CI/CD Security Cheat Sheet

Introduction

CI/CD pipelines and processes facilitate efficient, repeatable software builds and deployments; as such, they occupy an important role in the modern SDLC. However, given their importance and popularity, CI/CD pipelines are also an appealing target for malicious hackers, and their security cannot be ignored. This goal of this cheat sheet is to provide developers practical guidelines for reducing risk associated with these critical components. This cheat sheet will focus on securing the pipeline itself. It will begin by providing some brief background information before proceeding with specific CI/CD security best practices.

Definition and Background

CI/CD refers to a set of largely automated processes used to build and deliver software; it is often portrayed as a pipeline consisting of a series of sequential, discrete steps. The pipeline generally begins when code under development is pushed to a repository and, if all steps complete successfully, ends with the software solution built, tested, and deployed to a production environment. CI/CD may be decomposed into two distinct parts: continuous integration (CI) and continuous delivery and/or continuous deployment (CD). CI focuses on build and testing automation; continuous delivery focuses on promoting this built code to a staging or higher environment and, generally, performing additional automated testing. Continuous delivery and continuous deployment may not always be distinguished in definitions of CI/CD; however, according to NIST, continuous delivery requires code to be manually pushed to production whereas continuous deployment automates even this step.

The exact steps in a CI/CD pipeline may vary from organization to organization and from project to project; however, automation, and the repeatability and agility it brings, should be a core focus of any CI/CD implementation.

Understanding CI/CD Risk

Although CI/CD brings many benefits, it also increases an organization's attack surface. People, processes, and technology are all required for CI/CD and all can be avenues of attack; code repositories, automation servers such as Jenkins, deployment procedures, and the nodes responsible for running CI/CD pipelines are just a few examples of CI/CD components which can

be exploited by malicious entities. Furthermore, since CI/CD steps are frequently executed using high-privileged identities, successful attacks against CI/CD often have high damage potential. If an organization chooses to leverage the many benefits of CI/CD, it must also ensure it invests the resources required to properly secure it; the Codecov and SolarWinds breaches are just two sobering examples of the potential impact of CI/CD compromise.

The specific methods attackers use to exploit CI/CD environments are diverse; however, certain risks are more prominent than others. Although one should not restrict themselves to knowledge of them, understanding the most prominent risks to CI/CD environments can help organizations allocate security resources more efficiently. OWASP's Top 10 CI/CD Security Risks is a valuable resources for this purpose; the project identifies the following as the top 10 CI/CD risks:

- CICD-SEC-1: Insufficient Flow Control Mechanisms
- CICD-SEC-2: Inadequate Identity and Access Management
- CICD-SEC-3: Dependency Chain Abuse
- CICD-SEC-4: Poisoned Pipeline Execution (PPE)
- CICD-SEC-5: Insufficient PBAC (Pipeline-Based Access Controls)
- CICD-SEC-6: Insufficient Credential Hygiene
- CICD-SEC-7: Insecure System Configuration
- CICD-SEC-8: Ungoverned Usage of Third-Party Services
- CICD-SEC-9: Improper Artifact Integrity Validation
- CICD-SEC-10: Insufficient Logging and Visibility

The remainder of this cheat sheet will focus on providing guidance for mitigating against these top 10 and other CI/CD risks.

Secure Configuration

Time and effort must be invested into properly securing the components, such as SCM systems and automation servers (Jenkins, TeamCity, etc), that enable CI/CD processes. Regardless of the specific tools in use, one should never blindly rely on default vendor settings. At the same time, one must not adjust settings without fully understanding the implications, nor perform any needed configuration updates in an uncontrolled, entirely ad-hoc way. Change management and appropriate governance must be in place. Additionally, education is key; before leveraging a tool to perform critical, security sensitive operations such as code deployment, it is imperative one take time to understand the underlying technology. Secure configuration does not happen automatically; it requires education and planning.

Furthermore, one must also take steps to ensure the security of the operating systems, container images, web servers, or other infrastructure used to run or support the CI/CD components identified above. These systems must be kept appropriately patched, and an inventory of these assets, including software versions, should also be maintained. These technologies should be hardened according to standards such as CIS Benchmarks or STIGs where appropriate.

Beyond these general principles, some specific guideline relevant to CI/CD configuration will be explored below.

Secure SCM Configuration

CI/CD environments allow for code to be pushed to a repository and then deployed to a production environment with little to no manual intervention. However, this benefit can quickly become an attack vector if it allows untrusted, potentially malicious code to be deployed directly to a production system. Proper configuration of the SCM system can help mitigate this risk. Best practices include:

- Avoid the use of auto-merge rules in platforms such as Gitlab, Github, or Bitbucket.
- Require pull requests to be reviewed before merging and ensure this review step cannot be bypassed.
- Leverage protected branches.
- Require commits to be signed
- Carefully weigh the risk against the benefits of allowing ephemeral contributors. Limit the number and permissions of external contributions when possible.
- Enable MFA where available
- Avoid assigning default permissions for users and roles with access to your SCM assets. Carefully manage your permissions.
- Restrict the ability to fork private or internal repositories.
- Limit the option to change repository visibility to public. You can find a wide variety of additional policies in this documentation.

To help navigate SCM configuration challenges, there are tools available, such as Legitify, an open-source tool by Legit security. Legitify scans SCM assets and identifies misconfigurations and security issues, including policies for all the above best practices (available for GitHub and GitLab).

Pipeline and Execution Environment

In addition to SCM systems, it is imperative that the automation servers responsible for running the pipelines are also configured securely. Examples of these technologies include Travis, TeamCity, Jenkins, and CircleCI. While the exact hardening process will vary according to the specific platform used, some general best practices include:

- Perform builds in appropriately isolated nodes (see Jenkins example here)
- Ensure communication between the SCM and CI/CD platform is secured using widely accepted protocols such as TLS 1.2 or greater.
- Restrict access to CI/CD environments by IP if possible.
- If feasible, store the CI config file outside the repository that is hosting the code being built. If the file is stored alongside the code, it is imperative that the file is reviewed before any merge request is approved.
- Enable an appropriate level of logging (discussed more under Visibility and Monitoring below)
- Incorporate language appropriate SAST, DAST, IaC vulnerability scanning and related tools into the pipeline.
- Require manual approval and review before triggering production deployment.
- If pipelines steps are executed in Docker image, avoid using the `--privileged` flag ref
- Ensure the pipeline configuration code is version controlled (ref)
- Enforce MFA where possible

IAM

Identity and Access Management (IAM) is the process of managing digital identities and controlling their access to digital resources. Examples of identities include system accounts, roles, groups, or individual user accounts. IAM has wide applications well beyond CI/CD, but mismanagement of identities and their underlying credentials are among the most prominent risks impacting CI/CD environments. The following subsections will highlight some IAM related security best practices that are especially relevant to CI/CD environments.

Secrets Management

Secrets, such as API keys or passwords, are often required for a CI/CD pipeline to execute successfully. Secrets in CI/CD environment are often numerous, with at least some providing substantial access to sensitive systems or operations. This combination introduces a challenge: how does one securely manage secrets while also allowing automated CI/CD processes to access

them as needed? Following some simple guidelines can help substantially mitigate, though certainly not eliminate, risk.

First, one should take steps to reduce the likelihood that secrets can be stolen in a usable format. Secrets should **never** be hardcoded in code repositories or CI/CD configuration files. Employ tools such as git-leaks or git-secrets to detect such secrets. Strive to prevent secrets from ever being committed in the first place and perform ongoing monitoring to detect any deviations. Secrets must also be removed from other artifacts such as Docker images and compiled binaries. Secrets must always be encrypted using industry accepted standards. Encryption must be applied while secrets are at-rest in a file-system, vault, or similar store; however, one must also ensure these secrets are not disclosed or persisted in cleartext as a consequence of use in the CI/CD pipeline. For example, secrets must not be printed out to the console, logged, or stored in a system's command history files (such as `~/.bash-history`). A third-party solution such as HashiCorp Vault, AWS Secrets Manager, AKeyless, or CyberArk may be used for this purpose.

Second, one must take steps to reduce impact in the event that secrets are stolen in a format that is usable by an attacker. Using temporary credentials or OTPs is one method for reducing impact. Furthermore, one may impose IP based or other restrictions that prevent even valid credentials from accessing resources if these further requirements are not met.The Least Privilege and Identity Lifecycle Management sections below provide further guidance on techniques to mitigate risk related to secrets theft.

For additional guidance on securely managing secrets, please reference the Secrets Management Cheat Sheet.

Least Privilege

Least privilege, defined by NIST as:

> The principle that a security architecture is designed so that each entity is granted the minimum system resources and authorizations that the entity needs to perform its function".

In the context of CI/CD environments, this principle should be applied to at least three main areas: the secrets used within pipeline steps to access external resources, the access one pipeline or step has to other resources configured in the CI/CD platform (Palo Alto Networks, n.d.), and the permissions of the OS user executing the pipeline.

Regardless of the specific application, the general guidance remains the same: access must be justified, not assumed. One should adopt a "deny by default" mindset. Any identities used within the pipeline must be assigned only the minimum permissions necessary to do its job. For example, if a pipeline must access an AWS service in order to complete its task, the AWS credentials used in that pipeline must only be able to perform the specific operations on the specific services and

resources it requires to perform its task. Similarly, credential sharing across pipelines should be kept to a minimum; in particular, such sharing should not occur across pipelines having different levels of sensitivity or value. If pipeline A does not require access to the same secrets pipeline B requires, they should ideally not be shared. Finally, the OS accounts responsible for running the pipeline should not have root or comparable privileges; this will help mitigate impact in case of compromise.

Identity Lifecycle Management

Although proper secrets management and application of the principle of least privilege are necessary for secure IAM, they are not sufficient. The lifecycle of identities, from creation to deprovisioning, must be carefully managed to reduce risk for CI/CD and other environments.

In the initial or "Joiner" phase of Identity Management (as defined in the ILM Playbook), considerations include using a centralized IdP rather than allowing local accounts, disallowing shared accounts, disallowing self-provisioning of identities, and only allowing email accounts with domains controlled by the organization responsible for the CI/CD environment (OWASP, n.d.). Once provisioned, identities must be tracked, maintained, and, when necessary, deprovisioned. Of particular concern in complex, distributed CI/CD environments is ensuring that an accurate, comprehensive, and up-to-date inventory of identities is maintained. The format of this inventory can vary by organizational needs, but, in addition to the identity itself, suggested fields include identity owner or responsible party, identity provider, last used, last updated, granted permissions, and permissions actually used by the identity. Such an inventory will help one readily identify identities which may be over-privileged or which may be candidates for deprovisioning. Proper identity maintenance must not be overlooked; the "forgotten" identity can be the vector an attacker users to compromise a CI/CD system.

Managing Third-Party Code

Due, in part, to high-profile breaches such as SolarWinds, the concept of software supply chain security has received increasing attention in recent years. This issue is especially pressing in the context of CI/CD as such environments interact with third-party code in multiple ways. Two such areas of interaction, the dependencies used by projects running within the pipeline and the third-party integrations and plug-ins with the CI/CD system itself will be discussed below.

Dependency Management

Using third-party packages with known vulnerabilities is a well-known problem in software engineering, and many tools have been developed to address this. In the context of CI/CD,

automated use of SCA and comparable tools can actually assist in improving security in this area. However, the CI/CD environment itself is susceptible to a different, but related, risk: dependency chain abuse.

Dependency chain abuse involves the exploitation of flaws within a system's dependency chain and dependency resolution processes; a successful attack can result in the execution of malicious code from an attacker controlled package. The dependency chain itself refers to the set of packages, including internal, direct third-party, and transitive dependencies, that a software solution requires to function. An attacker can take advantage of this dependency chain through methods such as dependency confusion, typosquatting, or takeover of a valid package maintainer's account. Dependency chain abuse attacks can be quite complex and comprehensive defense is correspondingly so; however, basic mitigation are quite straightforward.

Mitigation techniques begin early on in the SDLC, well before the CI/CD pipeline begins execution. The project's package management technology (such as npm) should be configured in such a way as to ensure dependency references are immutable (CISA et al. 2022). Version pinning should be performed, the version chosen for pinning must be one that is known to be valid and secure, and the integrity of any package the system downloads should be validated by comparing its hash or checksum to a known good hash of the pinned package. The exact procedures to achieve this will vary depending on the project's underlying technology, but, in general, both version pinning and hash verification can be performed via a platform's "lock" or similar file (i.e. package-lock.json or Pipfile.lock). Remember to enforce the lockfile. Prefer using private repositories where possible, and configure the package manager to use only a single private feed (Microsoft, 2021). For private packages, leverage scoped NPM packages, ID prefixes for NuGet packages, or a comparable feature to reduce the risk of dependency confusion. Finally, regardless of the platform used, ensure the file(s) responsible for controlling these settings (such as `.npmrc` in node environments), is committed to source control and accessible in the CI/CD environment.

Plug-In and Integration Management

Most CI/CD platforms are extensible through means of plug-ins or other third-party integrations. While these extensions can introduce many benefits, including potentially improving the security capabilities of the system, they also increase the attack surface. This is not to say that plug-ins should necessarily be disallowed; rather, the risk must simply be considered and reduced to an acceptable level.

Installation of plug-ins or integration with third-party services should be treated like the acquisition of any software. These tools are often easy to install and setup, but this does not mean their installation and usage should go ungoverned. Least privileges must be enforced to ensure only a small subset of users even have the permissions required to extend CI/CD platforms. Additionally,

such extensions must be vetted before installation. Questions to consider are comparable to those that should be asked before any software acquisition:

- Is the vendor a recognized and respected developer or company?
- Does the vendor have a strong or weak history in regards to application security?
- How popular is the specific plug-in or integration endpoint?
- Is the plugin or integration actively maintained?
- Will the extension require configuration changes that could reduce security (such as exposing additional ports)?
- Does the organization have the experience and resources to properly configure and maintain the offering?

After a plug-in or other integration has been approved, it must be incorporated into the organization's configuration management processes. The software must be kept up-to-date, especially with any security patches that become available. The extension must also be continually reviewed for value; if it is no longer needed, the extension should be removed.

Integrity Assurance

CI/CD exploits often require attackers to insert themselves into the normal flow of the pipeline and modify the inputs and/or outputs of one or more steps. As such, integrity verification is an important method of reducing risk in CI/CD environments.

As with many other defensive actions, implementation of integrity related controls begins early in the SDLC. As noted earlier, the SCM should require commits to be signed before the code can be merged. Also, as discussed in Dependency Management, the package management platform should be configured to use hashes or comparable to verify the integrity of a package. Code signing should also be employed; technologies such as Sigstore or Signserver may be used for this purpose. However, it is important to note that code signing and related technologies are not absolute guarantors of security; the code signing processes itself can be exploited. Please see NIST's Security Considerations for Code Signing for additional guidance on securing the code signing processes. Finally, integration of the in-toto.to framework or similar can further assist in improving integrity within the CI/CD environment.

Visibility and Monitoring

CI/CD environments can be complex and may often seem like a opaque-box to developers. However, visibility into these systems is critical for detecting potential attacks, better understand

one's risk posture, and detecting and remediating vulnerabilities. Though their value is often underestimated, logging and log analysis are vital for providing visibility into CI/CD systems.

The first step in increasing CI/CD visibility, is ensuring that the logging configuration within the CI/CD environment is compliant with your organization's log management policy. Beyond adherence to internal policies, configure the system to log data in a readily parsable format such as JSON or syslog. Carefully consider what content needs to be logged and at what verbosity. Although proper logging should allow for end-to-end visibility of the pipeline, more logging is not inherently better. One must not only consider the storage costs associated with logs, but also take care to avoid logging any sensitive data. For example, most authentication related events likely should be logged. However, one should never log plaintext passwords, authentication tokens, API keys, or similar secrets.

Once an appropriate logging strategy has been defined and necessary configuration has been performed, one is ready to start utilizing these logs to reduce risk. Sending aggregate logs to a centralized log management system or, preferably, a SIEM is a first step for realizing the value of logs. If a SIEM is used, alert should be carefully configured, and regularly refined, in order to provide timely alerts of anomalies and potential attacks. The exact configuration will vary significantly depending on the CI/CD environment, SIEM platform, and other factors. For an overview of CI/CD observability within the context of the ELK Stack (a popular SIEM platform) refer to this article or reference this article for an alternative approach which can be readily adapted to a variety of CI/CD environments. It is important to keep in mind that SIEM alerts will never be 100% accurate in detecting CI/CD attacks. Both false positive and false negatives will occur. Such platforms should not be relied on unquestioningly, but they do provide important visibility into CI/CD environments and can act as important alert systems when thoughtfully configured.

References

General References

- CISA, NSA, & ODNI (2022). Securing the Software Supply Chain: Recommended Processes for Developers
- Microsoft (2021). 3 Ways to Mitigate Risks Using Private Package Feeds
- OWASP (n.d.). OWASP Top 10 CI/CD Security Risks
- Palo Alto Networks (n.d.). What is Insufficient Pipeline-Based Access Controls

CI/CD Platforms

- CircleCI

- Jenkins
- SignServer
- TeamCity
- TravisCI

IaC Scanning

- Checkov
- Kics
- SonarSource
- TerraScan

Integrity Verification and Signing

- In-toto
- SignServer
- SigStore
- SLSA

Secrets Management Tools

- AWS Secrets Manager
- Azure Key Vault
- CyberArk Secrets Management
- Google Cloud Key Management
- HashiCorp Vault