Modelagem de Ameaças em *Pipelines* de Desenvolvimento

Beatriz M. Reichert[1], **Rafael R. Obelheiro**[1]

[1]Universidade do Estado de Santa Catarina (UDESC) – Campus Joinville

beatrizreichert99@gmail.com, rafael.obelheiro@udesc.br

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. Este trabalho propõe desenvolver um modelo de ameaças para o pipeline de desenvolvimento e identificar mitigações para as ameaças encontradas. O artigo apresenta o pipeline e o modelo de ameaças para uma de suas etapas.*

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 [Simpson 2010]. Essa confiança envolve três objetivos: segurança, integridade e autenticidade. 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.

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 and McIntosh 2016].

Embora a segurança do *pipeline* de desenvolvimento tenha recebido atenção na literatura [Simpson 2010, Bass et al. 2015, Shaw 2017, Wheeler and Reddy 2018], uma lacuna diz respeito à identificação 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 restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta as etapas do *pipeline* de desenvolvimento, incluindo ameaças relativas a cada etapa e descrevendo casos documentados de incidentes de segurança. A Seção 3 introduz o modelo de ameaças proposto, com as mitigações correspondentes, aplicado a um subconjunto do *pipeline*. Finalmente, a Seção 4 conclui o artigo.

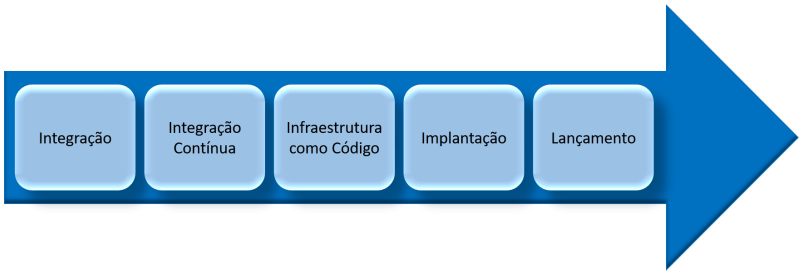

Figura 1. *Pipeline* típico de desenvolvimento de software

2. *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 existe uma definição rígida de quais são as etapas do *pipeline*; a Figura 1 representa um *pipeline* típico [Adams and 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 1, adotar esse *pipeline* como referência permite que todas as atividades do trajeto desenvolvedor → 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.

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 and McIntosh 2016]. Para auxiliar nesse processo, muitas organizações de software fazem uso de Sistemas de Controle de Versão (VCS, *Version Control Systems*), como o Subversion ou o Git [Adams and McIntosh 2016].

Ataques na etapa de integração envolvem modificações não autorizadas do código fonte. Exemplos de ataques reais incluem a tentativa de modificar o *kernel* do Linux para inserir uma vulnerabilidade [Corbet 2003], a inserção de código não autorizado no sistema operacional de equipamentos de rede Juniper, criando um *backdoor* para acesso remoto ao equipamento e permitindo monitorar e decifrar tráfego [Juniper 2015], e o caso onde os desenvolvedores do Chrome tiveram suas credenciais roubadas e os invasores puderam modificar as extensões, comprometendo milhões de usuários [Maunder 2017].

Integração Contínua: compilação e teste A Integração Contínua (IC) pesquisa continuamente o VCS em busca de novas revisões, compilando-as e executando um

conjunto inicial de testes para verificar se elas não causaram problemas no projeto [Adams and McIntosh 2016]. Para automatizar essas atividades pode-se fazer uso das ferramentas de IC como Jenkins ou similar. Todos os estágios de teste obtêm os produtos que foram construídos pelo processo de IC a partir do repositório de artefatos (*artifact repository*). 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 and McIntosh 2016].

Ataques na etapa de integração contínua envolvem inserção de vulnerabilidades no software por ferramentas, pacotes e bibliotecas corrompidas. Exemplos de casos reais incluem o uso de *typosquatting* para registrar pacotes com conteúdo malicioso no repositório npm para JavaScript usando nomes semelhantes aos de pacotes legítimos, visando infectar desenvolvedores que cometessem erros de digitação [Cimpanu 2017], e 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 4 mil aplicações infectadas na App Store do iOS [FireEye 2015].

Infraestrutura como código A infraestrutura como código (IaC, *Infrastructure as code*) automatiza a configuração de um ambiente (servidor, nuvem, contêiner ou máquina virtual) na qual uma nova versão do sistema deve ser implantada para teste ou produção [Adams and McIntosh 2016]. Essa fase gera o ambiente com base na especificação desenvolvida em uma linguagem de programação dedicada como Puppet, Chef, Ansible ou similar. Essas ferramentas automatizam 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. Especificações vulneráveis vão gerar infraestruturas vulneráveis, o que é particularmente perigoso quando se trata de ambientes de produção. Um relatório recente identificou quase 200.000 vulnerabilidades em especificações IaC [Palo Alto Networks 2020].

Implantação Depois que os produtos de software tiverem sido construídos e testados com sucesso, na fase de implantação (*deployment*) esses produtos são preparados para o lançamento (*release*). Por exemplo, para aplicativos Web, a implantação pode corresponder à cópia de um conjunto de arquivos pela rede para o diretório correto em um servidor Web [Adams and McIntosh 2016]. Em muitos casos o software permanece um tempo inativo entre as etapas de implantação e lançamento.

Ataques na etapa de implantação envolvem modificações não autorizadas do código executável. Exemplos de ataques reais incluem a disseminação do malware NotPetya por meio de atualizações adulteradas de um software de contabilidade [Wheeler and Reddy 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].

Lançamento O lançamento (*release*) é a etapa final do *pipeline*, na qual os produtos implantados são lançados e ficam disponíveis para os usuários [Adams and McIntosh 2016]. Existem várias formas de lançamento, como a oferta de um novo aplicativo ou versão em uma *app store*, a liberação de uma aplicação web, e a disponibilização de binários em um site. Ataques na etapa de lançamento envolvem substituir ou adulterar os executáveis e/ou componentes disponibilizados aos usuários. Exemplos de ataques re-

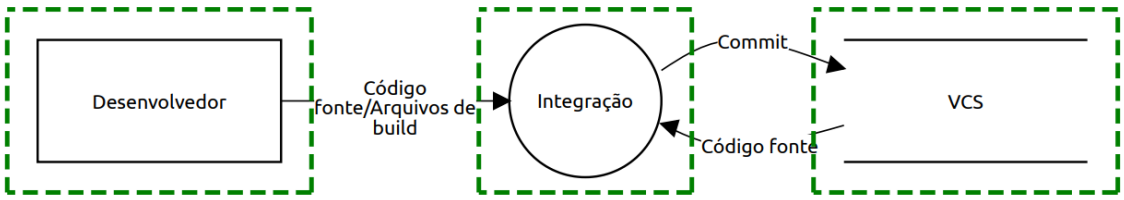

Figura 2. DFD para a etapa de Integração

ais incluem a inserção de código malicioso no software Windows baixado via Tor [Hern 2014] e a distribuição de uma versão infectada com malware da ferramenta CCleaner [Warren 2017, Brumaghin et al. 2017].

3. Modelagem de Ameaças

O objetivo deste trabalho é identificar ameaças para o *pipeline* de desenvolvimento de software, e discutir as possíveis mitigações. A abordagem adotada será a modelagem de ameaças baseada no modelo STRIDE, desenvolvida e usada pela Microsoft [Shostack 2014]. O acrônimo STRIDE representa as seguintes seis classes de ameaças:

- *Spoofing* (falsificação de identidade): alegação de uma identidade falsa;
- *Tampering* (manipulação): corrupção ou adulteração de dados;
- *Repudiation* (repudiação): 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;
- *Elevation of privilege* (elevação de privilégio): quando um usuário ou programa pode agir no sistema com privilégios além dos que lhe foram concedidos.

Uma modelagem de ameaças parte de um diagrama de fluxo de dados (DFD), que representa o fluxo de informações entre entidades externas, processos e depósitos de dados que compõem um sistema [Le Vie 2000]. Para a modelagem de ameaças adiciona-se um elemento extra, a fronteira de confiança, que indica componentes que estão em diferentes contextos de segurança (por exemplo, domínios de proteção) [Shostack 2014].

Para ilustrar a modelagem de ameaças do *pipeline* de desenvolvimento de software, será apresentado o modelo de ameaças para a etapa de Integração, que foi escolhida por ser bastante conhecida e por incluir todos os elementos dos DFDs. A Figura 2 apresenta a parte correspondente do DFD. Um Desenvolvedor (uma entidade externa) envia código fonte e arquivos de *build* (um fluxo de dados) para o processo de Integração. Outros dois fluxos de dados (Commit e Código fonte) representam a comunicação entre o processo Integração e o depósito de dados VCS. Para tornar a análise mais genérica, consideramos que cada elemento do diagrama está na sua própria fronteira de confiança (delimitada por uma linha tracejada).

Após a criação do diagrama foi usado o STRIDE-por-elemento, uma variante do STRIDE, para encontrar ameaças no DFD. No STRIDE-por-elemento, considera-se a quais ameaças cada elemento do DFD (entidade externa, processo, fluxo e depósito de dados) está sujeito. As ameaças encontradas para a entidade externa Desenvolvedor são:

- *Spoofing*: um atacante pode se passar por um Desenvolvedor e dessa forma pode inserir vulnerabilidades no código fonte. É possível mitigar usando autenticação. Outra ameaça de *spoofing* 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. É possível mitigar essa ameaça com o uso de certificados TLS (*Transport Layer Security*) [Rescorla 2018] para autenticar o servidor (o final desta seção traz considerações adicionais sobre o uso de TLS).
- *Repudiation*: o Desenvolvedor pode negar que enviou 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. Essa ameaça pode ser mitigada armazenando dados de log e também por meio da assinatura digital dos *commits*.

As ameaças identificadas para o processo Integração são:

- *Spoofing*: falsificação de servidor – o processo não sabe se está se conectando com um repositório legítimo. Para mitigar essa ameaça recomenda-se o uso de certificados TLS para autenticar o servidor.
- *Tampering*: o processo não sabe se os dados que está recebendo, seja de entidades externas ou de depósitos de dados, são confiáveis. É possível garantir dados não corrompidos fazendo uso de mecanismos como permissões e assinaturas digitais. As permissões garantem que apenas os principais (processos) autorizados podem modificar o conteúdo do repositório, enquanto que as assinaturas digitais garantem que os binários foram criados ou certificados pelos principais autorizados. No entanto, essas técnicas são insuficientes para a validação semântica, ou seja, elas não garantem que esse software seja correto. Isso pode ser mitigado usando ferramentas para detecção de vulnerabilidades no código fonte [Kupsch et al. 2017]. 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 et al. 2005].
- *Denial of Service*: se o repositório falhar, ou seja, seus serviços ficarem indisponíveis, teremos um problema de negação de serviço. Essa ameaça pode ser mitigada usando técnicas de alta disponibilidade [Atchison 2016]. Como a negação de serviço não impacta diretamente a integridade do software, que é o foco deste trabalho, não iremos nos aprofundar sobre a mitigação dessa classe de ameaças.

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. Este problema pode ser mitigado usando criptografia TLS, que 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, um vazamento do código fonte pode acarretar perda monetária. Para mitigar essa ameaça podem ser usadas as mesmas propostas da ameaça anterior: TLS ou outros canais cifrados de comunicação.

As ameaças relativas ao depósito de dados VCS são:

- *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. Para mitigar essa ameaça recomenda-se gerenciar adequadamente as permissões e criptografar os dados antes de armazená-los.
- *Information Disclosure*: no caso de software proprietário, existe a ameaça de que os dados sejam visualizados por usuários não autorizados, permitindo o roubo de código fonte ou binário. São aplicadas as mesmas mitigações da ameaça anterior.
- *Denial of Service*: repositórios podem sofrer ataques de negação de serviço, deixando seus serviços indisponíveis. É possível mitigar controlando o uso de recursos [Shostack 2014] e usando técnicas de alta disponibilidade [Atchison 2016].

Limitações do TLS O TLS é a solução padrão para proteção de fluxos de dados e para autenticação de servidores [Shostack 2014]. Fazendo uso de certificados digitais, o TLS fornece confidencialidade, integridade de dados e autenticação [Rescorla 2018].

Porém, o TLS possui limitações no seu modelo de confiança, notadamente a possibilidade de que qualquer autoridade certificadora (AC) emita um certificado para qualquer nome, independente da vontade ou consentimento do responsável pelo nome, que é acentuada pelo grande número de ACs tipicamente aceitas como confiáveis [Clark and Van Oorschot 2013]. Isso facilita a emissão de certificados fraudulentos que serão aceitos como válidos, oportunizando ataques de falsificação de servidor. Portanto, recomenda-se que sejam adotadas medidas para contornar essa limitação do TLS, tais como reduzir a lista de âncoras de confiança às ACs necessárias (que, em alguns casos, podem ser privadas), e/ou usar fixação (*pinning*), que é o processo de associar um servidor com um ou mais certificados ou chaves públicas esperados [OWASP 2020].

4. Conclusão

A integridade do software vem sendo uma preocupação crescente na área de segurança. Neste trabalho apresentamos um típico *pipeline* de desenvolvimento e fizemos uma modelagem de ameaças de uma etapa do *pipeline* usando STRIDE, identificando as ameaças à etapa de Integração e apontando possíveis mitigações. Na continuidade do trabalho pretende-se estender o modelo de ameaças às demais etapas do *pipeline*, e analisar o *pipeline* de projetos de código aberto para identificar ameaças não mitigadas.

Referências

Adams, B. and McIntosh, S. (2016). Modern release engineering in a nutshell–why researchers should care. In *IEEE SANER*, volume 5, pages 78–90.

Atchison, L. (2016). *Architecting for Scale: High Availability for Your Growing Applications*. O'Reilly Media.

Bass, L., Holz, R., Rimba, P., Tran, A. B., and Zhu, L. (2015). Securing a deployment pipeline. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 4–7.

Brumaghin, E., Gibb, R., Mercer, W., Molyett, M., and Williams, C. (2017). Ccleanup: A vast number of machines at risk. Talos Blog. `https://bit.ly/34JGPmw`

Cimpanu, C. (2017). Javascript packages caught stealing environment variables. Bleeping Computer. `https://bit.ly/2TE3aM3`

Clark, J. and Van Oorschot, P. C. (2013). SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symp. on Security and Privacy*.

Corbet, J. (2003). An attempt to backdoor the kernel. https://bit.ly/3jJ0SWH

FireEye (2015). Protecting our customers from XcodeGhost. https://bit.ly/3mA4WdC

Goodin, D. (2017). Devs unknowingly use "malicious" modules snuck into official python repository. Ars Technica. https://bit.ly/3iHcKZD

Hern, A. (2014). Tor users advised to check their computers for malware. *The Guardian*. https://bit.ly/3msg7Gd

Juniper (2015). Important announcement about ScreenOS. https://juni.pr/3c0WUGR

Kupsch, J. A., Heymann, E., Miller, B., and Basupalli, V. (2017). Bad and good news about using software assurance tools. *Software: Practice and Experience*, 47(1):143–156.

Le Vie, D. S. (2000). Understanding data flow diagrams. In *Annual Conference-Society for Technical Communication*, volume 47, pages 396–401.

Maunder, M. (2017). Psa: 4.8 million affected by chrome extension attacks targeting site owners. https://bit.ly/3oH5EHL

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do SBSeg*.

OWASP (2020). Pinning cheat sheet. https://bit.ly/3c7mNER

Palo Alto Networks (2020). Cloud threat report. https://bit.ly/2DV8V3T

Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://bit.ly/31ZE5jo

Shaw, R. A. (2017). Software supply chain attacks. https://bit.ly/2RyA4Nj

Shostack, A. (2014). *Threat modeling: Designing for security*. John Wiley & Sons.

Simpson, S. (2010). Software integrity controls–an assurance-based approach to minimizing risks in the software supply chain. Technical report, SAFECode.

Warren, T. (2017). Hackers hid malware in CCleaner software. The Verge. https://bit.ly/3kHXlJC

Wheeler, D. A. and Reddy, D. J. (2018). Securely using software assurance (SwA) tools in the software development environment. IDA Document P-9166, Institute for Defense Analysis.

Xiao, C. (2015). Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store. Palo Alto Networks. https://bit.ly/2HGEfFe