paloalto NETWORKS | CORTEX

Guide to the Top 10 CI/CD Security Risks

Introduction

CI/CD environments, processes, and systems are the beating heart of any modern software organization. They deliver code from an engineer's workstation to production. Combined with the DevOps discipline and microservice architectures, CI/CD systems and processes have reshaped the engineering ecosystem:

- The technical stack is more diverse, both in relation to coding languages as well as to technologies and frameworks adopted further down the pipeline (e.g., GitOps or Kubernetes).
- Adoption of new languages and frameworks is increasingly quicker, without significant technical barriers.
- There is an increased use of automation and infrastructure-as-code (IaC) practices.
- Third parties, both in the shape of external providers and as dependencies in code, have become a major part of any CI/CD ecosystem, with the integration of a new service typically requiring no more than adding 1–2 lines of code.

These characteristics allow faster, more flexible, and diverse software delivery. However, they have also reshaped the attack surface with a multitude of new avenues and opportunities for attackers.

CI/CD Services Increasingly Targeted for Attacks

Adversaries of all sophistication levels are shifting their attention to CI/CD, realizing CI/CD services provide an efficient path to reaching an organization's crown jewels. The industry is witnessing a significant rise in the amount, frequency, and magnitude of incidents and attack vectors focusing on abusing flaws in the CI/CD ecosystem, such as the following events:

- **SolarWinds build system compromise:** Spread malware to 18,000 customers.
- **Codecov breach:** Led to exfiltration of secrets stored within environment variables in thousands of build pipelines across numerous enterprises.
- **PHP breach:** Resulted in publishing a malicious version of PHP containing a backdoor.
- **Dependency Confusion flaw:** Affected dozens of large enterprises and abused flaws in the way external dependencies are fetched to run malicious code on developer workstations and build environments.
- **ua-parser-js, coa, and rc NPM package compromises:** With millions of weekly downloads each, resulted in malicious code running on millions of build environments and developer workstations.

While attackers have adapted their techniques to the new realities of CI/CD, many defender organizations are still early in their efforts to find the right ways to detect, understand, and manage the risks associated with these environments. Seeking the right balance between optimal security and engineering velocity, security teams are searching for the most effective security controls that will allow engineering to remain agile without compromising on security.

CI/CD Security Risks Initiative

This guide helps defenders identify focus areas for securing their CI/CD ecosystem. It highlights outcomes from extensive research into CI/CD-associated attack vectors and provides an analysis of high-profile breaches and security flaws.

This initiative was possible thanks to the numerous industry experts across multiple verticals and disciplines who collaborated with us on this document. Their knowledge, experience, and insights ensure the guide's relevance to today's threat landscape, risk surface, and challenges that defenders face in dealing with these risks. We are grateful to them for their contributions, reviews, and validations in helping us publish this document.

Authors

Omer Gil
Director of AppSec Research
Cortex Cloud

Daniel Krivelevich
CTO Application Security
Prisma Cloud

Reviewers

Iftach Ian Amit
Advisory CSO at Rapid7

Jonathan Jaffe
CISO at Lemonade Insurance

Ron Peled
Founder & CEO at ProtectOps
Former CISO at LivePerson

Hiroki Suezawa
Security Engineer at Mercari, Inc.

Noa Ginzbursky
DevOps Engineer at Prisma Cloud AppSec

Jonathan Claudius
Director of Security Assurance at Mozilla

Adrian Ludwig
Chief Trust Officer at Atlassian

Ty Sbano
CISO at Vercel

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

Asi Greenholts
Security Researcher at Prisma Cloud AppSec

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

Travis McPeak
Head of Product Security at Databricks

Astha Singhal
Director, Information Security at Netflix

Tyler Young
Head of Security at Relativity

Guide to Understanding the Risks

Each CI/CD security risk outlined in this guide follows a consistent structure:

- **Definition:** Concise definition about the nature of the risk.
- **Description:** Explanation of the context and adversary motivation.
- **Impact:** Details around the potential impact the risk can have on an organization.
- **Recommendations:** A set of measures and controls to help optimize your organization's CI/CD posture in relation to the risk in question.
- **References:** Real-world examples and precedents in which the risk in question was exploited.

This structure was compiled from extensive research and analysis based on:

- Analysis of the architecture, design, and security posture of hundreds of CI/CD environments across multiple verticals and industries.
- Profound discussions with industry experts.
- Publications detailing incidents and security flaws within the CI/CD security domain. Examples are provided where relevant.

List of the Top 10 CI/CD Security Risks

CICD-SEC-1

Insufficient Flow Control Mechanisms

Definition

This risk refers to when CI/CD environments lack the guardrails that enforce additional approval or review. When this happens, attackers can obtain permissions to a system within the CI/CD process—such as source code management (SCM), CI, and artifact repository—to push malicious code or artifacts down the pipeline.

Description

CI/CD flows are designed for speed. A developer can create new code on their machine and get it into production within minutes, often with full reliance on automation and minimal human involvement. CI/CD processes are essentially the highway to highly gated and secured production environments. Organizations, therefore, must continuously introduce measures and controls to ensure that no single entity (human or application) can push code or artifacts through the pipeline without being required to undergo a strict set of reviews and approvals.

Impact

An attacker with access to the SCM, CI, or systems further down the pipeline can abuse insufficient flow control mechanisms to deploy malicious artifacts.

Once code is created, the artifacts pass through the pipeline, potentially all the way to production without any approval or review. For example, an adversary might:

- Push code to a repository branch, which is automatically deployed through the pipeline to production.
- Push code to a repository branch and then manually trigger a pipeline that ships the code to production.
- Directly push code to a utility library that code running in a production system uses.
- Abuse an automerge rule in the CI that automatically merges pull requests that meet a predefined set of requirements, pushing malicious unreviewed code.
- Abuse insufficient branch protection rules, such as excluding specific users or branches, to bypass branch protection and push malicious unreviewed code.
- Upload an artifact, such as a package or container, to an artifact repository in the guise of a legitimate artifact the build environment created. In this scenario, a lack of controls or verifications can result in a pipeline picking up the artifact and deploying it to production.
- Access production and directly change application code or infrastructure (such as the AWS Lambda function) without any additional approval or verification.

Recommendations

Establish pipeline flow control mechanisms to ensure that no single entity (human, programmatic, or both) can ship sensitive code and artifacts through the pipeline without external verification or validation. You can achieve this by implementing the following measures:

- **Configure branch protection rules** on branches hosting code used in production and other sensitive systems. Where possible, avoid excluding user accounts or branches from branch protection rules. For user accounts granted permission to push unreviewed code to a repository, ensure such accounts do not have permission to trigger the deployment pipelines connected to the repository in question.
- **Limit the usage of automerge rules,** wherever they are in use. These rules apply to minimal amounts of context. Review all automerge rule code thoroughly to ensure the rules cannot be bypassed and avoid importing third-party code in the automerge process.
- **Prevent accounts from triggering production build and deployment pipelines** without additional approval or review, where applicable.
- **Allow artifacts to flow through the pipeline** only on the condition that a preapproved CI service account created them. Prevent artifacts uploaded by other accounts from flowing through the pipeline without secondary review and approval.
- **Detect and prevent drifts and inconsistencies between code** running in production and its CI/CD origin, as well as modify any resource that contains a drift.

References

- **PHP Git repository:** A backdoor was planted in the repository. The attackers pushed malicious unreviewed code directly to the PHP main branch, resulting in a formal PHP version spreading to all PHP websites.[1]

1. Nikita Popov, "Update on git.php.net incident," PHP Lists, April 6, 2021.

- **Homebrew:** RyotaK bypassed automerge rules. They used an automerge rule to merge insignificant changes that were susceptible to bypass into the main branch so adversaries could merge malicious code into the project.[2]
- **GitHub Actions:** The flaw leveraged GitHub Actions to bypass the required reviews mechanism and push unreviewed code to a protected branch.[3]

CICD-SEC-2

Inadequate Identity and Access Management

Definition

Inadequate identity and access management (IAM) risks stem from the challenges of managing vast amounts of identities across different systems in the engineering ecosystem, from source control to deployment. Poorly managed identities—both human and programmatic accounts—increase the potential and extent of damage when compromised.

Description

Software delivery processes consist of connected systems for moving code and artifacts from development to production. Each system provides multiple methods of access and integration—including username and password, personal access token, marketplace application, OAuth applications, plugins, and SSH keys. The different types of accounts and access methods can potentially have their own unique provisioning method, set of security policies, and authorization model. This complexity creates challenges in managing identities throughout the entire identity lifecycle and ensuring their permissions align with the principle of least privilege.

Also, in a typical environment, the average SCM, CI, or user account is highly permissive, because security teams traditionally have not prioritized their focus on these systems. Engineers mostly use these identities because they require the flexibility to create major changes in code and infrastructure.

The major concerns and challenges around IAM within the CI/CD ecosystem include:

- **Overly permissive identities:** The difficulty of enforcing the principle of least privilege for both user and service accounts. In practice, scoping an identity's access within source code management to only the necessary repositories and permissions is a complex but essential security control.
- **Stale identities:** Employees and systems that are inactive or no longer require access but have not had their human and programmatic accounts deprovisioned within all CI/CD systems.
- **Local identities:** Systems that do not have federated access with a centralized Internal Developer Platform (IDP) create identities managed locally within the system in question. Local accounts create challenges in enforcing consistent security policies (e.g., password policy, lockout policy, and multifactor authentication [MFA]), as well as properly deprovisioning access across all systems, such as when an employee leaves an organization.
- **External identities:**
  - **Employees registered with an email address from a domain that is unowned or managed by the organization:** In this scenario, the account security is highly dependent on the security of the external accounts they are assigned to. Because the organization does not manage these accounts, they are not necessarily compliant with the organization's security policy.

2. Markus Reiter, "Security Incident Disclosure," Homebrew, April 21, 2021.
3. Alvaro Muñoz, "Keeping your GitHub Actions and workflows secure Part 4: New vulnerability patterns and mitigation strategies," GitHub, January 16, 2025.

- **External collaborators:** When external collaborators are granted system access, the system's security level is derived from the level of the external collaborator's work environment, outside the organization's control.

- **Self-registered identities:** In systems that allow self-registration, a valid domain address is often the only prerequisite for access to CI/CD systems. Using a default or base set of permissions significantly expands the potential attack surface.
- **Shared identities:** Identities shared between human users, applications, or both increase the footprint of their credentials and create challenges with accountability in case of a potential investigation.

Impact

The large number of identities in a CI/CD ecosystem, including both human users and automated services, creates a significant security challenge. This risk is amplified by inadequate identity management and the common use of accounts with excessive permissions. Consequently, the compromise of a single account can grant an attacker substantial control, creating a direct path to breach the production environment.

Recommendations

- **Conduct a continuous analysis and mapping of identities** across all systems within the engineering ecosystem. For each identity, map the identity provider and levels of permissions both granted and used. Ensure the analysis covers programmatic access methods.
- **Remove unnecessary permissions for ongoing work** for each identity across the systems in your environment.
- **Determine an acceptable period for disabling or removing stale accounts** and disable or remove any identity that has surpassed the predetermined period of inactivity.
- **Avoid creating local user accounts.** Instead, create and manage identities using a centralized organization component (IdP). Whenever local user accounts are in use, ensure that you disable and remove the accounts that no longer require access and that security policies around all existing accounts match the organization's policies.
- **Continuously map all external collaborators** and align their identities with the principle of least privilege. When possible, grant permissions for both human and programmatic accounts with a predetermined expiration date, and then disable their account after the work is done.
- **Prevent employees from using their personal email addresses** or any address that belongs to an unowned and unmanaged domain by the organization on an SCM, CI, or any other CI/CD platform. Continuously monitor for nondomain addresses across your systems and remove noncompliant users.
- **Don't allow users to self-register to systems.** Instead, grant permission on an as-needed basis.
- **Don't grant base permissions to all users in a system** or to large groups where user accounts are automatically assigned.
- **Avoid using shared accounts.** Create dedicated accounts for each specific context, and grant the exact set of permissions required for the context in question.

References

- **Stack Overflow TeamCity:** The build server was compromised when the attacker was able to escalate their privileges in the environment because newly registered accounts were assigned administrative privileges upon access to the system.[4]

4. Dean Ward, "A deeper dive into our May 2019 security incident," Stack Overflow, January 25, 2021.

- **Mercedes Benz:** The source code was leaked after a self-maintained internet-facing GitLab server became available for access by self-registration.[5]
- **GitLab server:** The New York State government's self-managed GitLab server was exposed to the internet so anyone could self-register and log into the system that stored sensitive secrets.[6]
- **Gentoo Linux:** Malware was added to the distribution source code after the GitHub account password of a project maintainer was compromised.[7]

CICD-SEC-3

Dependency Chain Abuse

Definition

Dependency chain abuse risks refer to an attacker's ability to abuse flaws related to how engineering workstations and build environments fetch code dependencies. This type of risk results in inadvertently fetching a malicious package and executing locally when pulled.

Description

Using self-written code to manage dependencies and external packages is becoming complex, considering the number of systems involved in the process across an organization's development contexts. Packages are often fetched using a dedicated client per programming language. These packages are typically from both self-managed package repositories (e.g., JFrog Artifactory) and language-specific SaaS repositories (e.g., the Node.js, npm, and npm registry; PyPI for Python's pip, and RubyGems for Ruby gems).

Many organizations endeavor to detect usage of packages with known vulnerabilities and conduct static analysis of both self-written and third-party code. However, in the context of using dependencies, an equally important set of controls is required to secure the dependency ecosystem. These controls involve securing the process that defines how dependencies are pulled.

Inadequate configurations can cause an unsuspecting engineer—or worse, the build system—to download a malicious package, not the one intended to be pulled. In many cases, the package is downloaded and then immediately executed as intended by preinstalled scripts and similar processes.

This context has the following main attack vectors:

- **Dependency confusion:** Malicious packages are published in public repositories with the same name as internal package names. The goal is to trick clients into downloading the malicious package rather than the private one.
- **Dependency hijacking:** Control is obtained of a package maintainer account on the public repository to upload a new, malicious version of a widely used package. The intent is to compromise unsuspecting clients who pull the latest version of the package.
- **Typosquatting:** Malicious packages are published with names similar to those of popular packages. The intent is that a developer will misspell a package name and unintentionally fetch the typosquatted package.
- **Brandjacking:** Malicious packages are published in a consistent way as the naming convention or other characteristics of a specific brand's package. The goal is to lure unsuspecting developers to fetch these packages that are falsely associated with the trusted brand.

5. Catalin Cimpanu, "Mercedes-Benz onboard logic unit (OLU) source code leaks online," ZDNET, May 18, 2020.
6. Zack Whittaker, "An internal code repo used by New York State's IT office was exposed online," TechCrunch, June 24, 2021.
7. "Project: Infrastructure/Incident reports/2018-06-28 GitHub," Gentoo Linux Wiki, June 28, 2018.

Impact

The objective of adversaries that upload packages to public package repositories using one of the aforementioned techniques is to execute malicious code on a host pulling the package. This could be a developer's workstation or a build server pulling the package. Once the malicious code is running, the attacker can use it for credentials theft and lateral movement within the environment it is executed in.

Another potential scenario is for the attacker's malicious code to make its way to production environments from the build server. In many cases, the malicious package might continue to also maintain the original, safe functionality the user was expecting, resulting in a lower probability of discovery.

Recommendations

A wide range of mitigation methods are specific to the configuration of the different language-specific clients and the way internal proxies and external package repositories are used.

All recommended controls share the same guiding principles:

- **Do not allow any client that pulls code packages to fetch packages** from the internet or untrusted sources. Instead, implement the following controls:
  - **Whenever third-party packages are pulled from an external repository,** ensure all packages are pulled through an internal proxy, not from the internet. This way, you can deploy additional security controls at the proxy layer, as well as provide investigative capabilities around packages that are pulled in case of a security incident.
  - **Where applicable, disallow the pulling of packages directly from external repositories.** Configure all clients to pull packages from internal repositories that contain prevetted packages, and establish a mechanism to verify and enforce this client configuration.
- **Enable both checksum and signature verifications for pulled packages.**
- **Avoid configuring clients to pull the latest version of a package.** Instead, configure a prevetted version or range of versions. Use framework-specific techniques to continuously lock the package version required in your organization to a stable and secure version.
- **Align with the following scopes:**
  - Register all private packages under the organization's scope.
  - Use the package's scope for all code referencing a private package.
  - Force clients to fetch packages that are under your organization's scope solely from your internal registry.
- **Ensure that a separate context exists for installation scripts** when executing them as part of the package installation. These scripts should not have access to secrets and other resources in other stages of the build process.
- **Make sure internal projects always contain the configuration files of package managers** (such as .npmrc in NPM) within the code repository of the project to override any insecure configuration that might exist on a client fetching the package.
- **Avoid publishing names of internal projects in public repositories.**
- **Place an appropriate level of focus around detection, monitoring, and mitigation** so that, in case of an incident, it is identified as quickly as possible and has a minimal amount of potential damage. As a rule, complete prevention of third-party chain abuse is far from trivial, given the number of package managers and configurations in use simultaneously.

In this context, properly harden all relevant systems according to the guidelines in the "CICD-SEC-7: Insecure System Configuration" risk.

References

- **Attack vector tricks package managers and proxies:** Alex Birsan wrote how he created an attack vector that tricked package managers and proxies into fetching a malicious package from a public repository instead of the intended package of the same name from an internal repository.[8]
- **Amazon, Zillow, Lyft, and Slack Node.js apps:** Threat actors targeted these apps by using the Dependency Confusion vulnerability.[9]
- **ua-parser-js NPM library:** This library, with 9 million downloads a week, was hijacked to launch cryptominers and steal credentials.[10]
- **coa NPM library:** This library, also with 9 million downloads a week, was hijacked to steal credentials.[11]
- **rc NPM library:** This library, with 14 million downloads a week, was hijacked to steal credentials.[12]

CICD-SEC-4

Poisoned Pipeline Execution

Definition

Poisoned pipeline execution (PPE) risks refer to the ability of an attacker with access to source control systems—and without access to the build environment—to manipulate the build process. The attacker injects malicious code or commands into the build pipeline configuration, poisoning the pipeline and running malicious code as part of the build process.

Description

The PPE vector abuses permissions against an SCM repository, causing a CI pipeline to execute malicious commands. Users with permission to manipulate the CI configuration files—or other files the CI pipeline job relies on—can modify them to contain malicious commands, ultimately "poisoning" the CI pipeline that executes these commands.

Pipelines that execute unreviewed code, such as those that are triggered off pull requests or commits to arbitrary repository branches, are more susceptible to PPE. The reason is that these scenarios, by design, contain code that has not undergone any reviews or approvals. After an attacker executes the malicious code within the CI pipeline, they can conduct a wide array of malicious operations, all within the context of the pipeline's identity.

Direct PPE

In a direct PPE (D-PPE) scenario, an attacker modifies the CI configuration file in a repository they have access to. They either push the change directly to an unprotected remote branch on the repo or submit a push request with the change from a branch or a fork. The push events trigger the CI pipeline execution, which the commands in the modified CI configuration file define. This way, the attacker's malicious commands will ultimately run in the build node after the build pipeline is triggered.

Indirect PPE

A D-PPE is unavailable to an adversary that has access to an SCM repository if:

- The pipeline is configured to pull the CI configuration file from a separate, protected branch in the same repository.

8. Alex Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," Medium, February 9, 2021.
9. Lawrence Abrams, "Malicious NPM packages target Amazon, Slack with new dependency attacks," BleepingComputer, March 2, 2021.
10. Katarina Blagojevic, "Embedded malware in ua-parser-js," GitHub, July 28, 2023.
11. Katarina Blagojevic, "Embedded malware in coa," GitHub, January 11, 2023.
12. Katarina Blagojevic, "Embedded malware in rc," GitHub, January 11, 2023.

- The CI configuration file is stored in a separate repository from the source code, without the option for a user to edit it directly.
- The CI build is defined in the CI system itself, instead of in a file stored in the source code.

In such a scenario, the attacker can still poison the pipeline by injecting malicious code into files referenced by the pipeline configuration file, for example:

- **The make utility:** Executes commands defined in the Makefile.
- **Scripts referenced from within the pipeline configuration file:** These scripts are stored in the same repository as the source code itself (e.g., Python myscript.py, where an attacker would manipulate myscript.py).
- **Code tests:** Testing frameworks running on application code within the build process rely on dedicated files, stored in the same repository as the source code itself.
- **Attackers that are able to manipulate the code responsible for testing are also able to run malicious commands inside the build.**
- **Automatic tools:** Linters and security scanners used in the CI also commonly rely on a configuration file that resides in the repository. These configurations often involve loading and running external code from a location defined inside the configuration file.

Rather than poisoning the pipeline by inserting malicious commands directly into the pipeline definition file, in indirect PPE (I-PPE), an attacker injects malicious code into files referenced by the configuration file. The malicious code is ultimately executed on the pipeline node once the pipeline is triggered and runs the commands declared in the files in question.

Public-PPE

Execution of a PPE attack requires access to the repository that hosts the pipeline configuration file or to files it references. In most cases, the permission to do so is given to organization members, mainly engineers. Attackers would then typically have to be in possession of an engineer's permission to the repository to execute a D-PPE or I-PPE attack.

However, in some cases, poisoning CI pipelines is available to anonymous attackers on the internet. Public repositories, such as open-source projects, often allow any user to contribute, usually by creating pull requests and suggesting changes to the code. These projects are commonly tested and built automatically by using a CI solution, similar to private projects.

If the CI pipeline of a public repository runs unreviewed code suggested by anonymous users, it is susceptible to a public PPE (3PE) attack. This attack exposes internal assets, such as secrets of private projects, in cases where the pipeline of the vulnerable public repository runs on the same CI instance as private ones.

Example 1: Credential Theft via D-PPE in GitHub Actions

In this example, a GitHub repository is connected with a GitHub Actions workflow that fetches the code, builds it, runs tests, and eventually deploys artifacts to AWS. When new code is pushed to a remote branch in the repository, the code, including the pipeline configuration file, is fetched by the runner (the workflow node).

A D-PPE attack might be carried out as follows:

1. An attacker creates a new remote branch in the repository. There, they update the pipeline configuration file with malicious commands intended to access AWS credentials scoped to the GitHub organization. They then send them to a remote server.
2. The pushed update triggers the pipeline to fetch the code from the repository, including the malicious pipeline configuration file.

3. The pipeline runs according to the configuration file the attacker poisoned. Based on the attacker's malicious commands, the AWS credentials stored as repository secrets are loaded into memory.
4. The pipeline proceeds to execute the attacker's commands, sending the AWS credentials to a server that the attacker controls.
5. The attacker can then use the stolen credentials to access the AWS production environment.

Example 2: Credential Theft via Indirect-PPE (GitHub Actions)

In this example, a Jenkins pipeline fetches code from the repository, builds it, runs tests, and eventually deploys it to AWS. In the pipeline configuration, the file that describes the pipeline—the Jenkinsfile—is always fetched from the main branch in the repository, which is protected.

Therefore, the attacker cannot manipulate the build definition, because fetching secrets stored on the Jenkins credential store or running the job on other nodes is not possible. This does not mean the pipeline is risk free. In the build stage of the pipeline, AWS credentials are loaded as environment variables, making them available only to the commands running in this stage.

As shown in this example, the `make` command, which is based on the contents of the Makefile (also stored in the repository), runs as part of this stage.

An I-PPE attack might be carried out as follows:

1. An attacker creates a pull request in the repository and appends malicious commands to the Makefile file.
2. Because the pipeline is configured to be triggered upon any push request against the repository, the Jenkins pipeline is triggered and fetches the code from the repository, including the malicious Makefile.
3. The pipeline runs according to the configuration file stored in the main branch. When it gets to the build stage, it loads the AWS credentials into environment variables as defined in the original Jenkinsfile. Then, it runs the `make build` command, which executes the malicious command that was added into the Makefile.
4. The malicious build function defined in the Makefile is executed, sending the AWS credentials to a server controlled by the attacker.
5. The attacker can then use the stolen credentials to access the AWS production environment.

Impact

In a successful PPE attack, attackers execute malicious unreviewed code in the CI. This provides the attacker with the same abilities and level of access as the build job, including:

- Access to any secret available to the CI job, such as secrets injected as environment variables or additional secrets stored in the CI. Being responsible for building code and deploying artifacts, CI/CD systems typically contain dozens of high-value credentials and tokens, such as to a cloud provider, artifact registries, and the SCM repository itself.
- Access to external assets that the job node has permissions to, such as files stored in the node's file system, or credentials to a cloud environment that is accessible through the underlying host.
- The ability to ship code and artifacts further down the pipeline, in the guise of legitimate code built by the build process.
- The ability to access additional hosts and assets in the network or environment of the job node.

Recommendations

Preventing and mitigating the PPE attack vector involves multiple measures that span both SCM and CI systems:

- **Execute pipelines running unreviewed code on isolated nodes,** not exposed to secrets and sensitive environments.
- **Evaluate the need for triggering pipelines** on public repositories from external contributors. Where possible, refrain from running pipelines originating from forks, and consider adding controls, such as requiring manual approval for pipeline execution.
- **Correlate a branch protection rule in the SCM for each branch that is configured** to trigger a pipeline in the CI system. This applies particularly to sensitive pipelines, such as those that are exposed to secrets.
- **Before the pipeline runs, review each CI configuration file** to prevent any manipulation to it to run malicious code in the pipeline.

  Alternatively, manage the CI configuration file in a remote branch, separate from the branch that contains the code being built in the pipeline. Configure the remote branch as protected.
- **Remove permissions granted** on the SCM repository from users that do not need them.
- **Verify each pipeline only has access to the credentials it needs** to fulfill its purpose. The credentials must have the minimum required privileges.

References

- **Direct-PPE and 3PE attack vector techniques:** Learn about exploitation techniques of the Direct-PPE and 3PE attack vectors, targeting pipelines running unreviewed code.[13]
- **PPE:** The PPE vector abuses permissions against an SCM repository, in a way that causes a CI pipeline to execute malicious commands.[14]
- **AWS Indirect-PPE vulnerability:** The exposure happened in the CodeBuild pipeline of an AWS website.[15] It enabled anonymous attackers to modify a script run by the build configuration file with the creation of a pull request, resulting in compromised deployment credentials.
- **GitHub Actions:** GitHub Actions were used to mine cryptocurrency through pull requests that contained malicious code.[16]
- **Terraform provider cmdexec:** The cmdexec provider for Terraform demonstrates a poisoned pipeline vector by enabling the execution of arbitrary OS commands when the `terraform plan` command—an action often considered safe and read-only—is run within the CI/CD pipeline.[17]
- **The terraform plan command:** Running the `terraform plan` command on untrusted code in the CI/CD can lead to remote code execution.[18]
- **Teleport's CI implementation:** A vulnerability found in Teleport's CI implementation allowed attackers from the internet to execute a Direct-3PE attack.[19] The attackers could create a pull request in a public GitHub repository linked with a Drone CI pipeline and then modify the CI configuration file to execute a malicious pipeline.

13. Tyler Welton, "Exploiting Continuous Integration (CI) and Automated Build Systems," DEF CON 25 video on YouTube. Accessed September 4, 2025.
14. "CICD-SEC-4: Poisoned Pipeline Execution (PPE)," Open Worldwide Application Security Project, updated November 15, 2022.
15. xxsfox, "Build Pipeline Security," sprocketfox.io, February 18, 2021.
16. Tib, "Crypto-mining attack in my GitHub actions through Pull Request," DEV Community, October 14, 2021.
17. Hiroki Suezawa, "Terraform-provider-cmdexec," GitHub, March 13, 2020.
18. Alex Kaskasoli, "Terraform Plan RCE," alex.kaskaso.li, May 11, 2021.
19. Walt Della, "Anatomy of a Cloud Infrastructure Attack via a Pull Request," Teleport, September 16, 2021.

CICD-SEC-5

Insufficient Pipeline-Based Access Controls

Definition

Pipeline execution nodes have access to numerous resources and systems within and outside the execution environment. When adversaries run malicious code within a pipeline, they use insufficient pipeline-based access controls (PBAC) risks to abuse the permission granted to the pipeline for moving laterally within or outside the CI/CD system.

Description

Pipelines are the heart of CI/CD. Nodes that execute pipelines carry out the commands specified in the pipeline configuration and, by doing so, conduct an array of sensitive activities:

- Access the source code, and then build and test it.
- Obtain secrets from various locations, such as from environment variables, vaults, dedicated cloud-based identity services (like the AWS metadata service), and other locations.
- Create, modify, and deploy artifacts.

PBAC refers to the context in which each pipeline—and each step within that pipeline—is running. Given the highly sensitive and critical nature of each pipeline, your team must limit each pipeline to the exact set of data and resources they need to access. Ideally, they must restrict each pipeline and step so that it ensures that, if an adversary is able to execute malicious code within the context of the pipeline, the extent of potential damage is minimal.

PBAC includes controls relating to numerous elements related to the pipeline execution environment:

- Access within the pipeline execution environment to code, secrets, environment variables, and other pipelines.
- Permissions to the underlying host and other pipeline nodes.
- Ingress and egress filters to the internet.

Impact

A piece of malicious code that is able to run in the pipeline execution node context has full permissions of the pipeline stage it runs in. It can access secrets and the underlying host, as well as connect to any of the systems the pipeline in question can access. This might lead to exposure of confidential data, lateral movement within the CI environment—potentially accessing servers and systems outside the CI environment—and deployment of malicious artifacts down the pipeline, including to production.

The granularity of the PBAC in the environment determines the extent of the potential damage of a scenario where an adversary is able to compromise pipeline execution nodes or inject malicious code into the build process.

Recommendations

- **Use shared nodes only for pipelines with identical levels of confidentiality.** Do not use them for pipelines with different levels of sensitivity or that require access to different resources.
- **Apply the principle of least privilege to CI/CD secrets** to ensure each pipeline and step can access only the secrets essential for its task.
- **Revert the execution node to its pristine state after each pipeline execution.**
- **Apply the principle of least privilege** and ensure the pipeline's OS user has only the essential permissions required on the execution node.

- **Limit permissions on the controller node for CI and CD pipeline jobs.** Where applicable, run pipeline jobs on a separate, dedicated node.
- **Appropriately patch the execution node as needed.**
- **Configure network segmentation in the environment the job is running on** to allow access to the execution node for only the resources it requires within the network. Where possible, refrain from granting unlimited access toward the internet to build nodes.
- **When installation scripts are executed as part of the package installation,** ensure that a separate context exists for those scripts that do not have access to secrets and other sensitive resources available in other stages in the build process.

References

- **Codecov:** A popular code coverage tool used in the CI was compromised and used to steal environment variables from builds.[20]
- **Amazon, Zillow, Lyft, and Slack Node.js apps:** Threat actors targeted these apps by using the Dependency Confusion vulnerability.[21] The victim organizations of the Dependency Confusion attacks had malicious code executed on CI nodes, allowing the adversary to move laterally within the environment and abuse insufficient PBAC.
- **Teleport's CI implementation:** A vulnerability found in Teleport's CI implementation allowed attackers from the internet to execute a Direct-3PE attack.[22] The attackers could run a privileged container and escalate to root privilege on the node itself—leading to secret exfiltration, release of malicious artifacts, and access to sensitive systems.

CICD-SEC-6

Insufficient Credential Hygiene

Definition

Insufficient credential hygiene risks deal with an attacker's ability to obtain and use various secrets and tokens spread throughout the pipeline. This access occurs because of flaws related to access controls around the credentials, insecure secret management, and overly permissive credentials.

Description

CI/CD environments are built of multiple systems that communicate and authenticate against each other. They create great challenges around protecting credentials due to the large variety of contexts in which credentials can exist.

The application at runtime uses application credentials. Pipelines use credentials to production systems to deploy infrastructure, artifacts, and apps to production. And, engineers use credentials as part of their testing environments and within their code and artifacts.

This variety of contexts, paired with the large number of methods and techniques for storing and using them, creates a large potential for insecure usage of credentials. The following major flaws affect credential hygiene.

20. Jerrod Engelberg, "Bash Uploader Security Update," Codecov, April 15, 2021.
21. Abrams, "Malicious NPM packages target Amazon, Slack with new dependency attacks."
22. Della, "Anatomy of a Cloud Infrastructure Attack via a Pull Request."

Code Containing Credentials Pushed to an SCM Repository Branch

This flaw can be by mistake, without noticing the existence of the secret in the code, or deliberately without understanding the risk of doing that. From that moment, the credentials are exposed to anyone with read access to the repository. Even if they are deleted from the branch it was pushed into, they continue to appear in the commit history, available for anyone to view who has repository access.

Credentials Used Insecurely Inside Build and Deployment Processes

These credentials are used to access code repositories, read from and write to artifact repositories, and deploy resources and artifacts to production environments. Given the large number of pipelines and target systems they need access to, it is imperative to understand:

- In which context and with which method is each set of credentials used?
- Can each pipeline access only the credentials it needs to fulfill its purpose?
- Can unreviewed code flowing through the pipeline access the credentials?
- How are these credentials called and injected to the build: accessible only during runtime and only from the contexts where they are required?

Credentials in Container Image Layers

Credentials that were required only for building the image still exist in one of the image layers. They are now available to anyone who is able to download the image.

Credentials Printed to Console Output

Credentials used in pipelines are often printed to the console output, deliberately or inadvertently. This might leave credentials exposed in cleartext in logs, available to anyone to view who has access to the build results. These logs can potentially flow to log management systems, expanding their exposure surface.

Unrotated Credentials

Because the credentials are spread all over the engineering ecosystem, they are exposed to a large number of employees and contractors. Failing to rotate credentials results in a continuously growing number of people and artifacts that possess valid credentials. This is especially true for credentials used by pipelines. For example, deploy keys are often managed using the "If it is not broken, do not fix it" directive, leaving valid credentials unrotated for many years.

Impact

Credentials are the most sought-after object by adversaries. They seek to use them for accessing high-value resources or for deploying malicious code and artifacts. In this context, engineering environments provide attackers with multiple avenues to obtain credentials. The high potential for human error, paired with knowledge gaps around secure credentials management and concern over credential rotation breaking processes, risks compromising organizations' high-value resources when their credentials are exposed.

Recommendations

- **Establish procedures to continuously map credentials** found across the different systems in the engineering ecosystem—from code to deployment. Ensure each set of credentials follows the principle of least privilege and has been granted the exact set of permission needed by the service using it.
- **Avoid sharing the same set of credentials across multiple contexts.** This increases the complexity of achieving the principle of least privilege, as well as having a negative effect on accountability.

- **Prefer using temporary credentials over static credentials.** If you must use static credentials, establish a procedure to periodically rotate all static credentials and detect stale credentials.
- **Configure usage of credentials to be limited to predetermined conditions,** like scoping to a specific source IP or identity, to ensure that, if a compromise occurs, exfiltrated credentials cannot be used outside your environment.
- **Detect secrets pushed to and stored on code repositories.** Use such controls as an integrated development environment (IDE) plugin to identify secrets used in local changes, automatic scanning upon each code push, and periodical scans on the repository and its past commits.
- **Scope secrets used in CI/CD systems** to allow each pipeline and step to have access to only the secrets they require.
- **Use built-in vendor options or third-party tools to prevent secrets from being printed** to console outputs of future builds. Ensure all existing outputs do not contain secrets.
- **Verify that secrets are removed from any type of artifact,** such as from layers of container images, binaries, or Helm charts.

References

- **Codecov compromise:** Thousands of credentials, stored as environment variables, were stolen by attackers by compromising Codecov, a popular code coverage tool used in the CI.[23]
- **Travis CI compromise:** Travis CI injected secure environment variables of public repositories into pull request builds, causing them to be susceptible to compromise by anonymous users issuing pull requests against public repositories.[24]
- **TeamCity Build server compromise:** An attacker compromised the TeamCity Build server of Stack Overflow and was able to steal secrets due to their insecure storage method.[25]
- **Samsung compromise:** Samsung exposed overly permissive secrets in public GitLab repositories.[26]
- **Uber:** Attackers accessed Uber's private GitHub repositories that contained permissive and shared AWS tokens, leading to data exfiltration of millions of drivers and passengers.[27]
- **Homebrew:** The Homebrew Jenkins instance revealed environment variables of executed builds, including a GitHub token that allowed an attacker to make malicious changes to the Homebrew project itself.[28]

CICD-SEC-7

Insecure System Configuration

Definition

Insecure system configuration risks stem from flaws in the security settings, configuration, and hardening of various systems across the pipeline, such as an SCM, CI, and artifact repository. Attackers often look for "low-hanging fruits," such as these systems so they can expand their foothold in the environment.

23. Engelberg, "Bash Uploader Security Update."
24. "Security Bulletin," Travis CI, September, 2021.
25. Ward, "A deeper dive into our May 2019 security incident."
26. Zack Whittaker, "Samsung spilled SmartThings app source code and secret keys," TechCrunch, May 8, 2019.
27. "Uber Revised Consent Analysis," Federal Register/Vol. 83, No. 80, April 25, 2018.
28. Eric Holmes, "How I gained commit access to Homebrew in 30 minutes," Medium, August 7, 2018.

Description

CI/CD environments consist of multiple systems, provided by various vendors. To optimize CI/CD security, defenders are required to place strong emphasis both on the code and artifacts that flow through the pipeline, and on each system's posture and resilience. In a similar way to other systems storing and processing data, CI/CD systems involve various security settings and configurations on all levels: application, network, and infrastructure. These settings have a major influence on the security posture of CI/CD environments and their susceptibility to a potential compromise.

Adversaries of all levels of sophistication are always on the lookout for potential CI/CD vulnerabilities and misconfigurations that they can leverage to their benefit.

Potential hardening flaws include the following examples:

- A self-managed system, component, or both using an outdated version or lacking important security patches.
- A system having overly permissive network access controls.
- A self-hosted system that has administrative permissions on the underlying OS.
- A system with insecure system configurations. Configurations typically determine key security features having to do with authorization, access controls, logging and more. In many cases, the default set of configurations is not secure and requires optimization.
- A system with inadequate credential hygiene, such as default credentials that are not disabled and overly permissive programmatic tokens.

Using SaaS CI/CD solutions, rather than their self-hosted alternative, eliminates some of the potential risks associated with system hardening and lateral movement within the network. However, organizations are still required to be highly diligent in securely configuring their SaaS CI/CD solution. Each solution has its own set of unique security configurations and best practices that are essential for maintaining an optimal security posture.

Impact

An adversary can leverage a security flaw in one of the CI/CD systems to obtain unauthorized access to the system or worse—compromise the system and access the underlying OS. An attacker is then able to abuse these flaws to manipulate legitimate CI/CD flows, obtain sensitive tokens, and potentially access production environments. In some scenarios, these flaws might allow an attacker to move laterally within the environment and outside the context of CI/CD systems.

Recommendations

- **Maintain an inventory of systems and versions in use,** including mapping of a designated owner for each system. Continuously check for known vulnerabilities in these components. If a security patch is available, update the vulnerable component. Otherwise, consider removing the component or system. Or to reduce the potential impact of exploiting the vulnerability, restrict access to the system or its ability to perform sensitive operations.
- **Align network access to the systems with the principle of least-privileged access.**
- **Establish a process to periodically review all system configurations** for any settings that can affect the security posture of the system, and ensure all settings are optimal.
- **Ensure permissions to the pipeline execution nodes are granted according to the principle of least privilege.** A common misconfiguration in this context is around granting debug permissions on execution nodes to engineers. While this is a common practice in many organizations, consider that any user with the ability to access the execution node in debug mode can expose all secrets while they are loaded into memory. They can use the node's identity, effectively granting elevated permissions to any engineer with this permission.

References

- **SolarWinds:** The SolarWinds build system was compromised to spread malware through SolarWinds to 18,000 organizations.[29]
- **Backdoor planted in the PHP git repository.** The attackers pushed malicious unreviewed code directly to the PHP main branch, ultimately resulting in a formal PHP version being spread to all PHP users. The attack presumably originated in a compromise of the PHP self-maintained git server.[30]
- **Stack Overflow TeamCity:** An attacker compromised Stack Overflow's TeamCity build server that was accessible from the internet.[31]
- **Webmin build server:** Attackers compromised an unpatched Webmin build server and added a backdoor to the local copy of the code after it was fetched from the repository. This led to a supply chain attack on servers that use Webmin.[32]
- **Nissan:** This car manufacturer's source code leaked after a self-managed BitBucket instance left accessible from the internet with default credentials.[33]
- **Mercedes Benz:** This company's source code leaked after a self-maintained internet-facing GitLab server was made open for self-registration.[34]
- **New York State government:** A self-managed GitLab server of the New York State government was exposed to the internet, allowing anyone to self-register and log in to the system, which stored sensitive secrets.[35]

CICD-SEC-8

Ungoverned Usage of Third-Party Services

Definition

The CI/CD attack surface consists of an organization's organic assets, such as SCM or CI, and the third-party services that are granted access to those organic assets. Risks related to the ungoverned usage of third-party services rely on the ease of a third-party service being granted access to resources in CI/CD systems, expanding the organization's attack surface.

Description

It is rare to find an organization that does not have numerous third parties connected to its CI/CD systems and processes. Their ease of implementation, combined with their immediate value, has made third parties an integral part of the engineering day to day. The methods of embedding or granting access to third parties are becoming more diverse, and the complexities associated with implementing them are diminishing.

29. Oladimeji, Saheed, and Sean Michael Kerner, "SolarWinds hack explained: Everything you need to know," TechTarget, November 3, 2023.
30. Popov, "Update on git.php.net incident."
31. Ward, "A deeper dive into our May 2019 security incident."
32. "Hackers Planted Backdoor in Webmin, Popular Utility for Linux/Unix Servers," The Hacker News, August 20, 2019.
33. Catalin Cimpanu, "Nissan source code leaked online after Git repo misconfiguration," ZDNET, January 6, 2021.
34. Cimpanu, "Mercedes-Benz onboard logic unit (OLU) source code leaks online."
35. Whittaker, "An internal code repo used by New York State's IT office was exposed online."

For a common SCM, such as GitHub SaaS, third-party applications can be connected through one or more of these methods:

- GitHub Application.
- OAuth application.
- Provision an access token provided to a third-party application.
- Provision an SSH key provided to a third-party application.
- Configure webhook events to send to a third party.

Each method takes between seconds and minutes to implement. They also grant third parties numerous capabilities, from reading code in a single repository to fully administering the GitHub organization. Despite the potentially high level of permission these third parties are granted against the system, in many cases no special permissions or approvals are required by the organization before the actual implementation.

Build systems also allow easy integration of third parties. Integrating third parties into build pipelines is usually no more complex than adding 1–2 lines of code within the pipeline configuration file, or installing a plugin from the build system's marketplace (e.g., actions in GitHub Actions or Orbs in CircleCI). The third-party functionality is then imported and executed as part of the build process with full access to the resources available from the pipeline stage it is executed in.

Similar methods of connectivity are available in various forms across most CI/CD systems, creating a complex process of governing and maintaining least privilege around third-party usage across the entire engineering ecosystem. Organizations are grappling with the challenge of obtaining full visibility around which third parties have access to the different systems. They also struggle to understand the methods of access they have, the level of permission or access they have been granted, and the level of permissions or access they are using.

Impact

A lack of governance and visibility around third-party implementations prevents organizations from maintaining RBAC within their CI/CD systems. Given how permissive third parties tend to be, organizations are only as secure as the third parties they implement. Insufficient implementation of RBAC and least privilege around third parties, coupled with minimal governance and diligence around the process of third-party implementations create a significant increase of the organization's attack surface.

Given the highly interconnected nature of CI/CD systems and environments, attackers can use the compromise of a single third party to cause damage far outside the scope of the system the third party is connected to. For example, an adversary can use a third party with write permissions on a repository to push code to the repository that then triggers a build and runs their malicious code on the build system.

Recommendations

Implement governance controls around third-party services within every stage of the third-party usage lifecycle:

- **Approval:** Establish vetting procedures to ensure third parties granted access to resources anywhere across the engineering ecosystem are approved before being granted access to the environment. The level of permission they are granted must align with the principle of least privilege.
- **Integration:** Introduce controls and procedures to maintain continuous visibility over all third parties integrated to CI/CD systems, including:
    - Method of integration. Be sure to cover all integration methods for each system, including marketplace apps, plugins, OAuth applications, and programmatic access tokens.

  - › Level of permission granted to the third party.
  - › Level of permission in use by the third party.
- **Visibility over ongoing usage:** Limit and scope each third party to the specific resources it requires access to and remove unused, redundant, or both types of permissions. Run third parties that are integrated as part of the build process inside a scoped context with limited access to secrets and code, as well as with strict ingress and egress filters.
- **Deprovisioning:** Periodically review all integrated third parties and remove those that are no longer in use.

References

- **Codecov:** This popular code coverage tool used in the CI was compromised to steal environment variables from builds.[36]
- **DeepSource GitHub user account:** Attackers compromised a GitHub user account of a DeepSource (a static analysis platform) engineer.[37] Using the compromised account, they obtained the permissions of the DeepSource GitHub application, which granted them full access to the codebase of all DeepSource clients that installed the compromised GitHub application.
- **Waydev:** Attackers gained access to the database of Waydev, a git analytics platform. They stole their customers' GitHub and GitLab OAuth tokens.[38]

CICD-SEC-9

Improper Artifact Integrity Validation

Definition

Improper artifact integrity validation risks enable an attacker with access to one of the systems in the CI/CD process to push malicious (although seemingly benign) code or artifacts down the pipeline. The reason is because of insufficient mechanisms to ensure the validation of code and artifacts.

Description

CI/CD processes consist of multiple steps that are ultimately responsible for taking code from an engineer's workstation to production. Multiple resources are fed into each step, combining internal resources and artifacts with third-party packages and artifacts fetched from remote locations. The fact that the ultimate resource relies upon multiple sources across the steps, provided by multiple contributors, creates multiple entry points through which this ultimate resource can be tampered with.

If a tampered resource successfully infiltrated the delivery process, without raising suspicion or encountering security gates, it will most likely continue flowing through the pipeline to production in the guise of a legitimate resource.

Impact

Improper artifact integrity validation can be abused by an adversary with a foothold within the software delivery process to ship a malicious artifact through the pipeline. This ultimately results in the execution of malicious code—either on systems within the CI/CD process or worse—in production.

An adversary with a foothold in the software delivery process to ship a malicious artifact through the pipeline can abuse improper artifact integrity validation. The ultimate result is executing malicious code on systems within either the CI/CD process or worse—in production.

36. Engelberg, "Bash Uploader Security Update."
37. Jai Pradeesh, "Security incident on DeepSource's GitHub application," DeepSource Discuss, July 20, 2020.
38. "GitHub & GitLab OAuth - security update," Waydev, July 7, 2020.

Recommendations

Preventing improper artifact integrity validation risks requires a collection of measures across different systems and stages within the software delivery chain. Consider the following controls.

Validate the Resource Integrity

Implement processes and technologies to validate the integrity of resources from development to production. When a resource is generated, the process will include signing that resource using an external resource signing infrastructure. Before consuming the resource in the next steps down the pipeline, validate the resource's integrity against the signing authority.

In this context, consider the following prevalent measures among others:

- **Code signing:** SCM solutions provide the ability to sign commits using a unique key for each contributor. Your security team can then use this measure to prevent unsigned commits from flowing down the pipeline.
- **Artifact verification software:** Using tools to sign and verify code and artifacts helps to prevent unverified software from being delivered down the pipeline. For example, one such project is Sigstore, which was created by the Linux Foundation.
- **Configuration drift detection:** Measures aimed at detecting configuration drifts (such as cloud environment resources that are not managed using a signed IAC template) can potentially indicate resources that an untrusted source or process deployed.

Calculate the Resource Hash

Third-party resources fetched from build or deploy pipelines (such as imported or executed scripts as part of the build process) should follow a similar logic. That is, before using third-party resources, calculate the hash of the resource and then cross-reference it against the official published hash of the resource provider.

References

- **SolarWinds build system:** This system was hacked to spread malware through SolarWinds to 18,000 organizations.[39] The Orion software code was changed in the build system during the build process, leaving no trace in the codebase.
- **Codecov:** This popular code coverage tool used in the CI was compromised to steal environment variables from builds.[40] Attackers gained access to the Google Cloud Platform (now called Google Cloud) account that hosted the Codecov script and modified it to contain malicious code. A customer identified the attack by comparing the script hash stored on GitHub with the script downloaded from the Google Cloud Platform account.
- **PHP git repository:** A backdoor was planted in the PHP git repository that ultimately resulted in a formal PHP version spreading to all PHP users.[41] The attackers pushed malicious unreviewed code directly to the PHP main branch, committing the code as though it were made by known PHP contributors.
- **Webmin build server:** Attackers compromised this build server and added a backdoor to one of the application's scripts.[42] The backdoor continued to persist even after the compromised build server was decommissioned because that code was restored from a local backup, rather than the source control system. Webmin users were susceptible to RCE through a supply chain attack for a duration of over 15 months, until the backdoor was removed.

39. Oladimeji and Kerner, "SolarWinds hack explained: Everything you need to know."
40. Engelberg, "Bash Uploader Security Update."
41. Popov, "Update on git.php.net incident."
42. Hacker News, "Hackers Planted Backdoor in Webmin, Popular Utility for Linux/Unix Servers."

CICD-SEC-10

Insufficient Logging and Visibility

Definition

Insufficient logging and visibility risks allow an adversary to carry out malicious activities within the CI/CD environment without being detected during any phase of the attack lifecycle. This includes identifying the attacker's techniques, tactics, and procedures (TTP) as part of any postincident investigation.

Description

The existence of strong logging and visibility capabilities is essential for an organization's ability to prepare for, detect, and investigate a security-related incident.

Workstations, servers, network devices, and key IT and business applications are typically covered in depth within an organization's logging and visibility programs. Yet, it is often not the case with systems and processes in engineering environments.

Given the amount of potential attack vectors using engineering environments and processes, security teams must build the appropriate capabilities to detect these attacks as soon as they happen. Many of these vectors involve leveraging programmatic access against systems. Considering that challenge, a key aspect of addressing it is to create strong levels of visibility around both human and programmatic access.

With the sophisticated nature of CI/CD attack vectors, both your system audit logs and applicative logs require an equal level of importance. For the audit logs, for example, address user access, user creation, and permission modification. For applicative logs, for example, consider the push event to a repository, build execution, and artifact upload.

Impact

Adversaries are gradually shifting their focus to engineering environments as a means to achieve their goals. Organizations that do not ensure the appropriate logging and visibility controls around those environments might fail to detect a breach and face immense difficulties in mitigation and remediation due to minimal investigative capabilities.

Time and data are the most valuable commodities to an organization under attack. The existence of all relevant data sources in a centralized location might be the difference between a successful and devastating outcome in an incident response scenario.

Recommendations

Several elements achieve sufficient logging and visibility:

- **Map the environment.** To achieve strong visibility capabilities, security teams must be intimately familiar with all systems involved in potential threats. A breach can involve any of the systems that take part in the CI/CD processes, including SCM, CI, artifact repositories, package management software, container registries, CD, and orchestration engines (e.g., Kubernetes). Therefore, identify and build an inventory of all systems in use within the organization that contains every instance of these systems. This is specifically relevant for self-managed systems, such as Jenkins.
- **Identify and enable the appropriate log sources.** After identifying the relevant systems, ensure that all relevant logs are enabled, which is not the default state in the systems. Optimize visibility around both human access and programmatic access through the various measures it is allowed. Place an equal level of emphasis on identifying both relevant audit log sources and the applicative log sources.

- **Ship logs to a centralized location (e.g., SIEM).** Support the aggregation and correlation of logs between different systems for detection and investigation.
- **Create alerts to detect anomalies and potential malicious activity,** both in each system alone and the anomalies in the code shipping process. This involves multiple systems and requires deeper knowledge in the internal build and deployment processes.

References

Logging and visibility capabilities are essential and relevant for detecting and investigating any incident, regardless of the risk that was exploited in the incident. As any security incident in recent years involving CI/CD systems has required, victim organizations must have strong visibility to properly investigate and understand the extent of damage for the attack in question.

About Palo Alto Networks

As the global cybersecurity leader, Palo Alto Networks (NASDAQ: PANW) is dedicated to protecting our digital way of life via continuous innovation. Trusted by more than 70,000 organizations worldwide, we provide comprehensive AI-powered security solutions across network, cloud, security operations and AI, enhanced by the expertise and threat intelligence of Unit 42®. Our focus on platformization allows enterprises to streamline security at scale, ensuring protection fuels innovation. Explore more at www.paloaltonetworks.com.

paloalto® NETWORKS

3000 Tannery Way
Santa Clara, CA 95054

Main: +1.408.753.4000
Sales: +1.866.320.4788
Support: +1.866.898.9087

www.paloaltonetworks.com

cortex_guide-to-the-top-10-ci-cd-security-risks_092225