é um compilado priorizado de riscos que podem afetar uma organização ou seu ambiente. Nesse processo, riscos são identificados e descritos, contendo sua fonte, causa e potenciais consequências. A avaliação de risco define o critério que deve ser seguido para determinar quais são os riscos associados (LESZCZYNA, 2021).

Para apresentar a utilização prática dos *feature models* criados, este trabalho contém o desenvolvimento de módulos de um sistema bancário. Não foi realizada uma avaliação de risco formal para estes módulos. Portanto, as vulnerabilidades mais comuns em sistemas *web* foram utilizadas como principais riscos ao *software*. O relatório OWASP *Top Ten - 2021* (Open Worldwide Application Security Project, 2021) foi utilizado como parâmetro das vulnerabilidades mais comuns. Um exemplo de vulnerabilidade apresentada por este relatório são as falhas de identificação e autenticação, que podem ser evitadas com as práticas "Evitar senhas fracas" e "Utilizar mecanismos de autenticação fortes" do *framework* BSA.

Desta forma, as configurações de segurança compreendem práticas de segurança relacionadas à fase de desenvolvimento do código fonte, desenvolvidos com o objetivo de evitar vulnerabilidades comuns e gerar código fonte seguro.

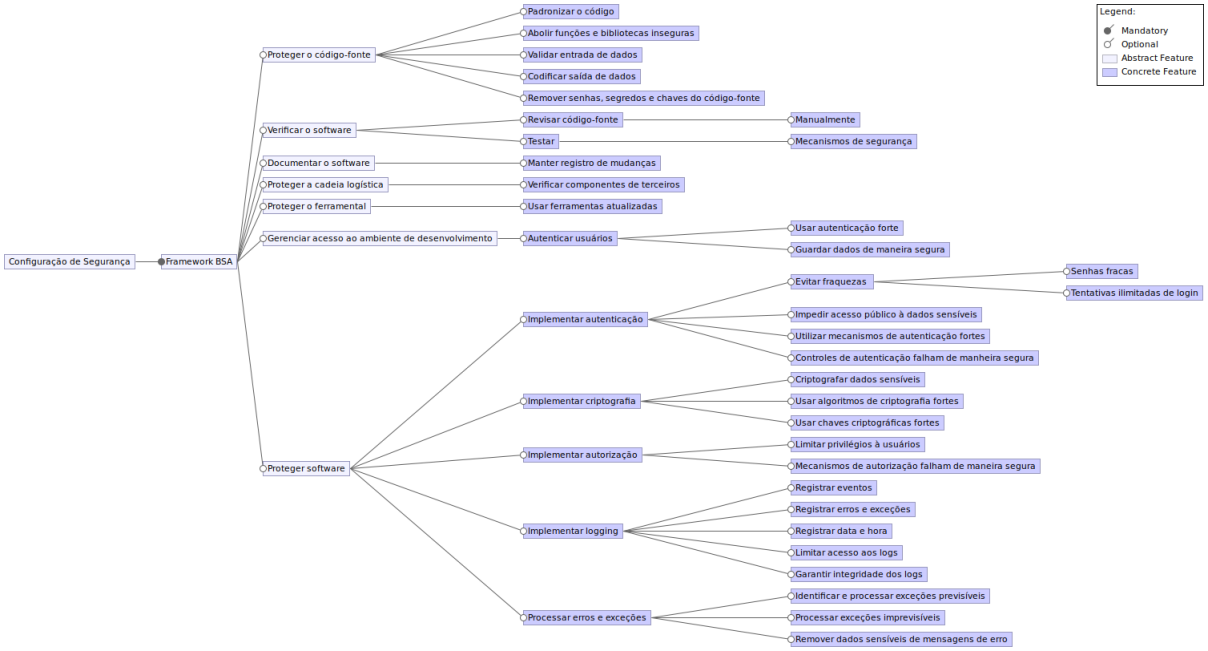
Além disso, práticas de fases anteriores e posteriores ao desenvolvimento foram ignoradas. As práticas das categorias "Manter o *software* seguro"(do *framework* BSA) e "Responder à vulnerabilidades"(do *framework* NIST) foram ignoradas por definirem práticas aplicáveis após a entrega do *software*. Adicionalmente, no *framework* NIST, foi ignorada a categoria "Preparar a organização", que define práticas organizacionais.

4.3 Configurações de segurança

Com base nos *feature models* criados, foram derivadas duas configurações de segurança com o apoio da ferramenta *FeatureIDE*. As configurações foram utilizadas para apoiar o desenvolvimento de dois módulos de um sistema bancário resistentes a vulnerabilidades comuns.

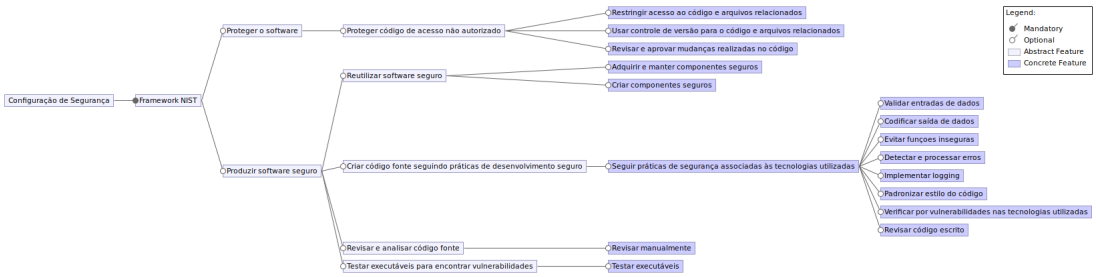
As configurações de segurança são apresentados nas Figuras 19 e 20.

Figura 19 – Configurações de segurança do *feature model* do *framework* BSA *Framework for Secure Software*



Fonte: Autoria própria (2023).

Figura 20 – Configurações de segurança do *feature model* do *framework* NIST SSDF



Fonte: Autoria própria (2023).

5 CASO DE ESTUDO: MÓDULOS DE UM SISTEMA BANCÁRIO

As configurações de segurança apresentadas no capítulo anterior foram utilizadas como base para definir as práticas a serem aplicadas nos módulos desenvolvidos. A configuração de segurança do *framework* BSA alcançou um número maior de práticas. Assim, os módulos desenvolvidos utilizaram as práticas do *framework* BSA.

Dois módulos de um sistema bancário foram desenvolvidos: (i) Autenticação, contendo funções de *login*, troca de senha e configuração de autenticação de múltiplos fatores, por meio de uma senha de uso único baseada em tempo (em inglês *Time-based One-Time Password*, ou apenas TOTP), gerada a partir de um código estático e o tempo atual; e (ii) Transferência de dinheiro, contendo funções para transferir saldo de uma conta para a outra e alteração do PIN utilizado para autorizar a transferência.

Os módulos foram desenvolvidos no formato de uma interface de programação de aplicações (do inglês *Application Programming Interface*, ou API) com comunicações via mensagens estruturadas no formato JSON (*JavaScript Object Notation*). A API é consumida por meio de um cliente (CLI) desenvolvido em uma interface gráfica instrucional de linha de comando.

A integração entre a API e a ferramenta CLI é dada da seguinte forma:

1. O cliente recebe a entrada de dados do usuário por meio de uma opção via teclado;
2. O cliente formata os dados recebidos do usuário em mensagens JSON e envia essas mensagens para a API;
3. A API realiza a validação dos dados recebidos pelo cliente e retorna mensagens em formato JSON informando o sucesso ou falha de uma operação;
4. Essas mensagens são mostradas para o usuário pela ferramenta CLI.

A API foi desenvolvida utilizando linguagem de programação *JavaScript* [1] no contexto do motor e ambiente de execução (*runtime*) JavaScript, chamado *Node.js* [2], bem como reutilizando recursos do framework *express* [3]. O cliente (CLI) instrucional que recebe comandos via linha de comando (terminal) foi desenvolvido utilizando linguagem de programação *Python* [4]. As mensagens enviadas via API seguem as práticas dos *frameworks*, e o CLI é responsável por mostrar via linha de comando as operações realizada via API.

Seguindo o critério de seleção das práticas mencionado anteriormente, a configuração de segurança do *framework* BSA foi escolhida com as principais práticas relacionadas ao processo de desenvolvimento do código-fonte. Essas práticas foram implementadas nos módulos de autenticação e transferência de dinheiro. Segue uma breve explicação de como as práticas

[1] *JavaScript*: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript
[2] *Node.js*: https://nodejs.org/
[3] *express*: https://www.npmjs.com/package/express
[4] *Python*: https://www.python.org/

(*features* concretas) foram implementadas, organizadas por suas categorias (*features* abstratas):

- **Proteger o código fonte**: Esta categoria aborda cinco práticas: (i) “Padronizar o código”, implementada utilizando um estilo padrão no código definido pelo formatador *Prettier* [5], instalado como extensão do editor de textos *Visual Studio Code (VS Code)* [6] e mantém uma estrutura e mensagens de retorno padrão; (ii) “Abolir funções e bibliotecas inseguras”, evita a utilização de funções perigosas, como a função *eval*, que executa uma *string* como comandos no servidor, tornando possível um ataque de injeção de código [7]; (iii) “Validar entrada de dados”, realiza a verificação de todos os dados fornecidos pelo usuário (tipo de dado e formato de dado); (iv) “Codificar saída de dados”, utiliza um padrão de mensagens de retorno para o usuário; e (v) “Remover senhas, segredos e chaves do código-fonte”, utiliza variáveis de ambiente para armazenar estes dados;

- **Verificar o *software***: As práticas selecionadas desta categoria foram: (i) “Revisar código-fonte manualmente”, todo o código foi revisado para que atendesse o padrão e erros fossem detectados; e (ii) “Testar mecanismos de segurança” por meio de requisições na API para garantir que os controles de segurança implementados estivessem corretos;

- **Documentar o *software***: Foi escolhida a prática “Manter registro de mudanças”, implementada com a utilização do gerenciador de versões *git* [8] e repositório *GitHub* [9];

- **Proteger a cadeia logística**: A prática “Verificar componentes de terceiro” foi selecionada para proteger a cadeia de logística. Os componentes e bibliotecas utilizados foram avaliados de acordo com sua popularidade e desenvolvimento ativo, selecionando aqueles que possuem comunidade ativa e grande adoção. Essa análise foi realizada pelo gerenciador de pacotes *npm* [10];

- **Proteger o ferramental**: Apenas a prática “Usar ferramentas atualizadas” foi selecionada desta categoria. A prática implementada garante que todas as ferramentas utilizadas (editor de textos, versões de bibliotecas, dentre outros) estejam em suas versões mais recentes;

- **Gerenciar acesso ao ambiente de desenvolvimento**: Para gerenciar o acesso ao ambiente de desenvolvimento, foi configurada autenticação de múltiplos fatores no

[5] *Prettier*: https://prettier.io/
[6] *Visual Studio Code*: https://code.visualstudio.com/
[7] Função *eval*: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/eval
[8] *git*: https://git-scm.com/
[9] *GitHub*: https://github.com/
[10] *npm*: https://www.npmjs.com/

GitHub, com os dados para autenticação salvos no gerenciador de senhas *Bitwarden* [11], o repositório com o código-fonte foi configurado com visibilidade privada, restringindo o acesso ao código-fonte.

As práticas da categoria **Proteger o *software*** do *framework* BSA foram selecionadas para adicionar funcionalidades de segurança aos módulos desenvolvidos:

- **Implementar autenticação**: O módulo de autenticação foi desenvolvido seguindo as seguintes diretrizes: (i) senhas de usuários devem possuir ao menos oito caracteres, uma letra maiúscula, uma letra minúscula, um número e um caractere especial, evitando senhas fracas no sistema; (ii) o módulo possibilita a configuração de um segundo fator de autenticação, sendo este um *token* de uso único baseado em tempo (TOTP), implementado com a biblioteca *OTPAuth* [12]; (iii) o módulo *express-rate-limit* [13] foi utilizada para evitar tentativas ilimitadas de login, configurado para permitir 10 tentativas de login a cada 5 minutos; (iv) as chaves criptográficas relacionadas ao módulo de autenticação foram armazenadas como variáveis de ambiente para evitar acesso público; e (v): sempre que o controle de autenticação falha com um erro interno, o usuário não recebe permissão de acesso ao sistema, mesmo se este usuário for legítimo;

- **Implementar criptografia**: Para proteger os dados dos usuários em caso de vazamentos, suas senhas (senha de acesso, PIN para transferência e chave criptográfica da senha de uso único) foram criptografados no banco de dados. Foi utilizado o algoritmo de *hashing* da biblioteca *bcrypt* [14] para a senha de acesso e PIN e o algoritmo criptográfico AES da biblioteca *crypto-js* [15] para criptografar a chave criptográfica da senha de uso único. Foram geradas chaves criptográficas fortes armazenadas como variáveis de ambiente.

- **Implementar autorização**: A autorização das operações de transferência de dinheiro, configuração da senha de uso único, troca de senha e troca de PIN foram realizadas por meio de um *token* de autorização enviado nos *header* das requisições. Se o *token* não for informado ou for inválido, a ação é bloqueada. Caso ocorra um erro durante à validação de autorização, o usuário não é autorizado a utilizar a função;

- **Implementar *logging***: Todos os eventos nos módulos são registrados por meio de *logs* gerados pela biblioteca *winston* [16]. Os *logs* salvam qual ação foi realizada e o horário do evento. Erros também são registrados. Para garantir a integridade dos *logs*, o nome do arquivo salvo contém o seu *hash*. Estes arquivos podem, então, ser armazenados

[11] Gerenciador de senhas *Bitwarden*: https://bitwarden.com/
[12] *OTPAuth*: https://www.npmjs.com/package/otpauth
[13] *express-rate-limit*: https://www.npmjs.com/package/express-rate-limit
[14] *bcrypt*: https://www.npmjs.com/package/bcrypt
[15] *crypto-js*: https://www.npmjs.com/package/crypto-js
[16] *winston*: https://www.npmjs.com/package/winston

sob sigilo em locais com restrição de acesso, seguindo os requerimentos do Marco Civil da Internet (Lei 12.965/2014)[17] e a Lei Geral de Proteção de Dados Pessoais (Lei 13.709/2018)[18];

- **Processar erros e exceções**: Todos os erros e exceções são processados pelos módulos utilizando blocos de código *try* e *catch*. Erros comuns são mapeados e mensagens de erro padrões são definidas. Por exemplo, um erro de senha inválida sempre retorna a mensagem “Login ou senha incorretos”. Erros incomuns retornam a mensagem “Erro interno do servidor”. Nenhuma mensagem de erro retorna dados sobre a infraestrutura da aplicação ou dados sensíveis.

A API com os módulos é constituída por oito rotas. O módulo de autenticação compõe as rotas de login, alterar senha, habilitar, verificar e desabilitar a senha de uso único. O módulo de transferência de dinheiro possui as rotas realizar transferência, verificar saldo e alterar PIN. Essas rotas são apresentadas abaixo:

- **Login**: Rota responsável pela autenticação. Recebe um login e uma senha que são validados. Se o login e a senha forem válidos e corretos, a API retorna um *token* de autorização. Caso a conta tenha a senha de uso único configurada, o *token* de autorização irá permitir acesso apenas à rota de verificar a senha de uso único;

- **Alterar senha**: Utilizada para trocar a senha de acesso de um usuário autenticado. Recebe o *token* de autorização, a senha atual, a nova senha e a confirmação desta senha. Se todos os dados forem válidos, a senha de acesso é alterada;

- **Habilitar senha de uso único**: Utilizada para habilitar a senha de uso único como segundo fator de autenticação. Recebe o *token* de autorização, retorna um código para configurar a senha de uso único em um aplicativo compatível, como o *Google Authenticator* e salva esse código criptografado no banco de dados;

- **Validar senha de uso único**: Rota utilizada para validar a senha de uso único fornecida pelo usuário na hora da autenticação. Recebe o *token* de autorização e a senha de uso único. Se todos os dados estiverem corretos, retorna um *token* de autorização que fornece acesso a todas as rotas;

- **Desabilitar senha de uso único**: Utilizada para desativar a senha de uso único. Recebe o *token* de autorização e remove o código utilizado para geração da senha de uso único do banco de dados;

[17] Marco Civil da Internet: https://www.planalto.gov.br/ccivil_03/_ato2011-2014/2014/lei/l12965.htm.

[18] Lei Geral de Proteção de Dados Pessoais: https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm.

- **Realizar transferência**: Rota responsável por efetuar uma transferência de saldo entre contas. Recebe *token* de autorização, a quantidade a ser transferida, o e-mail do destinatário e o PIN para transferência. Se todos os dados estiverem corretos e o usuário possuir saldo suficiente, a transferência é realizada e um comprovante é retornado para o usuário;
- **Verificar saldo**: Responsável por retornar o nome completo e o saldo atual do usuário autenticado. Recebe o *token* de autorização e retorna os dados se este *token* for válido;
- **Alterar PIN**: Responsável por trocar o PIN de transferência do usuário. Funciona de maneira análoga à rota de alterar senha.

O CLI implementado consome as oito rotas mencionadas via API e, sua interface gráfica principal, pode ser visualizada na Figura 21 (sem um usuário autenticado) e na Figura 22 (com o usuário autenticado).

Figura 21 – Tela principal sem usuário logado

Fonte: Autoria própria (2023).

Erros são exibidos para o usuário em caso de falhas. Estes erros podem ser visualizados na Figura 23 (erro na autenticação) e na Figura 24 (erro na transferência de dinheiro). No caso de uma transferência com sucesso, o CLI demonstra o comprovante (Figura 25).

Para ativar o uso de senhas de uso único, se deve utilizar o código fornecido pela API para configurar um aplicativo compatível, como o *Google Authenticator*. A Figura 26 apresenta a ativação desta funcionalidade. Após ativada, será necessário informar, além do e-mail e senha, a senha de utilização única em todo processo de autenticação (Figura 27).

Além da interface CLI, foram escritos dois *scripts* para testar as validações de entrada de dados, funcionalidades de bloqueio de tentativas ilimitadas de *login* e mecanismo de autenticação de dois fatores.

O primeiro *script* utiliza o módulo de autenticação e envia várias requisições com dados inválidos à rota de *login*, mostrando mensagens de entradas inválidas e a mensagem de blo-

Figura 22 – Tela principal com usuário logado

Fonte: Autoria própria (2023).

Figura 23 – Exemplo de erro de autenticação

Fonte: Autoria própria (2023).

Figura 24 – Exemplo de erro de transferência de dinheiro

Fonte: Autoria própria (2023).

Figura 25 – Exemplo de comprovante de transferência

Fonte: Autoria própria (2023).

Figura 26 – Exemplo de habilitação da senha de uso único

Fonte: Autoria própria (2023).

Figura 27 – Exemplo de verificação de senha de uso único

Fonte: Autoria própria (2023).

queio de requisições. O segundo *script* testa o controle da autenticação de dois fatores, fazendo *login* e não validando a senha de uso único, mostrando mensagens de erro ao tentar usar rotas protegidas.

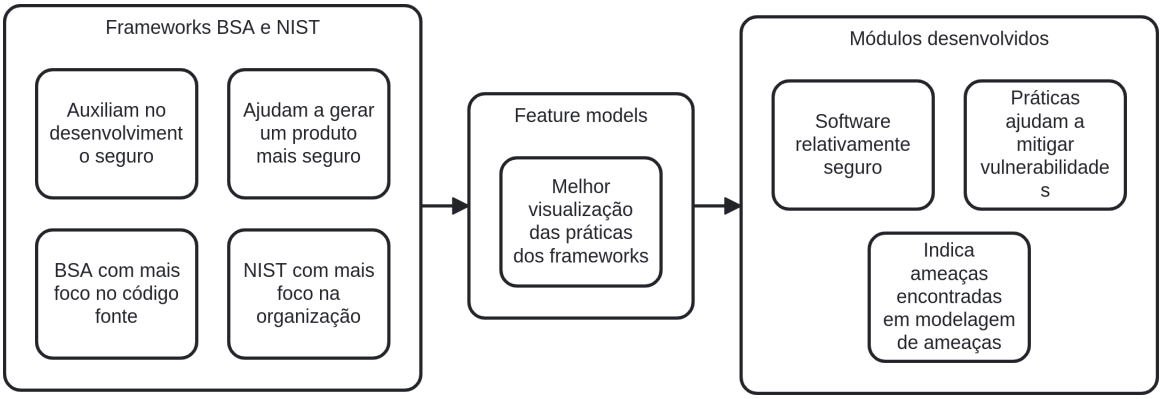
O código-fonte desta aplicação pode ser encontrado em um repositório público no GitHub [19].

[19] Repositório do código-fonte: https://github.com/ribeiro-julio/tcc-banking-application/.

6 DISCUSSÕES

As principais discussões deste trabalho são descritas a seguir e ilustradas na Figura 28. Os pontos principais de cada tópico avaliado são apresentados na Figura 28 (*frameworks* de segurança estudados, bem como os *feature models* criados e módulos desenvolvidos).

Figura 28 – Principais discussões



Fonte: Autoria própria (2023).

6.1 *Frameworks* utilizados: BSA e NIST

A utilização dos *frameworks* BSA e NIST para o desenvolvimento seguro de *software* se mostrou positiva e útil. Os *frameworks* auxiliam no desenvolvimento de software com processos e práticas implementáveis, que contribuem no aumento da segurança do produto final. Essas práticas não só incluem a etapa de desenvolvimento, mas um planejamento sistemático ou não-oportunístico prévio de diferentes tipos de organizações, bem como o gerenciamento de vulnerabilidades, *posteriori* à entrega do *software*.

Os *frameworks* BSA e NIST sintetizam o processo de gerenciamento de vulnerabilidades em diferentes práticas que englobam as funções de identificar, proteger, detectar, responder e recuperar, apresentadas pelo *framework* NIST *Cybersecurity Framework* (National Institute of Standards and Technology, 2023) e podem ser consideradas após a entrega do sistema, no esforço de manter o *software* seguro.

Comparando cada *framework*, é possível observar uma maior ênfase nas práticas relacionadas ao código-fonte no *framework* BSA. Tal fato é perceptível pelas práticas da categoria de “Capacidades de segurança”, nas quais são abordadas *features* de segurança do *software* detalhadamente (como autenticação, autorização, criptografia, dentre outras). Isso pode ser muito útil para desenvolvedores com menos experiência, que necessitam integrar práticas de segurança no código-fonte, se tornando um “material de apoio” na implementação de *features* de segurança ao código-fonte.

Do outro lado, o *framework* NIST apresenta uma abordagem mais orientada para a organização, com uma ênfase em como implementar práticas de segurança na estrutura organizacional, removendo o foco do código-fonte. Isso pode ser positivo para organizações que buscam integrar práticas de segurança em uma escala mais ampla, abordando não apenas aspectos técnicos, mas organizacionais acerca da segurança do software. Por exemplo, conscientizar desenvolvedores sobre problemas recorrentes de segurança em sistemas de informação.

Observando as configurações de segurança de cada *framework*, se percebe uma quantidade maior de práticas de segurança por parte do *framework* BSA. Essa situação é esperada devido ao critério de seleção das práticas, que se concentra em direcionar a segurança durante a etapa de desenvolvimento do código-fonte. A maior ênfase no produto de *software* é fornecida pelo *framework* BSA, que ocasionou em mais práticas na configuração de segurança quando comparado ao *framework* NIST. Todas as práticas contidas na configuração de segurança do *framework* NIST estão contidas no BSA. Por estes motivos, se escolheu desenvolver os módulos utilizando apenas a configuração de segurança do *framework* BSA.

As práticas do *framework* BSA ajudam a mitigar vulnerabilidades comuns, como as apresentadas pelo relatórios OWASP *Top-Ten* (Open Worldwide Application Security Project, 2021). Se pode citar como exemplo a prática “Implementar autenticação”, que ajuda a mitigar falhas de quebra de controle de acesso, que segundo este relatório, é a vulnerabilidade mais comum em sistemas *web*. Por outro lado, o *framework* não pode ser utilizado como única fonte de pesquisa, e as referências contidas em cada práticas devem ser consultadas para orientações mais específicas na implementação de cada prática.

6.2 *Feature models* desenvolvidos

Acredita-se que haja facilidade de visualizar as práticas ao utilizar *feature models*. A visualização permite observar de forma estruturada as práticas de segurança destacadas por cada *framework*. *Feature models* servem como uma ferramenta para a comunicação e discussão de práticas de segurança entre as diversas partes interessadas no projeto (*stakeholders*).

O *feature model* facilita a visualização das práticas e pode ser utilizado para apoiar a fase de projeto (*design*) da segurança no desenvolvimento do *software*. As informações neste modelo estão sintetizadas, é recomendado consultar os *frameworks* originais para que se entenda qual o objetivo e consequência da implementação de cada prática. Além disso, considerar as referências de cada *framework*, como material de consulta para entender como cada praticas devem ser implementadas. Os *feature model* criados apresentam “o que” pode ser implementado, mas não “como”.

No contexto deste trabalho, os *feature models* serviram como um recurso visual de apoio. Ao implementar as práticas escolhidas foi necessário consultar os *frameworks* e suas referências, para que se extraísse mais informações e detalhes sobre cada prática e como realizar a sua implementação.

6.3 Módulos implementados

A implementação dos módulos, guiada pelas práticas do *framework* BSA, tem a intenção de ser simples em relação a adoção de um *framework* de desenvolvimento seguro. A API e o cliente implementados, ao seguir as práticas apresentadas pelo *framework*, resultaram em uma solução com aspectos de segurança úteis, evidenciando que a aplicação de práticas de segurança ajudam a mitigar uma série de ameaças potenciais e vulnerabilidades no software.

A integração das práticas de configuração de segurança do *framework* BSA no desenvolvimento dos módulos ajuda a fortalecer a segurança dos módulos desenvolvidos. O *framework* BSA guiou o desenvolvimento, destacando diversas configurações e práticas que poderiam não ter sido implementadas sem a realização de uma modelagem de ameaças e análise de riscos. Isso indica o valor em adotar um *framework* de desenvolvimento seguro: ajudar na identificação e mitigação de potenciais vulnerabilidades durante o design do software, antes mesmo de se realizar uma modelagem de ameaças formal.

Por outro lado, não foi possível implementar práticas importantes, como revisar o código-fonte com ferramentas automatizadas e realizar testes de penetração, apresentadas pelos *frameworks*. Essas práticas necessitam de ferramentas externas e pessoas com expertise na área, e ajudam na detecção de falhas, geradas por uma possível má implementação de um controle de segurança que podem não ser encontradas em processos de revisão manual e testes comuns.

7 CONCLUSÃO

Este trabalho apresentou a utilização de *frameworks* de desenvolvimento seguro no processo de desenvolvimento dos módulos de autenticação e transferência de dinheiro de uma aplicação *web* bancária. *Feature models* apoiaram a visualização e escolha das práticas aplicadas no desenvolvimento, descritas pelas configurações de segurança utilizadas. A partir dessa configuração, foi desenvolvida uma API (que implementa as funcionalidades da aplicação) e um cliente CLI (que consome as funcionalidades da API). Estes se mostraram soluções com certo grau de segurança, mostrando que a aplicação de práticas de segurança ajudam a mitigar ameaças potenciais e vulnerabilidades no *software*.

Este trabalho contribui com:

- Uma síntese de diversas práticas de segurança que podem ser adicionadas durante todo o processo de desenvolvimento de *software*;
- Representações visuais/gráficas de cada *framework* por meio de *feature models*;
- A implementação das práticas relacionadas ao código-fonte de maneira prática, comentando tecnologias e bibliotecas utilizadas;
- Código-fonte de uma aplicação *web* segura que pode ser consultada e utilizada.

Os *feature models* facilitam a comunicação e escolha das práticas de segurança e servem como material de consulta às *features* escolhidas. Além disso, juntamente com os documentos dos *frameworks*, se mostraram grandes materiais de apoio para a implementação de cada prática por fornecerem práticas que contribuem com a mitigação de vulnerabilidades comuns e informam possíveis pontos de falha mesmo sem a realização de uma análise de riscos formal (descrita em detalhes).

Apesar disso, essas ferramentas não podem ser utilizadas como únicas fontes de informações na atividade de desenvolvimento de *software*, pois não fornecem afirmações sobre como implementar as práticas, apenas exemplos de implementação e resultados esperados. É necessário consultar outras fontes (começando com as referências indicadas em cada *framework*) para extrair mais informações.

No escopo deste trabalho, *feature models* oferecem um potencial para futuras automações. No contexto de desenvolvimento seguro de software é possível seguir com iniciativas como:

- Explorar a criação de uma nota de segurança relacionado às práticas selecionadas, proporcionando uma métrica que pode ser utilizada para avaliar e comparar a segurança de diferentes configurações ou versões do software;

- Indicar práticas de segurança recomendadas com base em um questionário ou na análise de requisitos não-funcionais de segurança específicos, facilitando a personalização e adaptação das práticas de segurança em *software* em desenvolvimento;
- Criação de um projeto que englobe as capacidades de segurança descritas pelo *framework* BSA, onde cada *feature* descrita por essa categoria seria implementada no formato de microsserviços de *software* reutilizáveis. Isso serve como uma base sobre a qual novas práticas de segurança podem ser criadas para atender à mudanças de requisitos ou desafios de segurança.

Finalmente, esta pesquisa fornece uma visão sobre as práticas de segurança dos *frameworks* BSA e NIST, e também uma base sobre a qual futuras pesquisas na área de desenvolvimento seguro de software podem ser conduzidas. Por meio da integração de práticas de segurança no processo de desenvolvimento de *software*, é possível não apenas mitigar vulnerabilidades e ameaças potenciais, mas desenvolver *softwares* intrinsecamente mais seguros e robustos.

REFERÊNCIAS

AJAYI, W. *et al.* ANALYSIS OF MODERN CYBERSECURITY THREAT TECHNIQUES AND AVAILABLE MITIGATING METHODS. **International Journal of Advanced Research in Computer Science**, 2022.

APEL, S. *et al.* **Feature-Oriented Software Product Lines**. [*S.l.*]: Springer Berlin, Heidelberg, 2013.

ASLAN, O. *et al.* A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. **Electronics**, 2023.

BENAVIDES, D.; SEGURA, S.; RUIZ-CORTES, A. Automated analysis of feature models 20 years later: A literature review. **Information Systems**, 2010.

BERGER, T. *et al.* What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. *In*: **Proceedings of the 19th International Conference on Software Product Line**. [*S.l.*: *s.n.*], 2015.

BERGER, T. *et al.* The state of adoption and the challenges of systematic variability management in industry. **Empirical Software Engineering**, 2020.

BOSCH, J.; CAPILLA, R.; HILLIARD, R. Trends in Systems and Software Variability [Guest editors' introduction]. **IEEE Software**, 2015.

BOSCH, J.; LEE, J. **Software Product Lines: Going Beyond**: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings. [*S.l.*]: Springer Berlin, Heidelberg, 2010.

BRITO, L. J. **Analise Automatica do Modelo de Features em Linha de Produtos de Software**. 2013 — Universidade de Brasília, 2013.

Business Software Alliance. **The BSA Framework for Secure Software: A New Approach to Securing the Software Lifecycle**. [*S.l.*], 2019.

BUTTNER, A. *et al.* A Secure Code Review Retrospective. *In*: **2020 IEEE Secure Development (SecDev)**. [*S.l.*: *s.n.*], 2020.

Committee on National Security Systems. **Committee on National Security Systems (CNSS) Glossary**. [*S.l.*], 2022.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Staged Configuration Using Feature Models. *In*: **Software Product Lines**. [*S.l.*: *s.n.*], 2004.

DAWSON, M. *et al.* Integrating Software Assurance into the Software Development Life Cycle (SDLC). **Journal of Information Systems Technology and Planning**, 2010.

European Union Agency for Cybersecurity. **ENISA Threat Landscape Report 2017**: 15 Top Cyber-Threats and Trends. [*S.l.*], 2018.

Federação Brasileira de Bancos. **Pesquisa Febraban de Tecnologia Bancária**. 2023. Disponível em: https://cmsarquivos.febraban.org.br/Arquivos/documentos/PDF/Pesquisa%20Febraban%20de%20Tecnologia%20Banc%C3%A1ria%202023%20-%20Volume%202.pdf. Acesso em: 16 out. 2023.

FELDERER, M. *et al.* Chapter One - Security Testing: A Survey. *In*: **Advances in Computers**. [*S.l.*]: Elsevier, 2016.

FEROZ, A. K.; ZO, H.; CHIRAVURI, A. Digital Transformation and Environmental Sustainability: A Review and Research Agenda. **Sustainability**, 2021.

GERALDI, R. T. **PROCESSO PARA MODELAGEM DE VARIABILIDADES DE SISTEMAS CIBER-FÍSICOS APOIADO POR FEATURE MODELS**. 2022. Tese (Doutorado) — Pontifícia Universidade Católica do Paraná, 2022.

GERALDI, R. T.; REINEHR, S.; MALUCELLI, A. Software product line applied to the internet of things: A systematic literature review. **Information and Software Technology**, 2020.

GHELANI, D.; HUA, T. K.; KODURU, S. K. R. A Model-Driven Approach for Online Banking Application Using AngularJS Framework. **American Journal of Information Science and Technology**, 2022.

GUEMBE, B. *et al.* The Emerging Threat of Ai-driven Cyber Attacks: A Review. **Applied Artificial Intelligence**, 2022.

HASSAN, M. M. *et al.* Quantitative Assessment on Broken Access Control Vulnerability in Web Applications. *In*: **Proceedings of the International Conference on Cyber Security and Computer Science (ICONCS'18)**. [*S.l.*: *s.n.*], 2018.

HASSAN, M. M. *et al.* Broken Authentication and Session Management Vulnerability: A Case Study Of Web Application. **International Journal of Simulation: Systems, Science Technology**, 2018.

HILTGEN, A.; KRAMP, T.; WEIGOLD, T. Secure Internet banking authentication. **IEEE Security Privacy**, 2006.

Information Systems Audit and Control Association. **ISACA® Glossary of Terms English-Brazilian Portuguese**. [*S.l.*], 2015.

Institute of Electrical and Electronics Engineers. **IEEE 1044-2009**: IEEE Standard Classification for Software Anomalies. [*S.l.*], 2010.

International Organization for Standardization. **ISO/IEC 27001**: Information security management systems. [*S.l.*], 2005.

International Organization for Standardization. **ISO/IEC 27005**: Information security, cybersecurity and privacy protection — Guidance on managing information security risks. [*S.l.*], 2008.

International Organization for Standardization. **ISO/IEC 25010**: Software and data quality. [*S.l.*], 2011.

JABIYEV, B. *et al.* Preventing server-side request forgery attacks. *In*: **Proceedings of the 36th Annual ACM Symposium on Applied Computing**. [*S.l.*: *s.n.*], 2021.

KAMAL, A. H. A. *et al.* **Risk Assessment, Threat Modeling and Security Testing in SDLC**. 2020.

KAMBOU, S.; BOUABDALLAH, A. A Strong Authentication Method for Web/Mobile Services. *In*: **2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)**. [*S.l.*: *s.n.*], 2019.

KANG, K. C. *et al.* **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. 1990.

KANG, K. C. *et al.* FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. **Annals of Software Engineering**, 1998.

KENNER, A. *et al.* Using Variability Modeling to Support Security Evaluations: Virtualizing the Right Attack Scenarios. *In*: **Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems**. [*S.l.*: *s.n.*], 2020.

KRüGER, J. *et al.* Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. *In*: **Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems**. [*S.l.*: *s.n.*], 2018.

KRUGER, J. *et al.* Where is my feature and what is it about? A case study on recovering feature facets. **Journal of Systems and Software**, 2019.

LEE, K.; KANG, K. C.; LEE, J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. *In*: **Software Reuse: Methods, Techniques, and Tools**. [*S.l.*]: Springer Berlin Heidelberg, 2002.

LESZCZYNA, R. Review of cybersecurity assessment methods: Applicability perspective. **Computers Security**, 2021.

LI, L. *et al.* Static analysis of android apps: A systematic literature review. **Information and Software Technology**, 2017.

LIN, T. C. W. Financial Weapons of War. **Minnesota Law Review**, 2016.

LINDEN, F.; SCHMID, K.; ROMMES, E. **Software Product Lines in Action**: The Best Industrial Practice in Product Line Engineering. [*S.l.*]: Springer Berlin, Heidelberg, 2007.

LU, D.; FEI, J.; LIU, L. A Semantic Learning-Based SQL Injection Attack Detection Technology. **Electronics**, 2023.

MARTINEZ, S.; COSENTINO, V.; CABOT, J. Model-based analysis of Java EE web security misconfigurations. **Computer Languages, Systems Structures**, 2017.

MEERTS, C. Security: Concepts and Definitions. *In*: ____. **Encyclopedia of Security and Emergency Management**. [*S.l.*]: Springer International Publishing, 2018. p. 1–3.

MEINICKE, J. *et al.* **Mastering Software Variability with FeatureIDE**. [*S.l.*]: Springer Cham, 2017.

National Institute of Standards and Technology. **The NIST Cybersecurity Framework 2.0**. [*S.l.*], 2023.

OLIINYK, O. *et al.* Structuring automotive product lines and feature models: an exploratory study at Opel. **Requirements Engineering**, 2015.

Open Worldwide Application Security Project. **OWASP Top 10 - 2017**: The Ten Most Critical Web Application Security Risks. 2017.

Open Worldwide Application Security Project. **OWASP Top 10:2021**. 2021.

POHL, K.; BöCKLE, G.; LINDEN, F. **Software Product Line Engineering**: Foundations, Principles and Techniques. [*S.l.*]: Springer Berlin, Heidelberg, 2005.

POHL, K.; RUPP, C. **Requirements Engineering Fundamentals**: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB compliant. [*S.l.*]: Rocky Nook, 2015.

PortSwigger. **Business logic vulnerabilities**. 2023. Disponível em: https://portswigger.net/web-security/logic-flaws. Acesso em: 04 jun. 2023.

REINEHR, S. **Engenharia de Requisitos**. [*S.l.*]: Grupo A, 2020.

SANTIAGO, M. R.; ZANETONI, J. P. L.; VITA, J. B. INCLUSÃO FINANCEIRA, INOVAÇÃO E PROMOÇÃO AO DESENVOLVIMENTO SOCIAL E ECONÔMICO ATRAVÉS DO PIX. **Revista Jurídica**, 2020.

SEMOLA, M. **Gestão da Segurança da Informação**: Uma Visão Executiva. [*S.l.*]: GEN LTC, 2013.

Snyk. **Secure Software Development Lifecycle (SSDLC)**. 2023. Disponível em: https://snyk.io/learn/secure-sdlc/. Acesso em: 04 jun. 2023.

SOUPPAYA, M.; SCARFONE, K.; DODSON, D. **Secure Software Development Framework (SSDF) Version 1.1**: Recommendations for Mitigating the Risk of Software Vulnerabilities. [*S.l.*], 2022.

STONEBURNER, G.; GOGUEN, A.; FERINGA, A. **Risk Management Guide for Information Technology Systems**. [*S.l.*], 2002.

THUM, T. *et al.* FeatureIDE: An extensible framework for feature-oriented software development. **Science of Computer Programming**, 2014.

VARELA-VACA, A. J. *et al.* AMADEUS: Towards the AutoMAteD SecUrity TeSting. *In*: **Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A**. [*S.l.*: *s.n.*], 2020.

VARELA-VACA, Á. J. *et al.* CyberSPL: A Framework for the Verification of Cybersecurity Policy Compliance of System Configurations Using Software Product Lines. **Applied Sciences**, 2019.

VARELA-VACA, Á. J. *et al.* CARMEN: A framework for the verification and diagnosis of the specification of security requirements in cyber-physical systems. **Computers in Industry**, 2021.

VARELA-VACA, J. *et al.* Feature models to boost the vulnerability management process. **Journal of Systems and Software**, 2023.

WOHLIN, C. *et al.* **Experimentation in Software Engineering**. [*S.l.*]: Springer, 2012.

XIONG, W.; LAGERSTROM, R. Threat modeling – A systematic literature review. **Computers & Security**, 2019.