Beatriz Michelson Reichert

Modelagem de Ameaças em Pipelines de Desenvolvimento de Software

Joinville

2021

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Beatriz Michelson Reichert

MODELAGEM DE AMEAÇAS EM *PIPELINES* DE DESENVOLVIMENTO DE SOFTWARE

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Dr. Rafael Rodrigues Obelheiro
Orientador

Joinville, Agosto de 2021

MODELAGEM DE AMEAÇAS EM *PIPELINES* DE DESENVOLVIMENTO DE SOFTWARE

Beatriz Michelson Reichert

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora:

Dr. Rafael Rodrigues Obelheiro - UDESC
(orientador)

Dr. Charles Christian Miers - UDESC

Dr. Ricardo José Pfitscher - UDESC

Agradecimentos

À minha família - meus pais e minha avó - pela força, paciência e motivação, sem estes eu não chegaria até aqui. Ao meu namorado pela paciência, por compreender a minha ausência e a necessidade de usar os finais de semana como horário de trabalho. Aos meus amigos pela motivação, incentivo, e por não me deixarem desanimar durante toda essa caminhada. Aos meus companheiros de quatro patas, por me proporcionarem momentos de distração e diversão. Ao meu orientador, o professor Rafael R. Obelheiro, pela orientação, apoio, confiança, paciência e pelo empenho dedicado à elaboração deste trabalho. A todos os professores pelos conhecimentos repassados, estes essenciais para o desenvolvimento deste trabalho. Ao Adam Williamson, membro da equipe de QA do Fedora, pelo esclarecimento de algumas dúvidas e por responder meus e-mails em poucas horas, inclusive no final de semana. A todos que de algum modo fizeram parte desta caminhada, o meu muito obrigada!

“Life is never complete without its challenges.”

Stan Lee

Resumo

Em anos recentes tem crescido a preocupação com a integridade de software, ou seja, a garantia de que o software não seja adulterado no caminho entre desenvolvedores e usuários. Esse caminho é representado por um *pipeline* de desenvolvimento de software. O *pipeline* adotado como referência neste trabalho consiste em cinco etapas (Integração, Integração contínua, Infraestrutura como código, Implantação e Lançamento), sendo que já foram reportadas na literatura vulnerabilidades ou ataques em todas essas etapas. Este trabalho desenvolve um modelo de ameaças para todo o *pipeline* de desenvolvimento de software usando a metodologia STRIDE, e identifica possíveis mitigações para cada ameaça. Esse modelo de ameaças é aplicado ao *pipeline* usado pela distribuição Fedora Linux para gerar imagens do sistema operacional e distribuí-las a usuários finais, com o propósito de descobrir quais ameaças são mitigadas e como. A modelagem do *pipeline* usado como referência resultou em um conjunto total de 17 ameaças. Dentre elas, sete não se aplicam ao *pipeline* do Fedora. Na modelagem do *pipeline* do projeto Fedora foram encontradas duas ameaças adicionais que não foram mencionadas no modelo do *pipeline* de referência, porque são específicas do processo de desenvolvimento do Fedora. Ou seja, a modelagem de ameaças para esse projeto resultou em um conjunto total de 12 ameaças, as quais são mitigadas totalmente ou parcialmente no Fedora.

Palavras-chaves: integridade de software; *pipeline* de desenvolvimento de software; modelagem de ameaças; STRIDE.

Abstract

In recent years, there has been a growing concern with software integrity, that is, the assurance that software has not been tampered with on the path between developers and users. This path is represented by a software development pipeline. The pipeline adopted as a reference in this work consists of five stages (Integration, Continuous Integration, Infrastructure as code, Deployment, and Release), and vulnerabilities or attacks in all these stages have already been reported in the literature. We develop a threat model for the entire software development pipeline using the STRIDE methodology, and identify possible mitigations for each threat. We apply this threat model to the pipeline used by the Fedora Linux distribution to generate operating system images and distribute them to end users, with the goal of finding out which threats are mitigated and how. Modeling the pipeline used as a reference resulted in a total set of 17 threats. Among them, seven do not apply to the Fedora pipeline. In the Fedora project pipeline modeling were found two additional threats that were not mentioned in the reference pipeline model because they are specific to the Fedora development process. That is, the threat modeling for this project resulted in a total set of 12 threats, which are fully or partially mitigated in Fedora.

Keywords: software integrity; software development pipeline; threat modeling; STRIDE.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

2FA	Autenticação de dois fatores
AC	Autoridade Certificadora
ACL	*Access Control List*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
CLI	*Command Line Interface*
DANE	*DNS-Based Authentication of Named Entities*
DDC	*Diverse Double Compiling*
DFD	Diagrama de Fluxo de Dados
DNS	*Domain Name System*
DNSSEC	*Domain Name System Security Extensions*
FAS	*Fedora Account System*
GPG	*GNU Privacy Guard*
HTTPS	*Hyper Text Transfer Protocol Secure*
IaC	*Infrastructure as Code*
IC	Integração Contínua
IP	*Internet Protocol*
ISO	*International Organization for Standardization*
MFA	*Multi Factor Authetication*
MITM	*Man-in-the-Middle*
MM	*MirrorManager*
NFS	*Network File System*
npm	*Node Package Manager*
OTP	*One Time Password*

PKI	*Public Key Infrastructure*
QA	*Quality Assurance*
releng	*Fedora Release Engineering*
RPM	*RPM Package Manager*
SDL	*Security Development Lifecycle*
SSH	*Secure Shell*
STRIDE	*Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege*
SwA	*Software Assurance*
TLS	*Transport Layer Security*
TOFU	*Trust on First Use*
TOTP	*Time-based One-time Password*
UI	*User Interface*
UML	*Unified Modeling Language*
VCS	*Version Control System*
VM	*Virtual Machine*
XML	*Extensible Markup Language*

1 Introdução

Uma preocupação crescente nos últimos anos tem sido a garantia de segurança de software (*software security assurance*), que pode ser definida como a confiança de que software, hardware e serviços estejam livres de vulnerabilidades intencionais e não intencionais, e que o software funcione como desejado (CRACIUN et al., 2020; TORRES-ARIAS et al., 2019; SONATYPE, 2020; SONATYPE, 2019). Um relatório da SAFEcode, uma organização dedicada à segurança de software que conta com vários representantes da indústria, define que essa confiança envolve três objetivos: segurança, integridade e autenticidade (SIMPSON, 2010). Segundo essa definição, a segurança foca em evitar erros de codificação, requisitos incompletos e implementação malfeita. A integridade consiste em garantir que o software não foi modificado durante as etapas de compilação e entrega ao usuário. A autenticidade garante que o software não é falsificado.

Os três objetivos podem ser ameaçados tanto por hardware quanto por software. Ataques via hardware são menos prevalentes; embora tenha sido alegado um ataque envolvendo a inserção de chips maliciosos em placas-mãe de servidores (ROBERTSON; RILEY, 2018), isso nunca ficou comprovado (OWEN, 2018; GALLAGHER, 2019). Portanto, neste trabalho são considerados apenas aspectos relacionados ao software.

No tocante ao software, o objetivo de segurança tem sido tradicionalmente o mais enfatizado. Porém, segundo Simpson (2010) uma área de preocupação emergente é a integridade do software, devido ao risco de introdução de código malicioso durante o ciclo de desenvolvimento do software. Logo, é necessário garantir que o software que chega até o usuário seja o mesmo produzido pelo desenvolvedor, sem que sejam introduzidas vulnerabilidades no meio do caminho. Esse trajeto compreende várias etapas e é conhecido como *pipeline* de desenvolvimento de software (ADAMS; MCINTOSH, 2016).

A modelagem de ameaças tem como objetivo identificar, em nível de projeto, as necessidades de segurança de um software, e é uma parte importante do *Security Development Lifecycle* (SDL) da Microsoft (HERNAN et al., 2006). Uma abordagem que pode ser usada para a modelagem de ameaças é conhecida como STRIDE, desenvolvida e usada pela Microsoft. Esse modelo consiste em decompor o sistema em componentes relevantes, analisar cada um

deles quanto à suscetibilidade às ameaças, e mitigar as ameaças encontradas.

Embora a segurança do *pipeline* de desenvolvimento tenha recebido atenção na literatura (SIMPSON, 2010; BASS et al., 2015; SHAW, 2017; WHEELER; REDDY; FONG, 2018; TORRES-ARIAS, 2020), uma lacuna diz respeito à identificação sistemática das ameaças que afetam o *pipeline* e suas possíveis mitigações. O objetivo deste trabalho é preencher essa lacuna por meio de uma modelagem de ameaças do *pipeline* de desenvolvimento e da discussão de mitigações para as ameaças encontradas, com ênfase em ameaças que afetem a integridade. O modelo de ameaças é aplicado ao *pipeline* da distribuição Fedora Linux, o qual está documentado publicamente. Esse *pipeline* é analisado para identificar as ameaças já mitigadas e as mitigações adotadas, realimentando assim o modelo, bem como eventuais ameaças não mitigadas para as quais é possível apontar mitigações aplicáveis.

É válido ressaltar que este trabalho está alinhado com o que é apresentado pela norma ISO/IEC 27002, dado que o objetivo do mesmo é garantir a segurança da cadeia de suprimentos de software, equiparando-se a referida norma, a qual propõe práticas para controle de segurança da informação, menciona políticas de desenvolvimento seguro, e apresenta considerações necessárias para um ambiente de desenvolvimento livre de vulnerabilidades (ABNT, 2013). Porém, a ISO/IEC 27002 aborda aspectos e considerações de segurança apenas de modo conceitual, ou seja, ela indica o que deve ser feito com relação à segurança durante o desenvolvimento de sistemas, mas não detalha ou mostra exemplos de como implementá-los na prática. Já o presente trabalho sugere, por meio das mitigações, mecanismos para tornar seguro o processo de desenvolvimento de software, isto é, aborda os aspectos e considerações de segurança de forma concreta.

1.1 Objetivos

Este trabalho tem como objetivo geral propor um modelo de ameaças que sistematize as ameaças a *pipelines* de desenvolvimento de software e suas mitigações, com ênfase em ameaças que afetem a integridade do software.

O objetivo geral pode ser desdobrado nos seguintes objetivos específicos:

1. Desenvolver um modelo de sistema para um *pipeline* de desenvolvimento de software;
2. Desenvolver um modelo de ameaças para o *pipeline* de desenvolvimento, identificando

ameaças e suas possíveis mitigações; e

3. Analisar ao menos um *pipeline* documentado publicamente com vistas a realimentar o modelo de ameaças e identificar eventuais ameaças não mitigadas.

1.2 Metodologia

Os métodos de pesquisa adotados neste trabalho são pesquisa referenciada e pesquisa aplicada. Inicialmente foi realizado um levantamento de conceitos a respeito de *pipelines* de desenvolvimento de software, e sobre o processo de modelagem de ameaças. Foi então adotado um *pipeline* típico, o qual possui cinco etapas, e para cada uma delas foram identificados incidentes de segurança mencionados na literatura. Em paralelo, foi realizada uma revisão bibliográfica com o intuito de encontrar trabalhos com temas semelhantes que pudessem auxiliar no desenvolvimento do presente trabalho.

Em seguida foram aplicados os conceitos aprendidos anteriormente, isto é, foi realizada a modelagem do *pipeline* de desenvolvimento de software, a qual apresenta as ameaças encontradas e as mitigações aplicáveis. A seguir foi realizada a leitura da documentação da distribuição Fedora Linux, com o intuito de entender o processo de desenvolvimento desse sistema. Por fim, aplicando todos os conceitos aprendidos nas etapas anteriores, desenvolveu-se a modelagem de ameaças para o *pipeline* de desenvolvimento do Fedora.

1.3 Organização do texto

O presente trabalho está organizado no seguinte formato: o Capítulo 2 apresenta o *pipeline* de desenvolvimento de software, descreve suas etapas, e para cada uma delas são identificadas vulnerabilidades ou ataques já mencionados na literatura. Este mesmo capítulo aborda os conceitos e exemplos relacionados a modelagem de ameaças, com foco no modelo STRIDE, e apresenta trabalhos relacionados. O Capítulo 3 aborda a modelagem do *pipeline* de desenvolvimento de software por meio de um Diagrama de Fluxo de Dados, e apresenta a modelagem de ameaças para o mesmo. Já o Capítulo 4 apresenta o *pipeline* de desenvolvimento da distribuição Fedora Linux, bem como a modelagem de ameaças para o mesmo. Por fim, o Capítulo 5 resume as realizações deste trabalho e apresenta as conclusões obtidas com o desenvolvimento do mesmo.

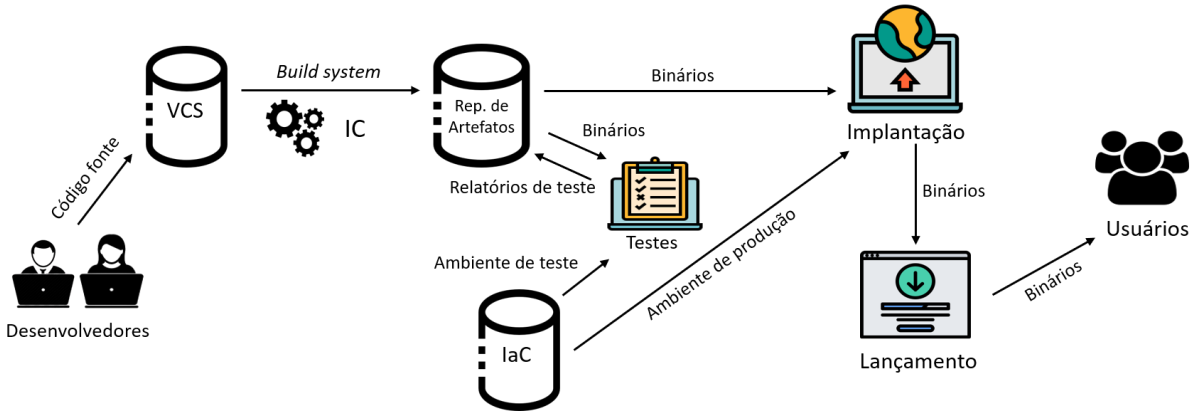
2 Revisão da literatura

Esse capítulo apresenta a revisão de literatura dos assuntos relacionados a este trabalho. A Seção 2.1 descreve as etapas do *pipeline* de desenvolvimento de software, introduzindo alguns riscos e incidentes de segurança relacionados a cada etapa. A Seção 2.2 discute sobre a modelagem de ameaças, incluindo um exemplo que envolve a modelagem de um sistema e faz uso da abordagem STRIDE para a modelagem de ameaças do mesmo. A Seção 2.3 faz uma revisão dos trabalhos relacionados na literatura. Por fim, a Seção 2.4 aborda algumas considerações sobre o exposto nesse capítulo.

2.1 *Pipeline* de desenvolvimento de software

Transformar o código fonte escrito por um desenvolvedor em um produto de software disponibilizado a um usuário final é um processo complexo, que envolve várias atividades, indo desde a compilação até a distribuição ou liberação de pacotes ou atualizações de software. Modernamente, convencionou-se chamar de *pipeline* de desenvolvimento de software a sequência de atividades que implementam esse processo complexo. Não foi encontrada uma definição rígida de quais são as etapas do *pipeline*; a Figura 2.1 representa um *pipeline* típico (ADAMS; MCINTOSH, 2016). Ele envolve as fases de Integração: ramificação e junção (*Integration: branching and merging*), Integração Contínua: compilação e teste (*Continuous Integration: building and testing*), Infraestrutura como código (*Infrastructure as code*), Implantação (*Deployment*) e Lançamento (*Release*). Embora nem todos os projetos tenham um *pipeline* idêntico ao da Figura 2.1, adotar esse *pipeline* como referência permite que todas as atividades do trajeto desenvolvedor $\rightarrow$ usuário sejam contempladas na modelagem de ameaças que é o objetivo deste trabalho. Projetos que tenham um *pipeline* mais simples poderão apenas desconsiderar partes do modelo de ameaças, e projetos que eventualmente tenham um *pipeline* mais complexo poderão usar nosso modelo de ameaças como uma *baseline* a ser complementada. As etapas do *pipeline* de referência são descritas a seguir.

Figura 2.1: *Pipeline* típico de desenvolvimento de software



Fonte: Adaptado de Adams e McIntosh (2016).

2.1.1 Integração: ramificação e junção

A primeira fase do *pipeline* envolve as mudanças que são feitas no código fonte de um software. Em geral, um desenvolvedor faz alterações de código em um ramo (*branch*) privado de desenvolvimento, que é uma versão local do código. Após essas alterações terem sido concluídas, elas são propagadas e consolidadas no ramo usado pela equipe em que esse desenvolvedor atua, e daí para o ramo principal do projeto (ADAMS; MCINTOSH, 2016).

Para que seja possível trazer alterações de código para o ramo principal do projeto sem reduzir a sua qualidade, muitas organizações de software fazem uso de Sistemas de Controle de Versão (do inglês *Version Control System* (VCS)), como o Subversion ou o Git (ADAMS; MCINTOSH, 2016). Com o controle de versão, para fazer uma modificação no código a partir do ramo principal do projeto é criado um novo ramo no qual o desenvolvedor pode fazer as mudanças necessárias, este ramo é isolado dos demais. Após fazer as alterações desejadas o desenvolvedor pode efetuar a junção (*merge*) do seu ramo no ramo principal do projeto, tornando as mudanças visíveis para os demais membros da equipe.

Uma outra forma de lidar com ramificações e junções são os *pull requests*, característicos de projetos de código aberto (BITBUCKET, 2021). Neste modelo, um desenvolvedor clona um repositório (mesmo que tenha apenas acesso de leitura) e efetua modificações em um ramo no seu repositório privado. Quando esse desenvolvedor quiser que um conjunto de modificações seja incorporado ao repositório original, ele submete um *pull request* indicando o ramo onde estão as modificações. Esse *pull request* é revisado por um desenvolvedor com acesso de escrita ao repositório original, que pode aceitar, rejeitar ou solicitar mudanças antes da aceitação. Essas mudanças podem ser realizadas no ramo privado do autor do *pull request*.

Quando a solicitação é aceita, o novo ramo é juntado ao repositório original. Esse processo tipicamente é assistido por ferramentas computacionais, como GitHub[1] e Bitbucket[2]. *Pull requests* auxiliam no desenvolvimento distribuído de software e possibilitam a dissociação do esforço da equipe de desenvolvimento da decisão de juntar ou não as mudanças de código no ramo principal do projeto. Em outras palavras, o trabalho é distribuído entre duas equipes, uma delas é composta por colaboradores/desenvolvedores, responsáveis pelo desenvolvimento de código, e outra equipe (principal) é encarregada de supervisionar o processo de junção e aceitar ou não as solicitações de *pull request* (GOUSIOS; PINZGER; DEURSEN, 2014). Essas equipes também podem ser nomeadas como Desenvolvedores (iniciantes e intermediários) e Desenvolvedores Sêniores respectivamente, a diferença entre ambos é o nível de permissão que cada um tem no sistema. Vale ressaltar que as aprovações das mudanças para o ramo principal do projeto são registradas pelo VCS.

Nesta etapa do *pipeline*, tem-se um código fonte sendo compartilhado com vários desenvolvedores e sendo armazenado em um repositório de controle de versão. Pensando nesses tópicos, foram encontrados alguns possíveis problemas do ponto de vista da integridade, por exemplo, caso um usuário mal intencionado obtenha acesso ao repositório ele pode alterar o código fonte e inserir um código malicioso, ou um atacante pode ter acesso ao sistema de controle de versão e injetar um *malware* com uma *backdoor* incorporada para inserir vulnerabilidades no código. Segundo Simpson (2010), o código fonte dos componentes e produtos de software deve ser armazenado com segurança, com os controles de acesso necessários.

Já foram registrados vários casos reais de ataques nesta etapa de integração. Exemplos incluem a inserção de código não autorizado no sistema operacional de equipamentos de rede Juniper, criando uma *backdoor* para acesso remoto ao equipamento e permitindo monitorar e decifrar tráfego (WORRALL, 2015), a tentativa de modificar o *kernel* do Linux para inserir uma vulnerabilidade (CORBET, 2003), e o caso no qual os desenvolvedores de extensões do Chrome tiveram suas credenciais roubadas e os invasores puderam modificar as extensões, comprometendo milhões de usuários (MAUNDER, 2017). Outro exemplo de roubo de credencial aconteceu com o OpenDev, um repositório de código fonte que abriga diversos projetos incluindo o gerenciador de nuvens OpenStack: o comprometimento de uma conta com privilégios administrativos em uma ferramenta de revisão de código permitia a adulteração dos repositórios de código fonte hospedados pelo OpenDev, entre outros danos (SHARWOOD, 2020).

[1] https://github.com/
[2] https://bitbucket.org/

2.1.2 Integração Contínua: compilação e teste

Integração Contínua (IC - *Continuous Integration*) refere-se à atividade de pesquisar continuamente o VCS em busca de novas confirmações (*commits*) ou junções, identificando essas revisões, compilando-as e executando um conjunto inicial de testes para verificar se as novas mudanças não causaram problemas no projeto (ADAMS; MCINTOSH, 2016). Para auxiliar nessas atividades pode-se fazer uso das ferramentas de IC como Jenkins[3], Bamboo[4], Team Foundation Server[5] e Buildbot[6]. Segundo Adams e McIntosh (2016), o ciclo de *feedback* rápido resultante da Integração Contínua tem um efeito positivo na produtividade da equipe, sem afetar significativamente na qualidade do código.

A IC tem início logo após a confirmação ou junção, devido a isso o *feedback* precisa ser rápido para que, caso ocorra algum problema, o desenvolvedor possa fazer as devidas alterações (ADAMS; MCINTOSH, 2016). Normalmente, após a compilação, a IC executa apenas alguns testes, aqueles que levam mais tempo são executados posteriormente porque as possíveis falhas destes geralmente indicam um problema mais abrangente, envolvendo um conjunto maior de funcionalidades (ADAMS; MCINTOSH, 2016). Porém, é provável que os testes mais rápidos terão um efeito maior no desenvolvimento do projeto.

Esta etapa envolve diferentes estágios de testes, por exemplo, testes de aceitação, testes de desempenho, teste de UI, testes manuais, entre outros (ADAMS; MCINTOSH, 2016). Ou seja, considera-se tanto testes automatizados quanto manuais. Segundo Humble e Farley (2010), os testes de aceitação automatizados costumam ser seguidos por testes manuais como teste exploratório, teste de usabilidade e *showcases*. Neste caso, o papel de um testador é garantir que os testes de aceitação automatizados tenham validado de forma genuína o comportamento do sistema, isto é, comprovando manualmente que os critérios de aceitação são atendidos (HUMBLE; FARLEY, 2010). Vale ressaltar que os testadores não têm permissão para modificar o código fonte ou arquivos binários.

Todos os estágios de teste obtêm os produtos que foram construídos pelo processo de IC a partir do repositório de artefatos (*artifact repository*) (ADAMS; MCINTOSH, 2016). Este é um servidor de arquivos que marca qualquer tipo de documento com metadados como IDs de confirmação ou nomes de versão. Dois exemplos de ferramentas de repositório de artefatos

[3]https://jenkins.io/
[4]https://www.atlassian.com/software/bamboo
[5]https://docs.microsoft.com/en-us/visualstudio/releasenotes/tfs2017-relnotes
[6]https://buildbot.net/

são o Artifactory[7] e o Nexus[8], estes podem localizar rapidamente os produtos corretos que foram lançados na versão ou compilação, caso seja relatado algum erro em uma versão (*release*) específica ou uma falha de teste em uma compilação de IC particular (ADAMS; MCINTOSH, 2016).

Um dos componentes mais importantes da etapa de Integração Contínua é o sistema de compilação (*build system*), que gera resultados do projeto como binários, bibliotecas ou pacotes a partir do código fonte (ADAMS; MCINTOSH, 2016). Um sistema de compilação é dividido em duas camadas, configuração e construção. A camada de configuração é usada para selecionar quais recursos devem ser compilados e incluídos nas entregas resultantes, bem como quais ferramentas de compilação (*build tools*) são necessárias para processar esses recursos (ADAMS; MCINTOSH, 2016). Já a camada de construção é usada para especificar as invocações da ferramenta de compilação (*build tool*) necessárias para gerar entregas a partir do código fonte, respeitando suas dependências (ADAMS; MCINTOSH, 2016).

Há diferentes ferramentas de compilação para diferentes linguagens de programação, podendo ser baseadas em arquivos, tarefas, ou ciclo de vida. Um exemplo de compilação baseada em arquivos é o GNU Make[9], baseado em tarefas tem-se o Ant[10], e o Maven[11] pode ser usado como exemplo de tecnologia de compilação baseado no ciclo de vida. Existem ainda geradores de sistemas de compilação, como o CMake[12], que fornecem uma linguagem de alto nível para especificar uma compilação. Um gerador processa uma especificação de modo a produzir um conjunto de arquivos de compilação para uma ferramenta de um dos três tipos anteriores.

Nesta fase do *pipeline* foram encontrados alguns possíveis problemas do ponto de vista da integridade, por exemplo um atacante pode inserir um *backdoor* na ferramenta de IC ou de compilação utilizada no projeto e assim inserir vulnerabilidades no software, ou ele pode usar a técnica conhecida como *typosquatting*[13] e alterar essa ferramenta de forma que o desenvolvedor faça *download* de um código malicioso para o projeto. Outro exemplo é um usuário

[7] https://jfrog.com/artifactory/
[8] https://www.sonatype.com/product-nexus-repository
[9] https://www.gnu.org/software/make/
[10] https://ant.apache.org/
[11] https://maven.apache.org/
[12] https://cmake.org/
[13] Neste contexto, o ataque de *typosquatting* consiste em carregar em um repositório um pacote fraudulento com nome similar ao de um pacote popular (TAYLOR et al., 2020). A expectativa é que, ao digitar erroneamente o nome do pacote, uma fração de usuários instalem por engano o pacote fraudulento em vez do popular. Frequentemente, um pacote fraudulento contém vulnerabilidades de segurança, que poderão ser usadas para comprometer software que faça uso do pacote.

mal intencionado poder ter acesso ao repositório de artefatos e alterar os arquivos de forma a torná-los vulneráveis. Um atacante também pode explorar o ambiente de compilação e encontrar alguma brecha, talvez um pacote ou uma biblioteca diretamente acessível de fora do ambiente. Se esse atacante for bem sucedido, ele pode obter os privilégios do processo (BASS et al., 2015).

Existem casos reais de ataques nesta etapa do *pipeline* de desenvolvimento, por exemplo o uso de *typosquatting* para registrar pacotes com conteúdo malicioso no repositório npm para JavaScript usando nomes semelhantes ao de pacotes legítimos, visando infectar desenvolvedores que cometessem erros de digitação (CIMPANU, 2017). Mais especificamente sobre o sistema de compilação pode-se citar o clássico compilador Trojan descrito por Ken Thompson em seu discurso de aceitação do prêmio Turing (THOMPSON, 1984). O compilador de Thompson foi um experimento em condições controladas, mas também existem casos reais. Um exemplo é o XcodeGhost, uma versão falsificada do ambiente de desenvolvimento Xcode da Apple que incluía código malicioso juntamente com o código do aplicativo real (XIAO, 2015), e que foi responsável por mais de 2500 aplicações infectadas na *App Store* do iOS (FRANCESCHI-BICCHIERAI, 2021). Outro exemplo é o caso no qual a ferramenta Webmin foi comprometida (WEBMIN, 2019). Os atacantes invadiram o servidor de *build*, no qual havia uma cópia local do código fonte, e introduziram vulnerabilidades em um arquivo sem alterar seu timestamp de modificação. Devido a isso, o Git não mostrava nenhuma diferença em relação à versão que estava no repositório VCS. Essa cópia local foi usada em *builds* distribuídos via SourceForge (que não eram gerados a partir do repositório do VCS). Vale ressaltar que uma das medidas tomadas após o incidente foi eliminar a cópia local, ou seja, todos os *builds* passaram a ser feitos a partir do repositório do VCS.

Um novo ataque a cadeia de suprimentos de software possibilitou violar os sistemas internos de mais de 35 grandes empresas, incluindo Microsoft, Apple, PayPal, Shopify, Netflix, Yelp, Tesla e Uber (BIRSAN, 2021). Segundo Birsan (2021), que foi o divulgador do ataque, ao receber um arquivo *package.json* de um pacote npm[14] usado internamente pelo PayPal, ele notou que esse arquivo continha uma mistura de dependências públicas e privadas, ou seja, pacotes públicos do npm, bem como nomes de pacotes não públicos, provavelmente hospedados internamente pelo Paypal. De acordo com Birsan (2021), se um pacote de dependência usado por um aplicativo existir tanto em um repositório de código aberto público quanto em sua construção privada, o pacote público tem prioridade. Com base nisso, o ataque consiste em criar

[14]npm é um gerenciador de pacotes para a linguagem de programação JavaScript. https://www.npmjs.com/

projetos falsificados em repositórios de código aberto, como npm, PyPI e RubyGems, e usar os mesmos nomes dos projetos privados. Esse tipo de vulnerabilidade foi chamado de *dependency confusion*, e permite entrar nas redes de algumas das maiores empresas de tecnologia, obtendo execução remota de código e, possivelmente, permitindo que invasores adicionem *backdoors* durante as compilações.

2.1.3 Infraestrutura como código

Nesta fase do *pipeline* pode-se usar o termo infraestrutura ou ambiente para referir-se ao servidor, nuvem, contêiner ou máquina virtual na qual uma nova versão do sistema deve ser implantada para teste ou produção (ADAMS; MCINTOSH, 2016). Para que não seja mais preciso configurar um ambiente manualmente, pode-se usar a infraestrutura como código (*Infrastructure as Code* - IaC) para automatizar essa etapa. Essa fase gera o ambiente com base na especificação desenvolvida em uma linguagem de programação dedicada como Puppet[15], Chef[16], CFEngine[17], Ansible[18] ou Salt[19] (ADAMS; MCINTOSH, 2016). Essas ferramentas permitem automatizar o provisionamento de infraestruturas virtuais e a instalação e configuração do sistema operacional e de serviços auxiliares, garantindo que a aplicação tenha um ambiente consistente e correto para executar.

Nessa etapa do *pipeline*, caso o código de infraestrutura esteja armazenado no mesmo repositório do código fonte do projeto, as pessoas que possuem acesso a esse repositório poderão alterar o código de infraestrutura, podendo prejudicar o ambiente de teste ou produção, comprometendo a integridade do projeto. Caso esse código de infraestrutura seja armazenado em um repositório próprio, um atacante pode ter acesso e inserir código malicioso, ou um usuário mal intencionado pode inserir um *backdoor* no ambiente como Puppet e Chef por exemplo, e este ser usado no projeto, inserindo vulnerabilidades no código. Outro exemplo é a geração, por meio de ferramentas como Puppet, Chef e Ansible, de *scripts* vulneráveis. Estes *scripts* podem propiciar, por exemplo, a criação de um ambiente para ataques de *Man-in-the-Middle* (MITM), em que um atacante pode criar elementos de arquitetura que forcem o tráfego a passar por um *proxy* que pode observar e/ou modificar esse tráfego.

Especificações vulneráveis vão gerar infraestruturas vulneráveis, o que é particu-

[15] https://puppet.com/
[16] https://www.chef.io/
[17] https://cfengine.com
[18] https://www.ansible.com/
[19] https://docs.saltstack.com/en/latest/topics/

larmente perigoso quando se trata de ambientes de produção. Um relatório recente identificou quase 200.000 vulnerabilidades em especificações IaC (Palo Alto Networks, 2020b). Rahman, Parnin e Williams (2019) identificaram 21.201 ocorrências de sete *security smells* em *scripts* do Puppet. Os *security smells* são padrões de codificação que podem indicar vulnerabilidades na segurança, sendo necessário uma investigação mais aprofundada dos mesmos (RAHMAN; PARNIN; WILLIAMS, 2019). Os mesmos autores replicaram o estudo usando 50.323 *scripts* Ansible e Chef, e encontraram 46.600 *security smells* (RAHMAN et al., 2021). Segundo Palo Alto Networks (2020a), quando os problemas de segurança dos modelos de infraestrutura como código não são verificados, o ambiente de nuvem pode ser exposto de forma indevida, por exemplo, os recursos de monitoramento da nuvem podem ser desabilitados e dessa forma a organização não consegue criptografar, da forma correta, os seus dados tanto em repouso quanto em trânsito.

2.1.4 Implantação

Depois que os produtos de software tiverem sido construídos e testados com sucesso, na fase atual esses produtos são preparados para o Lançamento (*Release*). Por exemplo, para aplicativos *web*, a Implantação (*Deployment*) pode corresponder à cópia de um conjunto de arquivos pela rede para o diretório correto em um servidor *web* (ADAMS; MCINTOSH, 2016). Entre a fase de Implantação e Lançamento o produto fica inativo, porém, há uma exceção que é a atividade de lançamento no escuro (*dark launching*). Esta atividade corresponde a implantar novos recursos que são liberados apenas para uma pequena base de usuários para obter um *feedback* e poder testar e melhorar as novidades (ADAMS; MCINTOSH, 2016). Quando a equipe obtiver um resultado satisfatório, estes recursos serão lançados para os demais usuários.

Existem diversas estratégias que são usadas para permitir a implantação progressiva de uma nova versão e/ou facilitar o retorno à versão anterior, como azul/verde (*blue/green*), canários (*canary*) e testes A/B (*A/B testing*) (HUMBLE; FARLEY, 2010). Essas estratégias podem auxiliar durante o desenvolvimento do produto, permitindo alterar, reverter, testar as funcionalidades, analisar o comportamento do produto em um ambiente real e ajudar a entender como cada recurso pode influenciar no resultado final. Elas podem ser utilizadas tanto individualmente como em conjunto.

Nesta fase do *pipeline* é abordado sobre a possibilidade de mais de uma versão para um mesmo produto, ou seja, possíveis atualizações do mesmo. Pensando nesses tópicos, foram

encontrados alguns possíveis problemas do ponto de vista da integridade, por exemplo, caso um usuário mal intencionado possua acesso ao repositório ele pode alterar o código executável e inserir um *malware* conhecido como Cavalo de Tróia, ou um atacante pode inserir código malicioso em uma ferramenta que será usada no desenvolvimento do projeto.

Existem casos reais de ataques nesta etapa de implantação (*deployment*), por exemplo a disseminação do *malware* NotPetya por meio de atualizações adulteradas de um software de contabilidade (WHEELER; REDDY; FONG, 2018) e a inserção, no repositório de pacotes de terceiros usado por desenvolvedores Python (PyPI), de bibliotecas adulteradas com o mesmo nome das oficiais da linguagem (as quais não devem ser instaladas usando PyPI) (GOODIN, 2017).

2.1.5 Lançamento

É nesta fase final do *pipeline* que os produtos implantados são lançados e ficam disponíveis para todos os usuários do sistema. Os mecanismos de lançamento (*releasing mechanisms*) permitem acesso de modo diferenciado para alguns usuários, como é o caso dos testes A/B citados anteriormente ou acesso a diferentes subconjuntos da funcionalidade do sistema (ADAMS; MCINTOSH, 2016). Um exemplo são aplicativos de jogos nos quais é possível jogar o jogo base, porém, se o jogador pagar uma quantia em dinheiro poderá ter acesso a versão "*Premium*" na qual irá conseguir itens melhores para o aplicativo, e possuirá novas funcionalidades que auxiliarão ou facilitarão o jogo.

Após o produto ser lançado é importante que haja um monitoramento do desempenho do aplicativo. Ou seja, monitorar os dados de telemetria e os *logs* de falhas (ADAMS; MCINTOSH, 2016). Para analisar e coletar esses dados existem diferentes ferramentas, por exemplo, para aplicativos da *web* podem ser usadas Nagios[20] e Splunk[21], para aplicativos móveis há estruturas personalizadas de terceiros, e para aplicativos *desktop*, as organizações desenvolvem suas próprias (ADAMS; MCINTOSH, 2016).

Nesta fase pode-se dizer que o produto está concluído, e segundo Simpson (2010), quando o produto de software estiver completo surgem necessidades de segurança adicionais. Isso inclui verificações antimalware e a disponibilização de um mecanismo que forneça aos clientes uma maneira de garantir a integridade do pacote entregue. Devido a isso, deve-se to-

[20] https://www.nagios.org/
[21] https://www.splunk.com/

mar cuidado caso seja usada alguma ferramenta de verificação de *malware*, pois essa ferramenta pode ter sido prejudicada por um atacante, de modo a permitir inserção de *malware* no software. Também pode ser levada em consideração a possibilidade de algum desenvolvedor ter sua credencial roubada, permitindo que um atacante tenha acesso e insira um *backdoor* no código, sendo assim, quando o usuário fizer o *download* do software, este estará prejudicado e afetará o usuário.

Existem casos reais de ataques nesta etapa de lançamento (*release*), por exemplo a inserção de código malicioso no software Microsoft Windows (versão não especificada) baixado via Tor (HERN, 2014), e a invasão no processo de desenvolvimento ou distribuição de software da Avast em algum estágio anterior à assinatura digital dos binários, o que permitiu que versões da ferramenta CCleaner infectadas com *malware* fossem distribuídas a mais de dois milhões de usuários (WARREN, 2017; BRUMAGHIN et al., 2017).

2.1.6 Discussão

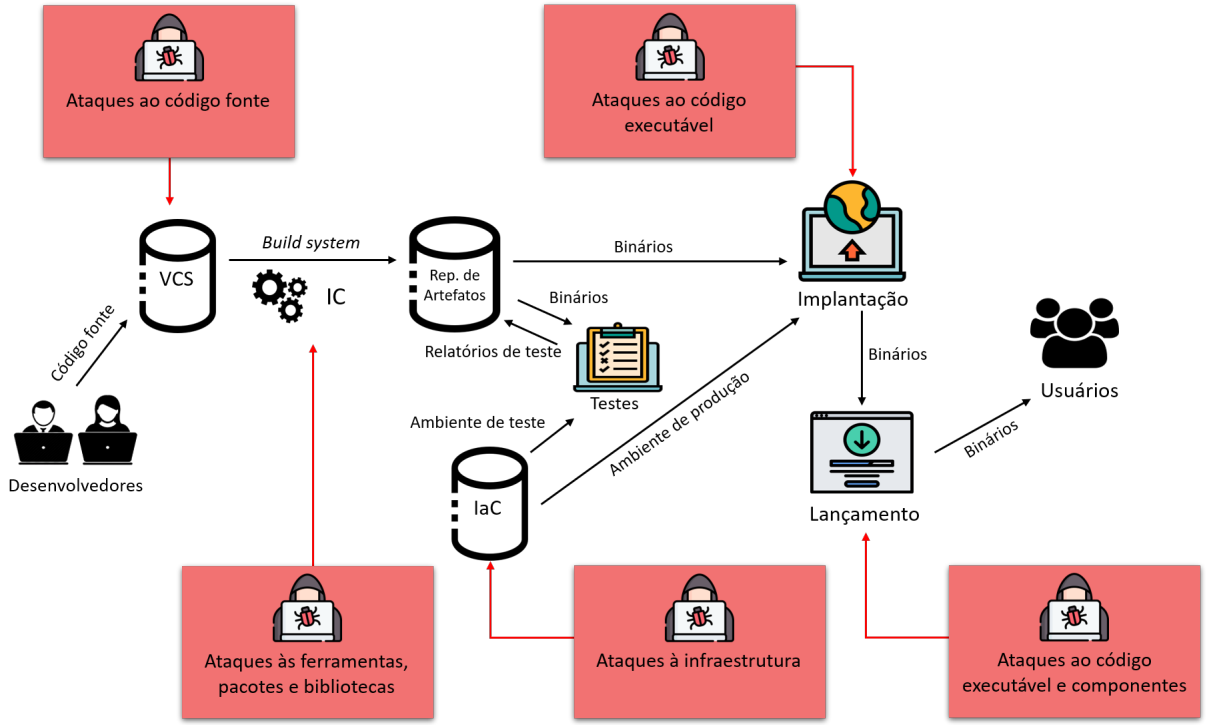
Para cada etapa do *pipeline* foram apresentados casos reais de ataques relatados na literatura. Um resumo desses incidentes de segurança pode ser observado na Figura 2.2. Nesta, observa-se que para a fase de Integração o foco dos ataques é a inserção de vulnerabilidades no código fonte. Já na etapa de Integração Contínua o alvo é o comprometimento das ferramentas, pacotes e bibliotecas que serão usadas no projeto, e a partir destas inserir vulnerabilidades no software.

A etapa de Infraestrutura como código faz uso de *scripts* para gerar ambientes de produção e teste, neste caso a ameaça está na geração de *scripts* vulneráveis, que consequentemente gerarão infraestruturas suscetíveis a ataques. Já na fase de Implantação, os incidentes de segurança encontrados na literatura envolvem a inserção de vulnerabilidades no código executável. Por fim, os ataques encontrados para a etapa de Lançamento envolvem a adulteração indevida do código executável e também o comprometimento dos componentes envolvidos nessa fase.

2.2 Modelagem de ameaças

Uma ameaça pode ser definida como uma possível violação de segurança (SHIREY, 2007). A modelagem de ameaças é um processo para identificar, documentar e mitigar ameaças a um

Figura 2.2: Resumo de incidentes de segurança relatados na literatura para todas as etapas do *pipeline*



Fonte: A autora (2021).

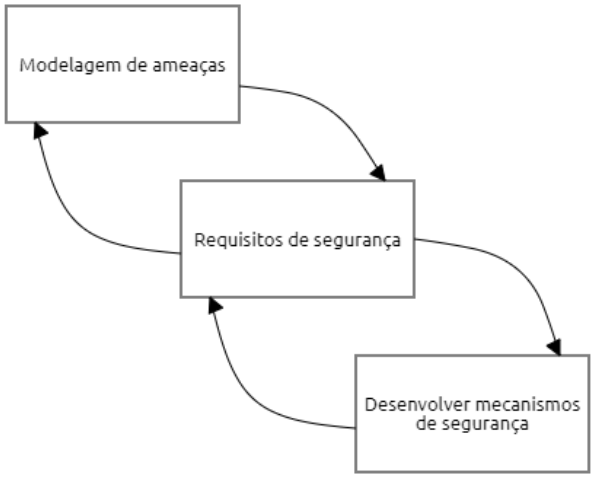
sistema (SHOSTACK, 2014). Esse processo de modelagem envolve algumas etapas e técnicas, as quais são apresentadas nas Seções 2.2.1 a 2.2.3.

2.2.1 O processo de modelagem de ameaças

A modelagem de ameaças tem como objetivo identificar, em tempo de projeto, as necessidades de segurança de um software. Ela envolve identificar os componentes principais do sistema, encontrar e categorizar as ameaças que podem afetar cada componente, priorizá-las com base em uma análise de risco e desenvolver estratégias de mitigação que serão incorporadas ao projeto (SWIDERSKI; SNYDER, 2004).

Na Figura 2.3 observa-se o processo de engenharia de segurança baseado em (MYAGMAR; LEE; YURCIK, 2005). A modelagem de ameaças consiste em entender o sistema e identificar suas possíveis ameaças. Já no decorrer da formação dos requisitos de segurança, essas ameaças são analisadas e é então decidido se as mesmas serão aceitas ou mitigadas (MYAGMAR; LEE; YURCIK, 2005). Após essas determinações, é então iniciado o desenvolvimento dos mecanismos de segurança. Segundo Myagmar, Lee e Yurcik (2005), esse desenvolvimento segue o ciclo da engenharia de software de projeto: implementação, teste

Figura 2.3: Engenharia de segurança do sistema



Fonte: Adaptado de Myagmar, Lee e Yurcik (2005).

e manutenção. Nesse processo de engenharia de segurança também observa-se que cada etapa volta para a anterior, permitindo corrigir erros sem deixar seus efeitos em cascata (MYAGMAR; LEE; YURCIK, 2005).

Uma identificação correta das ameaças ao sistema, juntamente com suas mitigações adequadas, permitem reduzir a possibilidade de eventuais atacantes serem bem sucedidos (MYAGMAR; LEE; YURCIK, 2005). Devido a isso, a modelagem de ameaças considera o sistema do ponto de vista do atacante, permitindo então prever em que pontos poderão ser efetuados ataques, possibilitando definir sobre o que o sistema foi projetado para proteger e de quem (MYAGMAR; LEE; YURCIK, 2005).

Segundo Shostack (2014), o processo de modelagem de ameaças envolve quatro questões principais, são elas:

1. O que está sendo construído?
2. O que pode dar errado?
3. O que pode ser feito a respeito das coisas que podem dar errado?
4. A análise foi bem feita?

Para melhor representar o que está sendo construído, recomenda-se a criação de um diagrama. Nesta etapa do processo de modelagem de ameaças é importante criar um modelo do sistema que enfatize seus principais componentes, interconexões e características (MYAG-

MAR; LEE; YURCIK, 2005). O sistema pode ser modelado usando diferentes diagramas, por exemplo, Diagrama de Fluxo de Dados (DFD), UML, diagrama de estados, entre outros.

A próxima etapa da modelagem consiste em encontrar ameaças à segurança do sistema, ou seja, o que pode dar errado. Para auxiliar nesta fase pode-se fazer uso de jogos como *Elevation of Privilege* (SHOSTACK, 2014) e OWASP Cornucopia (OWASP, 2020a), utilizar modelos como STRIDE, PASTA e LINDDUN, ou usar outros métodos como CVSS (SHEVCHENKO et al., 2018), árvores de ataques, entre outros (ENG, 2017). Em Shevchenko et al. (2018) são analisados 12 métodos diferentes de modelagem de ameaças (STRIDE, PASTA, LINDDUN, CVSS, Árvores de ataques, Persona non Grata, *Security Cards*, hTMM, *Quantitative Threat Modeling Method*, Trike, VAST e OCTAVE), o estudo aponta que o STRIDE é o mais maduro dentre eles, auxilia na identificação de técnicas de mitigação relevantes e é fácil de usar. Devido a isso, neste trabalho é feito uso do STRIDE, o qual será explicado em mais detalhes na Seção 2.2.3.

Após encontrar as ameaças, é necessário decidir o que fazer com cada uma. Segundo Shostack (2014), há quatro maneiras de gerenciar o risco das ameaças identificadas:

1. Mitigar o risco: apresentar contramedidas para reduzir o risco (MYAGMAR; LEE; YURCIK, 2005), por exemplo, utilizar autenticação para reduzir uma ameaça de falsificação.
2. Eliminar o risco: pode ser obtido por meio da remoção de funcionalidades ou recursos.
3. Transferir o risco: deixar que alguém ou outra coisa lide com o risco, por exemplo, passar as ameaças de autenticação ao sistema operacional.
4. Aceitar o risco: em alguns casos a mitigação de uma ameaça pode ter um custo muito alto, ou o risco dessa ameaça pode ser muito baixo, dessa forma a melhor alternativa é aceitar o risco.

Na última etapa da modelagem é realizada a verificação da análise, ou seja, é avaliado o diagrama, as ameaças e suas respectivas mitigações, se houver. Caso tenha alguma inconsistência ou algo faltando, é feita a correção e atualização do modelo.

2.2.2 Modelagem do sistema

Segundo Myagmar, Lee e Yurcik (2005), para realizar uma modelagem de ameaças é necessário entender as características fundamentais do sistema, ou seja, compreender seus componentes,

interações e dependências.

Conforme mencionado na Seção 2.2.1, o sistema pode ser modelado usando diferentes diagramas, por exemplo, diagrama de fluxo de dados, UML, diagrama de estados, entre outros. De acordo com Hernan et al. (2006), normalmente são usados os diagramas de fluxo de dados para representar um sistema, mas é possível utilizar diagramas diferentes desde que o sistema seja decomposto em partes e para cada uma delas seja exposto que a mesma não está sujeita às principais ameaças.

Segundo Myagmar, Lee e Yurcik (2005), usar o diagrama de fluxo de dados facilita a identificação de ameaças, porque permite visualizar como os fluxos de dados são processados pelo sistema, bem como a interação entre os seus componentes. O DFD é uma ferramenta clássica de Engenharia de Software (VIE, 2000) que representa como os dados fluem em um sistema usando os seguintes componentes:

- Entidade externa: entidade que recebe e/ou envia dados de/para o sistema, mas está fora do seu escopo. É representado por um retângulo como na Figura 2.4(a);
- Processo: componente que manipula dados de entrada para gerar dados de saída, transformando ou redirecionando os dados. É representado por um círculo como na Figura 2.4(b);
- Depósito de dados: recipiente onde um processo armazena dados que podem ser recuperados pelo mesmo ou por outro processo, ou por uma entidade externa. É representado por duas linhas em paralelo como na Figura 2.4(c);
- Fluxo de dados: é a movimentação de dados entre uma entidade, um processo e/ou um depósito de dados. É representado por uma seta como na Figura 2.4(d).

Para ajudar na identificação de ameaças, acrescenta-se no diagrama as fronteiras de confiança (ENG, 2017). Estas indicam os componentes que estão em diferentes contextos de segurança (por exemplo, domínios de proteção) (SHOSTACK, 2014). Elas são representadas por uma linha tracejada como na Figura 2.4(e). De acordo com Shostack (2014), as ameaças que cruzam essas fronteiras são, provavelmente, importantes e podem ser um ponto de partida interessante para a identificação das ameaças.

Para ilustrar o diagrama de fluxo de dados, considera-se um exemplo de Hernan et al. (2006). O analisador de banco de dados Fabrikam é um sistema que tem como objetivo obter

Figura 2.4: Símbolos de um diagrama de fluxo de dados

(a) Entidade externa (b) Processo (c) Depósito de dados (d) Fluxo de dados (e) Fronteira de confiança

Fonte: A autora (2021).

arquivos de um conjunto de vendedores, realizar algumas análises dos arquivos em um servidor centralizado e produzir relatórios semanais. O DFD correspondente pode ser encontrado na Figura 2.5. O funcionamento do sistema é o seguinte: cada Vendedor envia seus dados sobre as vendas para o processo de Coleta e análise, que recebe dados de análise do depósito Lista de vendedores e envia seus dados gerados para o depósito Processo de análise. O processo Geração de relatórios extrai dados do depósito Processo de análise para gerar e enviar relatórios para o Gerente.

A aplicação segue o modelo cliente-servidor. O lado do cliente possui apenas entidades externas que representam os vendedores. Já no lado do servidor, existem dois processos, uma entidade externa e dois depósitos de dados. Entre os lados do servidor e do cliente há uma fronteira de confiança.

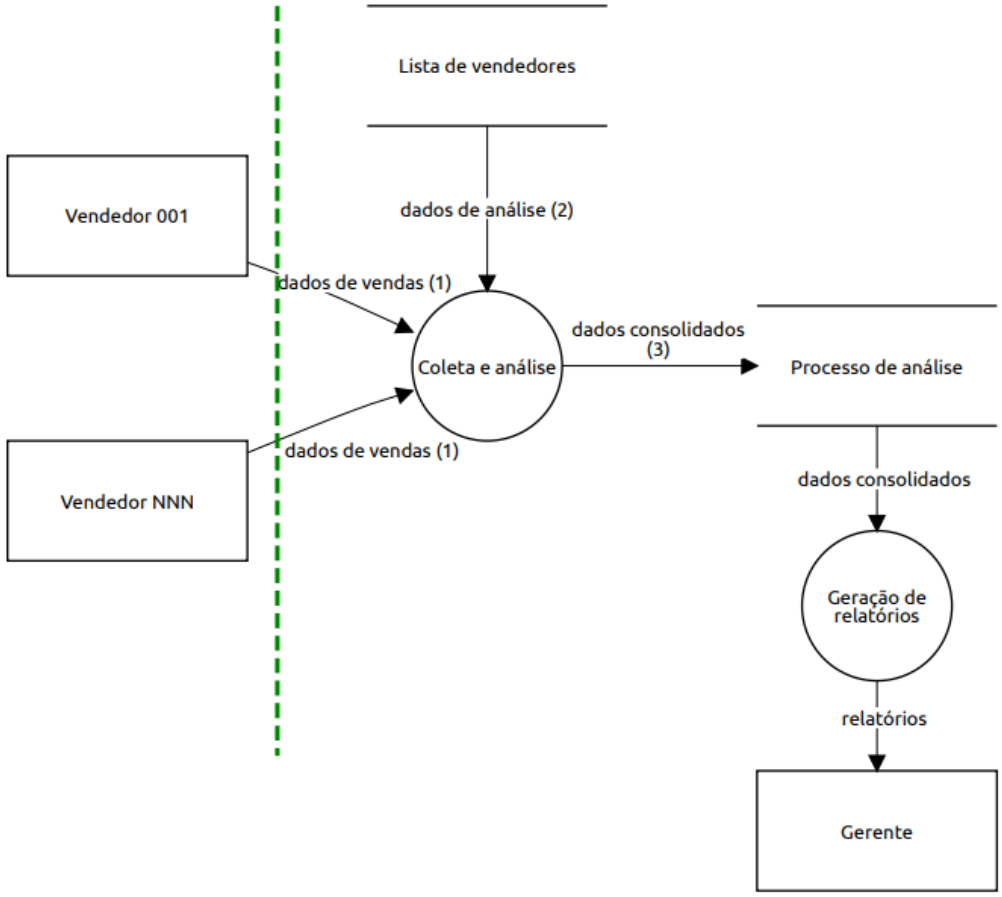
2.2.3 Modelagem de ameaças com STRIDE

Na Microsoft a modelagem de ameaças é uma parte importante do *Security Development Lifecycle* (HERNAN et al., 2006). O Microsoft SDL tem como objetivo ajudar os desenvolvedores a criar softwares seguros, apresentando considerações, técnicas, práticas e ferramentas de segurança e privacidade que podem ser aplicadas em todas as fases do processo de desenvolvimento, possibilitando a redução de custos no mesmo (MICROSOFT, 2020).

Uma abordagem que pode ser usada para a modelagem de ameaças é conhecida como STRIDE. Esta foi criada por Loren Kohnfelder e Praerit Garg na Microsoft em 1999 (ENG, 2017) e é um acrônimo que representa as seguintes seis classes de ameaças:

- *Spoofing* (falsificação de identidade): alegação de uma identidade falsa;
- *Tampering* (manipulação): corrupção ou adulteração de dados;

Figura 2.5: Diagrama de fluxo de dados da Fabrikam



Fonte: Adaptado de Hernan et al. (2006).

- *Repudiation* (retratação): negação da autoria de uma ação;
- *Information disclosure* (revelação de informações): divulgação de informações para usuários não autorizados;
- *Denial of service* (negação de serviço): interrupção no fornecimento de serviços a usuários legítimos; e
- *Elevation of privilege* (elevação de privilégio): quando um usuário ou programa pode agir no sistema com privilégios além dos que lhe foram concedidos.

A Tabela 2.1 mostra a correspondência entre as classes de ameaças do STRIDE e as propriedades de segurança que elas afetam.

Na modelagem de ameaças com STRIDE, os componentes do DFD são analisados quanto à sua suscetibilidade a ameaças de cada classe (HERNAN et al., 2006). Segundo Eng (2017), existem várias abordagens para estruturar essa análise, incluindo STRIDE-por-elemento, STRIDE-por-interação, DESIST e o jogo *Elevation of Privilege*. Este trabalho foca

Tabela 2.1: Ameaças e suas respectivas propriedades de segurança

Ameaça	**Propriedade de segurança**
Spoofing	Autenticação
Tampering	Integridade
Repudiation	Irretratabilidade
Information disclosure	Confidencialidade
Denial of service	Disponibilidade
Elevation of privilege	Autorização

Fonte: Adaptado de Hernan et al. (2006).

no STRIDE-por-elemento porque, segundo Shostack (2014), este é mais prescritivo, ajudando a identificar o que procurar, e torna mais fácil encontrar ameaças dado que foca em um conjunto de ameaças contra cada elemento. Além disso, o STRIDE-por-elemento apresentou melhores resultados que o STRIDE-por-interação em um estudo controlado (TUMA; SCANDARIATO, 2018).

No STRIDE-por-elemento, para cada elemento do diagrama (entidade externa, processo, fluxo e depósito de dados) considera-se a quais ameaças esse elemento está sujeito, de acordo com as possibilidades contidas na Tabela 2.2. Esta mostra, por exemplo, que entidades externas estão sujeitas apenas a ameaças de *spoofing* e retratação, não sendo necessário considerar as demais classes. No caso de depósitos de dados, a ameaça de retratação aplica-se apenas a *logs*. De acordo com Shostack (2014), a Tabela 2.2 é utilizada pela Microsoft como parte central de seu treinamento de modelagem de ameaças no *Security Development Lifecycle*.

Cabe ressaltar que, de modo geral, a análise baseada na Tabela 2.2 considera as ameaças que afetam um elemento, não as que ele pode causar (SHOSTACK, 2014). Por exemplo, quando uma entidade externa interage com um processo, a suscetibilidade da entidade a *spoofing* diz respeito à possibilidade de que ela interaja com um processo ilegítimo, e não que o processo interaja com um usuário ilegítimo (essa segunda possibilidade é uma ameaça de *spoofing* para o processo).

Tabela 2.2: STRIDE-por-elemento

	S	T	R	I	D	E
Entidade externa	✓		✓			
Processo	✓	✓	✓	✓	✓	✓
Fluxo de dados		✓		✓	✓	
Depósito de dados		✓	?	✓	✓	

Fonte: Adaptado de Shostack (2014).

Como exemplo de uso do STRIDE-por-elemento, considera-se o DFD da Figura 2.5. Para esse DFD, Hernan et al. (2006) apontam as seguintes ameaças:

Vendedor: *spoofing* é uma possível ameaça para essa entidade, porque um Vendedor pode ter sua credencial roubada, possibilitando que uma pessoa não autorizada seja considerada um Vendedor legítimo. Como ameaça de retratação, um Vendedor pode enviar algum dado comprometedor ou falso ao processo de coleta e negar sua ação.

Gerente: assim como a entidade externa Vendedor, o Gerente também está vulnerável à ameaças de *spoofing* e retratação.

Coleta e análise: o processo Coleta e análise não confia que os dados recebidos do Vendedor são legítimos, porque um atacante pode estar se passando por um Vendedor e enviar informações falsas para o processo de Coleta e análise. Um processo que finge ser o processo de Coleta e análise pode coletar todos os dados que os vendedores estão tentando enviar. Esta ameaça de *spoofing* pode levar a uma ameaça de divulgação de informação. Como retratação, o processo pode enviar dados de vendas revisados que sobrescrevem os dados existentes e pode alegar que não o fez. Este processo pode ser vítima de ataques de negação de serviço onde um invasor pode impedir que o servidor de coleta seja acessível aos vendedores. Um atacante também pode impedir a coleta normal de dados, causando uma negação de serviço. Neste caso se o invasor fizer isso no último dia de um trimestre, esta ação pode ter um impacto significativo na empresa. Neste exemplo não é possível identificar nenhuma ameaça das demais classes que afetam processos (divulgação de informação, manipulação e elevação de privilégio). Isso ocorre porque o DFD é considerado no nível conceitual, e essas ameaças (neste caso) só podem ser discutidas diante de escolhas concretas de implementação.

Geração de relatórios: assim como o processo anterior, este está vulnerável a ameaças de *spoofing*, *tampering*, divulgação de informação, retratação, negação de serviço e elevação de privilégio.

Fluxo de dados 1: seus dados podem ser modificados enquanto trafegam pela Internet. Outra ameaça é a divulgação de informação, onde os dados em trânsito podem ser lidos por pessoas que não deveriam ter acesso.

Fluxo de dados 2: assim como o fluxo de dados anterior, este está vulnerável a ameaças de *tampering*. Outra ameaça é a divulgação de informação, porque a lista de vendedores pode ser

de interesse para empresas concorrentes ou até mesmo para ex-funcionários.

Fluxo de dados 3: este está totalmente dentro de uma fronteira de confiança, ou seja, acredita-se que não há necessidade de se preocupar com suas possíveis ameaças, pois confia-se em tudo que está inteiramente dentro desta fronteira. Porém, se o depósito Processo de análise se encontra em uma máquina remota, este fluxo está vulnerável à ameaças de *tampering*, divulgação de informações e negação de serviço.

Processo de análise: está vulnerável a manipulação, ou seja, seus dados podem ser modificados por pessoas não autorizadas. Provavelmente, há informações proprietárias no banco de dados Fabrikam que os concorrentes gostariam de ver, devido a isso, este depósito pode ser vítima de divulgação de informação, onde os seus dados podem ser lidos por pessoas que não deveriam ter acesso. Outra ameaça são os ataques de negação de serviço, pois preencher o banco de dados é um ataque simples. Quando o banco de dados está cheio, as respostas comuns são: parar de atender a novas solicitações ou sobrescrever dados antigos.

Lista de vendedores: os seus dados podem ser alterados por pessoas não autorizadas, por exemplo, os invasores podem adicionar vendedores que podem causar danos no sistema ou excluir vendedores e impedi-los de realizar seu trabalho. Um atacante pode adulterar os dados inserindo um novo sistema no depósito Lista de vendedores, o qual pode permitir a entrada de dados falsos. Ameaças de divulgação de informação e negação de serviço também estão presentes.

Segundo Shostack (2014), para identificar ameaças e saber onde se aprofundar, a experiência e um repertório de cenários reais pode vir a ser muito útil. Porém, também é válido fazer uma revisão de literatura e, dependendo do sistema que está sendo trabalhado, bibliotecas de ataques como CAPEC e OWASP Top Ten podem ser proveitosas, outra forma de identificar os riscos ao sistema é por meio de árvores de ameaças que já foram construídas e apresentam casos recorrentes. Exemplos dessas árvores podem ser encontrados em (HOWARD; LIPNER, 2006, Capítulo 22) e (SHOSTACK, 2014, Apêndice B).

2.3 Trabalhos relacionados

O foco deste trabalho é encontrar ameaças à segurança do *pipeline* de desenvolvimento de software e apresentar contramedidas para os mesmos. Nesta seção são discutidos trabalhos que possuem enfoque na segurança do *pipeline* de desenvolvimento.

Em Simpson (2010) é discutida a segurança da cadeia de suprimentos de software, focando especificamente nas práticas e controles de integridade. São apresentadas listas de mecanismos para garantia de segurança de software adotados na indústria, considerando todos os três objetivos (segurança, integridade e autenticidade). Os mecanismos apresentados resumem-se basicamente à definição adequada de controles de acesso, entrega (verificação de *malware*, assinatura de código), autenticidade (componentes criptografados, tecnologia de notificação) e ao uso de criptografia, mas as ameaças consideradas não são explicitadas.

Ferramentas *Software Assurance* (SwA) são ferramentas de detecção de vulnerabilidades tanto estáticas (analisam o código fonte) quanto dinâmicas (analisam o código em execução). Neste trabalho pode-se usá-las nas fases de integração, integração contínua e infraestrutura como código para detectar vulnerabilidades no software durante o seu desenvolvimento. Essas ferramentas também podem vir a ser um risco, porque têm acesso privilegiado a informações como código fonte, pode assim, ser criada uma ferramenta mal intencionada e até mesmo as vulnerabilidades não intencionais de uma ferramenta SwA podem ser exploradas por atacantes. Em Wheeler, Reddy e Fong (2018) são identificadas abordagens práticas para proteger a cadeia de suprimentos dos riscos que podem ser causados pelas ferramentas SwA, é discutida a necessidade da implantação de um conjunto dessas ferramentas para detectar vulnerabilidades com cobertura suficiente para obter a garantia de segurança de software.

Em Shaw (2017) é apresentada uma lista de ataques concretos contra a cadeia de suprimentos de software. O autor associa os ataques a etapas específicas na cadeia de suprimentos, ou seja, é dito se cada ataque ocorreu nas ferramentas de desenvolvimento, no código fonte, na distribuição ou na atualização do software. Mostra-se também o quanto estão crescendo os números de ataques à cadeia de suprimentos de software. Embora diversos ataques sejam discutidos, não são apresentadas contramedidas para os mesmos.

Bass et al. (2015) discute sobre como um *pipeline* pode ser subvertido, apresentando três cenários. O primeiro é quando a imagem implantada não é válida, o segundo é quando uma imagem é implantada sem passar pelo *pipeline* completo e o terceiro é quando o ambiente de

produção é acessível a partir de um ambiente diferente. O trabalho tem como foco apenas o primeiro cenário apresentado. Para tornar o *pipeline* confiável é fornecido um processo de engenharia baseado em componentes confiáveis, eles limitam o acesso e permissões dos componentes, sendo assim, o atacante só pode acessar os componentes confiáveis. Esse trabalho tem um foco restrito, porque é feito uso de um *pipeline* específico envolvendo Chef, Jenkins, Docker, Github e AWS.

Paule (2018) apresenta abordagens e métodos que existem para detectar vulnerabilidades nos *pipelines* de entrega contínua (*continuous delivery*). O método de modelagem de ameaças utilizado foi o STRIDE. O trabalho também apresenta ferramentas para detecção ou mitigação de vulnerabilidades e faz um estudo de caso em uma empresa, o qual aplicou ferramentas selecionadas para verificar o nível de segurança existente em dois *pipelines* industriais de entrega contínua. O foco do trabalho é a detecção de vulnerabilidades, não mencionando mecanismos de proteção.

Torres-Arias (2020) apresenta o in-toto, um *framework* que tem como objetivo garantir criptograficamente a integridade da cadeia de suprimentos de software. A ideia dessa ferramenta é proteger, por meio de criptografia, os produtos gerados em cada etapa da cadeia de suprimentos, de modo a permitir que cada fase receba dados legítimos. Desse modo também é possível verificar por quais etapas o software passou. Porém, o trabalho não leva em consideração a adulteração de uma etapa, apenas garante a integridade do fluxo entre as fases, isto é, se os produtos não foram corrompidos durante o trajeto entre uma etapa e outra. Sendo assim, se uma etapa for comprometida, ela pode gerar produtos vulneráveis e estes servirão como entrada para a próxima etapa do *pipeline*.

A Tabela 2.3 apresenta a comparação dos trabalhos relacionados de acordo com as etapas do *pipeline* descritas na Seção 2.1. A marcação “A” significa que o trabalho discute ameaças para a etapa da coluna correspondente, e “C” indica que são discutidas contramedidas. É possível notar que nenhum dos trabalhos relacionados aborda tanto ameaças quanto contramedidas referentes a todas as etapas do *pipeline* de desenvolvimento, sendo este o principal diferencial do presente trabalho.

Tabela 2.3: Comparação dos trabalhos relacionados

Referência	Integração	IC	IaC	Implantação	Lançamento
Bass et al. (2015)	[C]	[A, C]	[A, C]	[C]	–
Paule (2018)	[A]	[A]	[A]	[A]	[A]
Shaw (2017)	[A]	[A]	–	[A]	[A]
Simpson (2010)	[C]	[C]	[C]	[C]	[C]
Torres-Arias (2020)	[A, C]	[A, C]	–	[A, C]	[A, C]
Wheeler, Reddy e Fong (2018)	[A, C]	[A, C]	–	[A, C]	[A]

Fonte: A autora (2021).

2.4 Considerações parciais

Neste capítulo foram apresentadas e detalhadas todas as etapas do *pipeline* de desenvolvimento de software (Integração, Integração contínua, Infraestrutura como código, Implantação e Lançamento), juntamente com exemplos de incidentes de segurança já relatados na literatura para cada fase mencionada. Em seguida foi descrito e exemplificado o processo de modelagem de ameaças, com ênfase na utilização do modelo STRIDE. Observa-se que o processo de modelagem não é trivial, ele requer a compreensão dos componentes, interações e dependências do sistema, implicando em um nível de abstração elevado. Nota-se também que o STRIDE é um método de modelagem de ameaças bem desenvolvido e de fácil uso, mas conforme a complexidade do sistema aumenta, o número de ameaças pode crescer rapidamente, consumindo muito tempo. O método STRIDE possui duas variantes: STRIDE-por-elemento e STRIDE-por-interação. O presente trabalho foca no primeiro deles, porque este torna mais fácil encontrar ameaças, dado que apresenta um conjunto de ameaças contra cada elemento do diagrama. A revisão de trabalhos relacionados mostrou a falta de uma visão mais abrangente de ameaças e contramedidas referentes a todas as etapas do *pipeline*.

Depois de apresentado o *pipeline* de desenvolvimento, como funciona o processo de modelagem de ameaças, e quais são os trabalhos relacionados, é possível aplicar esses conceitos para realizar a modelagem de ameaças, utilizando o modelo STRIDE, para o *pipeline* de desenvolvimento de software. O detalhamento desta modelagem é efetuado no Capítulo 3.

3 Modelagem de ameaças

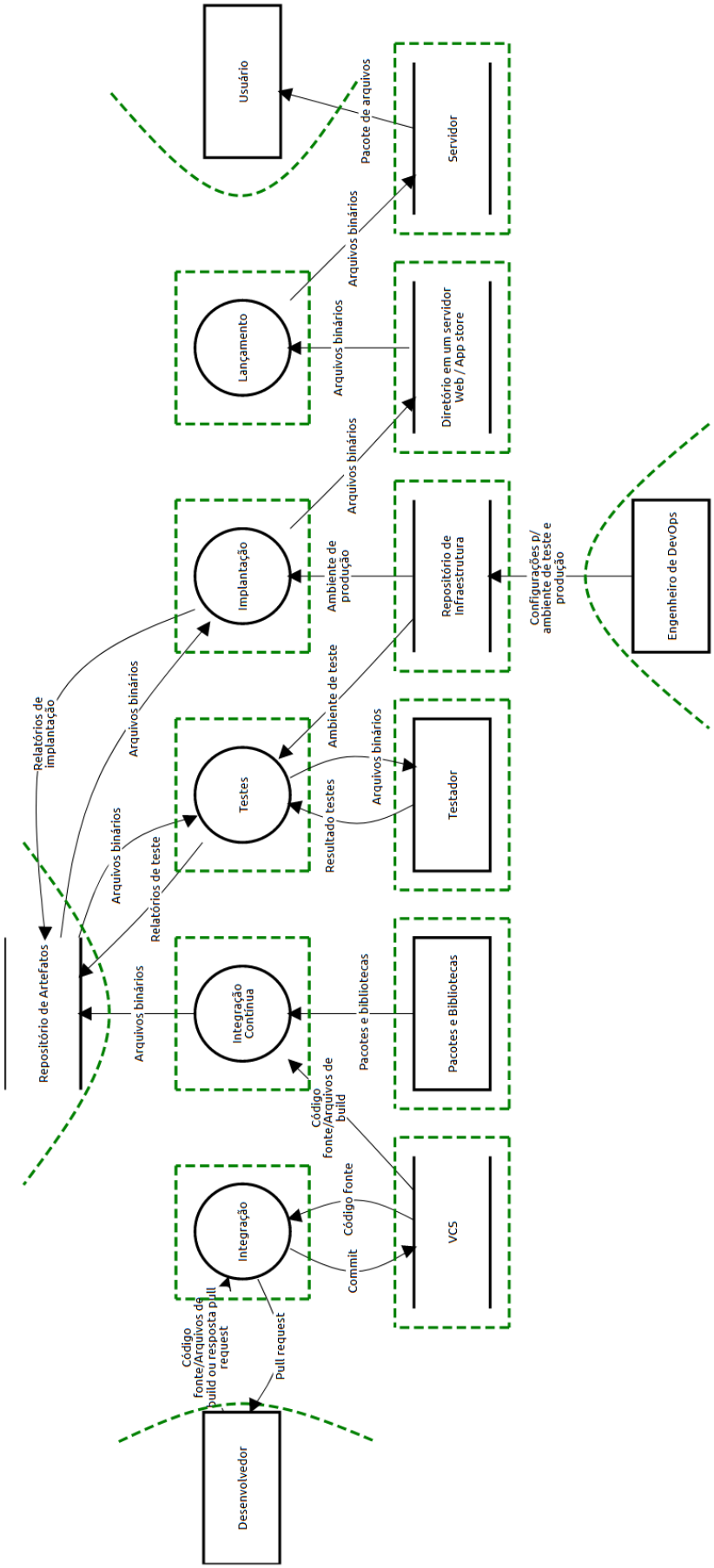
Esse capítulo apresenta a modelagem de ameaças para o *pipeline* de desenvolvimento de software descrito na Seção 2.1. Primeiramente, a Seção 3.1 mostra como o *pipeline* de desenvolvimento foi modelado usando um diagrama de fluxo de dados (DFD), que é a representação de sistemas adotada na abordagem STRIDE. Na Seção 3.2 é então apresentada a modelagem de ameaças, na qual é realizada uma discussão sobre possíveis mitigações para as ameaças encontradas, e são apresentados casos reais de incidentes de segurança. Por fim, na Seção 3.3 são apresentadas algumas considerações sobre o que foi apresentado nesse capítulo.

3.1 Modelagem do *pipeline* de desenvolvimento

Um dos objetivos deste trabalho é realizar uma modelagem de ameaças para o *pipeline* de desenvolvimento apresentado na Seção 2.1. Para que seja possível atingir este objetivo, foi desenvolvido um modelo de sistema para o *pipeline* mencionado, ou seja, foi produzido um Diagrama de Fluxo de Dados, o qual é apresentado na Figura 3.1.

No diagrama da Figura 3.1, observando da esquerda para a direita, é possível ver o Desenvolvedor acessando a primeira fase do *pipeline*, que é representada pelo processo de Integração. Aqui, o Desenvolvedor envia o código fonte e os arquivos de *build*, os quais são armazenados no repositório VCS e resgatados, sempre que necessário, pelo processo de Integração. Um Desenvolvedor usualmente classificado como sênior também recebe as solicitações de *pull request* e envia a aprovação ou a reprovação das mesmas ao processo de Integração. Finalizando esta fase, o código fonte e os arquivos de *build* são enviados para o processo de Integração Contínua. Esse processo pode incorporar pacotes e bibliotecas de uma entidade externa nomeada Pacotes e Bibliotecas, a qual está fora do controle da organização pois são pacotes e bibliotecas de terceiros que serão utilizadas como auxílio para o desenvolvimento do software. Encerrando a fase de Integração Contínua, os arquivos binários são enviados para o Repositório de Artefatos e podem ser usados pelos processos de Testes e Implantação. O processo de Testes envolve os testes automatizados. Já os testes manuais são realizados pela entidade externa Testador, que recebe como entrada os arquivos binários e

Figura 3.1: DFD para o *pipeline* de desenvolvimento



Fonte: A autora (2021).

retorna então o resultado dos testes manuais.

Neste diagrama também pode ser observada uma entidade externa nomeada Engenheiro de DevOps, que é responsável por enviar as configurações necessárias para a criação dos ambientes de teste e produção. Esses ambientes são armazenados no Repositório de Infraestrutura, do qual são recuperados pelos processos de Teste (ambiente de testes) e Implantação (ambiente de produção). Finalizando a fase de Implantação, os arquivos binários são enviados para o Diretório em um servidor Web ou para uma App store, a escolha fica por conta da equipe. Esses arquivos binários são então enviados para o processo de Lançamento, assim que este é finalizado os arquivos binários são enviados para o Servidor no qual o Usuário poderá fazer o *download* do software.

Para tornar a análise mais genérica, considera-se que cada elemento do diagrama está na sua própria fronteira de confiança (delimitada por uma linha tracejada). Quando há mais de um elemento dentro da mesma fronteira (por exemplo, dois processos que executam no mesmo sistema), algumas ameaças perdem o sentido, e torna-se desnecessário mitigá-las (DOTSON, 2019). Por exemplo, como os processos dentro de uma fronteira confiam um no outro, não é preciso levar em conta a ameaça de *spoofing* de processos. Assim, quando uma implementação de *pipeline* colocar múltiplos elementos dentro de uma mesma fronteira de confiança, algumas ameaças identificadas no modelo da Seção 3.2 (notadamente as que forem consequência de desconfiança mútua entre elementos) poderão ser desconsideradas.

3.2 Modelagem de ameaças do *pipeline* de desenvolvimento de software

Com o objetivo de desenvolver um modelo de ameaças para o *pipeline* de desenvolvimento apresentado na Seção 2.1, após a criação do diagrama de fluxo de dados demonstrado na Figura 3.1, foi usado STRIDE-por-elemento para encontrar ameaças no mesmo. Nas Seções 3.2.1 a 3.2.6 são apresentados os resultados da modelagem de ameaças e são discutidas mitigações para cada ameaça encontrada.

3.2.1 Entidades externas

A entidade externa Pacotes e Bibliotecas está fora do controle da organização, ela apenas fornece pacotes e bibliotecas de terceiros para auxiliarem no desenvolvimento do software. Dado que esta entidade está fora do controle da organização e é usada apenas para leitura, o importante é que o conteúdo importado pelo *pipeline* seja legítimo (essa ameaça é discutida na Seção 3.2.2). Neste caso, se uma entidade é ou não legítima (*i.e.*, uma possível ameaça de *spoofing*) é irrelevante, porque uma entidade ilegítima pode servir conteúdo legítimo, e uma entidade legítima comprometida pode servir conteúdo ilegítimo.

As ameaças encontradas para Desenvolvedor, Engenheiro de DevOps e Testador são:

- *Spoofing*: um atacante pode se passar por um Desenvolvedor, Engenheiro de DevOps, ou Testador e dessa forma pode inserir vulnerabilidades no código fonte, alterar os ambientes de teste e produção de modo a legitimar um software adulterado, ou modificar o resultado dos testes para aprovar um software vulnerável, respectivamente. Outra preocupação com relação à ameaça de falsificação da entidade externa Desenvolvedor é que um usuário sem privilégios de aprovação pode se passar por outro que possui esses privilégios e dessa forma pode aprovar uma solicitação de *pull request* de forma ilegítima. Esse caso não foi considerado como uma ameaça de elevação de privilégio, porque a aprovação de um *pull request* fica registrada no VCS. Então se torna mais viável para um atacante roubar as credenciais de um usuário que já possui os privilégios de aprovação, do que ele aprovar um *pull request* em seu nome, ou em nome de um Desenvolvedor sem privilégios, pois isso deixará claro nos registros do VCS que houve uma aprovação por um usuário indevido. Pode-se exemplificar essas ameaças com casos reais como quando os desenvolvedores de extensões do Google Chrome tiveram suas credenciais roubadas e os invasores puderam modificar as extensões, comprometendo milhões de usuários (MAUNDER, 2017). Outro exemplo é o caso no qual um atacante comprometeu a conta de um desenvolvedor e inseriu uma *backdoor* no pacote strong_password do Ruby Gems, possibilitando a execução remota do código em um ambiente infectado (COSTA, 2019). Por fim, o caso no qual houve a identificação de 11 pacotes infectados com *backdoor* no Ruby Gems (GOODIN, 2019). Em alguns casos os pacotes permitiam a mineração de criptomoedas e a execução remota dos códigos em servidores infectados. Foi possível concluir que um dos pacotes foi infectado devido ao comprometimento da conta de um desenvol-

vedor. É possível mitigar essa ameaça por meio da autenticação. Porém, recomenda-se usar Autenticação de dois fatores (2FA) ou *Multi Factor Authetication* (MFA) para contas importantes como de desenvolvedor (COSTA, 2019; GOODIN, 2019; WEINERT, 2019; MADDOX, 2018; NIST, 2019). Outra mitigação importante é a gestão das contas de usuários, que envolve processos para atribuir permissões de acesso a recursos segundo as necessidades de cada usuário, e conceder/revogar credenciais de acesso a recursos e sistemas (THEIS et al., 2019). Embora autenticação 2FA e/ou MFA possa ser uma solução adequada, é necessário atenção para a conveniência e segurança de uma implementação, visto que há casos reais de ataques envolvendo esse tipo de autenticação, por exemplo, quando a conta pessoal do CEO do Twitter, Jack Dorsey, foi comprometida no momento em que um hacker obteve acesso à autenticação de dois fatores baseada em SMS destinada a Dorsey (CHAWLA, 2019).

Outra ameaça é a falsificação de servidor, ou seja, a possibilidade de que o Desenvolvedor seja redirecionado para um processo de Integração ilegítimo que assume o lugar do verdadeiro (idem para o Engenheiro de DevOps e Repositório de Intraestrutura, e para o Testador e Testes). É possível mitigar essa ameaça com o uso de certificados TLS (*Transport Layer Security*) (RESCORLA, 2018) para autenticar o servidor (a Seção 3.2.5 traz considerações adicionais sobre o uso de TLS).

- *Repudiation*: o Desenvolvedor, o Engenheiro de DevOps e o Testador podem negar que enviaram dados para o sistema, e esses dados podem ser maliciosos, por exemplo, o Desenvolvedor pode enviar para o processo de Integração um código fonte que possui uma *backdoor* incorporada e dessa forma inserir vulnerabilidades no código. O Engenheiro de DevOps pode inserir um ambiente de teste que permite legitimar um software adulterado. Por fim, o Testador pode enviar resultados de testes adulterados que aprovam um software vulnerável. Essa ameaça pode ser mitigada armazenando dados de *log* (registrar quem fez as mudanças e quando essas mudanças foram feitas) e também por meio da assinatura digital dos *commits*.

Para as entidades Testador e Usuário foi encontrada a seguinte ameaça adicional:

- *Tampering*: neste caso específico foi encontrada uma ameaça de *tampering* na qual o Testador ou o Usuário não sabe se os dados que recebeu são confiáveis. Segundo The Linux Foundation (2020), os usuários muitas vezes não conseguem verificar se o software que

receberam é o software que desejavam, isto é, se o mesmo não é malicioso ou fraudulento. Um caso real, por exemplo, foi a inserção de código malicioso no software Microsoft Windows (versão não especificada) baixado via Tor (HERN, 2014). É possível garantir dados não corrompidos fazendo uso de técnicas como permissões e assinaturas digitais. As permissões garantem que apenas os principais (usuários ou processos) autorizados podem modificar o conteúdo do repositório, enquanto que as assinaturas digitais garantem que os binários foram criados ou certificados pelos principais autorizados. Porém, essas técnicas ajudam apenas na análise sintática, isto é, elas garantem que o software obtido pelo usuário é o mesmo que foi produzido por principais autorizados. No entanto, elas são insuficientes para a validação semântica, ou seja, não garantem que esse software seja correto. Isso pode ser mitigado usando ferramentas para detecção de vulnerabilidades no código fonte (KUPSCH et al., 2017; PISTOIA et al., 2007).

Para a entidade externa Usuário também considera-se a seguinte ameaça:

- *Spoofing*: o Usuário pode não ser legítimo. Por exemplo, se o software for pago, o Usuário pode fazer o *download* do mesmo sem realizar o pagamento. É possível mitigar essa ameaça por meio da autenticação, preferencialmente usando dois fatores ou multifatores.

3.2.2 Processos

Podem ser observados cinco processos no diagrama, são eles: Integração, Integração Contínua, Testes, Implantação e Lançamento. Esses processos estão sujeitos a algumas ameaças comuns que podem ser mitigadas da mesma forma (o que muda são as consequências, que dependem do processo):

- *Spoofing*: falsificação de servidor – os processos não sabem se estão se conectando com os repositórios legítimos. Para mitigar essa ameaça recomenda-se o uso de certificados TLS para autenticar o servidor.

- *Tampering*: os processos não sabem se os dados que estão recebendo, seja de entidades externas ou de depósitos de dados, são confiáveis. Como citado na Seção 3.2.1, é possível usar mecanismos que garantam que os dados foram gerados por principais autorizados e não foram modificados em trânsito, porém, elas ajudam apenas na análise sintática. Já

para a validação semântica recomenda-se ferramentas para detecção de vulnerabilidades no código fonte (KUPSCH et al., 2017; PISTOIA et al., 2007).

Outra ameaça encontrada é a falsificação local, ou seja, o processo recebe os dados corretos, mas grava dados falsos nos repositórios. É possível mitigar essa ameaça fazendo uso de técnicas de tolerância a intrusões (OBELHEIRO; BESSANI; LUNG, 2005), embora seja necessário avaliar a viabilidade dessa solução devido a seu custo.

- *Denial of Service*: se os repositórios falharem, ou seja, seus serviços ficarem indisponíveis, tem-se um problema de negação de serviço. Essa ameaça pode ser mitigada usando técnicas de alta disponibilidade (MARCUS; STERN, 2003; ATCHISON, 2016). Como a negação de serviço não impacta diretamente a integridade do software, que é o foco deste trabalho, a mitigação dessa classe de ameaças não é aprofundada.

Além dessas ameaças aplicáveis a todos os processos, foram encontradas ameaças específicas ao processo de Integração Contínua:

- *Tampering*: um atacante pode inserir uma *backdoor* na ferramenta de IC ou de compilação utilizada no desenvolvimento do software e assim inserir vulnerabilidades no mesmo. Há casos reais que exemplificam essa ameaça, como o ataque onde foi lançada uma versão falsificada da ferramenta Xcode, muitos desenvolvedores fizeram o *download* da mesma porque essa seria baixada mais rapidamente, de modo que quando os desenvolvedores instalaram o que pensavam ser a ferramenta segura da Apple, na verdade obtiveram uma versão adulterada que iria compilar o código malicioso juntamente com o código do aplicativo real (WHEELER; REDDY; FONG, 2018). Para evitar essa ameaça, é responsabilidade do desenvolvedor tomar os devidos cuidados ao fazer uso de ferramentas de terceiros. Compiladores adulterados podem ser mitigados usando diversidade, como na técnica de *Diverse Double Compiling* (DDC) (WHEELER, 2009; SKRIMSTAD, 2018). *Builds* verificáveis, um tema de pesquisa recente (TORRES-ARIAS, 2020), também podem servir como mitigação para esse problema.

  Outra ameaça encontrada é que o processo de Integração Contínua não sabe se os dados recebidos de Pacotes e Bibliotecas são confiáveis e não foram corrompidos. Se os pacotes e bibliotecas estiverem corrompidos, então podem ser inseridas vulnerabilidades no software. Pode ser um erro do desenvolvedor ou a entidade externa (Pacotes e Bibliotecas) pode estar comprometida. Há casos reais onde houve a criação, pelos atacantes,

de bibliotecas maliciosas com os nomes das bibliotecas internas da linguagem Python e os desenvolvedores baixaram as maliciosas (GOODIN, 2017). Outro exemplo é o caso no qual um invasor usou *typosquatting* para registrar pacotes com nomes semelhantes às bibliotecas originais, mas contendo erros de digitação em seus nomes, esse pacote continha a fonte do projeto legítimo da Mongoose, além de código malicioso extra (WHEELER; REDDY; FONG, 2018). A entidade externa Pacotes e Bibliotecas está fora do controle da organização, mas caso esta esteja corrompida, é possível minimizar os riscos, por exemplo, usando HTTPS para realizar a conexão com a entidade. Alguns fornecedores têm pacotes com assinaturas digitais, isso ajuda a dificultar que o atacante consiga acesso aos dados. Outro modo é não confiar em um único repositório, por exemplo, se a mesma versão de um pacote estiver disponível em vários repositórios diferentes, é possível comparar arquivos obtidos de vários repositórios para ver se são idênticos (se não forem, isso indica um possível comprometimento do repositório divergente).

Um caso adicional que pode ser levado em consideração é a possibilidade de haver dois arquivos distintos X e Y com o mesmo nome de pacote mas versões diferentes, que podem inclusive estar assinados. O pacote desejado é o arquivo X, porém, no momento do *build* é feito uso do arquivo Y. Um exemplo de caso real é o incidente de dependência com o pacote "*left-pad*", no qual um desenvolvedor removeu o pacote *left-pad* do npm causando uma série de problemas a outros desenvolvedores (GALLAGHER, 2016). Para resolver o problema, outro desenvolvedor substituiu o pacote excluído por um que era funcionalmente equivalente. Nesta substituição, um desenvolvedor mal intencionado poderia ter inserido algum código malicioso no pacote e prejudicado os softwares dependentes desse pacote. Esse incidente com o pacote *left-pad* também pode ser considerado um problema de versão. Outro exemplo de ameaça similar é a vulnerabilidade conhecida como *dependency confusion*, esta parte do princípio de que um arquivo pode conter uma mistura de dependências públicas e privadas, e se um aplicativo fizer uso de um pacote que existe tanto em um repositório de código aberto quanto em sua construção privada, o pacote público terá prioridade (BIRSAN, 2021). Esta ameaça permite que um atacante obtenha execução remota de código, podendo então inserir *backdoors* durante as compilações. Como mitigação, sugere-se que os projetos tenham um identificador de versão exclusivo para cada lançamento (The Linux Foundation, 2020).

3.2.3 Fluxos de dados

Os fluxos de dados do DFD estão suscetíveis a ameaças comuns, cujas consequências diferem ligeiramente em função da origem e do destino de cada fluxo:

- *Tampering*: um atacante pode alterar os dados durante a comunicação. Se o dado for o código fonte podem ser inseridas vulnerabilidades no mesmo. Em casos nos quais os dados que estão sendo manipulados são os arquivos binários estes podem ser modificados para chamar outras funções ou executar um conteúdo impróprio. Modificando o ambiente de teste é possível legitimar um software adulterado. Porém, independente da consequência, o problema pode ser mitigado usando criptografia TLS, que, conforme descrito na Seção 3.2.1, trata também a falsificação de servidores. Outros tipos de canais criptografados que ofereçam proteções compatíveis com as do TLS também podem ser usados.

- *Information Disclosure*: os dados que estão sendo manipulados podem ser visualizados por usuários não autorizados. Por exemplo, em empresas que desenvolvem software proprietário, se houver o vazamento do código fonte para pessoas não autorizadas, a empresa pode sofrer com a concorrência oferecendo um produto similar ou idêntico ao seu, podendo acarretar em uma grande perda monetária. Para mitigar essa ameaça podem ser usadas as mesmas propostas da ameaça anterior: TLS ou outros mecanismos para estabelecer canais cifrados de comunicação.

É válido ressaltar que as ameaças apresentadas nesta seção e a proposta de criptografia do canal de comunicação entre as partes envolvidas também são mencionadas nas considerações de segurança da informação da norma ISO/IEC 27002 (ABNT, 2013). Esta norma deixa claro que deve-se proteger as informações durante uma comunicação para prevenir a divulgação não autorizada das mesmas.

3.2.4 Depósitos de dados

No diagrama podem ser observados cinco depósitos de dados, são eles: VCS, Repositório de Artefatos, Repositório de Infraestrutura, Diretório em um servidor Web/App store e Servidor. Para cada um foram encontradas ameaças similares, porém, suas consequências são diferentes.

- *Tampering*: um atacante pode alterar os dados indevidamente. O repositório do VCS armazena o código fonte, e seu comprometimento pode levar à introdução de vulnerabilidades no fonte. Já no Repositório de Artefatos, a alteração nos relatórios de testes podem possibilitar legitimar um software adulterado, ou pode haver a alteração dos arquivos binários para chamar outras funções ou executar um conteúdo impróprio. Para mitigar essa ameaça recomenda-se gerenciar adequadamente as permissões e criptografar os dados antes de armazená-los. Porém, se a equipe de desenvolvimento optar por fazer uso de uma App store em vez de um Diretório em um servidor Web, a responsabilidade de evitar essa ameaça é inteiramente da App store (o mesmo se aplica as demais ameaças).

- *Information Disclosure*: os dados podem ser visualizados por usuários não autorizados, permitindo o roubo de código fonte ou binário (essa ameaça se aplica apenas a software proprietário). Aqui seguem as mesmas mitigações apresentadas para a ameaça de *tampering*.

- *Denial of Service*: repositórios podem sofrer ataques de negação de serviço, deixando seus serviços indisponíveis. Há exemplos de casos reais como o ataque distribuído de negação de serviço que o GitHub sofreu, deixando o site fora do ar por dez minutos (NEWMAN, 2018). É possível mitigar controlando o uso de recursos (SHOSTACK, 2014) e usando técnicas de alta disponibilidade (MARCUS; STERN, 2003; ATCHISON, 2016).

Segundo a norma ISO/IEC 27002 (ABNT, 2013), a informação é de suma importância para uma organização, devido a isso necessita ser protegida e armazenada adequadamente. A norma também destaca que repositórios seguros é um dos aspectos que devem ser considerados dentro de uma política de desenvolvimento seguro, reforçando a necessidade e a importância da identificação das ameaças e da implementação das mitigações apresentadas nesta seção.

3.2.5 Contornando limitações do TLS

O TLS é a solução padrão para proteção de fluxos de dados e para autenticação de servidores (SHOSTACK, 2014), e foi apontado como mitigação para diversas ameaças na modelagem proposta. Este é um protocolo de segurança que fornece uma comunicação segura entre cliente

e servidor. Fazendo uso de certificados digitais, o TLS fornece confidencialidade, integridade de dados e autenticação (RESCORLA, 2018).

O TLS tem problemas conhecidos que se dividem em dois grandes grupos: vulnerabilidades criptográficas e limitações do modelo de confiança (CLARK; OORSCHOT, 2013). Com relação às vulnerabilidades criptográficas, a melhor mitigação é usar a versão mais atual do protocolo, que no momento é o TLS 1.3 (RESCORLA, 2018), desabilitando as versões anteriores.

Conforme mencionado, o TLS também possui algumas limitações no seu modelo de confiança. Clientes TLS (como um navegador *web*) tipicamente possuem um conjunto de Autoridade Certificadoras (ACs) que são âncoras de confiança, ou seja, certificados emitidos por uma dessas ACs (ou por uma outra AC a quem uma das ACs confiáveis delegue o direito de emitir certificados) são aceitos como legítimos. Somado a isso, o modelo de confiança permite que qualquer AC emita um certificado para qualquer nome, independente da vontade ou consentimento do responsável por aquele nome. Com isso, diversos cenários podem levar à emissão de certificados forjados que serão aceitos como legítimos: o comprometimento de qualquer âncora de confiança (isto é, qualquer AC ou chave privada), a inclusão fraudulenta de uma AC ou chave na lista de âncoras de confiança de um cliente, ou algum ataque que convença uma AC genuína a emitir um certificado impróprio. Isso oportuniza ataques de MITM e falsificação de servidor, por exemplo, um atacante intercepta o fluxo de dados, no qual ele apresenta um certificado como se fosse o servidor original, fazendo com que a proteção criptográfica termine nesse interceptador, permitindo ao atacante substituir os dados. Outra possibilidade é explorar a dependência de DNS, na qual a conexão é feita pelo endereço IP, então se o atacante conseguir, de alguma forma, fazer com que o cliente vá para um endereço IP errado, manipulando o DNS, e apresentar o certificado falso, o cliente vai aceitar.

Existem diversas propostas para contornar as limitações do modelo de confiança (CLARK; OORSCHOT, 2013; DIAZ-SANCHEZ et al., 2019). As mais realistas são:

- Enxugar a lista de âncoras de confiança às ACs necessárias. Isso é mais factível quando o conjunto de servidores está sob o controle da mesma organização, mas pode ter problemas de continuidade em caso de revogação de chaves.
- Usar fixação (*pinning*), que é o processo de associar um servidor com um ou mais certificados ou chaves públicas esperados (OWASP, 2020b). Duas formas usuais de realizar fixação são carregando previamente no cliente o certificado ou a chave pública, ou então

confiando no primeiro certificado ou chave pública recebida (conhecido como *Trust on First Use* (TOFU)). Uma outra forma de *pinning* envolve o uso de *DNS-Based Authentication of Named Entities* (DANE), este tem como objetivo associar nomes de domínios a certificados utilizando DNSSEC. O DANE permite a inclusão, por parte dos proprietários de domínio, de informações sobre as credenciais de autenticação de seus serviços permanentes em seu DNS (DIAZ-SANCHEZ et al., 2019).

Portanto, quando o TLS for usado para mitigar ameaças, recomenda-se que seja adotado algum dos mecanismos citados, ou um equivalente, para minimizar as limitações do seu modelo de confiança.

3.2.6 Discussão

Um resumo do modelo de ameaças é fornecido na Tabela 3.1, onde a primeira coluna informa qual é o elemento do diagrama que será analisado. A segunda coluna indica qual é o tipo de ameaça (STRIDE) a qual o elemento está suscetível. Na terceira coluna observa-se a descrição dessa ameaça. Por fim, na quarta coluna são apontadas mitigações para a ameaça encontrada. No total foram encontradas 17 ameaças, a Tabela 3.2 oferece um resumo quantitativo das mesmas.

Vale ressaltar que há ameaças que foram mitigadas parcialmente, como é o caso da ameaça de recebimento de dados não confiáveis. Observa-se também mitigações que envolvem um alto custo, como é o caso do uso de técnicas de tolerância a intrusões. Foram ainda encontradas ameaças de negação de serviço, as quais não afetam diretamente na integridade do software, mas não deixam de ser importantes.

3.3 Considerações parciais

Este capítulo de modelagem foi descrito na seguinte ordem: modelagem do sistema e modelagem de ameaças. Essa sequência é necessária porque a modelagem de ameaças usando STRIDE é feita sobre um modelo do sistema. Com esse objetivo foi desenvolvido um diagrama de fluxo de dados, este foi apresentado e detalhado suas funcionalidades no presente capítulo. Para tornar a análise mais genérica optou-se por representar cada elemento do diagrama dentro da sua própria fronteira de confiança.

Tabela 3.1: Resumo das ameaças e mitigações encontradas para o DFD

Elemento do diagrama	Tipo de ameaça	Ameaça	Mitigação
Desenvolvedor, Engenheiro de DevOps e Testador	*Spoofing*	Falsificação do Desenvolvedor ou do Engenheiro de DevOps ou do Testador	Autenticação
		Falsificação de servidor	Certificados TLS
	Repudiation	Negar o envio de dados para o sistema	Armazenamento de dados de *log* e assinatura digital dos *commits*
Usuário	*Spoofing*	Usuário ilegítimo	Autenticação
Testador e Usuário	*Tampering*	Recebimento de dados não confiáveis	Permissão, assinatura digital e ferramentas para detecção de vulnerabilidades no código fonte
Integração, Integração Contínua, Testes, Implantação e Lançamento	*Spoofing*	Falsificação de servidor	Certificados TLS
	Tampering	Recebimento de dados não confiáveis	Permissão, assinatura digital e ferramentas para detecção de vulnerabilidades no código fonte
		Falsificação local	Técnicas de tolerância a intrusões
	Denial of Service	Serviços indisponíveis	Técnicas de alta disponibilidade
Integração Contínua	*Tampering*	Ferramentas corrompidas	Prudência do desenvolvedor, técnica DDC e *builds* verificáveis
		Recebimento de dados não confiáveis	Conexão HTTPS, assinatura digital e não confiar em apenas um repositório
		Pacotes com o mesmo nome	Identificador de versão exclusivo para cada lançamento
Fluxos de dados (todos)	*Tampering*	Alteração dos dados durante a comunicação	Criptografia TLS ou outros canais cifrados de comunicação
	Information Disclosure	Exposição dos dados aos usuários não autorizados	Criptografia TLS ou outros canais cifrados de comunicação
VCS, Repositório de Artefatos, Repositório de Infraestrutura, Diretório em um servidor Web/App store e Servidor	*Tampering*	Alteração indevida dos dados	Gerenciar as permissões e criptografar os dados antes de armazená-los
	Information Disclosure	Exposição dos dados aos usuários não autorizados	Gerenciar as permissões e criptografar os dados antes de armazená-los
	Denial of Service	Serviços indisponíveis	Controlar o uso de recursos e usar técnicas de alta disponibilidade

Fonte: A autora (2021).

Tabela 3.2: Resumo quantitativo das ameaças encontradas para o DFD do *pipeline* de desenvolvimento de software

	Entidades externas	Processos	Fluxos de dados	Depósitos de dados	Total
S	3	1	-	-	4
T	1	5	1	1	8
R	1	-	-	-	1
I	-	-	1	1	2
D	-	1	-	1	2
E	-	-	-	-	0
Total					17

Fonte: A autora (2021).

Em seguida foram encontradas possíveis ameaças a esse sistema (usando STRIDE-por-elemento), juntamente com casos reais de incidentes de segurança apresentados na literatura. Foram identificadas 17 ameaças no total, mas duas delas não impactam diretamente na integridade do software, como é o caso dos ataques de negação de serviço. Então, para completar a modelagem de ameaças, foram propostas mitigações para cada ameaça encontrada. Observa-se que para a ameaça de recebimento de dados não confiáveis foi possível mitigar apenas parcialmente, isto é, ainda haverá risco residual. Algumas mitigações também possuem um alto custo, como é o caso do uso de técnicas de tolerância a intrusões.

Após apresentado todo o processo realizado para a modelagem de ameaças do *pipeline* usado como referência, o Capítulo 4 traz a modelagem de ameaças para o *pipeline* de desenvolvimento da distribuição Fedora Linux.

4 Modelagem de ameaças para a distribuição Fedora Linux

Um dos objetivos deste trabalho é realizar a análise de ao menos um *pipeline* documentado publicamente. Para escolher o primeiro projeto pensou-se em um sistema operacional de código aberto, dado que são projetos que distribuem software a usuários finais e para os quais é possível encontrar documentação sobre o *pipeline*. Foram avaliadas as seguintes opções: OpenBSD[1], FreeBSD[2], Debian[3], Arch Linux[4], Ubuntu[5] e Fedora[6]. Dentre essas opções, a distribuição Fedora Linux foi a que apresentou a documentação mais ampla. Esta característica foi considerada importante pela autora, porque era necessário um entendimento amplo sobre o ciclo de desenvolvimento do software, para então realizar a modelagem de ameaças do mesmo.

Portanto, esse capítulo apresenta a modelagem de ameaças para o *pipeline* de desenvolvimento do Fedora. Primeiramente, a Seção 4.1 mostra uma visão geral do que é a distribuição Fedora Linux, e como esta é organizada. A Seção 4.2 descreve o *pipeline* de desenvolvimento, detalhando cada uma de suas etapas. A Seção 4.3 mostra como o *pipeline* foi modelado usando diagramas de fluxo de dados. Já na Seção 4.4 é apresentada a modelagem de ameaças, a qual faz uso do STRIDE, e discute possíveis mitigações para as ameaças encontradas. A Seção 4.5 traz uma discussão sobre o que foi abordado até então neste capítulo, e a Seção 4.6 apresenta as limitações do mesmo. Por fim, na Seção 4.7 são apresentadas algumas considerações sobre o que foi apresentado no presente capítulo.

4.1 Visão geral da distribuição Fedora Linux

Uma distribuição Linux é um sistema operacional pronto para uso, que geralmente inclui o *kernel* Linux, um instalador, interfaces com o usuário e um gerenciador de pacotes (SUSE, 2021). As distribuições Linux agregam software de diferentes origens, tipicamente de código

[1] https://www.openbsd.org/
[2] https://www.freebsd.org/
[3] https://www.debian.org/index.pt.html
[4] https://archlinux.org/
[5] https://ubuntu.com/
[6] https://getfedora.org/en/

aberto, embora algumas também incluam componentes de código fechado. O papel de uma distribuição é integrar essas diversas partes em um sistema coeso, simplificando a instalação do sistema operacional e o gerenciamento de seus componentes.

O Fedora Linux é uma distribuição Linux criada em 2003, e que contém apenas componentes de software livre (FEDORA PROJECT, 2019). O Fedora é organizado como uma coleção de pacotes. Cada pacote representa uma integração de um componente de software e sua adequação às diretrizes de empacotamento da distribuição (*Packaging Guidelines*). Essas adequações envolvem aspectos como a organização do sistema de arquivos, correções de *bugs* e problemas de segurança, e opções de compilação e instalação do software. No Fedora, cada pacote tem um ou mais mantenedores, que são as pessoas responsáveis por tarefas como garantir que o pacote é corretamente compilado e instalado, atualizar o pacote quando uma nova versão do componente original estiver disponível, corrigir *bugs* e problemas de segurança encontrados no componente original ou no empacotamento.

A distribuição Fedora Linux faz uso de pacotes RPMs[7], que normalmente contêm um arquivo de especificações (spec), arquivos de fontes *upstream* e patches (LACHMAN, 2019). Esses pacotes possuem campos como nome, descrição e outros metadados. Segundo Tunka (2014), cada pacote contém compilações (*builds*), e cada compilação é construída em um ambiente isolado denominado *build-root*, este é abordado em mais detalhes na Seção 4.2.2. É válido ressaltar que existem dois tipos de pacotes RPM: RPM fonte (*Source RPM*, SRPM) e RPM binário. A diferença entre ambos é o seu conteúdo. Segundo Miller et al. (2020), um SRPM contém o código fonte, patches, e um arquivo spec, este último descreve como construir um RPM binário a partir do código fonte. Já um RPM binário contém os binários construídos a partir de um RPM fonte. Outro ponto importante é que um arquivo spec pode definir mais de um pacote, e estes pacotes adicionais são chamados subpacotes. Na maioria dos casos, os subpacotes são criados com o objetivo de particionar o conteúdo de um determinado fonte *upstream*, por exemplo, dividir os componentes cliente e servidor de uma aplicação (alguns usuários podem querer instalar apenas um desses componentes) ou separar a documentação e os binários de um software (se a documentação ocupar muito espaço, usuários podem querer instalar apenas os binários para economizar disco).

Versões estáveis do Fedora Linux são lançadas aproximadamente a cada 6 meses, e cada versão é mantida (isto é, seus pacotes recebem atualizações) por aproximadamente 13 meses após o seu lançamento (FEDORA PROJECT, 2021d). O Fedora adota um esquema

[7] https://rpm-packaging-guide.github.io/#rpm-packages

com dois ramos de desenvolvimento, *Rawhide* e *Branched* (FEDORA PROJECT, 2021h). *Rawhide* é o ramo de desenvolvimento contínuo, isto é, contém o código mais recente e a última compilação de todos os pacotes do Fedora atualizados diariamente, este não garante estabilidade (FEDORA PROJECT, 2021j). Já o *Branched* é um ramo de desenvolvimento derivado do *Rawhide* para estabilização de pré-lançamento, ou seja, se tornará a próxima versão estável do Fedora (FEDORA PROJECT, 2021h). Este contém compilações de todos os pacotes que foram atualizados pelos mantenedores com o objetivo de estabilizar antes do lançamento e corrigir possíveis problemas (FEDORA PROJECT, 2021i).

Por fim, o objetivo desse ciclo de desenvolvimento do Fedora é entregar ao usuário final uma nova versão dessa distribuição, esta consiste em um conjunto de imagens, que podem ser usadas para executar e/ou instalar (dependendo da imagem) o sistema operacional. Para que isso seja possível, a etapa de composição, apresentada na Seção 4.2.2, realiza a união dos pacotes e gera as imagens necessárias. Atualmente são geradas cinco imagens diferentes (Workstation, Server, IoT, CoreOS e Silverblue), para variadas arquiteturas, incluindo x86-64 e ARM (FEDORA, 2021).

4.2 *Pipeline* de desenvolvimento do projeto Fedora

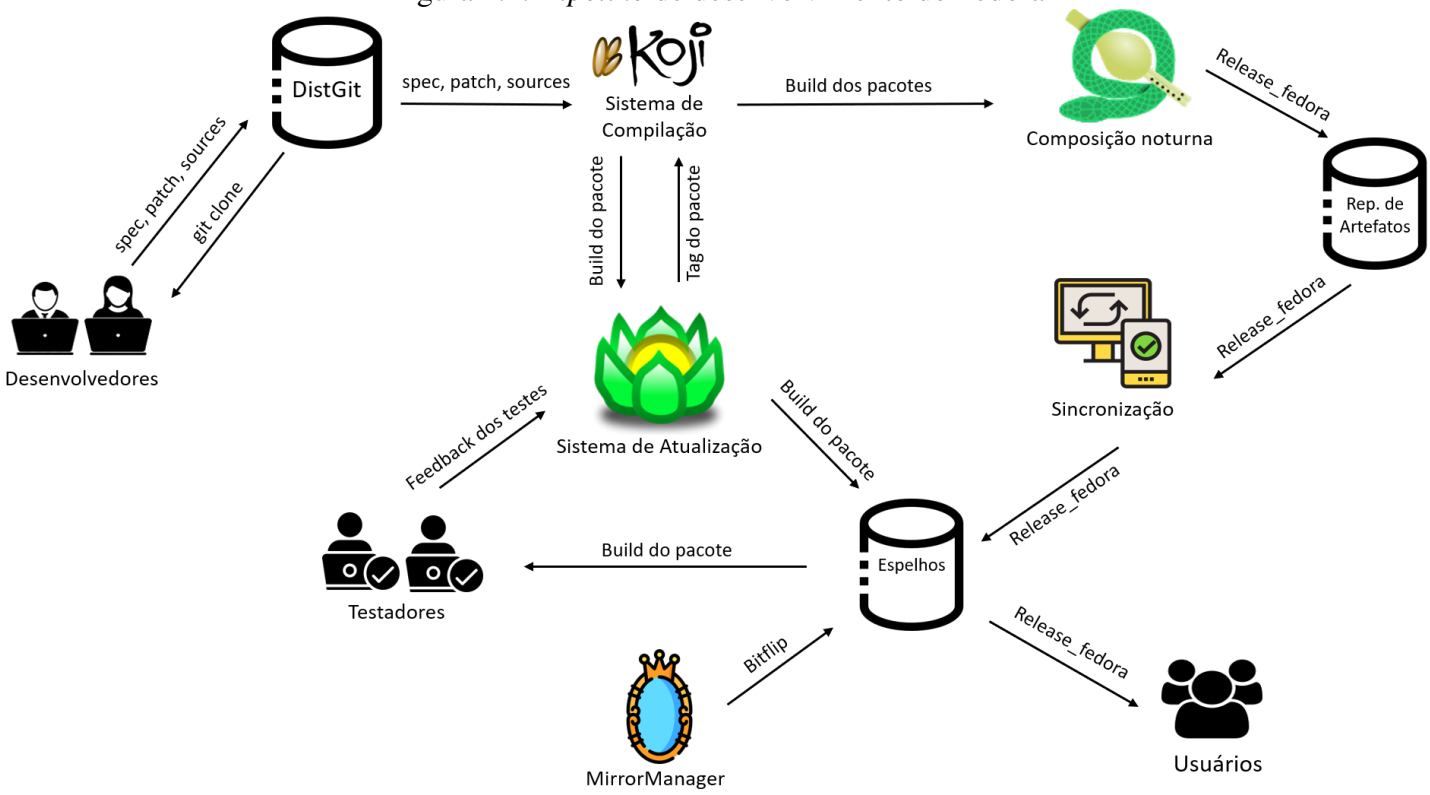
A Figura 4.1 apresenta o *pipeline* de desenvolvimento do projeto Fedora. Este envolve as fases de Integração, Integração Contínua, Implantação e Lançamento, as quais são apresentadas nas Seções 4.2.1 a 4.2.4.

4.2.1 Integração

De acordo com Lachman (2019), os passos que um mantenedor/desenvolvedor precisa fazer para propor uma nova versão *upstream* aos usuários do Fedora são:

1. Baixar o repositório dist-git;
2. Para cada *branch* do Fedora:
   (a) Atualizar o arquivo spec;
   (b) Atualizar os arquivos de patch;
   (c) Atualizar o arquivo de sources;

Figura 4.1: *Pipeline* de desenvolvimento do Fedora



Fonte: A autora (2021).

(d) Realizar o *commit* das mudanças, *i.e.*, enviar as alterações para o repositório dist-git;

(e) Propor uma compilação no Koji (Sistema de compilação do Fedora); e

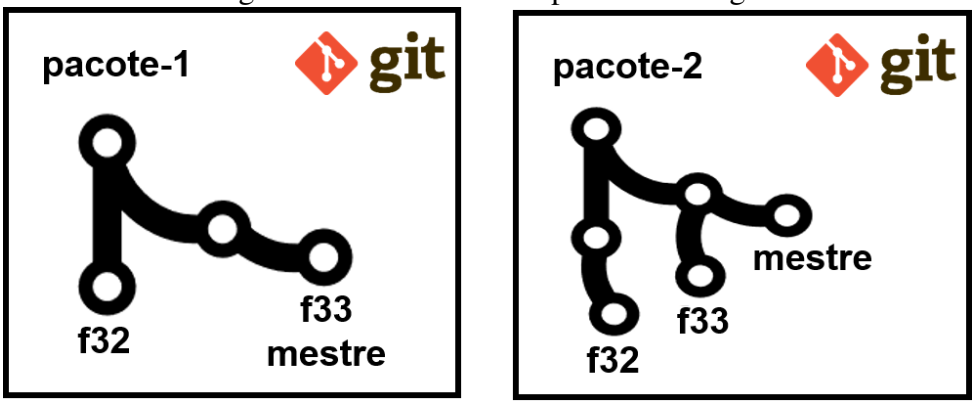
(f) Criar uma atualização no Bodhi (Sistema de atualização do Fedora).

Os dois últimos passos (e, f) são explicados em mais detalhes na Seção 4.2.2. A etapa de Integração envolve os cinco primeiros itens (1, 2-a, 2-b, 2-c, 2-d), ou seja, o desenvolvedor deve fazer o *download* do repositório dist-git para ter acesso às versões atuais dos arquivos spec, patch e sources. Esse *download* pode ser feito por meio da ferramenta fedpkg[8], ou diretamente via Git. Em seguida, o desenvolvedor deve fazer as devidas alterações nos arquivos desejados. Por fim, deve enviar essas modificações para o repositório dist-git.

O dist-git é um servidor Git que possui um repositório para cada pacote, e é usado como *package-backend* pela distribuição Fedora Linux (GITHUB, 2021b). Além de repositórios Git, esse servidor possui um *lookaside cache*, o qual é usado para armazenar arquivos fontes, e contém *scripts* para gerenciar os repositórios e a *cache*. Cada repositório dist-git possui um *branch* para cada lançamento e um *branch* mestre para *Rawhide* (LACHMAN, 2019). A estrutura desse repositório pode ser observada na Figura 4.2.

[8]https://pagure.io/fedpkg

Figura 4.2: Estrutura do repositório dist-git



Fonte: Adaptado de Lachman (2019).

Conforme observado na Seção 4.1, a distribuição Fedora Linux faz uso de pacotes RPMs, e estes possuem arquivos de fontes *upstream*, que são os fontes originais do software contido no pacote. O Fedora coloca sob controle de versão apenas os arquivos de metadados de pacotes RPM, mas não os fontes *upstream*. Para estes são usados os arquivos disponibilizados originalmente (*pristine sources*), que são tipicamente arquivos comprimidos no formato .tar.gz. Porém, como esses arquivos são binários eventualmente grandes, não é adequado armazená-los em um repositório Git (GITHUB, 2021b). Portanto, estes arquivos binários são armazenados no *lookaside cache*, o qual está presente no servidor dist-git. Devido a isso, é adicionado no pacote um arquivo sources, o qual contém o nome, o *hash* e o link do(s) fonte(s) *upstream(s)* que estão salvos no *lookaside cache* (LACHMAN, 2019).

O *lookaside cache* usa comunicação HTTPS, e todos os usuários do dist-git precisam (GITHUB, 2021b):

1. Ter acesso ao servidor SSH com autenticação de chave privada;
2. Estar em um grupo de “empacotadores” no servidor; e
3. Usar um certificado TLS para se autenticar no *lookaside cache*, esse certificado de cliente é emitido pelo provedor de serviços do dist-git e é complementar ao acesso SSH.

Segundo o que foi apresentado na Seção 4.1, o Fedora usa duas árvores de desenvolvimento: *Rawhide* e *Branched*. *Rawhide* é o *branch* mestre, isto é, o ponto de entrada para todo o desenvolvimento do Fedora, e é o tronco de onde todos os ramos divergem (FEDORA RELEASE ENGINEERING, 2016c). Vale ressaltar que nenhum lançamento é feito diretamente desta árvore, os lançamentos são ramificados do *Rawhide*. *Branched* é o nome dado a uma versão do

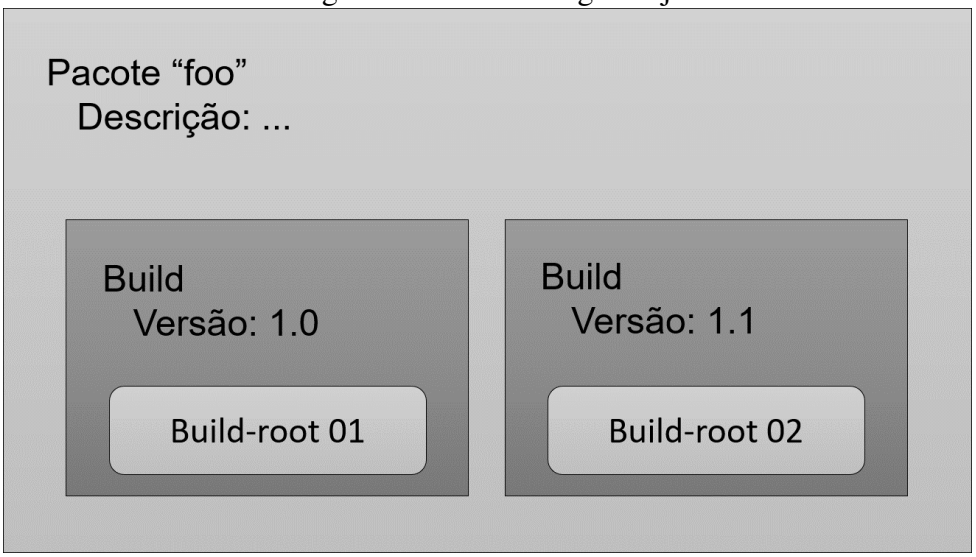
Fedora que se ramificou a partir do *Rawhide* e se tornará a próxima versão estável do Fedora (FEDORA PROJECT, 2021i).

4.2.2 Integração Contínua

Conforme mencionado na Seção 4.2.1, o penúltimo passo que o mantenedor/desenvolvedor precisa fazer para propor uma nova versão *upstream* aos usuários do Fedora é propor uma compilação no Koji, o sistema de compilação do Fedora (*Fedora's RPM buildsystem*). Este tem como objetivo fornecer compilações RPM a partir do conteúdo do repositório dist-git (LACHMAN, 2019).

O Koji trabalha principalmente com pacotes, e de acordo com o que foi apresentado na Seção 4.1, cada pacote contém compilações, e cada compilação é construída em um ambiente isolado denominado *build-root*. Este é configurado individualmente para cada compilação, e todas as ferramentas e dependências que forem necessárias são instaladas do zero. Isto é feito para garantir que não haverá problemas de dependências ao implantar o conteúdo em outras máquinas (TUNKA, 2014). A Figura 4.3 apresenta uma ideia geral dessa terminologia usada pelo Koji.

Figura 4.3: Terminologia Koji



Fonte: Adaptado de Tunka (2014).

De acordo com McLean et al. (2017b), no Koji às vezes é necessário distinguir entre um pacote em geral, uma compilação (*build*) de um pacote, e os arquivos RPMs criados por uma compilação:

- Pacote: é o nome de um SRPM, isto é, refere-se ao pacote em geral, por exemplo: kernel, glibc, entre outros.

- Compilação: é uma compilação de um pacote em particular, isto é, envolve todas as arquiteturas e subpacotes. Por exemplo: kernel-2.6.9-34.EL, glibc-2.3.4-2.19.

- RPM: é um RPM particular, isto é, refere-se a uma arquitetura e um subpacote específicos de uma compilação. Por exemplo: kernel-2.6.9-34.EL.x86_64, kernel-devel-2.6.9-34.EL.s390.

O Koji organiza os pacotes usando *tags*. Uma *tag* é similar a uma coleção de pacotes, esta suporta herança e cada *tag* tem sua própria lista de pacotes válidos (herdáveis) (MCLEAN et al., 2017b). Essas *tags* são de suma importância para o processo de composição, o qual é explicado em mais detalhes no final dessa seção.

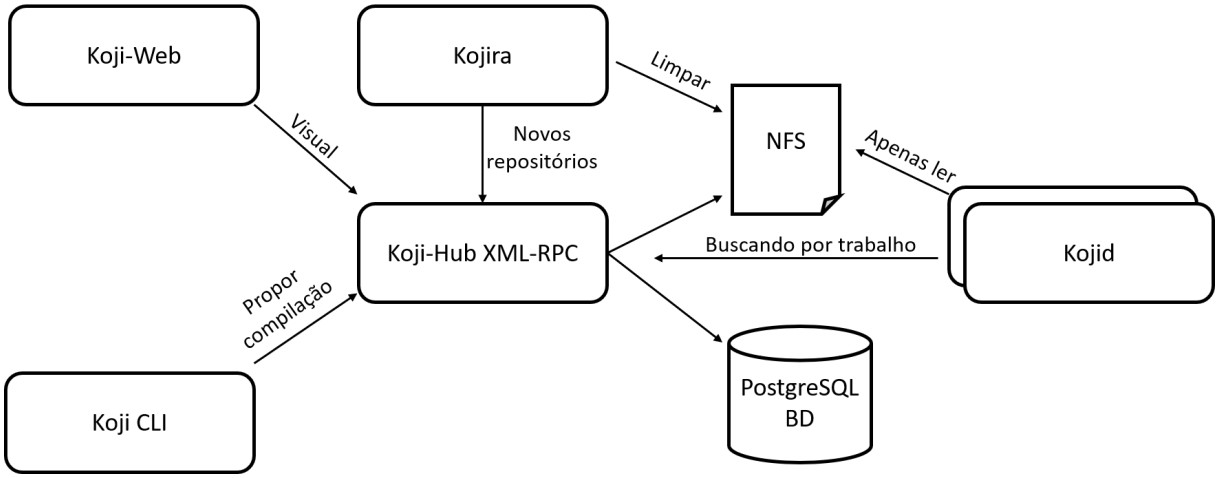
Koji tem uma arquitetura centralizada, isto é, há um servidor mestre que atua como um hub, este é responsável por distribuir a carga por vários *hosts* de compilação, controlar a comunicação com os demais componentes, e armazenar os artefatos resultantes (TUNKA, 2014). A Figura 4.4 apresenta a arquitetura do Koji, os seus componentes são detalhados a seguir:

- Koji-Hub: é um servidor XML-RPC executado no centro das operações do Koji. Este é considerado um componente passivo pois aguarda uma comunicação XML-RPC, a qual é iniciada por outros componentes do Koji (TUNKA, 2014). O Koji-Hub é o único componente a ter acesso direto ao banco de dados PostgreSQL, e tem permissão para gravar no sistema de arquivos (MCLEAN et al., 2017b).

- Kojid: é o *daemon* que está executando em cada uma das máquinas de compilação (MCLEAN et al., 2017b). Este componente pesquisa continuamente o Koji-Hub em busca de novas solicitações de compilação e, em seguida, determina que tipo de compilação deve ser executada (TUNKA, 2014). O Kojid também é responsável pela criação de imagens de instalação, e pela criação do *build-root* (MCLEAN et al., 2017b).

- Kojira: é o *daemon* responsável por manter as informações sobre os *build-roots* atualizadas (MCLEAN et al., 2017b), isto inclui solicitar a recompilação de novos repositórios quando necessário. Este componente também executa a coleta de lixo, removendo *build-roots* antigos e realizando a limpeza assim que a solicitação de compilação for completada (TUNKA, 2014).

Figura 4.4: Arquitetura Koji



Fonte: Adaptado de Tunka (2014).

- Koji-Web: o site do Koji exibe todas as informações sobre os pacotes, compilações, *build-roots*, artefatos, *tags* de compilação, entre outros (TUNKA, 2014). Não é permitido realizar operações como propor uma compilação, o Koji-Web é uma ferramenta que permite apenas a visualização de dados.

- Koji CLI: é um cliente de console simples que permite que os usuários tenham acesso a várias informações, e também executem ações como propor uma compilação (MCLEAN et al., 2017b). O Koji CLI usa autenticação Kerberos (FAS, *Fedora Account System*) (TUNKA, 2014).

- Sistema de arquivos NFS: Koji usa NFS para armazenar os *build-roots*, artefatos gerados pelas compilações, e dados de *log* (TUNKA, 2014).

- API XML-RPC: esta API é responsável pela comunicação entre os componentes do Koji. A API XML-RPC usa principalmente o método de autenticação simples que é composto por *login* e senha, porém, Tunka (2014) afirma que o *login* OpenSSL também é suportado.

Conforme mencionado na Seção 4.2.1, o último passo que o mantenedor/desenvolvedor precisa fazer para propor uma nova versão *upstream* aos usuários do Fedora é criar uma atualização no Bodhi, o sistema de atualização do Fedora (*Fedora Updates System*). Este é projetado para democratizar os testes das atualizações de pacote. O Bodhi fornece uma interface *web* na qual os desenvolvedores podem criar atualizações, e possui uma interface na qual os testadores podem deixar *feedback* para as atualizações (GITHUB, 2021a). Esse retorno dos testadores é feito por meio do sistema de karma +1/-1, o qual é abordado em mais detalhes ainda nesta seção.

O Bodhi tem o seguinte fluxo de trabalho (FEDORA PROJECT, 2021b):

1. Quando o empacotador/desenvolvedor propõe uma compilação no Koji, ele pode criar uma atualização no Bodhi.
2. O Bodhi então envia a versão recém construída para o repositório *updates-testing*[9].
3. Os usuários/testadores do Fedora podem então instalar essa versão atualizada que está no repositório *updates-testing*, e podem votar karma +1 ou -1 para essa atualização.
4. Se a atualização atingir o nível de karma desejado (geralmente 3), ela pode ser enviada para o repositório *updates*[10], sendo então declarada como estável.

Depois de enviadas as atualizações para o Bodhi, os pacotes podem passar pelos seguintes estados principais (FEDORA PROJECT, 2021c):

- Pendente: a atualização ainda não foi enviada para o repositório de testes.
- Teste: o pacote está no repositório *updates-testing* para as pessoas/testadores testarem. Neste caso é adicionada uma *tag* "-updates-testing" para este pacote no Koji.
- Estável: o pacote foi marcado como estável e foi lançado no repositório principal de *updates*. Neste caso é adicionada uma *tag* "-updates" para este pacote no Koji.
- Obsoleto: ao enviar uma nova versão de um pacote, o Bodhi tornará automaticamente obsoletas quaisquer atualizações pendentes, ou de teste, que não tenham uma solicitação ativa.

É possível enviar uma atualização para estável manualmente ou usando o karma. O karma é um sistema de votação +1/-1, o qual permite que os testadores forneçam *feedback* para uma atualização, isto é, informar se ela funciona ou não. A soma do karma de todos os comentários da atualização é o que determina o karma final da mesma (FEDORA PROJECT, 2021c). Geralmente, se a atualização receber o valor de karma igual a 3, a mesma será declarada estável automaticamente. Porém, se a atualização receber valor de karma igual a -3, então esta não será declarada estável. Esses valores de karma podem ser modificados caso necessário (FEDORA PROJECT, 2021c). Vale ressaltar que uma atualização pode ser configurada para

[9] https://dl.fedoraproject.org/pub/fedora/linux/updates/testing/
[10] https://dl.fedoraproject.org/pub/fedora/linux/updates/

ser enviada para estável automaticamente após atingir o nível de karma desejado, mas se a mesma receber qualquer karma negativo a opção de estabilização automática é desabilitada. Porém, o responsável pela atualização pode enviá-la manualmente para estável mesmo que esta tenha karma negativo. Outra possibilidade para enviar uma atualização para estável é baseada no número de dias que a mesma está no repositório *updates-testing*, conhecida como "estável por tempo" (*stable by time*) (FEDORA PROJECT, 2021k). Para *Branched* e *Rawhide*, quando uma atualização é tornada estável por meio do Bodhi, a *tag* "fXX" é aplicada a ela no Koji. "XX" é substituído pelo número da nova versão, por exemplo, "f35".

A ferramenta Fedora_Easy_Karma[11] tem como objetivo facilitar o processo de relatar o resultado dos testes (FEDORA PROJECT, 2021g). Esta permite o *feedback* para apenas um item: "A atualização é geralmente funcional?", este é o item mais significativo, porque apenas as respostas a este serão usadas para calcular o karma da atualização (FEDORA PROJECT, 2021f). Outra possibilidade para relatar um *feedback* é por meio da interface *web* do Bodhi (FEDORA PROJECT, 2021g). Tanto para usar a ferramenta como para usar a interface *web*, é necessário fazer o *login* usando uma conta do Fedora (FAS).

Uma "composição" é o mecanismo pelo qual a distribuição Fedora Linux obtém as compilações do Koji, com base nas *tags* dos pacotes, e gera os produtos que os usuários do Fedora irão "consumir", isto inclui imagens e repositórios voltados ao público. As composições se enquadram em duas categorias: elas podem ser candidatas à lançamento (*release candidates*) criadas sob demanda, também chamadas de composições candidatas (*candidate composes*), ou podem ser composições noturnas (*nightly composes*), as quais são definidas para serem executadas automaticamente todos os dias em um horário agendado (FEDORA RELEASE ENGINEERING, 2016a).

As composições candidatas podem incluir, de forma manual, pacotes que ainda não foram declarados estáveis. Essas composições não são automatizadas, elas são feitas manualmente pela equipe de QA e pela equipe de Engenharia de Lançamento (releng). A equipe de QA é responsável por solicitar uma composição candidata por meio de um tíquete do Pagure[12], este é um sistema de tíquetes criado pela equipe de infraestrutura para rastrear problemas, solicitações de recursos, e qualquer outra atividade relacionada à infraestrutura do Fedora (FEDORA PROJECT, 2017b). A equipe de releng é então responsável por acionar a composição, a qual é reali-

[11] https://fedoraproject.org/wiki/Fedora_Easy_Karma
[12] https://pagure.io/

zada por meio do *script* release-candidate.sh que se encontra no repositório pungi-fedora[13]. Essa composição é feita usando Pungi, uma ferramenta de composição de distribuição (*distribution compose tool*), a qual é apresentada em mais detalhes ainda nesta seção.

Assim que a composição é concluída, são executados os testes manuais e automatizados do openQA[14]. O openQA é o sistema de testes automatizados do Fedora (FEDORA PROJECT, 2020c). Essa distribuição Linux usa o openQA para testes de validação de lançamento, isto é, para verificar se a composição está de acordo com os critérios básicos de lançamento (*Basic Release Criteria*[15]) (FEDORA PROJECT, 2020c). Se a composição passar nos testes ela é então sincronizada com o repositório /pub/alt/stage/[16] nos espelhos, esse processo é feito manualmente pela equipe de Engenharia de Lançamento.

Já o processo de composição noturna (para *Rawhide* e *Branched*) usa os pacotes estáveis como entrada, isto é, pacotes marcados com a *tag* "fXX" para o lançamento em questão. Por exemplo, se em um determinado momento as composições do *Rawhide* usam todos os pacotes marcados como "f35", quando a versão F35 se ramifica, a composição para o *Branched* usará todos os pacotes marcados como "f35", e a composição do *Rawhide* usará todos os pacotes marcados como "f36".

A composição noturna é realizada por meio do *script* nightly.sh, o qual se encontra no repositório pungi-fedora. Essa composição é feita usando Pungi, este consiste em vários arquivos executáveis, onde o ponto de entrada é o *script* pungi-koji, o qual é responsável por carregar a configuração de composição e iniciar o processo (PAGURE, 2016a). O Pungi garante que todos os comandos são chamados na ordem correta, com os argumentos certos, e também move os artefatos para os locais adequados, porém quem faz o trabalho real é o Koji (PAGURE, 2016a). Ou seja, os arquivos executáveis enviam tarefas para o Koji, e este as executa de forma auditável.

A composição no Pungi é feita em fases, e cada etapa é responsável por gerar alguns artefatos em disco e atualizar o objeto de composição que passa por todas as fases (PAGURE, 2016a). Cada invocação do pungi-koji consiste em um conjunto de etapas, estas podem ser observadas na Figura 4.5. A maioria das fases é executada sequencialmente (da esquerda para a direita), mas há casos onde elas podem ser executadas em paralelo (PAGURE, 2016b). Esta seção aborda apenas a parte de Integração Contínua, ou seja, engloba apenas as etapas apresen-

[13] https://pagure.io/pungi-fedora
[14] https://openqa.fedoraproject.org/
[15] https://fedoraproject.org/wiki/Basic_Release_Criteria
[16] https://dl.fedoraproject.org/pub/alt/stage/

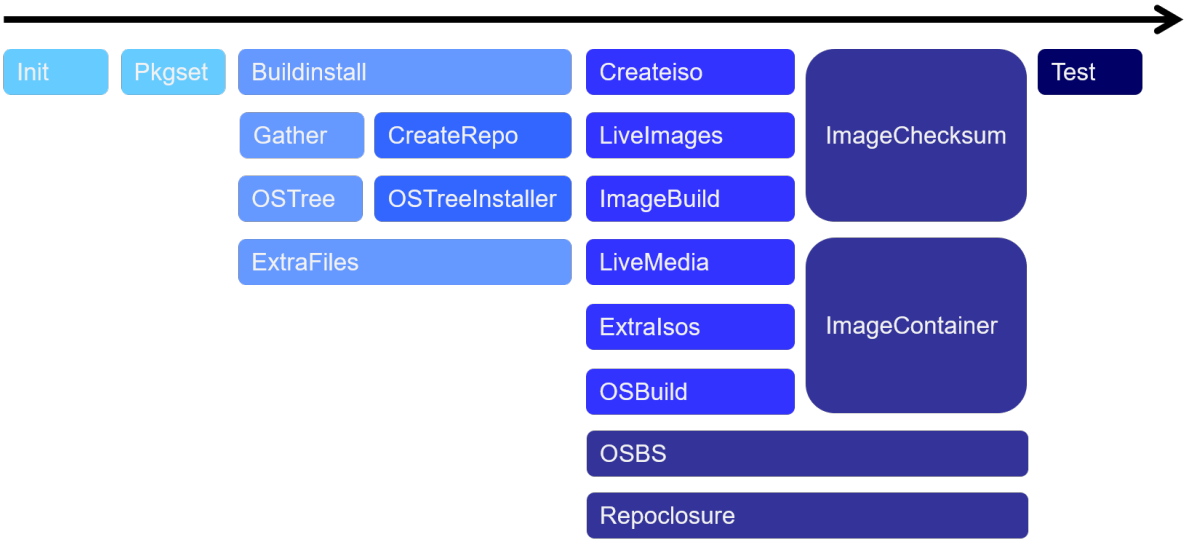
tadas na Figura 4.6. As fases correspondentes à geração das imagens são abordadas na Seção 4.2.3, a qual apresenta a etapa de Implantação do *pipeline*. Portanto, as fases abordadas nesta seção são (PAGURE, 2016b):

- Init: esta fase gera os arquivos comps para as variantes. comps é um arquivo XML usado por várias ferramentas do Fedora para realizar o agrupamento de pacotes em grupos funcionais (PAGURE, 2021).
- Pkgset: esta fase carrega um conjunto de pacotes que devem ser compostos, e tem como resultado uma estrutura de dados com mapeamento de pacotes para as arquiteturas.
- Buildinstall: nesta fase é criado o boot.iso e outros arquivos de configuração de inicialização.
- Gather: esta fase usa os dados coletados pela etapa Pkgset para saber quais pacotes devem estar em cada variante. Assim que o mapeamento é finalizado, os pacotes são vinculados aos locais apropriados e o manifesto rpms.json é criado. O arquivo rpms.json informa quais pacotes devem ser incluídos nos repositórios RPMs das arquiteturas.
- Createrepo: esta fase cria repositórios RPM para cada variante, e usa o manifesto rpms.json para descobrir quais pacotes devem ser incluídos.
- OSTree: esta fase atualiza o repositório ostree[17] por meio de um *commit* com pacotes da composição.
- OSTreeInstaller: esta fase cria mídia inicializável que carrega um repositório ostree como uma carga útil.
- ExtraFiles: esta fase coleta os arquivos extras de configuração e os copia para o diretório de composição.

O foco deste trabalho é apenas a composição noturna, devido ao processo de composição candidata ser manual, e possuir poucas diferenças de configuração/nomenclatura. Outro ponto importante são os testes automatizados. Se a composição noturna for bem sucedida, a mesma será sincronizada com o sistema de espelhos automaticamente (a Seção 4.2.3 traz mais detalhes sobre esse processo), ou seja, os resultados dos testes openQA não são levados em consideração pelo processo de sincronização, são meramente "consultivos". Porém,

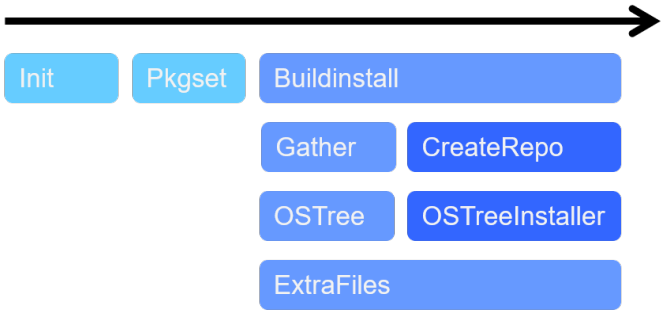
[17] https://fedoramagazine.org/what-is-silverblue/

Figura 4.5: Fases do pungi-koji



Fonte: Adaptado de PAGURE (2016b).

Figura 4.6: Fases do pungi-koji realizadas na etapa de Integração Contínua



Fonte: Adaptado de PAGURE (2016b).

esses testes são realizados e o *feedback* é enviado para as equipes de QA, Desenvolvimento e Engenharia de Lançamento, responsáveis por gerenciar de forma manual os possíveis problemas encontrados durante a execução dos testes automatizados. É responsabilidade do usuário verificar, por meio da interface do openQA, se a composição é instalável ou não (FEDORA PROJECT, 2021j). Conclui-se então que o processo de testes automatizados não interfere no *pipeline* de desenvolvimento, portanto, estes não são levados em consideração neste trabalho.

4.2.3 Implantação

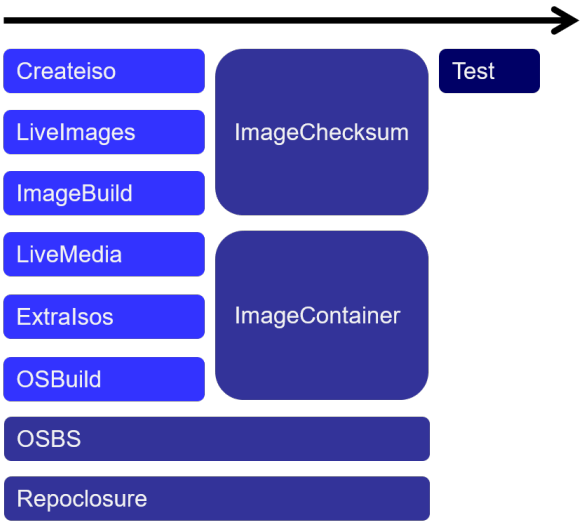
A etapa de Implantação consiste na continuação do processo de composição, isto é, a geração de imagens, e a sincronização da composição noturna com o sistema de espelhos. A Figura 4.7 apresenta as fases do pungi-koji que são executadas nesta etapa de Implantação, as quais são responsáveis pela geração das imagens e finalização da composição. Segue a introdução do que

cada uma dessas etapas executa (PAGURE, 2016b):

- Createiso: esta fase gera arquivos ISO e acumula metadados suficientes para criar o manifesto image.json, porém, este ainda não é criado nesta etapa. Os arquivos incluem um repositório com todos os RPMs da variante.
- LiveImages, LiveMedia: estas fases criam mídia no Koji, e quando esta é concluída as imagens são copiadas para o diretório da composição, e os metadados das imagens são atualizados.
- ImageBuild: esta fase termina a construção da imagem no Koji, e atualiza os metadados responsáveis pelo manifesto image.json.
- ExtraIsos: esta fase combina o conteúdo de diversas variantes em uma única imagem. Os pacotes e arquivos extras de cada variante configurada são colocados em um subdiretório.
- OSBuild: esta fase cria uma tarefa *osbuild* no Koji. Ele basicamente usa o OSBuild Composer[18] para criar imagens.
- OSBS: esta fase cria imagens de base de contêiner em OSBS[19]. As imagens finalizadas não são baixadas diretamente na composição, elas ficam registradas no OSBS. Os metadados da criação da imagem são registrados em osbs.json.
- Repoclosure: esta fase consiste em executar o comando repoclosure em cada repositório para verificar se há erros nos mesmos.
- ImageChecksum: esta fase é responsável por gerar *checksums* para as imagens.
- ImageContainer: esta fase cria uma imagem de contêiner no OSBS e armazena os metadados no mesmo arquivo da fase OSBS (osbs.json). A imagem criada aqui é uma VM para ambientes de contêineres.
- Test: esta fase executa algumas verificações simples na composição final. É verificado se as imagens estão corretas, isto é, se estão no tamanho adequado e se o formato corresponde a um arquivo ISO.

[18] https://www.osbuild.org/
[19] https://osbs.readthedocs.io/en/latest/index.html

Figura 4.7: Fases do pungi-koji realizadas na etapa de Implantação



Fonte: Adaptado de PAGURE (2016b).

Ao finalizar a composição noturna, a mesma é então enviada para o repositório de artefatos /mnt/koji/compose/rawhide/[20]. Se a composição for concluída com sucesso, isto é, nenhuma fase essencial falhar, a composição é sincronizada com o sistema de espelhos mestres, são eles: /pub/fedora/linux/development/[21], /pub/alt/development/[22], /pub/fedora-secondary/development/[23].

O Fedora possui alguns espelhos mestres, os quais possuem funções diferentes, são eles (FEDORA INFRASTRUCTURE TEAM, 2012; FEDORA PROJECT, 2020b):

- /pub/archive[24]: usado para arquivos de versões históricas do Fedora.
- /pub/fedora-secondary: usado para arquiteturas secundárias, por exemplo, s390x, ppc64le, aarch64.
- /pub/alt: usado para conteúdos diversos.
- /pub/epel[25]: usado para pacotes extras do Linux.
- /pub[26]: usado para versões oficiais do Fedora, amplamente espelhadas. Contém conteúdo pré-bitflip (a Seção 4.2.4 traz mais detalhes sobre o bitflip).

[20] https://kojipkgs.fedoraproject.org/compose/rawhide/
[21] https://dl.fedoraproject.org/pub/fedora/linux/development/
[22] https://dl.fedoraproject.org/pub/alt/development/
[23] https://dl.fedoraproject.org/pub/fedora-secondary/development/
[24] https://archive.fedoraproject.org/pub/archive/
[25] https://dl.fedoraproject.org/pub/epel/
[26] https://dl.fedoraproject.org/pub/

- /pub/fedora: contém todo o conteúdo atual do Fedora, incluindo conteúdo pré-bitflip.

Os servidores de espelho do Fedora usam um sistema de hierarquização, por meio do qual alguns espelhos rápidos selecionados obtêm acesso de leitura aos servidores mestres, e todos os outros espelhos são puxados desses espelhos (FEDORA PROJECT, 2021e). Em outras palavras, os espelhos mestres são os servidores de propriedade do Fedora, os quais foram apresentados acima. Os espelhos de nível 1 são rápidos e puxam o conteúdo de um espelho mestre, já os espelhos de nível 2 são os que puxam o conteúdo dos servidores de nível 1 (FEDORA PROJECT, 2021e). Segundo Domsch (2008), o Fedora também permite a criação de espelhos privados, estes puxam o conteúdo dos espelhos de nível 1 ou 2.

MirrorManager (MM) é uma ferramenta desenvolvida para gerenciar/atualizar os espelhos e facilitar a entrega do software Fedora aos usuários finais, isto é, esta ferramenta rastreia todos os espelhos e direciona automaticamente os usuários para um espelho local, rápido e atual (DOMSCH, 2008). O *MirrorManager* também auxilia os administradores de espelhos a atender de forma eficiente os seus usuários locais e globais. De acordo com Domsch (2008), os usuários também têm acesso a páginas *web* do *MirrorManager*, as quais permitem a visualização de listas de espelhos públicos que estão disponíveis, bem como as propriedades, país, largura de banda, entre outras informações sobre os mesmos.

4.2.4 Lançamento

Na etapa de Implantação, apresentada na Seção 4.2.3, o software final foi sincronizado com o sistema de espelhos, porém, esse conteúdo possui permissões restritas, isto é, não é legível para todos. A etapa de Lançamento é representada pelo bitflip, que nada mais é do que o ajuste das permissões no repositório para liberar acesso geral, o conteúdo já vai estar no repositório (FEDORA PROJECT, 2020b). Esse bitflip pode ser efetivado pelo *MirrorManager* mediante agendamento, e também por meio da atualização do espelho, esta vai apenas mudar as permissões, não será necessário baixar todo o conteúdo novamente (FEDORA PROJECT, 2020b). Após esse processo de ajuste de permissões, os usuários finais poderão ter acesso a nova versão do software da distribuição Fedora Linux.

Para uma composição candidata, o bitflip só pode ser efetuado se a nova versão for aprovada em uma reunião Go/No Go. Esta é realizada pelas equipes de QA, Desenvolvimento e Engenharia de Lançamento. Nesta reunião é discutido se os critérios de lançamento foram

atendidos, em caso afirmativo, e se o novo lançamento for unanimemente declarado GOLD, o mesmo pode ser disponibilizado para os usuários finais (FEDORA PROJECT, 2017a).

4.3 Modelagem do *pipeline* de desenvolvimento do Fedora

Um dos objetivos deste trabalho é realizar a modelagem de ameaças para o *pipeline* de desenvolvimento do Fedora. Para isso, primeiramente foi desenvolvido um modelo de sistema para o *pipeline* apresentado na Seção 4.2, isto é, foram criados três Diagramas de Fluxo de Dados, um para a etapa de Integração, outro para Integração Contínua e um terceiro representando as fases de Implantação e Lançamento. Estes DFDs são apresentados em mais detalhes nas Seções 4.3.1 a 4.3.3.

As fronteiras de confiança, dos três diagramas, são delimitadas de acordo com o contexto dos componentes. Por exemplo, os repositórios Git se encontram dentro da mesma fronteira porque são considerados como um único serviço. Vale ressaltar que cada fronteira é nomeada de acordo com o contexto, e suas particularidades são discutidas nas Seções 4.3.1 a 4.3.3. Separar as fronteiras de acordo com o contexto é apenas uma das abordagens possíveis, dado que no Fedora os componentes/produtos são protegidos mesmo estando dentro de uma fronteira de confiança, isto é, não há uma confiança total. Dentro da fronteira ainda há proteção de canais, mecanismos de assinatura, entre outras técnicas de segurança.

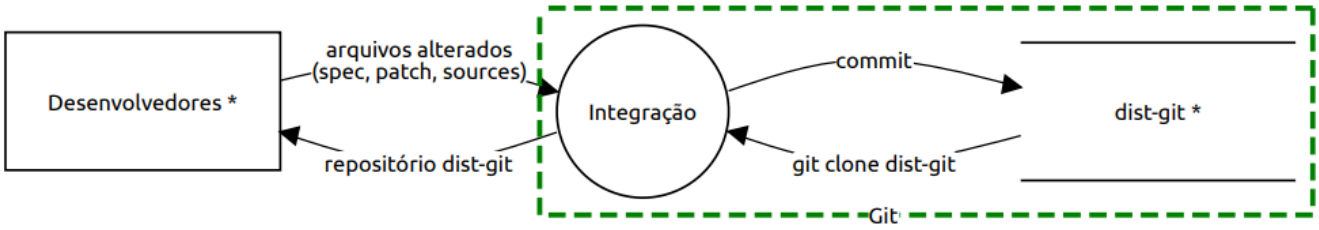
4.3.1 Diagrama de fluxo de dados da etapa de Integração

O diagrama da Figura 4.8 representa a etapa de Integração do *pipeline* de desenvolvimento do Fedora. Neste é possível observar, da esquerda para a direita, que os Desenvolvedores devem fazer o *download* do repositório dist-git para ter acesso aos arquivos atuais do mesmo. Em seguida eles podem alterar os arquivos spec, patch e sources desejados. Por fim, os Desenvolvedores devem realizar o *commit* das mudanças, isto é, enviar os novos arquivos para o repositório dist-git.

A fronteira de confiança (delimitada por uma linha tracejada) é determinada de acordo com o contexto. Neste diagrama observa-se o depósito dist-git, o qual é um servidor Git, devido a isso ele é considerado dentro da fronteira rotulada Git. De acordo com Vie (2000), não é recomendado representar uma relação direta entre uma entidade externa e um depósito

de dados. Portanto, foi adicionado um processo nomeado Integração para servir como uma "casca" intermediária entre a entidade externa Desenvolvedores e o depósito dist-git. Devido a isso, o processo de Integração é considerado dentro da mesma fronteira de confiança do depósito dist-git.

Figura 4.8: DFD para a etapa de Integração do *Pipeline* de desenvolvimento do Fedora



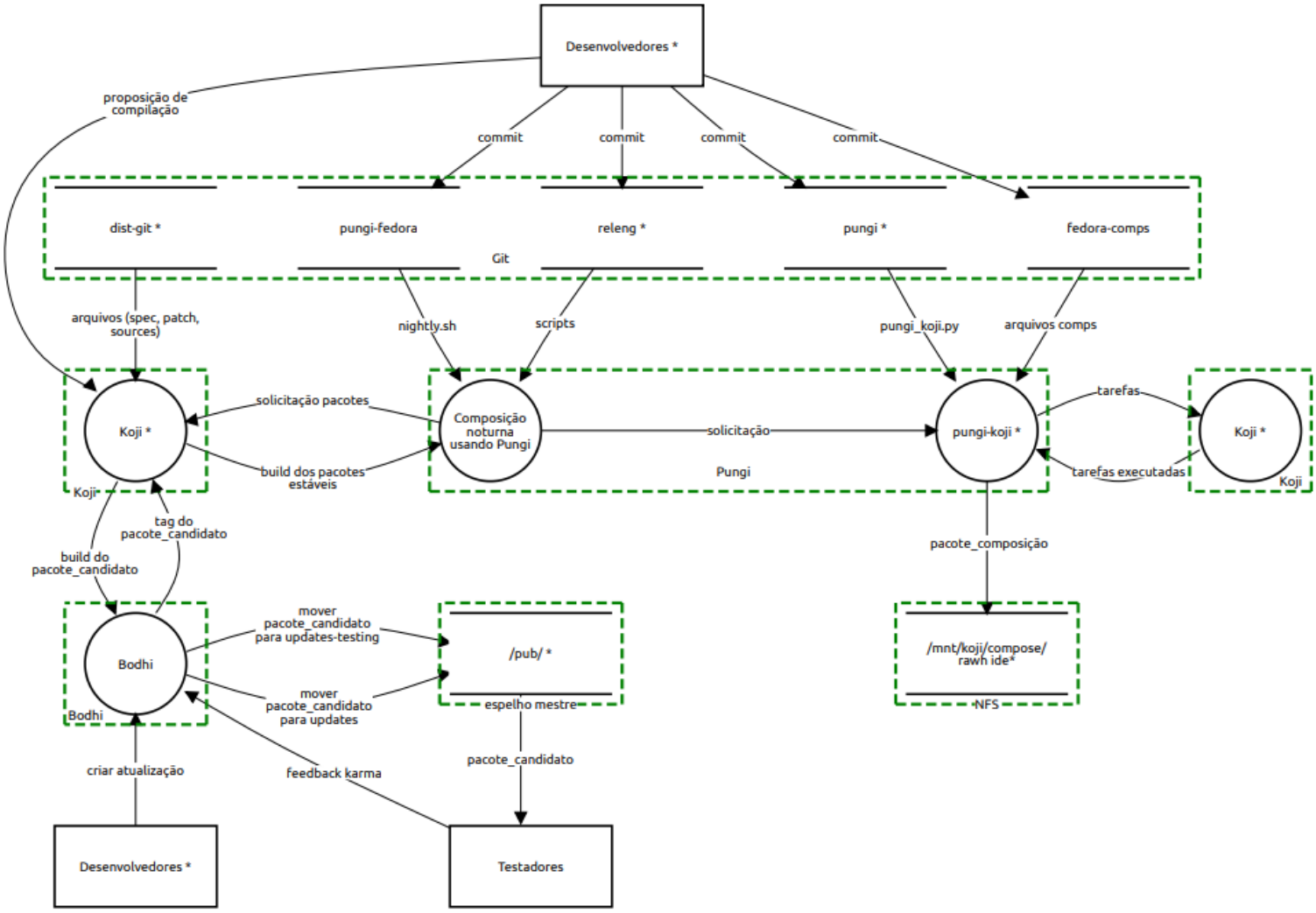
Fonte: A autora (2021).

4.3.2 Diagrama de fluxo de dados da etapa de Integração Contínua

O DFD da etapa de Integração Contínua é apresentado na Figura 4.9. Neste observa-se que os Desenvolvedores devem propor uma compilação no Koji. Este irá gerar compilações RPM (RPMs binários) a partir dos arquivos (spec, patch e sources) armazenados no depósito dist-git. Conforme apresentado na Seção 4.2.2, o Koji possui um sistema de armazenamento de arquivos interno, portanto as compilações geradas não são disponibilizadas em um repositório público, estas ficam armazenadas internamente no Koji. Dado o nível de abstração que está sendo considerado no presente trabalho, todos os componentes internos do Koji são representados por um único processo (Koji). Devido a isso, para os processos (Composição noturna usando Pungi, Bodhi e pungi-koji) que têm como entrada os produtos gerados pelo Koji, é representado um fluxo direto entre processos.

Após propor uma compilação, os Desenvolvedores podem criar uma atualização no Bodhi. Este é responsável por enviar a versão recém construída no Koji (nomeada pacote_candidato) para o repositório updates-testing, o qual é representado pelo depósito /pub/. Em seguida os Testadores podem fazer o *download* do pacote_candidato, e devem votar karma +1, se a atualização é funcional, ou -1 caso contrário. Ou seja, os Testadores devem enviar o seu *feedback* de karma para o Bodhi. Quando a atualização atingir o nível de karma desejado, ela é enviada para o repositório updates, o qual também é representado pelo depósito /pub/, porque está localizado neste espelho mestre do Fedora. Conforme os pacotes mudam de estado dentro do Bodhi, *tags* como: "-updates-testing", "-updates" e "fXX", são adicionadas aos respectivos pacotes no Koji.

Figura 4.9: DFD para a etapa de Integração Contínua do *Pipeline* de desenvolvimento do Fedora



Fonte: A autora (2021).

De acordo com o que foi apresentado na Seção 4.2.2, as composições noturnas são definidas para serem executadas automaticamente todos os dias em um horário agendado. Estas composições são representadas pelo processo Composição noturna usando Pungi. Este recebe como entrada os pacotes declarados estáveis no Koji, e o *script* nightly.sh, o qual se encontra no depósito pungi-fedora. Este nightly.sh é responsável pela chamada de outros *scripts* disponíveis no depósito releng, os quais são necessários para o processo de composição. O ponto principal do *script* nightly.sh é a chamada do pungi-koji, este é responsável por realizar a composição em fases. As fases que são realizadas na etapa de Integração Contínua foram apresentadas na Seção 4.2.2. Neste diagrama o pungi-koji é representado por um processo, o qual recebe como entrada o próprio *script* pungi_koji.py, que está armazenado no depósito pungi, e recebe os arquivos comps necessários para a configuração da composição. Conforme mencionado na Seção 4.2.2, o pungi-koji garante que todos os comandos serão chamados na ordem correta, mas quem faz o trabalho real é o Koji, este é invocado por meio de tarefas que são enviadas pelo pungi-koji. Os artefatos gerados até então (nomeados pacote_composição) pelas fases da composição relacionadas à etapa de Integração Contínua são armazenados no depósito /mnt/koji/compose/rawhide.

Observa-se neste diagrama alguns fluxos diretos entre uma entidade externa e um depósito. Embora normalmente tais fluxos sejam evitados, e tendo em vista que este modelo visa a representar o *pipeline* em nível de projeto (e não de implementação), o presente trabalho segue a ideia de que o depósito é acessado indiretamente por um processo, o qual é apenas uma "casca". Dado que a fase de Integração Contínua já é bastante complexa, a inserção de mais processos intermediários, apenas aumentaria essa complexidade e não agregaria informações importantes, portanto estes não são representados no diagrama.

Todos os depósitos Git (dist-git, pungi-fedora, releng, pungi e fedora-comps) podem ser acessados pelos Desenvolvedores, e estes podem realizar *commits* quando necessário. Estes depósitos são considerados um único serviço, portanto estão na mesma fronteira de confiança (nomeada Git). Os processos Composição noturna usando Pungi e pungi-koji estão na mesma máquina, devido a isso estão na mesma fronteira, a qual é rotulada Pungi. Os processos Koji e Bodhi possuem sua própria fronteira, porque estão em contextos diferentes dos demais componentes, assim como os depósitos /pub/ e /mnt/koji/compose/rawhide.

Os fluxos pungi-fedora → Composição noturna usando Pungi e pungi → pungi-koji são representados para mostrar o repositório de origem dos *scripts* usados na composição noturna. Portanto, estes fluxos não ocorrem sempre, ou seja, a máquina que exe-

cuta a composição terá esses *scripts* no seu sistema de arquivos local. Porém, analisando a documentação, não foi possível determinar os mecanismos de proteção e verificação da integridade dos *scripts* locais.

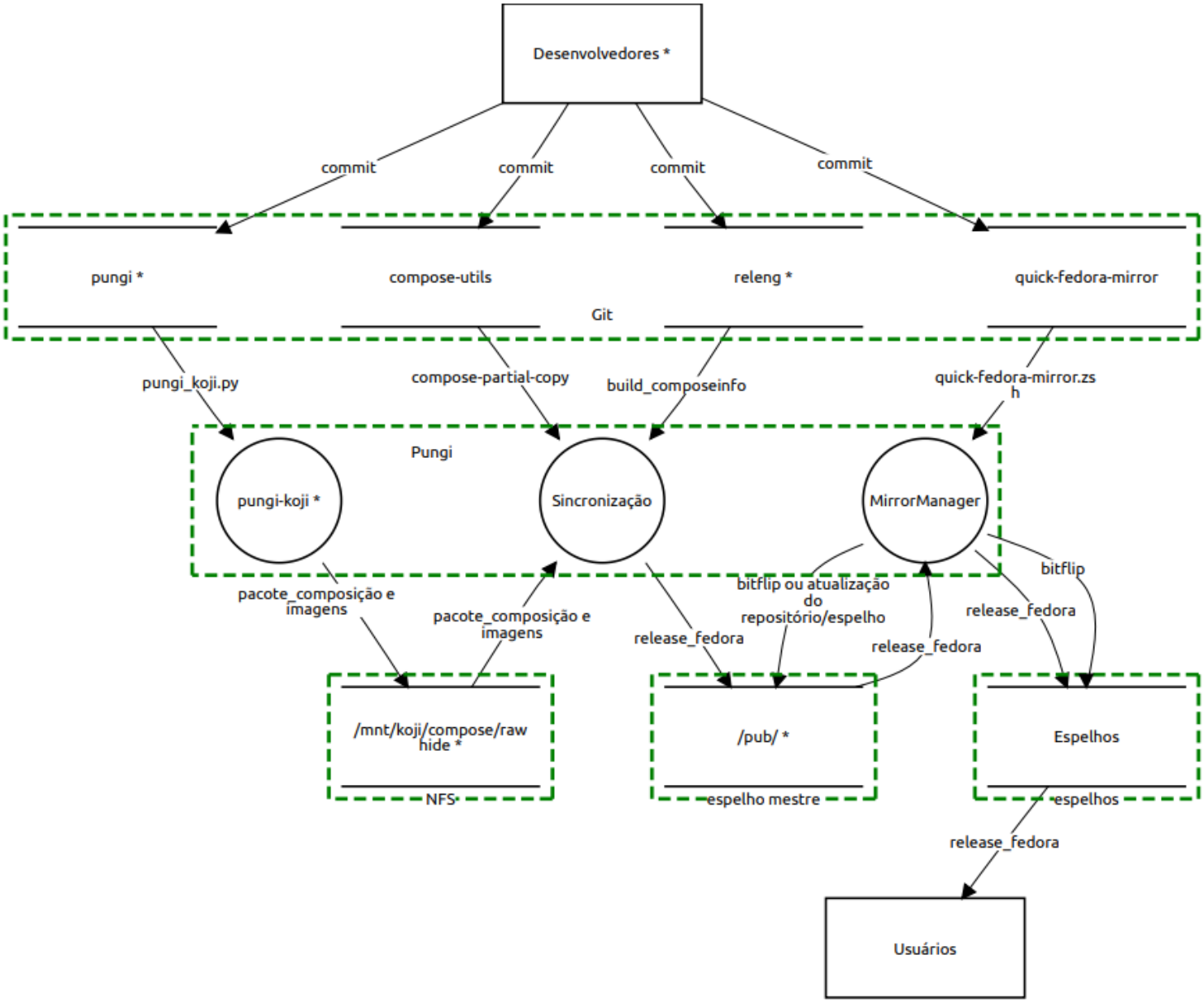
4.3.3 Diagrama de fluxo de dados das etapas de Implantação e Lançamento

O DFD das etapas de Implantação e Lançamento é apresentado na Figura 4.10. Conforme descrito na Seção 4.2.3, a etapa de Implantação consiste na continuação do processo de composição. Isto é, o processo pungi-koji gera as imagens e finaliza a composição. Os artefatos e as imagens geradas são armazenadas no depósito de dados /mnt/koji/compose/rawhide. Em seguida, o processo Sincronização é responsável por sincronizar a composição com os espelhos mestres do Fedora, representados pelo depósito /pub/. O processo de Sincronização faz uso de dois *scripts*: compose-partial-copy e build_composeinfo. O *script* compose-partial-copy é armazenado no depósito compose-utils, e é usado para copiar partes da composição para outro local via *rsync*, especificando quais variantes e/ou arquiteturas serão copiadas. Já o *script* build_composeinfo é armazenado no depósito releng, e é responsável por gerar arquivos contendo informações sobre a composição, isto é, quais variantes e/ou arquiteturas foram copiadas, bem como o local de destino de cada uma delas.

Conforme apresentado na Seção 4.2.3, os servidores de espelho do Fedora usam um sistema de hierarquização, e a ferramenta *MirrorManager* é responsável por realizar o espelhamento. Neste diagrama esta ferramenta é representada pelo processo MirrorManager, o qual tem como entrada a composição (nomeada release_fedora) que está armazenada nos espelhos mestres do Fedora. Uma de suas funções é realizar o espelhamento desse conteúdo para os demais espelhos de nível 1 e 2 (representados pelo depósito Espelhos). Para auxiliar nesse espelhamento é feito uso do *script* quick-fedora-mirror.zsh, o qual é armazenado no depósito quick-fedora-mirror. A outra função do processo MirrorManager é realizar o bitflip e/ou a atualização dos espelhos para ajustar as permissões, e dessa forma tornar disponível o conteúdo para todos os Usuários.

Assim como no diagrama apresentado na Seção 4.3.2, este DFD também possui fluxos diretos entre uma entidade externa e um depósito. Portanto, a justificativa dessa representação mantém-se a mesma. Isto é, o presente trabalho tem como princípio que o depósito é acessado indiretamente por um processo intermediário, o qual é apenas uma "casca",

Figura 4.10: DFD para as etapas de Implantação e Lançamento do *Pipeline* de desenvolvimento do Fedora



Fonte: A autora (2021).

por isso este não é representado no diagrama.

Todos os depósitos Git (compose-utils e quick-fedora-mirror) que não foram apresentados nos diagramas das Seções 4.3.1 e 4.3.2 estão na mesma fronteira de confiança dos demais (nomeada Git), isto é, são considerados um único serviço. Conforme apresentado na Seção 4.3.2, o processo pungi-koji está dentro da fronteira rotulada Pungi. Devido ao fato de que os processos pungi-koji, Sincronização e MirrorManager estão no mesmo contexto, isto é, um chama o outro e executam na mesma máquina, estes são considerados dentro da mesma fronteira de confiança (Pungi). Já os depósitos /mnt/koji/compose/rawhide, /pub/ e Espelhos possuem sua própria fronteira, dado que estão em contextos diferentes.

Os fluxos compose-utils → Sincronização e quick-fedora-mirror → MirrorManager são representados para mostrar o repositório de origem dos *scripts* usados na sincronização e espelhamento da composição. Portanto, estes fluxos não ocorrem sempre, isto é, a máquina que executa a sincronização e o espelhamento terá esses *scripts* no seu sistema de arquivos local. Porém, analisando a documentação, não foi possível determinar os mecanismos de proteção e verificação da integridade dos *scripts* locais.

4.4 Modelagem de ameaças do *pipeline* de desenvolvimento do Fedora

Com o objetivo de desenvolver um modelo de ameaças para o *pipeline* de desenvolvimento do projeto Fedora descrito na Seção 4.2, foi usado STRIDE-por-elemento para encontrar ameaças nos DFDs introduzidos na Seção 4.3. Nas Seções 4.4.1 a 4.4.4 são apresentados os resultados da modelagem de ameaças para os três diagramas propostos, e são discutidas mitigações para cada ameaça encontrada.

Vulnerabilidades de implementação nas ferramentas usadas no *pipeline* (Koji, Bodhi, Pungi e MirrorManager) estão fora do escopo deste trabalho. Assim, a modelagem pressupõe que essas ferramentas funcionam corretamente, e considera apenas as ameaças em nível de projeto. Devido a isso, os componentes do Koji, apresentados na Seção 4.2.2, e as fases do Pungi, descritas nas Seções 4.2.2 e 4.2.3, não são aprofundados.

4.4.1 Entidades externas

Nos diagramas apresentados na Seção 4.3 podem ser observadas três entidades externas, são elas: Desenvolvedores, Testadores e Usuários. As ameaças encontradas para as entidades Desenvolvedores e Testadores são:

- *Spoofing*: foram identificadas algumas ameaças de *spoofing*:

    1. Um atacante pode se passar por um Desenvolvedor e dessa forma inserir vulnerabilidades nos arquivos (spec, fontes *upstream*, e patches) dos pacotes. Se essas modificações não gerarem problemas perceptíveis a ponto de quebrar o *build* ou prejudicar o *feedback* de karma do pacote, há grandes chances deste ser declarado estável posteriormente.
    2. Um Desenvolvedor mal intencionado também pode alterar os *scripts* armazenados nos repositórios Git, os quais são usados para a composição noturna, sincronização e espelhamento, e desse modo permitir a composição de pacotes vulneráveis, ou a sincronização e espelhamento de pacotes ilegítimos.
    3. Um atacante se passando por um Desenvolvedor é capaz de propor a compilação de um pacote vulnerável no Koji, e pode criar uma atualização indevida no Bodhi.
    4. Um atacante também pode se passar por um Testador, e dessa forma é capaz de dar um *feedback* de karma positivo para uma atualização vulnerável. Se um atacante conseguir acesso a várias contas de Testadores, este pode dar *feedback* de karma positivo suficiente para declarar uma atualização vulnerável como estável.

    O projeto Fedora tenta mitigar essas ameaças por meio da autenticação, isto é, para propor uma compilação no Koji, criar uma atualização no Bodhi, e enviar um *feedback* de karma, é necessário estar logado em uma conta do Fedora (FAS). FAS suporta 2FA com *tokens* OTP. Por padrão, este sistema é responsável por gerar *tokens* de senha única baseada em tempo, também conhecido como *Time-based One-time Password* (TOTP), estes podem ser usados com aplicativos de autenticação como o FreeOTP[27] (FEDORA PROJECT, 2021l). Porém, adicionar autenticação de dois fatores é opcional, isto facilita o roubo de credenciais que usam apenas autenticação simples (*login* e senha). Devido a isso, e ao fato de que o projeto Fedora possui centenas de contribuidores espalhados

[27] https://freeotp.github.io/

por todo o mundo, outra mitigação importante e que foi apresentada na Seção 3.2.1, é a gestão de contas de usuários. Esta envolve a atribuição de permissões de acesso a recursos de acordo com a necessidade de cada contribuidor, e envolve conceder/revogar credenciais de acesso sempre que necessário. O Koji possui uma política de controle de acesso, a qual é definida em McLean et al. (2017a). Porém, para as demais ferramentas não foi possível encontrar uma documentação que definisse essas permissões. Para mitigar as ameaças relacionadas a inserção de vulnerabilidades em *scripts* armazenados nos repositórios Git, o projeto Fedora também faz uso de autenticação. Ou seja, para acessar o Git do projeto Fedora é necessário conexão SSH + chave. (FEDORA PROJECT, 2021a). Essa conexão SSH ocorre por meio de um *host* bastião, isto é, os usuários se conectam primeiramente ao bastião e, em seguida, há outra conexão SSH do bastião ao destino final (servidores Git do projeto Fedora). No caso do projeto Fedora, a chave é certificada (YLONEN, 2019). Porém, o SSH não tem PKI de certificação, ou seja, a chave SSH irá identificar o usuário/contribuidor para os servidores Git, e a chave de *host* do servidor Git vai permitir que a máquina do usuário/contribuidor verifique o servidor remoto. Como não há certificação envolvida, a chave garante a criptografia, mas não garante a autenticação do servidor remoto. As chaves de certificação estão disponíveis na URL https://admin.fedoraproject.org/ssh_known_hosts, cabendo ao Desenvolvedor verificar manualmente sua autenticidade. Portanto, essa ameaça de *spoofing* é mitigada parcialmente pelo projeto Fedora.

Outra ameaça é a falsificação de servidor, ou seja, a possibilidade de que os Desenvolvedores e os Testadores sejam redirecionados para um processo ou repositório ilegítimo. Conforme mencionado anteriormente, para um Desenvolvedor acessar o Git do projeto Fedora é necessário conexão SSH + chave. Porém, não há certificação neste processo, ou seja, o mesmo não garante a autenticação do servidor remoto. Já nos fluxos entre entidades (Desenvolvedores e Testadores) e processos (Integração, Koji e Bodhi) é feito uso de certificados TLS para autenticar o servidor. Portanto, essa ameaça é mitigada parcialmente pelo projeto Fedora.

- *Repudiation*: os Desenvolvedores e os Testadores não podem negar que enviaram dados para o sistema, porque para realizar qualquer ação dentro do *pipeline* é necessário estar conectado em uma conta do Fedora (FAS). Esse *login* ficará registrado no sistema juntamente com a ação realizada pelo usuário. Logo, essa ameaça é mitigada pelo projeto Fedora.

Para as entidades externas Testadores e Usuários foi encontrada a seguinte ameaça adicional:

- *Tampering*: os Testadores e Usuários não sabem se os dados que receberam são confiáveis. No início do desenvolvimento de um novo lançamento, uma nova chave GPG de assinatura de pacote é criada. Esta chave é usada para assinar todos os pacotes do lançamento final (FEDORA RELEASE ENGINEERING, 2016b). Sigul[28] é o servidor de assinatura que contém essas chaves (este servidor é considerado correto, isto é, suas vulnerabilidades estão fora do escopo do presente trabalho). É responsabilidade da equipe de Engenharia de Lançamento assinar todos os pacotes com a chave GPG do Fedora (FEDORA RELEASE ENGINEERING, 2016d). Estas assinaturas garantem que os pacotes que o usuário instala são produzidos pelo projeto Fedora e não foram alterados (acidentalmente ou maliciosamente) por nenhum espelho que esteja fornecendo os pacotes (FEDORA, 2020). Porém, é responsabilidade do Testador verificar se o pacote baixado foi assinado pelos principais (desenvolvedores ou processos) autorizados. Bem como é responsabilidade do Usuário verificar o *download* por meio de arquivos de *checksum* (FEDORA, 2020). Depois de baixar uma imagem do Fedora, o usuário deve importar as chaves GPG do Fedora. Em seguida deve verificar se o arquivo *checksum* é correto, este deve ter uma assinatura válida de uma das chaves importadas. Por fim, o usuário deve verificar se a soma de verificação do *download* está correta. Conforme apresentado no Capítulo 3, essa verificação ajuda apenas na análise sintática, isto é, ela garante que o software obtido pelos Usuários e Testadores é o mesmo que foi produzido pelos principais autorizados. No entanto, ela é insuficiente para a validação semântica, ou seja, não garante que o software esteja correto. Essa ameaça pode ser mitigada usando ferramentas para detecção de vulnerabilidades no código fonte (KUPSCH et al., 2017; PISTOIA et al., 2007). Portanto, essa ameaça de *tampering* é mitigada parcialmente pelo projeto Fedora.

Para a entidade externa Usuários também considera-se a seguinte ameaça apresentada no Capítulo 3:

- *Spoofing*: a ameaça de que um Usuário pode ser ilegítimo não se aplica neste caso, porque a distribuição Fedora Linux é um software livre e de código aberto. Portanto, um Usuário não precisa se autenticar para fazer o *download* do mesmo.

[28] https://pagure.io/sigul

4.4.2 Processos

Podem ser observados sete processos nos DFDs, são eles: Integração, Koji, Bodhi, Composição noturna usando Pungi, pungi-koji, Sincronização, e MirrorManager. Estes processos estão sujeitos a algumas ameaças em comum que podem ser mitigadas da mesma forma, o que muda são as consequências, que dependem do processo. Porém, as ameaças de *spoofing* e *tampering* não se aplicam à conexão entre o processo de Integração e o depósito dist-git, porque estes estão na mesma fronteira de confiança. O mesmo se tem para a conexão entre os processos Composição noturna usando Pungi e pungi-koji. As ameaças encontradas são:

- *Spoofing*: falsificação de servidor - os processos não sabem se estão se conectando com os repositórios legítimos ou, em casos onde há fluxo direto entre processos, estes não sabem se estão se conectando com os processos corretos. Nestas conexões é feito uso de certificados TLS para autenticar o servidor. Portanto, essa ameaça é mitigada pelo projeto Fedora.

- *Tampering*: os processos não sabem se os dados que estão recebendo, seja de entidades externas, depósitos de dados ou de outros processos, são confiáveis. Conforme mencionado na Seção 4.4.1, os pacotes são assinados com chaves GPG, e o conteúdo do Koji é assinado pela equipe de Engenharia de Lançamento (FEDORA RELEASE ENGINEERING, 2016b). Porém, a assinatura dos pacotes garante apenas que estes foram gerados por principais autorizados, isto é, realizam a análise sintática. Já para a validação semântica recomenda-se ferramentas para detecção de vulnerabilidades no código fonte (KUPSCH et al., 2017; PISTOIA et al., 2007). Logo, essa ameaça é mitigada parcialmente pelo projeto Fedora.

  Outra ameaça encontrada é a falsificação local. Porém, devido ao fato de que as vulnerabilidades específicas de cada ferramenta (Koji, Bodhi, Pungi, e MirrorManager) estão fora do escopo deste trabalho (acredita-se que estas estão funcionando corretamente), esta ameaça não é considerada.

- *Denial of Service*: se os repositórios falharem, isto é, seus serviços ficarem indisponíveis, haverá um problema de negação de serviço. No projeto Fedora, boa parte dos seus serviços são replicados e operam com balanceamento de carga, ou seja, são aplicadas técnicas de alta disponibilidade (FEDORA PROJECT, 2020a). Portanto, essa ameaça é

mitigada pelo projeto Fedora.

Além dessas ameaças aplicáveis a todos os processos, foi encontrada uma ameaça específica ao processo Bodhi:

- *Tampering*: conforme observado na Seção 4.2.2, é possível enviar automaticamente uma atualização para estável com base no número de dias que a mesma está no repositório *updates-testing*. Se a atualização não receber nenhum *feedback* de karma negativo, ela será enviada para estável assim que completar o tempo mínimo configurado. Então, se um atacante alterar/excluir os *feedbacks* negativos de uma determinada atualização vulnerável, a mesma será enviada para estável. Essa ameaça pode ser mitigada gerenciando adequadamente as permissões, e por meio do armazenamento de dados de *log*, isto é, registrar quem enviou o *feedback* de karma, quem alterou/excluiu o *feedback*, e quando essas ações foram feitas. Para enviar um *feedback* de karma ao Bodhi é necessário estar logado em uma conta FAS. O nome de usuário e o *feedback* ficam registrados no Bodhi. Porém, não foi possível encontrar uma documentação que definisse se há o gerenciamento de permissões ou o armazenamento dos dados de quem alterou/excluiu um *feedback* no Bodhi.

Para o processo Koji, além das ameaças comuns a todos os processos, foi encontrada a seguinte ameaça:

- *Tampering*: de acordo com o que é apresentado na Seção 4.2.2, o Koji organiza os pacotes usando *tags*. O processo Composição noturna usando Pungi recebe como entrada apenas os pacotes marcados com a *tag* estável. Então, se um atacante alterar as *tags* dos pacotes no Koji, será possível adicionar pacotes vulneráveis na composição. Conforme mencionado na Seção 4.4.1, o Koji possui uma política de controle de acesso, a qual permite o gerenciamento das *tags*. O Koji também armazena dados de *log*. Como mitigação adicional sugere-se a proteção criptográfica (assinatura) dessas *tags*.

O Capítulo 3 também apresenta algumas ameaças específicas para o processo de Integração Contínua. Estas não são aplicáveis para o projeto Fedora, porque a ferramenta de compilação (Koji) é suposta correta neste trabalho, sendo assim, não se considera a ameaça de inserção de vulnerabilidades no software por meio do comprometimento desta ferramenta. No *pipeline* do Fedora não são considerados pacotes e bibliotecas de terceiros, portanto, a ameaça

de recebimento desses dados comprometidos não é aplicável. Por fim, os pacotes do Fedora possuem nomes únicos. Logo, não existe a possibilidade de que no momento do *build* seja compilado um pacote duplicado malicioso.

4.4.3 Fluxos de dados

Os fluxos de dados dos diagramas estão suscetíveis a ameaças comuns, cujas consequências diferem ligeiramente em função da origem e do destino de cada fluxo. Porém, as ameaças encontradas não se aplicam aos fluxos entre o processo de Integração e o depósito dist-git, porque estes estão dentro da mesma fronteira de confiança. O mesmo se tem para o fluxo entre os processos Composição noturna usando Pungi e pungi-koji. As ameaças encontradas são:

- *Tampering*: um atacante pode alterar os dados durante a comunicação. Se o dado for os arquivos do pacote, o *build* de um pacote, ou a própria composição, podem ser inseridas vulnerabilidades nos mesmos. Se um pacote vulnerável for declarado estável (ameaça apresentada na Seção 4.4.1), um atacante pode modificar os arquivos comps e inserir este pacote ilegítimo na composição noturna. Em casos onde os dados que estão sendo manipulados são *scripts*, estes podem ser modificados de modo a gerar composições vulneráveis, ou realizar a sincronização e o espelhamento de composições ilegítimas. No fluxo que envolve o *feedback* de karma, se este for negativo, um atacante pode alterá-lo para positivo com o objetivo de tornar estável uma atualização vulnerável. As tarefas realizadas pelo Koji também podem ser alteradas de modo a inserir vulnerabilidades na composição. Porém, independente da consequência, esta ameaça é mitigada pelo projeto Fedora por meio do uso de conexão SSH + chave, e criptografia TLS. Os fluxos entre Desenvolvedores e repositórios Git (pungi-fedora, releng, pungi, fedora-comps, compose-utils e quick-fedora-mirror), e o fluxo entre Desenvolvedores e Integração fazem uso da conexão SSH + chave. Já os demais fluxos empregam criptografia TLS. Recomenda-se que os Usuários façam o *download* de espelhos que utilizam HTTPS, dado que nem todos os espelhos de nível 1 e 2 fazem uso do mesmo. Vale ressaltar que, conforme apresentado nas Seções 4.3.2 e 4.3.3, alguns fluxos não ocorrem sempre. Portanto, há uma preocupação menor quanto a eles.

- *Information Disclosure*: a ameaça de que os dados que estão sendo manipulados podem ser visualizados por usuários ilegítimos não se aplica neste caso, porque a distribuição Fedora Linux é um software livre e de código aberto.

4.4.4 Depósitos de dados

Nos DFDs podem ser observados dez depósitos de dados, são eles: dist-git, pungi-fedora, releng, pungi, fedora-comps, /pub/, /mnt/koji/compose/rawhide, compose-utils, quick-fedora-mirror, e Espelhos. Para cada um foram encontradas ameaças similares, porém, suas consequências são diferentes.

- *Tampering*: um atacante pode alterar os dados indevidamente. O repositório dist-git armazena os pacotes com os fontes *upstream*, o seu comprometimento pode levar à introdução de vulnerabilidades no código fonte dos pacotes. Já nos demais repositórios Git (pungi-fedora, releng, pungi, fedora-comps, compose-utils, quick-fedora-mirror) são armazenados os *scripts* e arquivos necessários para a composição noturna, sincronização e espelhamento. O comprometimento desses depósitos pode acarretar na geração de composições vulneráveis, e sincronização e espelhamento de composições ilegítimas. O repositório /mnt/koji/compose/rawhide é onde o Pungi armazena os artefatos gerados pela composição. Se este depósito for corrompido, o processo de Sincronização irá copiar dados ilegítimos para os espelhos mestres, e como consequência, os espelhos de nível 1 e 2 também terão arquivos vulneráveis. O mesmo se aplica aos depósitos /pub/ e Espelhos, ou seja, se /pub/ for comprometido, o depósito Espelhos também será prejudicado. Se este último estiver corrompido, os Usuários receberão uma versão ilegítima da distribuição Fedora Linux. Para mitigar essa ameaça recomenda-se gerenciar adequadamente as permissões e criptografar os dados antes de armazená-los. Para o controle de acesso aos repositórios Git, o Fedora faz uso do gitolite[29] (FEDORA PROJECT, 2018). Este verificará o nome do usuário em relação a uma ACL pré-gerada. Essa ACL é gerada usando os dados disponibilizados pelo mantenedor do pacote no Pagure, isto é, os mantenedores do pacote definem quem tem acesso de *commit*. Já a ferramenta *MirrorManager* é responsável pelo gerenciamento dos espelhos, ou seja, esta verifica constantemente se o conteúdo dos espelhos (nível 1 e 2) é o mesmo dos espelhos mestres (FEDORA INFRASTRUCTURE TEAM, 2017). Os espelhos de nível 1 e 2 também possuem ACLs gerenciadas pelos seus administradores (FEDORA PROJECT, 2021e), que são responsáveis por definir quem tem acesso de leitura e escrita nos espelhos. A recomendação é que o acesso a estes espelhos seja apenas de leitura (FEDORA PROJECT, 2020b). Presume-se que os espelhos mestres (/pub/) também te-

[29] https://github.com/sitaramc/gitolite

nham um mecanismo igual ou semelhante para gerenciar as permissões de acesso, mas não foi possível encontrar uma documentação que tratasse desse assunto. Para os demais depósitos também não foi possível encontrar uma documentação descrevendo como suas permissões são gerenciadas.

- *Information Disclosure*: a ameaça de que os dados podem ser visualizados por usuários ilegítimos não se aplica neste caso, porque a distribuição Fedora Linux é um software livre e de código aberto.

- *Denial of Service*: os repositórios podem sofrer ataques de negação de serviço, deixando seus serviços indisponíveis. Conforme mencionado na Seção 4.4.2, boa parte dos serviços são replicados e operam com balanceamento de carga, isto é, são aplicadas técnicas de alta disponibilidade. Portanto, essa ameaça é mitigada pelo projeto Fedora.

4.5 Discussão

O Fedora Linux é uma distribuição Linux que possui apenas componentes de software livre, e é organizado como uma coleção de pacotes. O projeto Fedora possui uma documentação ampla, devido a isso, esta distribuição foi escolhida para a modelagem. Porém, durante o desenvolvimento do presente trabalho, notou-se que partes da documentação estão confusas, desorganizadas, e/ou desatualizadas. Isso dificultou o entendimento do ciclo de desenvolvimento.

Neste capítulo foi apresentado o *pipeline* de desenvolvimento do projeto Fedora, no qual observou-se que há quatro fases: Integração, Integração Contínua, Implantação, e Lançamento. Em comparação com as cinco etapas apresentadas no *pipeline* típico do Capítulo 2, o *pipeline* do projeto Fedora não possui a fase de Infraestrutura como Código. Isso se tem porque o Fedora objetiva gerar uma distribuição instalável que será disponibilizada aos usuários finais, mas não instalar essa distribuição em sua própria infraestrutura.

Na etapa de Integração Contínua, apresentada na Seção 4.2.2, observou-se o processo de composição, o qual é de suma importância para o projeto Fedora. As composições podem ser categorizadas como: composições candidatas ou composições noturnas. O presente trabalho tem como objetivo apresentar apenas a composição noturna, porque esta é feita automaticamente, e é executada todos os dias em um determinado horário agendado. Outro ponto importante é que os testes automatizados são meramente informativos para as composições noturnas. Os *feedbacks* desses testes são tratados de forma manual pelas equipes de QA, De-

senvolvimento e Engenharia de Lançamento. Ou seja, os testes automatizados não interferem no *pipeline* de desenvolvimento. Logo, não são levados em consideração neste trabalho.

Após a apresentação do *pipeline* de desenvolvimento do Fedora, foi realizada a modelagem do mesmo. Com o objetivo de tornar a visualização do DFD mais simples e legível, nesta modelagem foi optado por desenvolver três diagramas, um para a etapa de Integração, outro para Integração Contínua, e um terceiro representando as fases de Implantação e Lançamento. Observou-se nesses diagramas alguns fluxos diretos entre uma entidade externa e um depósito de dados. Embora não seja recomendado fazer essa representação (VIE, 2000), e considerando que este modelo visa a representar o *pipeline* em nível de projeto, o presente trabalho sugere que o depósito é acessado indiretamente por um processo, o qual é apenas uma "casca". Dado que as fases de Integração Contínua e Implantação já são complexas, optou-se por essa representação para não aumentar essa complexidade. Outra peculiaridade desta modelagem é que há fluxos diretos entre processos, isso se teve porque o Koji possui um sistema de armazenamento de arquivos interno, portanto as compilações geradas por este processo ficam armazenadas internamente no Koji. Então, dado o nível de abstração do presente trabalho, todos os componentes internos do Koji são representados por um único processo.

Em seguida foi realizada a modelagem de ameaças para o *pipeline* do projeto Fedora. Um resumo do modelo de ameaças é fornecido na Tabela 4.1, onde a primeira coluna informa qual é o elemento do diagrama que será analisado. A segunda coluna indica qual é o tipo de ameaça (STRIDE) a qual o elemento está suscetível. Na terceira coluna observa-se a descrição dessa ameaça. Por fim, na quarta coluna são apontadas as mitigações adotadas pelo Fedora para a ameaça encontrada. No total foram encontradas 12 ameaças, a Tabela 4.2 oferece um resumo quantitativo das mesmas.

Conforme mencionado, a modelagem de ameaças do *pipeline* da distribuição Fedora Linux resultou em 12 ameaças no total, duas delas não foram mencionadas no modelo do *pipeline* de referência porque são específicas do ciclo de desenvolvimento do projeto Fedora. Das 17 ameaças encontradas na Seção 3.2, sete não se aplicam ao *pipeline* da distribuição Fedora Linux. Entre as 12 ameaças encontradas, seis delas são mitigadas totalmente pelo projeto Fedora, e as outras seis são mitigadas parcialmente, isto é, há uma lacuna nas mitigações empregadas pelas equipes do Fedora e/ou na documentação do projeto.

Na modelagem de ameaças para o *pipeline* de desenvolvimento do Fedora, observou-se que as vulnerabilidades específicas de cada ferramenta (Koji, Bodhi, Pungi, e Mir-

Tabela 4.1: Resumo das ameaças e mitigações encontradas para os DFDs do projeto Fedora

Elemento do diagrama	**Tipo de ameaça**	**Ameaça**	**Mitigação**
Desenvolvedores e Testadores	*Spoofing*	Falsificação do Desenvolvedor ou do Testador	Autenticação (FAS) e permissões
		Falsificação de servidor	Certificados TLS ou conexão SSH + chave
	Repudiation	Negar o envio de dados para o sistema	Armazenamento de dados de *log*
Testadores e Usuários	*Tampering*	Recebimento de dados não confiáveis	Assinatura dos pacotes
Integração, Koji, Bodhi, Composição noturna usando Pungi, pungi-koji, Sincronização e MirrorManager	*Spoofing*	Falsificação de servidor	Certificados TLS
	Tampering	Recebimento de dados não confiáveis	Assinatura dos pacotes
	Denial of Service	Serviços indisponíveis	Técnicas de alta disponibilidade
Bodhi	*Tampering*	Alteração indevida de *feedbacks*	Armazenamento de dados de *log*
Koji	*Tampering*	Alteração indevida das *tags*	Políticas de controle de acesso e armazenamento de dados de *log*
Fluxos de dados (com exceção dos fluxos: Integração $\leftrightarrow$ dist-git, e Composição noturna usando Pungi $\rightarrow$ pungi-koji)	*Tampering*	Alteração dos dados durante a comunicação	Criptografia TLS ou conexão SSH + chave
Depósitos de dados (todos)	*Tampering*	Alteração indevida dos dados	Políticas de controle de acesso
	Denial of Service	Serviços indisponíveis	Técnicas de alta disponibilidade

Fonte: A autora (2021).

Tabela 4.2: Resumo quantitativo das ameaças encontradas para os DFDs do projeto Fedora

	Entidades externas	Processos	Fluxos de dados	Depósitos de dados	Total
S	2	1	-	-	3
T	1	3	1	1	6
R	1	-	-	-	1
I	-	-	-	-	0
D	-	1	-	1	2
E	-	-	-	-	0
Total					12

Fonte: A autora (2021).

rorManager) estão fora do escopo deste trabalho, ou seja, pressupõe-se que elas estão funcionando corretamente. Devido a isso, as ameaças de falsificação local apresentadas no Capítulo 3 não se aplicam neste modelo. Outras ameaças que não são encontradas nestes diagramas são as de revelação de informações, e *spoofing* de Usuários, dado que o Fedora Linux contém apenas componentes de software livre. As ameaças de *tampering* específicas para o processo de Integração Contínua, as quais são apresentadas no Capítulo 3, também não são aplicáveis para o projeto Fedora.

Na modelagem de ameaças observou-se que a ameaça de retratação das entidades externas apresentada no Capítulo 3 foi totalmente mitigada no projeto Fedora, porque para um Desenvolvedor realizar *commits*, propor uma compilação no Koji e criar uma atualização no Bodhi, ele precisa estar logado em uma conta Fedora. Independente da ação que o mesmo realizou, o nome do usuário fica registrado no sistema. O mesmo se aplica aos Testadores quando estes enviam um *feedback* ao Bodhi. A ameaça de falsificação de servidor encontrada para os processos, e a ameaça de alteração indevida de dados durante uma comunicação também são mitigadas pelo projeto Fedora, dado que este faz uso do protocolo TLS. As ameaças de negação de serviço são mitigadas por meio do uso de técnicas de alta disponibilidade. Notou-se também que algumas ameaças apresentadas no Capítulo 3 foram mitigadas parcialmente pelo projeto Fedora, como é o caso da ameaça de *spoofing* dos Desenvolvedores e Testadores. As ameaças de recebimento de dados não confiáveis também são mitigadas parcialmente pelo projeto Fedora, porque são aplicadas assinaturas GPG aos pacotes, mas estas são úteis apenas para a análise sintática, e não para a validação semântica. Além dessas ameaças, foi possível observar outras que são consideradas mitigadas parcialmente, porque não foi possível encontrar

uma documentação que as esclarecesse. Por exemplo, a ameaça de alteração indevida de dados nos depósitos.

Na modelagem de ameaças notou-se também a descoberta de duas novas ameaças específicas para o *pipeline* de desenvolvimento do Fedora. Uma delas é a alteração/exclusão dos *feedbacks* de karma negativos de uma determinada atualização vulnerável, possibilitando o envio desta para estável. Esta foi considerada mitigada parcialmente devido a falta de documentação. Já a outra ameaça encontrada é a alteração das *tags* dos pacotes no Koji, de modo a permitir a inserção de pacotes corrompidos na composição. Esta ameaça foi mitigada pelo projeto Fedora.

4.6 Limitações

O presente modelo de ameaças possui algumas limitações. Por exemplo, a modelagem foi largamente baseada na documentação existente, cujas limitações já foram apontadas no texto, e em algumas referências da literatura. Portanto, não se pode descartar a possibilidade de que tenha ficado alguma discrepância entre o modelo e a realidade (*e.g.*, devido a alguma mudança na infraestrutura ainda não documentada).

A modelagem de ameaças também não leva em conta vulnerabilidades de implementação nas ferramentas (por uma razão de escopo), isso faz com que algumas ameaças deixam de ser consideradas. Por exemplo, a ameaça de falsificação local, na qual o processo recebe os dados corretos, mas grava dados falsos nos repositórios.

Em alguns casos há mecanismos de mitigação presentes, mas não é possível garantir que eles estejam sendo empregados de forma correta. Um bom exemplo é o gerenciamento de permissões. Foi possível encontrar a política de controle de acesso para o Koji, esta é definida em McLean et al. (2017a), e para os repositórios Git (FEDORA PROJECT, 2018). Porém, para as demais ferramentas e depósitos não foi possível encontrar uma documentação que definisse as políticas das mesmas. Se houvesse essa documentação seria possível ter um grau maior de confiança de que as permissões são gerenciadas corretamente.

O projeto Fedora faz uso do protocolo TLS para mitigar algumas ameaças (*e.g.*, alteração de dados durante a comunicação). Porém, não foi possível encontrar na documentação se existe algum gerenciamento de âncoras de confiança e/ou se os certificados TLS são efetivamente validados.

4.7 Considerações parciais

Neste capítulo foi exibida a modelagem de ameaças para o *pipeline* de desenvolvimento do Fedora. Primeiramente foi apresentado uma visão geral desse projeto. Em seguida foram apresentadas e detalhadas todas as etapas do *pipeline* de desenvolvimento (Integração, Integração contínua, Implantação e Lançamento). Conforme já mencionado no presente trabalho, todo o entendimento sobre o ciclo de desenvolvimento do Fedora e a modelagem foram baseados na limitada documentação existente. Portanto, é possível que existam algumas diferenças entre o que foi apresentado no presente capítulo e a realidade.

Após apresentado o *pipeline* do projeto Fedora, foi realizada a modelagem do mesmo. Dada a complexidade do ciclo de desenvolvimento, optou-se por construir três diagramas de fluxos de dados, ou seja, um para a etapa de Integração, outro para Integração contínua, e um terceiro para as etapas de Implantação e Lançamento. Estes foram apresentados e detalhados suas funcionalidades no presente capítulo.

Em seguida foi feita a modelagem de ameaças para esse sistema, esta fez uso do STRIDE-por-elemento e teve como base a modelagem apresentada no Capítulo 3. Das 17 ameaças encontradas para o *pipeline* de referência, sete delas não se aplicam ao projeto Fedora, dado que este é um software livre e de código aberto, as vulnerabilidades específicas de cada ferramenta não foram consideradas, pacotes e bibliotecas de terceiros não são considerados neste *pipeline*, e os pacotes do Fedora possuem nomes únicos. Além dessas ameaças foram encontradas duas ameaças adicionais, específicas para o projeto Fedora. Todas as ameaças relativas a distribuição Fedora Linux já são mitigadas totalmente ou parcialmente pelas equipes de Desenvolvimento, QA e Engenharia de Lançamento.

Depois de apresentado o *pipeline* de desenvolvimento do Fedora, e realizar a modelagem de ameaças para o mesmo, o Capítulo 5 traz as considerações finais do trabalho, juntamente com os trabalhos futuros.

5 Considerações e Trabalhos futuros

A integridade do software vem sendo uma preocupação crescente na área de segurança, devido ao risco de inserção de código malicioso durante o ciclo de desenvolvimento do software. Ou seja, é necessário garantir que o *pipeline* de desenvolvimento de software não seja violado, isto é, não sejam introduzidas vulnerabilidades em nenhuma de suas etapas.

Neste trabalho foi apresentado um típico *pipeline* de desenvolvimento e foi realizada uma modelagem de ameaças para o mesmo. Na modelagem de ameaças foi criado um diagrama de fluxo de dados, e para todos os elementos envolvidos foram encontrados possíveis ameaças à integridade do software e exemplos de ataques reais que já foram registrados em trabalhos anteriores. Também foi apresentada uma discussão sobre possíveis mitigações para as ameaças encontradas. Foram encontradas 17 ameaças no total.

Durante a análise das ameaças e as discussões das mitigações, foi possível observar que algumas propostas apresentadas para tentar solucionar os problemas encontrados envolvem um alto custo, como é o caso do uso de técnicas de tolerância a intrusões. Há ameaças as quais foi possível mitigar parcialmente, ou seja, ainda haverá risco residual que não será resolvido. Também foram encontradas ameaças que não impactam diretamente na integridade do software, mas existem e são importantes, como é o caso dos ataques de negação de serviço.

Por fim, foi realizada a modelagem de ameaças do *pipeline* da distribuição Fedora Linux. Esta modelagem resultou em 12 ameaças no total, as quais são mitigadas totalmente ou parcialmente (*i.e.*, há uma lacuna nas mitigações empregadas pelas equipes do Fedora e/ou na documentação do projeto) pelas equipes do Fedora. As mitigações adotadas pelo Fedora servem de exemplo do que pode ser feito na prática para mitigar efetivamente essas ameaças. Com o objetivo de melhorar/reforçar algumas dessas mitigações, este trabalho também sugere melhorias para as mesmas.

O objetivo geral do presente trabalho era propor um modelo de ameaças a um *pipeline* de desenvolvimento de software e suas mitigações. Esse objetivo geral foi desdobrado em três objetivos específicos: (i) desenvolver um modelo de sistema para o *pipeline*, (ii) desenvolver um modelo de ameaças, identificando possíveis mitigações, para o *pipeline*, e (iii) analisar pelo menos um *pipeline* de projeto de código aberto para então realimentar o modelo de ameaças e

identificar possíveis ameaças não mitigadas. É possível concluir que esses objetivos específicos, e por conseguinte o objetivo geral, foram totalmente atingidos.

A principal dificuldade para o desenvolvimento do presente trabalho foi a documentação do projeto Fedora, pois algumas partes estão confusas, desorganizadas, e/ou desatualizadas. Isto causou uma certa confusão durante a leitura da mesma e demandou um gasto de tempo maior do que o esperado pela autora. A dificuldade em encontrar determinadas informações nesta documentação também resultou em algumas dúvidas durante a modelagem de ameaças, pois não se sabe até que ponto as mitigações adotadas pelo projeto são implementadas de forma correta.

Resultados preliminares deste trabalho foram publicados em:

- REICHERT, B. M.; OBELHEIRO, R. R. Modelagem de Ameaças em *Pipelines* de Desenvolvimento. In: V Workshop Regional de Segurança da Informação e de Sistemas Computacionais. *Anais da XVIII ERRC*. Porto Alegre, RS: Editora da SBC, 2020.

Como proposta de trabalho futuro, sugere-se o aprofundamento da modelagem de ameaças para a distribuição Fedora Linux, isto é, estudar as ferramentas (Koji, Bodhi, entre outras) para identificar possíveis ameaças para as mesmas. Ademais, recomenda-se realizar a modelagem de ameaças para outros projetos de código aberto, possibilitando realizar comparações entre diferentes *pipelines* de desenvolvimento.

Referências

ABNT. *ABNT NBR ISO/IEC 27002 - Tecnologia da Informação - Técnicas de Segurança – Código de Prática para controles de segurança da informação*. [S.l.], 2013.

ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell – why researchers should care. In: IEEE. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2016. v. 5, p. 78–90.

ATCHISON, L. *Architecting for Scale: High Availability for Your Growing Applications*. 1. ed. Sebastopol, CA: O'Reilly Media, 2016. ISBN 978-1-491-94339-7.

BASS, L. et al. Securing a deployment pipeline. In: IEEE. *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. [S.l.], 2015. p. 4–7.

BIRSAN, A. *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies*. 2021. Medium. Disponível em: <https://medium.com/@alex.birsan-/dependency-confusion-4a5d60fec610>. Acesso em: 10 fev. 2021.

BITBUCKET. *Making a Pull Request*. 2021. Atlassian Bitbucket. Disponível em: <https://www.atlassian.com/git/tutorials/making-a-pull-request>. Acesso em: 30 jul. 2021.

BRUMAGHIN, E. et al. *CCleanup: A Vast Number of Machines at Risk*. 2017. Talos Blog. Disponível em: <https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html>. Acesso em: 13 out. 2020.

CHAWLA, V. *Jack Dorsey's Twitter Hack Proves SMS-Based Two-Factor Authentication Is Not Foolproof*. 2019. Analytics India Magazine. Disponível em: <https://analyticsindiamag-.com/dorsey-twitter-hack-sms-two-factor-authentication/>. Acesso em: 18 nov. 2020.

CIMPANU, C. *JavaScript Packages Caught Stealing Environment Variables*. 2017. Bleeping Computer. Disponível em: <https://www.bleepingcomputer.com/news/security/javascript-packages-caught-stealing-environment-variables/>. Acesso em: 13 out. 2020.

CLARK, J.; OORSCHOT, P. C. van. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In: IEEE. *2013 IEEE Symposium on Security and Privacy*. [S.l.], 2013. p. 511–525.

CORBET, J. *An attempt to backdoor the kernel*. 2003. Disponível em: <https://lwn.net-/Articles/57135/>. Acesso em: 24 out. 2020.

COSTA, T. *strong_password v0.0.7 rubygem hijacked*. 2019. Disponível em: <https:/-/withatwist.dev/strong-password-rubygem-hijacked.html>. Acesso em: 14 dez. 2020.

CRACIUN, V. et al. Malware in the SGX supply chain: Be careful when signing enclaves! *IEEE Transactions on Dependable and Secure Computing*, p. 1–1, sep 2020.

DIAZ-SANCHEZ, D. et al. TLS/PKI Challenges and Certificate Pinning Techniques for IoT and M2M Secure Communications. *IEEE Communications Surveys & Tutorials*, IEEE, v. 21, n. 4, p. 3502–3531, may 2019.

DOMSCH, M. Getting the bits out: Fedora mirrormanager. *Linux Symposium*, Ottawa, Ontario, Canada, v. 1, p. 107–112, jul 2008.

DOTSON, C. *Practical Cloud Security: A Guide for Secure Design and Deployment*. 1. ed. Sebastopol, CA: O'Reilly Media, 2019. ISBN 978-1-492-03751-4.

ENG, D. *Integrated Threat Modelling*. Dissertação (Mestrado) — University of Oslo, 2017.

FEDORA. *Fedora keeps you safe*. 2020. Disponível em: <https://getfedora.org/en/security/>. Acesso em: 12 jul. 2021.

FEDORA. *Welcome to Freedom*. 2021. Disponível em: <https://getfedora.org/en/>. Acesso em: 12 jul. 2021.

FEDORA INFRASTRUCTURE TEAM. *Content Hosting Infrastructure SOP*. 2012. Disponível em: <https://fedora-infra-docs.readthedocs.io/en/latest/sysadmin-guide/sops-/contenthosting.html>. Acesso em: 01 jul. 2021.

FEDORA INFRASTRUCTURE TEAM. *MirrorManager Infrastructure SOP*. 2017. Disponível em: <https://fedora-infra-docs.readthedocs.io/en/latest/sysadmin-guide/sops-/mirrormanager.html>. Acesso em: 27 jul. 2021.

FEDORA PROJECT. *Go No Go Meeting*. 2017. Disponível em: <https://fedoraproject.org-/wiki/Go_No_Go_Meeting>. Acesso em: 03 jul. 2021.

FEDORA PROJECT. *Infrastructure/Tickets*. 2017. Disponível em: <https://fedoraproject.org-/wiki/Infrastructure/Tickets>. Acesso em: 07 jul. 2021.

FEDORA PROJECT. *Package Source Control*. 2018. Disponível em: <https://fedoraproject-.org/wiki/Package_Source_Control>. Acesso em: 24 jul. 2021.

FEDORA PROJECT. *Fedora's Mission and Foundations*. 2019. Disponível em: <https://docs-.fedoraproject.org/en-US/project/>. Acesso em: 02 jul. 2021.

FEDORA PROJECT. *Infrastructure/Architecture*. 2020. Disponível em: <https:/-/fedoraproject.org/wiki/Infrastructure/Architecture>. Acesso em: 22 jul. 2021.

FEDORA PROJECT. *Infrastructure/Mirroring*. 2020. Disponível em: <https://fedoraproject-.org/wiki/Infrastructure/Mirroring>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *OpenQA*. 2020. Disponível em: <https://fedoraproject.org/wiki-/OpenQA>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *Account System*. 2021. Disponível em: <https://fedoraproject.org/wiki-/Account_System>. Acesso em: 12 jul. 2021.

FEDORA PROJECT. *Bodhi*. 2021. Disponível em: <https://bodhi.fedoraproject.org/docs/>. Acesso em: 27 jun. 2021.

FEDORA PROJECT. *Bodhi*. 2021. Disponível em: <https://fedoraproject.org/wiki/Bodhi>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *Fedora Release Life Cycle*. 2021. Disponível em: <https://fedoraproject-.org/wiki/Fedora_Release_Life_Cycle>. Acesso em: 07 jul. 2021.

FEDORA PROJECT. *Infrastructure/Mirroring/Tiering*. 2021. Disponível em: <https://fedoraproject.org/wiki/Infrastructure/Mirroring/Tiering>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *QA:Update feedback guidelines*. 2021. Disponível em: <https://fedoraproject.org/wiki/QA:Update_feedback_guidelines>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *QA:Updates Testing*. 2021. Disponível em: <https://fedoraproject.org/wiki/QA:Updates_Testing>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *Releases*. 2021. Disponível em: <https://fedoraproject.org/wiki/Releases>. Acesso em: 30 jun. 2021.

FEDORA PROJECT. *Releases/Branched*. 2021. Disponível em: <https://fedoraproject.org/wiki/Releases/Branched>. Acesso em: 27 jun. 2021.

FEDORA PROJECT. *Releases/Rawhide*. 2021. Disponível em: <https://fedoraproject.org/wiki/Releases/Rawhide>. Acesso em: 30 jun. 2021.

FEDORA PROJECT. *Updates policy*. 2021. Disponível em: <https://docs.fedoraproject.org/en-US/fesco/Updates_Policy/>. Acesso em: 01 jul. 2021.

FEDORA PROJECT. *Using Fedora Accounts*. 2021. Disponível em: <https://docs.fedoraproject.org/en-US/fedora-accounts/user/>. Acesso em: 12 jul. 2021.

FEDORA RELEASE ENGINEERING. *Composing Fedora*. 2016. Disponível em: <https://docs.pagure.org/releng/sop_composing_fedora.html>. Acesso em: 01 jul. 2021.

FEDORA RELEASE ENGINEERING. *Create Release Signing Key*. 2016. Disponível em: <https://docs.pagure.org/releng/sop_create_release_signing_key.html>. Acesso em: 12 jul. 2021.

FEDORA RELEASE ENGINEERING. *Fedora Release Engineering Overview*. 2016. Disponível em: <https://docs.pagure.org/releng/overview.html>. Acesso em: 27 jun. 2021.

FEDORA RELEASE ENGINEERING. *Release Package Signing*. 2016. Disponível em: <https://docs.pagure.org/releng/sop_release_package_signing.html>. Acesso em: 12 jul. 2021.

FRANCESCHI-BICCHIERAI, L. *The Fortnite Trial Is Exposing Details About the Biggest iPhone Hack on Record*. 2021. VICE. Disponível em: <https://www.vice.com/en/article/n7bbmz/the-fortnite-trial-is-exposing-details-about-the-biggest-iphone-hack-of-all-time>. Acesso em: 16 maio 2021.

GALLAGHER, S. *Rage-quit: Coder unpublished 17 lines of JavaScript and "broke the Internet"*. 2016. Ars Technica. Disponível em: <https://arstechnica.com/information-technology/2016/03/rage-quit-coder-unpublished-17-lines-of-javascript-and-broke-the-internet/>. Acesso em: 14 dez. 2020.

GALLAGHER, W. *Editorial: A year later, Bloomberg silently stands by its 'Big Hack' iCloud spy chip story*. 2019. AppleInsider. Disponível em: <https://appleinsider.com/articles/19/10/04/editorial-a-year-later-bloomberg-silently-stands-by-its-big-hack-icloud-spy-chip-story>. Acesso em: 11 dez. 2020.

GITHUB. *Bodhi*. 2021. GitHub. Disponível em: <https://github.com/fedora-infra/bodhi>. Acesso em: 27 jun. 2021.

GITHUB. *DistGit*. 2021. GitHub. Disponível em: <https://github.com/release-engineering-/dist-git>. Acesso em: 26 jun. 2021.

GOODIN, D. *Devs unknowingly use "malicious" modules snuck into official Python repository*. 2017. Ars Technica. Disponível em: <https://arstechnica.com/information-technology/2017-/09/devs-unknowingly-use-malicious-modules-put-into-official-python-repository/>. Acesso em: 20 abr. 2020.

GOODIN, D. *The year-long rash of supply chain attacks against open source is getting worse*. 2019. Ars Technica. Disponível em: <https://arstechnica.com/information-technology-/2019/08/the-year-long-rash-of-supply-chain-attacks-against-open-source-is-getting-worse/>. Acesso em: 14 dez. 2020.

GOUSIOS, G.; PINZGER, M.; DEURSEN, A. v. An exploratory study of the pull-based software development model. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. p. 345–355.

HERN, A. *Tor users advised to check their computers for malware*. 2014. The Guardian. Disponível em: <https://www.theguardian.com/technology/2014/oct/28/tor-users-advised-check-computers-malware>. Acesso em: 20 abr. 2020.

HERNAN, S. et al. Threat modeling - uncover security design flaws using the stride approach. *MSDN Magazine-Louisville*, San Francisco, CA: CMP Media Inc., c2000-, p. 68–75, 2006.

HOWARD, M.; LIPNER, S. *The Security Development Lifecycle*. [S.l.]: Microsoft Press, 2006. ISBN 978-0-735-62214-2.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1. ed. [S.l.]: Pearson Education, 2010.

KUPSCH, J. A. et al. Bad and good news about using software assurance tools. *Software: Practice and Experience*, Wiley Online Library, v. 47, n. 1, p. 143–156, 2017.

LACHMAN, B. F. *Modern development workflow for Fedora GNU/Linux distribution*. Dissertação (Mestrado) — Masaryk University Faculty of Informatics, 2019.

MADDOX, I. *12 best practices for user account, authentication and password management*. 2018. Google Cloud Platform. Disponível em: <https://cloud.google.com/blog/products/gcp-/12-best-practices-for-user-account>. Acesso em: 15 dez. 2020.

MARCUS, E.; STERN, H. *Blueprints for high availability*. 2. ed. Indianopolis, Indiana: John Wiley & Sons, 2003. ISBN 0-471-43026-9.

MAUNDER, M. *PSA: 4.8 Million Affected by Chrome Extension Attacks Targeting Site Owners*. 2017. Disponível em: <https://www.wordfence.com/blog/2017/08/chrome-browser-extension-attacks/>. Acesso em: 17 abr. 2020.

MCLEAN, M. et al. *Access Controls*. 2017. Disponível em: <https://docs.pagure.org/koji-/access_controls/>. Acesso em: 22 jul. 2021.

MCLEAN, M. et al. *Koji HOWTO*. 2017. Disponível em: <https://docs.pagure.org/koji-/HOWTO/>. Acesso em: 27 jun. 2021.

MICROSOFT. *Microsoft Security Development Lifecycle (SDL)*. 2020. Disponível em: <https://www.microsoft.com/en-us/securityengineering/sdl/>. Acesso em: 05 out. 2020.

MILLER, A. et al. *RPM Packages*. 2020. Disponível em: <https://rpm-packaging-guide-.github.io//#rpm-packages>. Acesso em: 30 jun. 2021.

MYAGMAR, S.; LEE, A. J.; YURCIK, W. Threat Modeling as a Basis for Security Requirements. In: CITESEER. *Symposium on requirements engineering for information security (SREIS)*. [S.l.], 2005. v. 2005, p. 1–8.

NEWMAN, L. H. *Github Survived the Biggest DDoS Attack Ever Recorded*. 2018. Wired. Disponível em: <https://www.wired.com/story/github-ddos-memcached/>. Acesso em: 15 out. 2020.

NIST. *Back to basics: Multi-factor authentication (MFA)*. 2019. NIST. Disponível em: <https://www.nist.gov/itl/applied-cybersecurity/tig/back-basics-multi-factor-authentication>. Acesso em: 15 dez. 2020.

OBELHEIRO, R. R.; BESSANI, A. N.; LUNG, L. C. Analisando a viabilidade da implementaçao prática de sistemas tolerantes a intrusões. *Anais do V Simpósio Brasileiro em Segurança da Informaç ao e de Sistemas Computacionais-SBSeg 2005*, 2005.

OWASP. *OWASP Cornucopia*. 2020. Disponível em: <https://owasp.org/www-project-cornucopia/>. Acesso em: 13 out. 2020.

OWASP. *Pinning Cheat Sheet*. 2020. Disponível em: <https://cheatsheetseries.owasp-.org/cheatsheets/Pinning_Cheat_Sheet.html#pinning-cheat-sheet>. Acesso em: 24 ago. 2020.

OWEN, M. *Apple denies claim China slipped spy chips into its iCloud server hardware*. 2018. AppleInsider. Disponível em: <https://appleinsider.com/articles/18/10/04/apple-denies-claims-china-attacked-icloud-server-supply-chain-to-spy-on-us>. Acesso em: 11 dez. 2020.

PAGURE. *About Pungi*. 2016. Disponível em: <https://docs.pagure.org/pungi/about.html>. Acesso em: 01 jul. 2021.

PAGURE. *Phases*. 2016. Disponível em: <https://docs.pagure.org/pungi/phases.html>. Acesso em: 01 jul. 2021.

PAGURE. *Fedora Comps*. 2021. Disponível em: <https://pagure.io/fedora-comps>. Acesso em: 03 jul. 2021.

Palo Alto Networks. *Cloud Threat Report*. 2020. Disponível em: <https://www-.paloaltonetworks.com/content/dam/pan/en_US/assets/pdf/reports/Unit_42/cloud-threat-report-spring-2020.pdf>. Acesso em: 12 set. 2020.

Palo Alto Networks. *Unit 42 Cloud Threat Report: Spring 2020*. 2020. Disponível em: <https://unit42.paloaltonetworks.com/cloud-threat-report-intro/>. Acesso em: 12 set. 2020.

PAULE, C. *Securing DevOps: detection of vulnerabilities in CD pipelines*. Dissertação (Mestrado) — Institute of Software Technology and University of Stuttgart, 2018.

PISTOIA, M. et al. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, IBM, v. 46, n. 2, p. 265–288, 2007.

RAHMAN, A.; PARNIN, C.; WILLIAMS, L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2019. p. 164–175.

RAHMAN, A. et al. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Transactions on Software Engineering and Methodology*, v. 30, n. 1, jan 2021.

REICHERT, B. M.; OBELHEIRO, R. R. Modelagem de Ameaças em *Pipelines* de Desenvolvimento. In: V Workshop Regional de Segurança da Informação e de Sistemas Computacionais. *Anais da XVIII ERRC*. Porto Alegre, RS: Editora da SBC, 2020.

RESCORLA, E. RFC, *The Transport Layer Security (TLS) Protocol Version 1.3*. Fremont, CA, USA: RFC Editor, 2018. RFC 8446 (Proposed Standard). (Internet Request for Comments, 8446). Disponível em: <https://tools.ietf.org/html/rfc8446>. Acesso em: 03 ago. 2020.

ROBERTSON, J.; RILEY, M. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. 2018. Bloomberg. Disponível em: <https://www.bloomberg.com/news-/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies>. Acesso em: 01 dez. 2020.

SHARWOOD, S. *OpenStack haven OpenDev yanks Gerrit code review tool after admin account compromised for two weeks*. 2020. The Register. Disponível em: <https://www-.theregister.com/2020/10/21/opendev_gerrit_attack/>. Acesso em: 04 nov. 2020.

SHAW, R. A. *Software Supply Chain Attacks*. 2017. Disponível em: <https://csrc.nist-.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter-/NCSC_Placemat.pdf>. Acesso em: 25 mar. 2020.

SHEVCHENKO, N. et al. *Threat modeling: a summary of available methods*. [S.l.], 2018.

SHIREY, R. W. *Internet Security Glossary, Version 2*. RFC Editor, 2007. RFC 4949. (Request for Comments, 4949). Disponível em: <https://rfc-editor.org/rfc/rfc4949.txt>.

SHOSTACK, A. *Threat modeling: Designing for security*. Indianopolis, Indiana: John Wiley & Sons, 2014. ISBN 978-1-118-80999-0.

SIMPSON, S. *Software integrity controls–an assurance-based approach to minimizing risks in the software supply chain*. [S.l.], 2010.

SKRIMSTAD, Y. *Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds*. Dissertação (Mestrado) — University of Oslo, 2018.

SONATYPE. *2019 Software Supply Chain Report*. 2019. Disponível em: <https://www-.sonatype.com/resources/white-paper-state-of-software-supply-chain-report-2019>. Acesso em: 16 abr. 2021.

SONATYPE. *2020 State of the Software Supply Chain*. 2020. Disponível em: <https://www-.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020>. Acesso em: 16 abr. 2021.

SUSE. *Linux Distribution*. 2021. Disponível em: <https://susedefines.suse.com/definition-/linux-distribution/>. Acesso em: 07 jul. 2021.

SWIDERSKI, F.; SNYDER, W. *Threat Modeling*. [S.l.]: Microsoft Press, 2004.

TAYLOR, M. et al. SpellBound: Defending Against Package Typosquatting. *arXiv preprint arXiv:2003.03471*, mar 2020.

The Linux Foundation. *Open Source Software Supply Chain Security*. 2020. Disponível em: <https://www.linuxfoundation.org/en/resources/publications/open-source-software-supply-chain-security/>. Acesso em: 22 nov. 2020.

THEIS, M. et al. *Common Sense Guide to Mitigating Insider Threats*. Sixth. Pittsburgh, PA, 2019. Disponível em: <http://resources.sei.cmu.edu/library/asset-view-.cfm?AssetID=540644>. Acesso em: 15 dez. 2020.

THOMPSON, K. Reflections on trusting trust. *Communications of the ACM*, ACM New York, NY, USA, v. 27, n. 8, p. 761–763, aug 1984.

TORRES-ARIAS, S. *In-toto: Practical Software Supply Chain Security*. Tese (Doutorado) — New York University Tandon School of Engineering, 2020.

TORRES-ARIAS, S. et al. in-toto: Providing farm-to-table guarantees for bits and bytes. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019. p. 1393–1410. ISBN 978-1-939133-06-9. Disponível em: <https://www-.usenix.org/conference/usenixsecurity19/presentation/torres-arias>. Acesso em: 17 abr. 2021.

TUMA, K.; SCANDARIATO, R. Two architectural threat analysis techniques compared. In: SPRINGER. *European Conference on Software Architecture*. [S.l.], 2018.

TUNKA, V. *Jenkins-Koji Integration Plugin*. Dissertação (Mestrado) — Univerzita Tomáše Bati ve Zlíně, 2014.

VIE, D. S. L. Understanding data flow diagrams. In: *Annual Conference-Society for Technical Communication*. [S.l.: s.n.], 2000. v. 47, p. 396–401.

WARREN, T. *Hackers hid malware in CCleaner software*. 2017. The Verge. Disponível em: <https://www.theverge.com/2017/9/18/16325202/ccleaner-hack-malware-security>. Acesso em: 13 out. 2020.

WEBMIN. *Webmin 1.890 Exploit - What Happened?* 2019. Webmin. Disponível em: <https://www.webmin.com/exploit.html>. Acesso em: 27 out. 2020.

WEINERT, A. *Your Pa$$word doesn't matter*. 2019. Microsoft. Disponível em: <https:-//techcommunity.microsoft.com/t5/azure-active-directory-identity/your-pa-word-doesn-t-matter/ba-p/731984>. Acesso em: 15 dez. 2020.

WHEELER, D. A. Fully Countering Trusting Trust through Diverse Double-Compiling (DDC). nov 2009.

WHEELER, D. A.; REDDY, D. J.; FONG, E. K. *Securely Using Software Assurance (SwA) Tools in the Software Development Environment*. [S.l.], 2018.

WORRALL, B. *Important Announcement about ScreenOS*. 2015. Disponível em: <https://forums.juniper.net/t5/Security-Incident-Response/Important-Announcement-about-ScreenOS/ba-p/285554>. Acesso em: 08 out. 2020.

XIAO, C. *Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store*. 2015. Palo Alto Networks. Disponível em: <https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/>. Acesso em: 12 set. 2020.

YLONEN, T. Ssh key management challenges and requirements. In: *2019 10th IFIP International Conference on New Technologies, Mobility and Security*. United States: IEEE, 2019. (International Conference on New Technologies Mobility and Security). ISBN 978-1-7281-1543-6.