

CIS Software Supply Chain Security Guide

v1.0

JUNE 2022

Terms of Use

Please use the following link for our current terms of use:

<https://www.cisecurity.org/terms-of-use-for-non-member-cis-products/>

Contents

Overview	1
Intended Audience	3
Consensus Guidance	3
Acknowledgments	3
1 Source Code	4
1.1 Code Changes	4
1.1.1 Ensure any changes to code are tracked in a version control platform	4
1.1.2 Ensure any change to code can be traced back to its associated task	5
1.1.3 Ensure any change to code receives approval of two strongly authenticated users (Automated)	5
1.1.4 Ensure previous approvals are dismissed when updates are introduced to a code change proposal	6
1.1.5 Ensure there are restrictions on who can dismiss code change reviews	6
1.1.6 Ensure code owners are set for extra sensitive code or configuration	7
1.1.7 Ensure code owner's review is required when a change affects owned code	7
1.1.8 Ensure inactive branches are periodically reviewed and removed	8
1.1.9 Ensure all checks have passed before merging new code	8
1.1.10 Ensure open Git branches are up to date before they can be merged into code base	9
1.1.11 Ensure all open comments are resolved before allowing code change merging	9
1.1.12 Ensure verification of signed commits for new changes before merging	10
1.1.13 Ensure linear history is required	10
1.1.14 Ensure branch protection rules are enforced for administrators	11
1.1.15 Ensure pushing or merging of new code is restricted to specific individuals or teams	11
1.1.16 Ensure force push code to branches is denied	12
1.1.17 Ensure branch deletions are denied	12
1.1.18 Ensure any merging of code is automatically scanned for risks	12
1.1.19 Ensure any changes to branch protection rules are audited	13
1.2 Repository Management	14
1.2.1 Ensure all public repositories contain a SECURITY.md file	14
1.2.2 Ensure repository creation is limited to specific members	14
1.2.3 Ensure repository deletion is limited to specific users	15
1.2.4 Ensure issue deletion is limited to specific users	15
1.2.5 Ensure all copies (forks) of code are tracked and accounted for	15
1.2.6 Ensure all code projects are tracked for changes in visibility status	16
1.2.7 Ensure inactive repositories are reviewed and archived periodically	16

1.3 Contribution Access	16
1.3.1 Ensure inactive users are reviewed and removed periodically	17
1.3.2 Ensure team creation is limited to specific members	17
1.3.3 Ensure minimum number of administrators are set for the organization	17
1.3.4 Ensure Multi-Factor Authentication (MFA) is required for contributors of new code	18
1.3.5 Ensure the organization is requiring members to use Multi-Factor Authentication (MFA)	18
1.3.6 Ensure new members are required to be invited using company-approved email	19
1.3.7 Ensure two administrators are set for each repository	19
1.3.8 Ensure strict base permissions are set for repositories	20
1.3.9 Ensure an organization's identity is confirmed with a "Verified" badge	20
1.3.10 Ensure Source Code Management (SCM) email notifications are restricted to verified domains	21
1.3.11 Ensure an organization provides SSH certificates	21
1.3.12 Ensure Git access is limited based on IP addresses	22
1.3.13 Ensure anomalous code behavior is tracked	22
1.4 Third-Party	23
1.4.1 Ensure administrator approval is required for every installed application	23
1.4.2 Ensure stale applications are reviewed and inactive ones are removed	23
1.4.3 Ensure the access granted to each installed application is limited to the least privilege needed	24
1.5 Code Risks	24
1.5.1 Ensure scanners are in place to identify and prevent sensitive data in code	24
1.5.2 Ensure scanners are in place to secure Continuous Integration (CI) pipeline instructions	25
1.5.3 Ensure scanners are in place to secure Infrastructure as Code (IaC) instructions	25
1.5.4 Ensure scanners are in place for code vulnerabilities	26
1.5.5 Ensure scanners are in place for open-source vulnerabilities in used packages	26
1.5.6 Ensure scanners are in place for open-source license issues in used packages	27
2 Build Pipelines	28
2.1 Build Environment	28
2.1.1 Ensure each pipeline has a single responsibility	28
2.1.2 Ensure all aspects of the pipeline infrastructure and configuration are immutable	29
2.1.3 Ensure the build environment is logged	29
2.1.4 Ensure the creation of the build environment is automated	30
2.1.5 Ensure access to build environments is limited	30
2.1.6 Ensure users must authenticate to access the build environment	31
2.2 Build Worker	31
2.2.1 Ensure build workers are single-used	31
2.2.2 Ensure build worker environments and commands are passed and not pulled	32
2.2.3 Ensure the duties of each build worker are segregated	32
2.2.4 Ensure build workers have minimal network connectivity	33
2.2.5 Ensure run-time security is enforced for build workers	33
2.2.6 Ensure build workers are automatically scanned for vulnerabilities	34
2.2.7 Ensure build workers' deployment configuration is stored in a version control platform	34
2.2.8 Ensure resource consumption of build workers is monitored	35

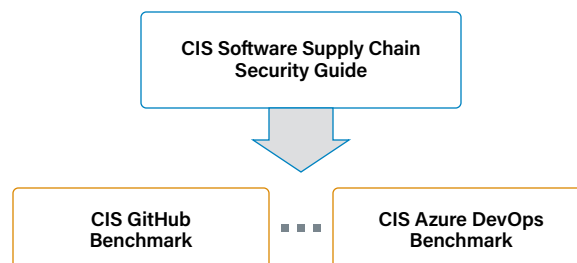
2.3 Pipeline Instructions	35
2.3.1 Ensure all build steps are defined as code	35
2.3.2 Ensure steps have clearly defined build stage input and output	36
2.3.3 Ensure output is written to a separate, secured storage repository	36
2.3.4 Ensure changes to pipeline files are tracked and reviewed	36
2.3.5 Ensure access to build process triggering is minimized	37
2.3.6 Ensure pipelines are automatically scanned for misconfigurations	37
2.3.7 Ensure pipelines are automatically scanned for vulnerabilities	37
2.3.8 Ensure scanners are in place to identify and prevent sensitive data in pipeline files (Automated)	38
2.4 Pipeline Integrity	38
2.4.1 Ensure all artifacts on all releases are signed	38
2.4.2 Ensure all external dependencies used in the build process are locked	39
2.4.3 Ensure dependencies are validated before being used	39
2.4.4 Ensure the build pipeline creates reproducible artifacts	39
2.4.5 Ensure pipeline steps produce a Software Bill of Materials (SBOM)	40
2.4.6 Ensure pipeline steps sign the SBOM produced	40
3 Dependencies	41
3.1 Third-Party Packages	41
3.1.1 Ensure third-party artifacts and open-source libraries are verified	41
3.1.2 Ensure SBOM is required from all third-party suppliers	42
3.1.3 Ensure signed metadata of the build process is required and verified	42
3.1.4 Ensure dependencies are monitored between open-source components	43
3.1.5 Ensure trusted package managers and repositories are defined and prioritized	43
3.1.6 Ensure a signed SBOM of the code is supplied	43
3.1.7 Ensure dependencies are pinned to a specific, verified version	44
3.1.8 Ensure all packages used are more than 60 days old	44
3.2 Validate Packages	44
3.2.1 Ensure an organization-wide dependency usage policy is enforced	45
3.2.2 Ensure packages are automatically scanned for known vulnerabilities	45
3.2.3 Ensure packages are automatically scanned for license implications	45
3.2.4 Ensure packages are automatically scanned for ownership change	46
4 Artifacts	47
4.1 Verification	47
4.1.1 Ensure all artifacts are signed by the build pipeline itself	47
4.1.2 Ensure artifacts are encrypted before distribution	47
4.1.3 Ensure only authorized platforms have decryption capabilities of artifacts	48

4.2 Access to Artifacts	48
4.2.1 Ensure factor authorization to certify certain artifacts is limited	48
4.2.2 Ensure number of permitted users who may upload new artifacts is minimized	49
4.2.3 Ensure user access to the package registry utilizes Multi-Factor Authentication (MFA)	49
4.2.4 Ensure user management of the package registry is not local	50
4.2.5 Ensure anonymous access to artifacts is revoked	50
4.3 Package Registries	50
4.3.1 Ensure all signed artifacts are validated upon uploading the package registry	51
4.3.2 Ensure all versions of an existing artifact have their signatures validated	51
4.3.3 Ensure changes in package registry configuration are audited	51
4.3.4 Ensure webhooks of the package registry are secured	52
4.4 Origin Traceability	52
4.4.1 Ensure artifacts contain information about their origin	52
4.4.2 Ensure private artifacts are not allowed to be pulled from external registries	53
5 Deployment	54
5.1 Deployment Configuration	54
5.1.1 Ensure deployment configuration files are separated from source code	54
5.1.2 Ensure changes in deployment configuration are tracked	55
5.1.3 Ensure scanners are in place to identify and prevent sensitive data in deployment configuration	55
5.1.4 Ensure access to deployment configurations are limited to specific members	56
5.1.5 Ensure scanners are in place to secure Infrastructure as Code (IaC) instructions	56
5.1.6 Ensure deployment configuration manifests are verified	57
5.1.7 Ensure deployment configuration manifests are pinned to a specific, verified version	57
5.2 Deployment Environment	58
5.2.1 Ensure deployments are automated	58
5.2.2 Ensure the deployment environment is reproducible	58
5.2.3 Ensure access to production environment is limited	59
5.2.4 Ensure default passwords are not used	59

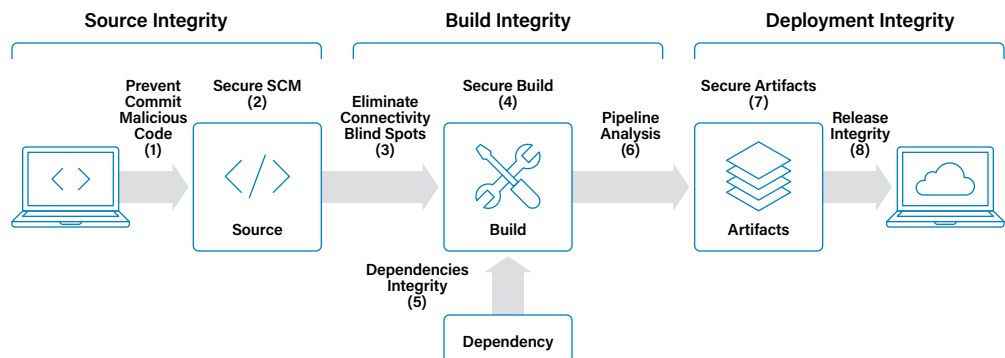
Overview

Argon (now part of Aqua Security) approached the Center for Internet Security (CIS) with the idea of developing a CIS Benchmark for Software Supply Chain Security. CIS has developed and published secure configuration guidance (i.e., CIS Benchmarks) covering a wide variety of technologies for many years, but the concept of creating a Benchmark for Software Supply Chain Security presented a new set of issues. There are a variety of technologies and platforms commonly used for developing modern software, so which should be covered? How do we ensure consistent security recommendations across the various platforms?

It was decided that instead of diving into creating a specific Benchmark initially, a more generic guidance set would be created first to act as the parent for the more specific guidance to come. Thus, the *CIS Software Supply Chain Security Guide* was born. The hope with the publication of this Guide is to elicit feedback from the global community that will help ensure the future platform-specific guidance (CIS Benchmarks) is even more accurate and relevant.



The Guide follows the phases of the software supply chain, as described in the below chart, from the moment a contributor adds code to the moment the application is delivered to the customer.



The Guide currently consists of 100+ recommendations organized into five main categories:

- 1 Source Code:** Security recommendations for proper source code management of any application developed by the organization.
 - This is the first phase of the software supply chain and is considered the only source of truth for the rest of the process. Because of that, it needs to be protected from the code itself, with the vulnerabilities, misconfigurations, and sensitive data it might hold, to the platform it is stored on.
- 2 Build Pipelines:** Security recommendations for the management and security of the build pipeline components.
 - Build components include build pipelines—a set of instructions dedicated to taking raw files of source code and running a series of tasks on them to achieve some final artifact as output, the environment they are running on, their management and execution, and more. This second phase of the software supply chain is targeted increasingly at supply chain attacks (e.g., the Codecov attack or SolarWinds).
- 3 Dependencies:** Security recommendations for the management of various dependencies introduced as part of the software build and release process.
 - Dependencies are a huge part of the software supply chain, as they are comprised of anything that goes into application code or is used by the build pipelines themselves. They are often written by third-party developers and might be vulnerable to certain attacks (e.g., the log4j attack).
- 4 Artifacts:** Security recommendations for the management of artifacts produced by build pipelines, as well as ones used by the application in the build process itself.
 - Artifacts are packaged versions of software. They are stored in package registries (or artifact managers) and require securing from the moment they are created, through the time they are copied and updated, and up to deployment to their relevant environment.
- 5 Deployment:** Security recommendations for the management of the application deployment process, the configurations, and the files that come with it.
 - This is the final phase of the software supply chain. After that, the client already uses the application, and it is running in production. It is important to secure all of these to deliver the software to the client safely.

The overall vision of the Guide and ultimately of the CIS Benchmarks is to support key emerging standards like Supply-chain Levels for Software Artifacts (SLSA) and The Update Framework (TUF) with foundational recommendations for setting and auditing configurations on the Benchmark-supported platforms.

By publishing the *CIS Software Supply Chain Security Guide*, CIS and Aqua Security hope to build a vibrant set of communities interested in developing the platform-specific Benchmark guidance to come. They are calling on subject matter experts (SMEs) that develop or work with these platforms to help create this guidance in the collaborative and consensus-based manner CIS is known for.

To date, the Guide has been reviewed by SMEs from Aqua Security, CIS, Microsoft, PayPal, Red Hat, CyberArk, Axonius, and others. By publishing the current work, CIS and Aqua Security want an even wider audience of SMEs to contribute to this project, for the benefit of all.

To contribute to this or other CIS Benchmark projects, please contact the CIS Benchmarks

Development Team at benchmarkinfo@cisecurity.org.

Intended Audience

This CIS Guide is intended for DevOps and application security administrators, security specialists, auditors, help desks, and platform deployment personnel who plan to develop, deploy, assess, or secure solutions to build and deploy software updates through automated means of DevOps pipelines.

Consensus Guidance

This Guide was created using a consensus review process comprised of a global community of subject matter experts. The process combines real-world experience with data-based information to create technology-specific guidance to assist users to secure their environments. Consensus participants provide perspective from a diverse set of backgrounds including consulting, software development, audit and compliance, security research, operations, government, and legal.

Acknowledgments

This Guide exemplifies the great things a community of users, vendors, and subject matter experts can accomplish through consensus collaboration. The CIS community thanks the entire consensus team with special recognition to the following individuals who contributed greatly to the creation of this Guide:

Authors

Eylam Milner, Aqua Security
Resheet Kosef, Aqua Security

Contributors

Yossi Weizman, Microsoft
Erez Dasa, PayPal
Michael Kotelnikov, Red Hat
Moshik Barak, CyberArk
Ofir Shapira, Axonius
Mor Weinberger, Aqua Security
Yakir Kadkoda, Aqua Security
Randy Mowen, Center for Internet Security
Stephen Keller, Center for Internet Security
Phil White, Center for Internet Security
Andrew Dannenberger, Center for Internet Security
Kari Byrd, Independent Consultant
Ori Zerah, Aqua Security
Lucas Aides, Aqua Security
Marina Segal, Sysdig

1 Source Code

This section consists of security recommendations for proper source code management of any application developed by the organization. This is the first phase of the software supply chain, and is considered the single source of truth for the rest of the process.

It is critical to secure both the source code itself, as well as the platform with which it is managed, in order to protect the integrity of a software release. From the developers who commit changes, to the sensitive data or vulnerabilities that could be placed within it, and ultimately to the source code management platform in which it is stored, verification of the integrity of the source code is imperative in order to keep every software update secure.

1.1 Code Changes

This section consists of security recommendations for code changes and how they should be done. It contains recommendations to protect the main branch of the application code. This branch is the most important one, because it contains the actual code that is being delivered to the customer. It should be protected from any mistake or malicious deed in order to keep the software secured.

1.1.1 Ensure any changes to code are tracked in a version control platform

Description

Manage all code projects in a version control platform.

Rationale

Version control platforms keep track of every modification to code. They represent the cornerstone of code security, as well as allow for better code collaboration within engineering teams. With granular access management, change tracking, and key signing of code edits, version control platforms are the first step in securing the software supply chain.

Audit

Ensure that all code activity is managed through a version control platform for every microservice or application developed by an organization.

Remediation

Upload existing code projects to a dedicated version control platform and create an identity for each active team member who might contribute or need access to it.

1.1.2 Ensure any change to code can be traced back to its associated task

Description

Use a task management system to trace any code back to its associated task.

Rationale

The ability to trace each piece of code back to its associated task simplifies the Agile and DevOps process by enabling transparency of any code changes. This allows faster remediation of bugs and security issues, while also making it harder to push unauthorized code changes to sensitive projects. Additionally, using a task management system simplifies achieving compliance, as it is easier to track each regulation.

Audit

Ensure every code change can be traced back to its origin task in a task management system.

Remediation

Use a task management system to manage tasks as the starting point for each code change. Whether it is a new feature, bug fix, or security fix—all should originate from a dedicated task (ticket) in your organization's task management system. These tasks should also be linked to the code changes themselves in a way that is easy to follow: from code to task, and from task back to code.

1.1.3 Ensure any change to code receives approval of two strongly authenticated users (Automated)

Description

Ensure that every code change is reviewed and approved by two authorized contributors who are both strongly authenticated, from the team relevant to the code change.

Rationale

To prevent malicious or unauthorized code changes, the first layer of protection is the process of code review. This process involves engineer teammates reviewing each other's code for errors, optimizations, and general knowledge-sharing. With proper peer reviews in place, an organization can detect unwanted code changes very early in the process of release. In order to help facilitate code review, companies should employ automation to verify that every code change has been reviewed and approved by at least two team members before it is pushed into the code base. These team members should be from the team that is related to the code change, so it will be a meaningful review.

NOTE To enforce a code review requirement, verification for a minimum of two reviewers must be put into place. This will ensure new code will not be able to be pushed to the code base before it has received two independent approvals.

Audit

For every code repository in use, verify that two approvals from the specific code repository team are required in order to push new code to the code base.

Remediation

An organization can protect specific code branches—for example, the “main” branch, which often is the version deployed to production—by setting protection rules. These rules secure your code repository from unwanted or unauthorized changes. You may set requirements for any code change to that branch, and thus specify a minimum number of reviewers required to approve a change.

1.1.4 Ensure previous approvals are dismissed when updates are introduced to a code change proposal

Description

Ensure that when a proposed code change is updated, previous approvals are declined and new approvals are required.

Rationale

An approval process is necessary when code changes are suggested. Through this approval process, however, changes can still be made to the original proposal even after some approvals have already been given. This means malicious code can find its way into the code base even if the organization has enforced a review policy. To ensure this is not possible, outdated approvals must be declined when changes to the suggestion are introduced.

NOTE If new code changes are pushed to a specific proposal, all previously accepted code change proposals must be declined.

Audit

For each code repository in use, validate that each updated code suggestion declines the previously received approvals.

Remediation

For each code repository in use, enforce an organization-wide policy to dismiss given approvals to code change suggestions if those suggestions were updated.

1.1.5 Ensure there are restrictions on who can dismiss code change reviews

Description

Only trusted users should be allowed to dismiss code change reviews.

Rationale

Dismissing a code change review permits users to merge new suggested code changes without going through the standard process of approvals. Controlling who can perform this action will prevent malicious actors from simply dismissing the required reviews to code changes and merging malicious or dysfunctional code into the code base.

NOTE In cases where a code change proposal has been updated since it was last reviewed and the person who reviewed it is not available for approval, a general collaborator would not be able to merge their code changes until a user with "dismiss review" abilities could dismiss the open review. Users who are not allowed to dismiss code change reviews will not be permitted to do so, and thus are unable to waive the standard flow of approvals.

Audit

For each code repository in use, ensure that only trusted users are allowed to dismiss code change reviews.

Remediation

For each code repository in use, do not grant the permission to dismiss code change reviews unless it is really necessary. If it is obligatory, carefully select the individual collaborators or groups whom you trust with the ability to dismiss code change reviews.

1.1.6 Ensure code owners are set for extra sensitive code or configuration

Description

Code owners are trusted users that are responsible for reviewing and managing an important piece of code or configuration. An organization is advised to set code owners for every extremely sensitive code or configuration.

Rationale

Configuring code owners protects data by verifying that trusted users will notice and review every edit, thus preventing unwanted or malicious changes from potentially compromising sensitive code or configurations.

NOTE Code owner users will receive notifications for every change that occurs to the code and subsequently added as reviewers of pull requests automatically.

Audit

For every code repository in use, ensure code owners are set for sensitive code or configuration.

Remediation

For every code repository in use, identify particularly sensitive parts of code and configurations and set trusted users to be their code owners.

1.1.7 Ensure code owner's review is required when a change affects owned code

Description

Ensure trusted code owners are required to review and approve any code change proposal made to their respective owned areas in the code base.

Rationale

Configuring code owners ensures that no code, especially code that could prove malicious, will slip into the source code or configuration files of a repository. This allows an organization to mark areas in the code base that are especially sensitive or more prone to an attack. It can also enforce review by specific individuals who are designated as owners to those areas so that they may filter out unauthorized or unwanted changes beforehand.

NOTE If an organization enforces code owner-based reviews, some code change proposals would not be able to be merged to the code base before specific, trusted individuals approve them.

Audit

For each repository in use, verify that code owners are required to review all code change proposals relevant to areas they own.

Remediation

For each repository in use, configure code owner-required approvals for each change proposal related to code they own.

1.1.8 Ensure inactive branches are periodically reviewed and removed

Description

Keep track of code branches that are inactive for a lengthy period of time and periodically remove them.

Rationale

Git branches that have been inactive (i.e., no new changes introduced) for a long period of time are enlarging the surface of attack for malicious code injection, sensitive data leaks, and CI pipeline exploitation. They potentially contain outdated dependencies that may leave them highly vulnerable. They are more likely to be improperly managed, and could possibly be accessed by a large number of members of the organization.

NOTE Removing inactive Git branches means that any code changes they contain would be removed along with them, thus work done in the past might not be accessible after auditing for inactivity.

Audit

For each code repository in use, verify that all existing Git branches are active or have yet to be checked for inactivity within a specified period.

Remediation

For each code repository in use, review existing Git branches and remove those that have not been active for a prescribed period.

1.1.9 Ensure all checks have passed before merging new code

Description

Before a code change request can be merged to the code base, all predefined checks must successfully pass.

Rationale

On top of manual reviews of code changes, a code protect should contain a set of prescriptive checks that validate each change. Organizations should enforce those status checks so that changes can only be introduced if all checks have successfully passed. This set of checks should serve as the absolute quality, stability, and security conditions that must be met in order to merge new code to a project.

NOTE Code changes in which all checks do not pass successfully would not be able to be pushed into the code base of the specific code repository.

Audit

Ensure that for each code repository in use, status checks are required to pass before allowing any code change proposal merge.

Remediation

Configure each code repository to require all status checks to pass before permitting a merge of new code.

1.1.10 Ensure open Git branches are up to date before they can be merged into code base

Description

Organizations should make sure each suggested code change is in full sync with the existing state of its origin code repository before allowing merging.

Rationale

Git branches can easily become outdated since the origin code repository is constantly being edited. This means engineers working on separate code branches can accidentally include outdated code with potential security issues that might have already been fixed, overriding the potential solutions for those security issues when merging their own changes.

NOTE If enforced, outdated branches would not be able to be merged into their origin repository without first being updated to contain any recent changes.

Audit

For each code repository in use, verify that open branches must be updated before merging is permitted.

Remediation

For each code repository in use, enforce a policy to only allow merging open branches if they are current with the latest change from their origin repository.

1.1.11 Ensure all open comments are resolved before allowing code change merging

Description

Organizations should enforce a “no open comments” policy before allowing code change merging.

Rationale

In an open code change proposal, reviewers can leave comments containing their questions and suggestions. These comments can also include potential bugs and security issues. Requiring all comments on a code change proposal to be resolved before it can be merged ensures that every concern is properly addressed or acknowledged before the new code changes are introduced to the code base.

NOTE Code change proposals containing open comments would not be able to be merged into the code base.

Audit

For every code repository in use, verify that each merged code change does not contain open, unattended comments.

Remediation

For each code repository in use, require open comments to be resolved before the relevant code change can be merged.

1.1.12 Ensure verification of signed commits for new changes before merging

Description

Ensure every commit in a pull request is signed and verified before merging.

Rationale

Signing commits, or requiring to sign commits, gives other users confidence about the origin of a specific code change. It ensures that the author of the change is not hidden and is verified by the version control system, thus the change comes from a trusted source.

NOTE Pull requests with unsigned commits cannot be merged.

Audit

Ensure only signed commits can be merged for every branch, especially the main branch, via branch protection rules.

Remediation

For each repository in use, enforce the branch protection rule of requiring signed commits, and make sure only signed commits are capable of merging.

1.1.13 Ensure linear history is required

Description

Linear history is the name for Git history where all commits are listed in chronological order, one after another. Such history exists if a pull request is merged either by rebase merge (reorders the commits history) or squash merge (squashes all commits to one). Ensure that linear history is required by requiring the use of rebase or squash merge when merging a pull request.

Rationale

Enforcing linear history produces a clear record of activity, and as such it offers specific advantages: it is easier to follow, it is easier to revert a change, and bugs can be found more easily.

NOTE Pull requests cannot be merged except by squash or rebase merge.

Audit

For each repository in use, ensure that linear history is required and/or that only squash merge and rebase merge are allowed.

Remediation

For each repository in use, require linear history and/or allow only rebase merge and squash merge.

1.1.14 Ensure branch protection rules are enforced for administrators

Description

Ensure administrators are subject to branch protection rules.

Rationale

Administrators by default are excluded from any branch protection rules. This means these privileged users (on both the repository and organization levels) are not subject to protections meant to prevent untrusted code insertion, including malicious code. This is extremely important since administrator accounts are often targeted for account hijacking due to their privileged role.

NOTE Administrator users will not be able to push code directly to the protected branch without being compliant with listed branch protection rules.

Audit

For each repository in use, validate branch protection rules also apply to administrator accounts.

Remediation

For each repository in use, enforce branch protection rules on administrators, as well.

1.1.15 Ensure pushing or merging of new code is restricted to specific individuals or teams

Description

Ensure that only trusted users can push or merge new code to protected branches.

Rationale

Requiring that only trusted users may push or merge new changes reduces the risk of unverified code, especially malicious code, to a protected branch by reducing the number of trusted users who are capable of doing such.

NOTE Only administrators and trusted users can push or merge to the protected branch.

Audit

For each repository that is being used, ensure only trusted and responsible users can push or merge new code.

Remediation

For each repository in use, allow only trusted and responsible users to push or merge new code.

1.1.16 Ensure force push code to branches is denied

Description

The "force push" option allows users with "push" permissions to force their changes directly to the branch without a pull request, and thus should be disabled.

Rationale

The "force push" option allows users to override the existing code with their own code. This can lead to both intentional and unintentional data loss, as well as data infection with malicious code. Disabling the "force push" option prohibits users from forcing their changes to the main branch, which ultimately prevents malicious code from entering source code.

NOTE Users cannot "force push" to protected branches.

Audit

For each repository in use, validate that no one can "force push" code.

Remediation

For each repository in use, block the option to "force push" code.

1.1.17 Ensure branch deletions are denied

Description

Ensure that users with only push access are incapable of deleting a protected branch.

Rationale

When enabling deletion of a protected branch, any user with at least push access to the repository can delete a branch. This can be potentially dangerous, as a simple human mistake or a hacked account can lead to data loss if a branch is deleted. It is therefore crucial to prevent such incidents by denying protected branch deletion.

NOTE Protected branches cannot be deleted.

Audit

For each repository that is being used, verify that protected branches cannot be deleted.

Remediation

For each repository that is being used, block the option to delete protected branches via branch protection rules.

1.1.18 Ensure any merging of code is automatically scanned for risks

Description

Ensure that every pull request is required to be scanned for risks.

Rationale

Scanning pull requests to detect risks allows for early detection of vulnerable code and/or dependencies and helps mitigate potentially malicious code.

Audit

For each repository in use, ensure that every pull request must be scanned for risks.

Remediation

For every repository in use, enforce risk scanning on every pull request.

1.1.19 Ensure any changes to branch protection rules are audited

Description

Ensure that changes in the branch protection rules are audited.

Rationale

Branch protection rules should be configured on every repository. The only users who may change such rules are administrators. In a case of an attack on an administrator account or of human error on the part of an administrator, protection rules could be disabled, and thus decrease source code confidentiality as a result. It is important to track and audit such changes to prevent potential incidents as soon as possible.

Audit

Ensure a tracking system is in place that logs changes in branch protection rules (webhooks, etc.).

Remediation

Use, maintain, or create a tracking system that tracks changes in branch protection rules (webhooks, etc.).

1.2 Repository Management

This section consists of security recommendations for proper code repository management.

Code repositories are where the application code is stored and organized. It is important to keep code repositories organized and maintained to avoid data loss, data theft, and other attacks that may happen unknowingly when a repository is not maintained well. The recommendations of this section are setting guides to do so.

1.2.1 Ensure all public repositories contain a SECURITY.md file

Description

A SECURITY.md file is a security policy file that offers instruction on reporting security vulnerabilities in a project. When someone creates an issue within a specific project, a link to the SECURITY.md file will subsequently be shown.

Rationale

A SECURITY.md file provides users with crucial security information. It can also serve an important role in project maintenance, encouraging users to think ahead about how to properly handle potential security issues, updates, and general security practices.

Audit

For each repository in use, verify that it has a SECURITY.md file in the documents or root directory of the repository.

Remediation

For each repository in use, create a SECURITY.md file and save it in the documents or root directory of the repository.

1.2.2 Ensure repository creation is limited to specific members

Description

Limit the ability to create repositories to trusted users and teams.

Rationale

Restricting repository creation to trusted users and teams is recommended in order to keep the organization properly structured, track fewer items, prevent impersonation, and to not overload the version control system. It will allow administrators easier source code tracking and management capabilities, as they will have fewer repositories to track. The process of detecting potential attacks becomes far more straightforward, as well, since the easier it is to track the source code, the easier it is to detect malicious acts within it. Additionally, the possibility of a member creating a public repository and sharing the organization's data externally is significantly decreased.

NOTE Specific users will not be permitted to create repositories.

Audit

Verify that only trusted users and teams can create repositories.

Remediation

Restrict repository creation to trusted users and teams only.

1.2.3 Ensure repository deletion is limited to specific users

Description

Ensure only a limited number of trusted users can delete repositories.

Rationale

Restricting the ability to delete repositories protects the organization from intentional and unintentional data loss. This ensures that users cannot delete repositories or cause other potential damage — whether by accident or due to their account being hacked — unless they have the correct privileges.

NOTE Certain users will not be permitted to delete repositories.

Audit

Verify that only a limited number of trusted users can delete repositories.

Remediation

Enforce repository deletion by a few trusted and responsible users only.

1.2.4 Ensure issue deletion is limited to specific users

Description

Ensure only trusted and responsible users can delete issues.

Rationale

Issues are a way to keep track of things happening in repositories, such as setting new milestones or requesting urgent fixes. Deleting an issue is not a benign activity, as it might harm the development workflow or attempt to hide malicious behavior. Because of this, it should be restricted and allowed only by trusted and responsible users.

NOTE Certain users will not be permitted to delete issues.

Audit

Verify that only trusted and responsible users can delete issues.

Remediation

Restrict issue deletion to a few trusted and responsible users only.

1.2.5 Ensure all copies (forks) of code are tracked and accounted for

Description

Track every fork of code and ensure it is accounted for.

Rationale

A fork is a copy of a repository. On top of being a plain copy, any updates to the original repository itself can be pulled and reflected by the fork under certain conditions. A large number of repository copies (forks) become difficult to manage and properly secure. New and sensitive changes can often be pushed into a critical repository without developer knowledge of an updated copy of the very same repository. If there is no limit on doing this, then it is recommended to track and delete copies of organization repositories as needed.

NOTE Disabling forks completely may slow down the development process as more actions will be necessary to take in order to fork a repository.

Audit

Verify that forks are tracked and examined regularly.

Remediation

Track forks and examine them regularly.

1.2.6 Ensure all code projects are tracked for changes in visibility status

Description

Ensure every change in visibility of projects is tracked.

Rationale

Visibility of projects determines who can access a project and/or fork it: anyone, designated users, or only members of the organization. If a private project becomes public, this may point to a potential attack, which can ultimately lead to data loss, the leaking of sensitive information, and finally to a supply chain attack. It is crucial to track these changes in order to prevent such incidents.

Audit

Ensure that every change in project visibility is tracked and investigated.

Remediation

Track every change in project visibility and investigate if suspicious behavior occurs.

1.2.7 Ensure inactive repositories are reviewed and archived periodically

Description

Track inactive repositories and remove them periodically.

Rationale

Inactive repositories (i.e., no new changes introduced for a long period of time) can enlarge the surface of a potential attack or data leak. These repositories are more likely to be improperly managed, and thus could possibly be accessed by many users in an organization.

NOTE Bug fixes and deployment of necessary changes could prove complicated for archived repositories.

Audit

Verify that all the repositories in the organization are active, and those that are not are reviewed or archived.

Remediation

Review all inactive repositories and archive them periodically.

1.3 Contribution Access

This section consists of security recommendations for managing access to the application code. This includes managing both internal and external access, administrator accounts, permissions, identification methods, etc. Securing these items is important for software safety, because every security constraint on access is an obstacle in the way of attacks.

This section differentiates between common user account and admin account. It is important to understand that due to the high permissions of the admin account, it should be used only for administrative work and not for everyday tasks.

1.3.1 Ensure inactive users are reviewed and removed periodically

Description

Track inactive user accounts and periodically remove them.

Rationale

User accounts that have been inactive for a long period of time are enlarging the surface of attack. Inactive users with high-level privileges are of particular concern, as these accounts are more likely to be targets for attackers. This could potentially allow access to large portions of an organization should such an attack prove successful. It is recommended to remove them as soon as possible in order to prevent this.

Audit

For each repository in use, verify that all user accounts are active.

Remediation

For each repository in use, review inactive user accounts (members that left the organization, etc.) and remove them.

1.3.2 Ensure team creation is limited to specific members

Description

Limit the ability to create teams to trusted and specific users.

Rationale

The ability to create new teams should be restricted to specific members in order to keep the organization orderly and ensure users have access to only the lowest privilege level necessary. Teams typically inherit permissions from their parent team; thus, if base permissions are less restricted and any user has the ability to create a team, a permission leverage could occur in which certain data is made available to users who should not have access to it. Such a situation could potentially lead to the creation of shadow teams by an attacker. Restricting team creation will also reduce additional clutter in the organizational structure, and as a result will make it easier to track changes and anomalies.

NOTE Only specific users will be able to create new teams.

Audit

For every organization, ensure that team creation is limited to specific, trusted users.

Remediation

For every organization, limit team creation to specific, trusted users.

1.3.3 Ensure minimum number of administrators are set for the organization

Description

Ensure the organization has a minimum number of administrators.

Rationale

Organization administrators have the highest level of permissions, including the ability to add/remove collaborators, create or delete repositories, change branch protection policy, and convert to a publicly accessible repository. Due to the permissive access granted to an organization administrator, it is highly recommended to keep the number of administrator accounts as minimal as possible.

Audit

Set the minimum number of administrators in your organization.

Remediation

Set the minimum number of administrators in your organization.

1.3.4 Ensure Multi-Factor Authentication (MFA) is required for contributors of new code

Description

Require collaborators from outside the organization to use Multi-Factor Authentication (MFA) in addition to a standard user name and password when authenticating to the source code management platform.

Rationale

By default, every user authenticates within the system by password only. If the password of a user is compromised, however, the user account and every repository to which they have access is in danger of data loss, malicious code commits, and data theft. It is therefore recommended that each user has Multi-Factor Authentication enabled. This adds an additional layer of protection to ensure the account remains secure even if the user's password is compromised.

NOTE A member without enabled Multi-Factor Authentication cannot contribute to the project. They must enable Multi-Factor Authentication before they can contribute any code.

Audit

For each repository in use, verify that Multi-Factor Authentication is enforced for contributors and is the only way to authenticate.

Remediation

For each repository in use, enforce Multi-Factor Authentication as the only way to authenticate for contributors.

1.3.5 Ensure the organization is requiring members to use Multi-Factor Authentication (MFA)

Description

Require members of the organization to use Multi-Factor Authentication (MFA) in addition to a standard user name and password when authenticating to the source code management platform.

Rationale

By default, every user authenticates within the system by password only. If the password of a user is compromised, however, the user account and every repository to which they have access is in danger of data loss, malicious code commits, and data theft. It is therefore recommended that each user has Multi-Factor Authentication enabled. This adds an additional layer of protection to ensure the account remains secure even if the user's password is compromised.

NOTE Members could be removed from the organization if they do not have Multi-Factor Authentication already enabled. If this is the case, it is recommended that an invitation be sent to reinstate the user's access and former privileges. They must enable Multi-Factor Authentication in order to accept the invitation.

Audit

For every organization that exists in your source code management platform, verify that Multi-Factor Authentication is enforced and is the only way to authenticate.

Remediation

Use the built-in setting to ensure the enforcement of Multi-Factor Authentication for each member of the organization.

1.3.6 Ensure new members are required to be invited using company-approved email

Description

Existing members of an organization can invite new members to join; however, new members must only be invited with their company-approved email.

Rationale

Ensuring new members of an organization have company-approved email prevents existing members of the organization from inviting arbitrary new users to join. Without this verification, they can invite anyone who is using the organization's version control system or has an active email account, thus allowing outside users (and potential threat actors) to easily gain access to company private code and resources. This practice will subsequently reduce the chance of human error or typos when inviting a new member.

NOTE Existing members would not be able to invite new users who do not have a company-approved email address.

Audit

For each organization in use, verify for every invitation that the invited email address is company-approved.

Remediation

For each organization, allow only users with company-approved email to be invited. If a user was invited without company-approved email, cancel the invitation and investigate the reason they were invited.

1.3.7 Ensure two administrators are set for each repository

Description

Ensure every repository has two users with administrative permissions.

Rationale

Repository administrators have the highest permissions to said repository. These include the ability to add/remove collaborators, change branch protection policy, and convert to a publicly accessible repository. Due to the liberal access granted to a repository administrator, it is highly recommended that only two contributors occupy this role.

NOTE Removing administrative users from a repository would result in them losing high-level access to that repository.

Audit

For every repository in use, verify there are two administrators.

Remediation

For every repository in use, set two administrators.

1.3.8 Ensure strict base permissions are set for repositories

Description

Base permissions define the permission level automatically granted to all organization members. Define strict base access permissions for all of the repositories in the organization, including new ones.

Rationale

Defining strict base permissions is the best practice in every role-based access control (RBAC) system. If the base permission is high — for example, “Write” permission — every member of the organization will have “Write” permission to every repository in the organization. This will apply regardless of the specific permissions a user might need, which generally differ between organization repositories. The higher the permission, the higher the risk for incidents such as bad code commit or data breach. It is therefore recommended to set the base permissions to the strictest level possible.

NOTE Users might not be able to access organization repositories or perform some acts as commits. These specific permissions should be granted individually for each user or team, as needed.

Audit

Verify that strict base permissions are set for the organization repositories — either “None” or “Read.”

Remediation

Set strict base permissions for the organization repositories — either “None” or “Read.”

1.3.9 Ensure an organization's identity is confirmed with a “Verified” badge

Description

Confirm the domains an organization owns with a “Verified” badge.

Rationale

Verifying the organization's domains gives developers assurance that a given domain is truly the official home for a public organization. Attackers can pretend to be an organization and steal information via a faked/spoofed domain; therefore, the use of a “Verified” badge instills more confidence and trust between developers and the open-source community.

Audit

Ensure the organization has a “Verified” badge next to its name.

Remediation

Verify the organization's domains and secure a “Verified” badge next to its name.

1.3.10 Ensure Source Code Management (SCM) email notifications are restricted to verified domains

Description

Restrict the organization's Source Code Management (SCM) email notifications to approved domains only.

Rationale

Restricting Source Code Management email notifications to verified domains only prevents data leaks, as personal emails and custom domains are more prone to account takeover via DNS hijacking or password breach.

NOTE Only members with approved email would be able to receive Source Code Management notifications.

Audit

Ensure Source Code Management email notifications are restricted to approved domains only.

Remediation

Restrict Source Code Management email notifications to approved domains only.

1.3.11 Ensure an organization provides SSH certificates

Description

As an organization, become an SSH Certificate Authority (CA) and provide SSH keys for accessing repositories.

Rationale

There are two ways for remotely working with Source Code Management: via HTTPS, which requires authentication by user/password, or via SSH, which requires the use of SSH keys. SSH authentication is better in terms of security; key creation and distribution, however, must be done in a secure manner. This can be accomplished by implementing SSH certificates, which are used to validate the server's identity. A developer will not be able to connect to a Git server if its key cannot be verified by the SSH Certificate Authority (CA) server. As an organization, one can verify the SSH certificate signature used to authenticate if a CA is defined and used. This ensures that only verified developers can access organization repositories, as their SSH key will be the only one signed by the CA certificate. This reduces the risk of misuse and malicious code commits.

NOTE Members with unverified keys will not be able to clone organization repositories. Signing, certification, and verification might also slow down the development process.

Audit

Verify that the organization has an SSH Certificate Authority server and provides an SSH certificate with which to sign keys.

Remediation

Deploy an SSH Certificate Authority server and configure it to provide an SSH certificate with which to sign keys.

1.3.12 Ensure Git access is limited based on IP addresses

Description

Limit Git access based on IP addresses by having an allowlist of IP addresses from which connection is possible.

Rationale

Allowing access to Git repositories (source code) only from specific IP addresses adds yet another layer of restriction and reduces the risk of unauthorized connection to the organization's assets. This will prevent attackers from accessing Source Code Management (SCM), as they would first need to know the allowed IP addresses to gain access to them.

NOTE Only members with whitelisted IP addresses will be able to access the organization's Git repositories.

Audit

For every repository in use, ensure that access is allowed only by IP allowlist, and that access is forbidden for all others IPs.

Remediation

Create an IP allowlist and forbid all other IPs from accessing the source code.

1.3.13 Ensure anomalous code behavior is tracked

Description

Track code anomalies.

Rationale

Carefully analyze any code anomalies within the organization. For example, a code anomaly could be a push made outside of working hours. Such a code push has a higher likelihood of being the result of an attack, as most if not all members of the organization would likely be outside the office. Another example is an activity that exceeds the average activity of a particular user. Tracking and auditing such behaviors creates additional layers of security and can aid in early detection of potential attacks.

Audit

For every repository in use, ensure code anomalies relevant to the organization are promptly investigated.

Remediation

For every repository in use, track and investigate anomalous code behavior and activity.

1.4 Third-Party

This section consists of security recommendations for using third-party applications in the code repositories.

Applications are typically automated integrations that improve the workflow of an organization — for example, OAuth applications or Github applications. Those applications are written by third-party developers and therefore should be reviewed carefully before use. It is important to monitor their use and permissions because unused applications or unnecessary high permissions can enlarge the attack surface.

1.4.1 Ensure administrator approval is required for every installed application

Description

Ensure an administrator approval is required when installing applications.

Rationale

Applications are typically automated integrations that improve the workflow of an organization. They are written by third-party developers and therefore should be validated before using in case they are malicious or cannot be trusted. Because administrators are expected to be the most qualified and trusted members of the organization, they should review the applications being installed and decide whether they are both trusted and necessary.

NOTE Applications will not be installed without administrator approval.

Audit

Verify that applications are installed only after receiving administrator approval.

Remediation

Require administrator approval for every installed application.

1.4.2 Ensure stale applications are reviewed and inactive ones are removed

Description

Ensure stale (inactive) applications are reviewed and removed if no longer in use.

Rationale

Applications that have been inactive for a long period of time are enlarging the surface of attack for data leaks. They are more likely to be improperly managed, and could possibly be accessed by third-party developers as a tool for collecting internal data of the organization or repository in which they are installed. It is important to remove these inactive applications as soon as possible.

Audit

Verify that all the applications in the organization are actively used, and remove those that are no longer in use.

Remediation

Review all stale applications and periodically remove them.

1.4.3 Ensure the access granted to each installed application is limited to the least privilege needed

Description

Ensure installed application permissions are limited to the lowest privilege level required.

Rationale

Applications are typically automated integrations that can improve the workflow of an organization. They are written by third-party developers and therefore should be reviewed carefully before use. It is recommended to use the “principle of least privilege,” granting applications the lowest level of permissions required. This may prevent harm from a potentially malicious application with unnecessarily high-level permissions leaking data or modifying source code.

Audit

Verify that each installed application has the least privilege needed.

Remediation

Grant permissions to applications by the “principle of least privilege,” meaning the lowest possible permission necessary.

1.5 Code Risks

This section consists of recommendations for many security code scanners. This includes, for example, looking for hard-coded secrets, common misconfigurations that are vulnerable to attack or restrictive licenses. Because an application code has a lot of components, it is important to scan each part that can lead to attack—from secrets to licenses.

1.5.1 Ensure scanners are in place to identify and prevent sensitive data in code

Description

Detect and prevent sensitive data in code, such as confidential ID numbers, passwords, etc.

Rationale

Having sensitive data in the source code makes it easier for attackers to maliciously use such information. In order to avoid this, designate scanners to identify and prevent the existence of sensitive data in the code.

Audit

For every repository in use, verify that scanners are set to identify and prevent the existence of sensitive data in code.

Remediation

For every repository in use, designate scanners to identify and prevent sensitive data in code.

1.5.2 Ensure scanners are in place to secure Continuous Integration (CI) pipeline instructions

Description

Detect and prevent misconfigurations and insecure instructions in Continuous Integration (CI) pipelines.

Rationale

Detecting and fixing misconfigurations or insecure instructions in CI pipelines decreases the risk for a successful attack through or on the CI pipeline. The more secure the pipeline, the less risk there is for potential exposure of sensitive data, a deployment being compromised, or external access mistakenly being granted to the CI infrastructure or the source code.

Audit

For every CI pipeline, verify that scanners are set to identify and prevent misconfigurations and insecure instructions.

Remediation

For every CI pipeline, set scanners to identify and prevent misconfigurations and insecure instructions.

1.5.3 Ensure scanners are in place to secure Infrastructure as Code (IaC) instructions

Description

Detect and prevent misconfigurations or insecure instructions in Infrastructure as Code (IaC) files, such as Terraform files.

Rationale

Detecting and fixing misconfigurations and/or insecure instructions in IaC (Infrastructure as Code) files decreases the risk for data leak or data theft. It is important to secure IaC instructions in order to prevent further problems of deployment, exposed assets, or improper configurations, which can ultimately lead to easier ways to attack and steal organization data.

Audit

For every IaC instructions file, verify that scanners are set to identify and prevent misconfigurations and insecure instructions.

Remediation

For every IaC instructions file, set scanners to identify and prevent misconfigurations and insecure instructions.

1.5.4 Ensure scanners are in place for code vulnerabilities

Description

Detect and prevent known open-source vulnerabilities in the code.

Rationale

Open-source code blocks are used a lot in developed software. This has its own advantages, but it also has risks. Because the code is open for everyone, attackers can publish or add malicious code to these open-source code blocks, or use their knowledge to find vulnerability in an existing code. Detecting and fixing such code vulnerabilities by SCA (software composition analysis) prevents insecure flaws from reaching production. It gives another opportunity for developers to secure the source code before it is deployed in production, where it is far more exposed and vulnerable to attacks.

Audit

For every repository that is in use, verify that scanners are set to identify and prevent code vulnerabilities.

Remediation

For every repository that is in use, set scanners that will identify and prevent code vulnerabilities.

1.5.5 Ensure scanners are in place for open-source vulnerabilities in used packages

Description

Detect, prevent and monitor known open-source vulnerabilities in packages that are being used.

Rationale

Open-source vulnerabilities might exist before one starts to use a package, but they are also discovered over time. New attacks and vulnerabilities are announced every now and then. It is important to keep track of these and to monitor whether the dependencies used are affected by the recent vulnerability. Detecting and fixing those packages' vulnerabilities decreases the attack surface within deployed and running applications that use such packages. It prevents security flaws from reaching the production environment that could eventually lead to a security breach.

Audit

For every repository that is in use, verify that scanners are set to monitor, identify, and prevent open-source vulnerabilities in packages.

Remediation

For every repository that is in use, set scanners that will monitor, identify, and prevent open-source vulnerabilities in packages.

1.5.6 Ensure scanners are in place for open-source license issues in used packages

Description

Detect open-source license problems in used dependencies and fix them.

Rationale

A software license is a legal document that establishes several key conditions between a software company or developer and a user in order to allow the use of software. Software licenses have the potential to create code dependencies. Not following the conditions in the software license can also lead to lawsuits. When using packages with a software license, especially commercial ones (which are the most permissive), it is important to verify what is allowed by that license in order to be protected against lawsuits.

Audit

For every package in use, ensure scanners are set to identify open-source license problems.

Remediation

For every package in use, designate scanners to identify open-source license problems and fix them.

2 Build Pipelines

This section consists of security recommendations for the management of application build pipelines developed by an organization.

Build pipelines are a set of instructions dedicated to taking raw files of source code and running a series of tasks on them to achieve some final artifact as output. This artifact represents the final form of the recent version of software, which is subsequently packaged for convenient storing, handling, and deploying. Build pipelines are a general name for the environment in which this compilation process takes place, the pipeline files that orchestrate the process, and all sets of instructions related to them.

2.1 Build Environment

This section consists of security recommendations for the build pipelines environment.

Build environment is everything related to the infrastructure of the organization's artifacts build — the orchestrator, the pipeline executor, where the build workers are running — while pipeline is a set of commands that runs in the build environment. Most of the build environment recommendations are relevant for self-hosted build platforms only, such as a CircleCI that is self-hosted.

2.1.1 Ensure each pipeline has a single responsibility

Description

Ensure each pipeline has a single responsibility in the build process.

Rationale

Build pipelines generally have access to multiple secrets depending on their purposes. There are, for example, secrets of the test environment for the test phase, repository, and artifact credentials for the build phase, etc. Limiting access to these credentials/secrets is therefore recommended by dividing pipeline responsibilities, as well as having a dedicated pipeline for each phase with the lowest privilege instead of a single pipeline for all. This will ensure that any potential damage caused by attacks on a workflow will be limited.

Audit

For each pipeline, ensure it has only one responsibility in the build process.

Remediation

Divide each multi-responsibility pipeline into multiple pipelines, each having a single responsibility with the least privilege. Additionally, create all new pipelines with a sole purpose going forward.

2.1.2 Ensure all aspects of the pipeline infrastructure and configuration are immutable

Description

Ensure the pipeline orchestrator and its configuration are immutable.

Rationale

An immutable infrastructure is one that cannot be changed during execution of the pipeline. This can be done, for example, by using Infrastructure as Code for configuring the pipeline and the pipeline environment. Utilizing such infrastructure creates a more predictable environment because updates will require redeployment to prevent any previous configuration from interfering. Because it is dependent on automation, it is easier to revert changes. Testing code is also simpler because it is based on virtualization. Most importantly, an immutable pipeline infrastructure ensures that a potential attacker seeking to compromise the build environment itself would not be able to do so if the orchestrator, its configuration, and any other component cannot be changed. Verifying that all aspects of the pipeline infrastructure and configuration are immutable, therefore, keeps them safe from malicious tampering attempts.

Audit

Verify that the pipeline orchestrator, its configuration, and all other aspects of the build environment are immutable.

Remediation

Use an immutable pipeline orchestrator, and ensure that its configuration and all other aspects of the build environment are immutable as well.

2.1.3 Ensure the build environment is logged

Description

Keep build logs of the build environment detailing configuration and all activity within it. Also, consider storing them in a centralized organizational log store.

Rationale

Logging the environment is important for two primary reasons: one, for debugging and investigating the environment in case of a bug or security incident; and two, for reproducing the environment easily when needed. Storing these logs in a centralized organizational log store allows the organization to generate useful insights and identify anomalies in the build process faster.

Audit

Verify that the build environment is logged and stored in a centralized organizational log store.

Remediation

Keep logs of the build environment. For example, use the .buildinfo file for Debian build workers. Also, store the logs in a centralized organizational log store.

2.1.4 Ensure the creation of the build environment is automated

Description

Automate the creation of the build environment.

Rationale

Automating the deployment of the build environment reduces the risk for human mistakes — such as a wrong configuration or exposure of sensitive data — because it requires less human interaction and intervention. It also eases redeployment of the environment. It is best to automate with Infrastructure as Code because it offers more control over changes made to the environment creation configuration and stores to a version control platform.

Audit

Verify that the deployment of the build environment is automated and can be easily redeployed.

Remediation

Automate the deployment of the build environment.

2.1.5 Ensure access to build environments is limited

Description

Restrict access to the build environment (orchestrator, pipeline executor, their environment, etc.) to trusted and qualified users only.

Rationale

A build environment contains sensitive data such as environment variables, secrets, and the source code itself. Any user that has access to this environment can make changes to the build process, including changes to the code within it. Restricting access to the build environment to trusted and qualified users only will reduce the risk for mistakes such as exposure of secrets or misconfiguration. Limiting access also reduces the number of accounts that are vulnerable to hijacking in order to potentially harm the build environment.

NOTE Reducing the number of users who have access to the build process means those users would lose their ability to make direct changes to that process.

Audit

Verify each build environment is accessible only to known and authorized users.

Remediation

Restrict access to the build environment to trusted and qualified users.

2.1.6 Ensure users must authenticate to access the build environment

Description

Require users to log in to access the build environment—the orchestrator, the pipeline executor, where the build workers are running, etc.

Rationale

Requiring users to authenticate, and disabling anonymous access to the build environment, allows an organization to track every action on that environment, good or bad, to its actor. This will help in recognizing an attack and its attacker because authentication is required.

NOTE Anonymous users will not be able to access the build environment.

Audit

Ensure authentication is required to access the build environment.

Remediation

Require authentication to access the build environment and disable anonymous access.

2.2 Build Worker

This section consists of security recommendations for build workers management and use.

Build workers are often called “runners.” They are the infrastructure on which the pipeline runs. Build workers are considered sensitive because usually they have access to multiple, if not all, software supply chain components. One worker can run code checkout with source code management access, run tests, and push to the registry that requires access to it. Also, some of the pipeline commands running in a build worker can be vulnerable to attack and enlarge the attack surface. Because of all of that, it is especially important to ensure that the build workers are protected.

2.2.1 Ensure build workers are single-used

Description

Use a clean instance of build worker for every pipeline run.

Rationale

Using a clean instance of build worker for every pipeline run eliminates the risks of data theft, data integrity breaches, and unavailability. It limits the pipeline's access to data stored on the file system from previous runs, and the cache is volatile. This prevents malicious changes from affecting other pipelines or the Continuous Integration/Continuous Delivery (CI/CD) system itself.

NOTE Data and cache will not be saved in different pipeline runs.

Audit

Ensure that every pipeline that is being run has its own clean, new runner.

Remediation

Create a clean build worker for every pipeline that is being run, or use build platform-hosted runners, as they typically offer a clean instance for every run.

2.2.2 Ensure build worker environments and commands are passed and not pulled

Description

A worker's environment can be passed (for example, a pod in a Kubernetes cluster in which an environment variable is passed to it). It also can be pulled, like a virtual machine that is installing a package. Ensure that the environment and commands are passed to the workers and not pulled from it.

Rationale

Passing an environment means additional configuration happens in the build time phase and not in run time. It will also pass locally and not remotely. Passing a worker environment, instead of pulling it from an outer source, reduces the possibility for an attacker to gain access and potentially pull malicious code into it. By passing locally and not pulling from remote, there is also less chance of an attack based on the remote connection, such as a man-in-the-middle or malicious scripts that can run from remote. This therefore prevents possible infection of the build worker.

Audit

For each build worker, ensure its environment and commands are passed and not pulled.

Remediation

For each build worker, pass its environment and commands to it instead of pulling it.

2.2.3 Ensure the duties of each build worker are segregated

Description

Separate responsibilities in the build workflow, such as testing, compiling, pushing artifacts, etc., to different build workers so that each worker will have a single duty.

Rationale

Separating duties and allocating them to many workers makes it easier to verify each step in the build process and ensure there is no corruption. It also limits the effect of an attack on a build worker, as such an attack would be less critical if the worker has less access and fewer duties that are subject to harm.

Audit

For each build worker, ensure it has the least responsibility possible, preferably only one duty.

Remediation

For each build worker, limit its responsibility to one duty.

2.2.4 Ensure build workers have minimal network connectivity

Description

Ensure that build workers have minimal network connectivity.

Rationale

Restricting the network connectivity of build workers decreases the possibility that an attacker would be capable of entering the organization from the outside. If the build workers are connected to the public internet without any restriction, it is far simpler for attackers to compromise them. Limiting network connectivity between build workers also protects the organization in case an attacker was successful and subsequently attempts to spread the attack to other components of the environment.

NOTE Developers will not have connectivity to every resource they might need from the outside. Workers will also only be able to exchange data through shareable storage.

Audit

Verify that build workers, environment, and any other components have only the required minimum of network connectivity.

Remediation

Limit the network connectivity of build workers, environment, and any other components to the necessary minimum.

2.2.5 Ensure run-time security is enforced for build workers

Description

Add traces to build workers' operating systems and installed applications so that in run time, collected events can be analyzed to detect suspicious behavior patterns and malware.

Rationale

Build workers are exposed to data exfiltration attacks, code injection attacks, and more while running. It is important to secure them from such attacks by enforcing run-time security on the build worker itself. This will identify attempted attacks in real time and prevent them.

Audit

Verify that a run-time security solution is enforced on every active build worker.

Remediation

Deploy and enforce a run-time security solution on build workers.

2.2.6 Ensure build workers are automatically scanned for vulnerabilities

Description

Scan build workers for vulnerabilities. It is recommended that this be done automatically.

Rationale

Automatic scanning for vulnerabilities detects known weaknesses in environmental sources in use, such as docker images or kernel versions. Such vulnerabilities can lead to a massive breach if these environments are not replaced as fast as possible, since attackers also know about these vulnerabilities and often try to take advantage of them. Setting automatic scanning that scans environmental sources ensures that if any new vulnerability is revealed, it can be replaced quickly and easily. This protects the worker from being exposed to attacks.

Audit

For each build worker, ensure the environmental sources it uses are scanned for vulnerabilities.

Remediation

For each build worker, automatically scan its environmental sources, such as docker images, for vulnerabilities.

2.2.7 Ensure build workers' deployment configuration is stored in a version control platform

Description

Store the deployment configuration of build workers in a version control platform, such as Github.

Rationale

Build workers are a sensitive part of the build phase. They generally have access to the code repository, the Continuous Integration platform, the deployment platform, etc. This means that an attacker gaining access to a build worker may compromise other platforms in the organization and cause a major incident. One thing that can protect workers is to ensure that their deployment configuration is safe and well-configured. Storing the deployment configuration in version control enables more observability of these configurations because everything is catalogued in a single place. It adds another layer of security, as every change will be reviewed and noticed, and thus malicious changes will theoretically occur less. In the case of a mistake, bug, or security incident, it also offers an easier way to "revert" back to a safe version or add a "hot fix" quickly.

NOTE Changes in deployment configuration may only be applied by declaration in the version control platform. This could potentially slow down the development process.

Audit

Verify that the deployment configuration of build workers is stored in a version control platform.

Remediation

Document and store every deployment configuration of build workers in a version control platform.

2.2.8 Ensure resource consumption of build workers is monitored

Description

Monitor the resource consumption of build workers and set alerts for high consumption that can lead to resource exhaustion.

Rationale

Resource exhaustion is when machine resources or services are highly consumed until exhausted. Resource exhaustion may lead to DOS (Denial of Service). When such a situation happens to build workers, it slows down and even stops the build process, which harms the production of artifacts and the organization's ability to deliver software on schedule. To prevent that, it is recommended to monitor resource consumption in the build workers and set alerts to notify when they are highly consumed. That way, resource exhaustion can be acknowledged and prevented at an early stage.

Audit

Verify that there is monitoring of resource consumption for each build worker.

Remediation

Set resource consumption monitoring for each build worker.

2.3 Pipeline Instructions

This section consists of security recommendations for pipeline instructions and commands.

Pipeline instructions are dedicated to taking raw files of source code and running a series of tasks on them to achieve some final artifact as output. They are most of the time written by third-party developers so they should be treated carefully and can also be vulnerable to attack in certain situations. Pipeline instructions files are considered very sensitive, and it is important to secure all their aspects — instructions, access, etc.

2.3.1 Ensure all build steps are defined as code

Description

Use pipeline as code for build pipelines and their defined steps.

Rationale

Storing pipeline instructions as code in a version control system means automation of the build steps and less room for human error, which could potentially lead to a security breach. Additionally, it creates the ability to revert to a previous pipeline configuration in order to pinpoint the affected change should a malicious incident occur.

Audit

Verify that all build steps are defined as code and stored in a version control system.

Remediation

Convert pipeline instructions into code-based syntax and upload them to the organization's version control platform.

2.3.2 Ensure steps have clearly defined build stage input and output

Description

Define clear expected input and output for each build stage.

Rationale

In order to have more control over data flow in the build pipeline, clearly define the input and output of the pipeline steps. If anything malicious happens during the build stage, it will be recognized more easily and stand out as an anomaly.

Audit

For each build stage, verify that the expected input and output are clearly defined.

Remediation

For each build stage, clearly define what is expected for input and output.

2.3.3 Ensure output is written to a separate, secured storage repository

Description

Write pipeline output artifacts to a secured storage repository.

Rationale

To maintain output artifacts securely and reduce the potential surface for attack, store such artifacts separately in secure storage. This separation enforces the Single Responsibility Principle by ensuring the orchestration platform will not be the same as the artifact storage, which reduces the potential harm of an attack. Using the same security considerations as the input (for example, the source code) will protect artifacts stored and will make it harder for a malicious actor to successfully execute an attack.

Audit

For each pipeline that produces output artifacts, ensure that they are written to a secured storage repository.

Remediation

For each pipeline that produces output artifacts, write them to a secured storage repository.

2.3.4 Ensure changes to pipeline files are tracked and reviewed

Description

Track and review changes to pipeline files.

Rationale

Pipeline files are sensitive files. They have the ability to access sensitive data and control the build process, thus it is just as important to review changes to pipeline files as it is to verify source code. Malicious actors can potentially add harmful code to these files, which may lead to sensitive data exposure and hijacking of the build environment or artifacts.

Audit

For each pipeline file, ensure changes to it are being tracked and reviewed.

Remediation

For each pipeline file, track changes to it and review them.

2.3.5 Ensure access to build process triggering is minimized

Description

Restrict access to pipeline triggers.

Rationale

Build pipelines are used for multiple reasons. Some are very sensitive, such as pipelines that deploy to production. In order to protect the environment from malicious acts or human mistakes, such as a developer deploying a bug to production, it is important to apply the “principle of least privilege” to pipeline triggering. This principle requires restrictions placed on which users can run which pipeline. It allows for sensitive pipelines to only be run by administrators, who are generally the most trusted and skilled members of the organization.

Audit

For every pipeline in use, verify only the necessary users have permission to trigger it.

Remediation

For every pipeline in use, grant only the necessary users permission to trigger it.

2.3.6 Ensure pipelines are automatically scanned for misconfigurations

Description

Scan the pipeline for misconfigurations. It is recommended that this be performed automatically.

Rationale

Automatic scans for misconfigurations detect human mistakes and misconfigured tasks. This protects the environment from backdoors caused by such mistakes, which create easier access for attackers. For example, a task that mistakenly configures credentials to persist on the disk makes it easier for an attacker to steal them. This type of incident can be prevented by auto-scanning.

Audit

For each pipeline, verify that it is automatically scanned for misconfigurations.

Remediation

For each pipeline, set automated misconfiguration scanning.

2.3.7 Ensure pipelines are automatically scanned for vulnerabilities

Description

Scan pipelines for vulnerabilities. It is recommended that this be implemented automatically.

Rationale

Automatic scanning for vulnerabilities detects known vulnerabilities in pipeline instructions and components, allowing faster patching in case one is found. These vulnerabilities can lead to a potentially massive breach if not handled as fast as possible, as attackers might also be aware of such vulnerabilities.

Audit

For each pipeline, verify that it is automatically scanned for vulnerabilities.

Remediation

For each pipeline, set automated vulnerability scanning.

2.3.8 Ensure scanners are in place to identify and prevent sensitive data in pipeline files (Automated)

Description

Detect and prevent sensitive data, such as confidential ID numbers, passwords, etc., in pipelines.

Rationale

Sensitive data in pipeline configuration, such as cloud provider credentials or repository credentials, create vulnerabilities with which malicious actors could steal such information if they gain access to a pipeline. In order to mitigate this, set scanners that will identify and prevent the existence of sensitive data in the pipeline.

Audit

For every pipeline that is in use, verify that scanners are set to identify and prevent the existence of sensitive data within it.

Remediation

For every pipeline that is in use, set scanners that will identify and prevent sensitive data within it.

2.4 Pipeline Integrity

This section consists of security recommendations for keeping pipeline integrity.

Integrity means ensuring that the pipelines, the dependencies they use, and their artifacts are all authentic and what they intended to be. Securing the pipeline integrity is to verify that every change and process running during the build pipeline run is what it is supposed to be. One way to do that, for example, is to lock each dependency to a certain secured version. It is important to insist on securing that because this is the way to set trust with the customer.

2.4.1 Ensure all artifacts on all releases are signed

Description

Sign all artifacts in all releases with user or organization keys.

Rationale

Signing artifacts is used to validate both their integrity and security. Organizations signal that artifacts may be trusted and they themselves produced them by ensuring that every artifact is properly signed. The presence of this signature also makes potentially malicious activity far more difficult.

Audit

Ensure every artifact in every release is signed.

Remediation

For every artifact in every release, verify that all are properly signed.

2.4.2 Ensure all external dependencies used in the build process are locked

Description

External dependencies may be public packages needed in the pipeline, or perhaps the public image being used for the build worker. Lock these external dependencies in every build pipeline.

Rationale

External dependencies are sources of code that are not under organizational control. They might be intentionally or unintentionally infected with malicious code or have known vulnerabilities, which could result in sensitive data exposure, data harvesting, or the erosion of trust in an organization. Locking each external dependency to a specific, safe version gives more control and less chance for risk.

Audit

Ensure every external dependency being used in pipelines is locked.

Remediation

For all external dependencies being used in pipelines, verify they are locked.

2.4.3 Ensure dependencies are validated before being used

Description

Validate every dependency of the pipeline before use.

Rationale

To ensure that a dependency used in a pipeline is trusted and has not been infected by a malicious actor (e.g., the Codecov incident), validate dependencies before using them. This can be accomplished by comparing the checksum of the dependency to its checksum in a trusted source. If a difference arises, this is a sign that an unknown actor has interfered and may have added malevolent code. If this dependency is used, it will infect the environment, which could end in a massive breach and leave the organization exposed to data leaks, etc.

Audit

For every dependency used in every pipeline, ensure it has been validated.

Remediation

For every dependency used in every pipeline, validate each one.

2.4.4 Ensure the build pipeline creates reproducible artifacts

Description

Verify that the build pipeline creates reproducible artifacts, meaning that an artifact of the build pipeline is the same in every run when given the same input.

Rationale

A reproducible build is a build that produces the same artifact when given the same input data. Ensuring that the build pipeline produces the same artifact when given the same input helps verify that no change has been made to the artifact. This action allows an organization to trust that its artifacts are built only from safe code that has been reviewed and tested and has not been tainted or changed abruptly.

Audit

Ensure that build pipelines create reproducible artifacts.

Remediation

Create build pipelines that produce the same artifact given the same input (for example, artifacts that do not rely on timestamps).

2.4.5 Ensure pipeline steps produce a Software Bill of Materials (SBOM)

Description

An SBOM is a file that specifies each component of software or a build process. Generate an SBOM after each run of a pipeline.

Rationale

Generating an SBOM after each run of a pipeline will validate the integrity and security of that pipeline. Recording every step or component role in the pipeline ensures that no malicious acts have been committed during the pipeline's run.

Audit

For each pipeline, ensure it produces an SBOM on every run.

Remediation

For each pipeline, configure it to produce an SBOM on every run.

2.4.6 Ensure pipeline steps sign the SBOM produced

Description

An SBOM is a file that specifies each component of software or a build process. It should be generated after every pipeline run. After it is generated, it must then be signed.

Rationale

An SBOM is a file used to validate the integrity and security of a build pipeline. Signing it ensures that no one tampered with the file when it was delivered. Such interference can happen if someone tries to hide unusual activity. Validating the SBOM signature can detect this activity and prevent much greater incident.

Audit

For each pipeline, ensure it signs the SBOM it produces on every run.

Remediation

For each pipeline, configure it to sign its produced SBOM on every run.

3 Dependencies

This section consists of security recommendations for the management of various dependencies introduced as part of the software build and release process. These are comprised of anything that goes into application code or is used by build pipelines themselves.

Dependencies are a huge part of the software supply chain, as they are integrated in a lot of important phases. They are often written by third-party developers and might be vulnerable to certain attacks, such as the "log4j" attack. Because of that it is particularly important to secure them and their use in the supply chain.

3.1 Third-Party Packages

This section consists of security recommendations for the use and management of third-party dependencies and packages. As a consumer of various third-party packages, you need to ensure certain conditions exist to trust them and use them safely. Using third-party packages affects not only the software, but also its customers, so it is important to carefully examine each one of these packages.

3.1.1 Ensure third-party artifacts and open-source libraries are verified

Description

Ensure third-party artifacts and open-source libraries in use are trusted and verified.

Rationale

Verify third-party artifacts used in code are trusted and have not been infected by a malicious actor before use. This can be accomplished, for example, by comparing the checksum of the dependency to its checksum in a trusted source. If a difference arises, this may be a sign that someone interfered and added malicious code. If this dependency is used, it will infect the environment and could end in a massive breach, leaving the organization exposed to data leaks and more.

Audit

For every artifact and open-source library, ensure verification before use.

Remediation

Verify every artifact and open-source library in use.

3.1.2 Ensure SBOM is required from all third-party suppliers

Description

An SBOM is a file that specifies each component of software or a build process. Require an SBOM from every third-party provider.

Rationale

An SBOM for every third-party artifact helps to ensure an artifact is safe to use and fully compliant. This file lists all important metadata, especially all the dependencies of an artifact, and allows for verification of each dependency. If one of the dependencies/artifacts is attacked or has a new vulnerability (e.g., the “SolarWinds” or even “log4j” attack), it is easier to detect what has been affected by this incident because dependencies in use are listed in the SBOM file.

Audit

For every third-party dependency in use, ensure it has an SBOM.

Remediation

For every third-party dependency in use, require an SBOM from its supplier.

3.1.3 Ensure signed metadata of the build process is required and verified

Description

Require and verify signed metadata of the build process for all dependencies in use.

Rationale

The metadata of a build process lists every action that took place during an artifact build. It is used to ensure that an artifact has not been compromised during the build, that no malicious code was injected into it, and that no nefarious dependencies were added during the build phase. This creates trust between user and vendor that the software supplied is exactly the software that was promised. Signing this metadata adds a checksum to ensure there have been no revisions since its creation, as this checksum changes when the metadata is altered. Verification of proper metadata signature with Certificate Authority confirms that the signature was produced by a trusted entity.

Audit

For each artifact used, ensure it was supplied with verified and signed metadata of its build process. The signature should be the organizational signature and should be verifiable by common Certificate Authority servers.

Remediation

For each artifact in use, require and verify signed metadata of the build process.

3.1.4 Ensure dependencies are monitored between open-source components

Description

Monitor, or ask software suppliers to monitor, dependencies between open-source components in use.

Rationale

Monitoring dependencies between open-source components helps to detect if software has fallen victim to attack on a common open-source component. Swift detection can aid in quick application of a fix. It also helps find potential compliance problems with components usage. Some dependencies might not be compatible with the organization's policies, and other dependencies might have a license that is not compatible with how the organization uses this specific dependency. If dependencies are monitored, such situations can be detected and mitigated sooner, potentially deterring malicious attacks.

Audit

For each open-source component, ensure its dependencies are monitored.

Remediation

For each open-source component, monitor its dependencies.

3.1.5 Ensure trusted package managers and repositories are defined and prioritized

Description

Prioritize trusted package registries over others when pulling a package.

Rationale

When pulling a package by name, the package manager might look for it in several package registries, some of which may be untrusted or badly configured. If the package is pulled from such a registry, there is a higher likelihood that it could prove malicious. In order to avoid this, configure packages to be pulled from trusted package registries.

Audit

For each package registry in use, ensure it is trusted.

Remediation

For each package to be downloaded, configure it to be downloaded from a trusted source.

3.1.6 Ensure a signed SBOM of the code is supplied

Description

An SBOM is a file that specifies each component of software or a build process. When using a dependency, demand its SBOM and ensure it is signed for validation purposes.

Rationale

An SBOM creates trust between its provider and its users by ensuring that the software supplied is the software described, without any potential interference in between. Signing an SBOM creates a checksum for it, which will change if the SBOM's content was changed. With that checksum, a software user can be certain nothing had happened to it during the supply chain, engendering trust in the software. When there is no such trust in the software, the risk surface is increased because one cannot know if the software is potentially vulnerable. Demanding a signed SBOM and validating it decreases that risk.

Audit

For every artifact supplied, ensure it has a validated, signed SBOM.

Remediation

For every artifact supplied, require, and verify a signed SBOM from its supplier.

3.1.7 Ensure dependencies are pinned to a specific, verified version

Description

Pin dependencies to a specific version. Avoid using the “latest” tag or broad version.

Rationale

When using a wildcard version of a package, or the “latest” tag, the risk of encountering a new, potentially malicious package increases. The “latest” tag pulls the last package pushed to the registry. This means that if an attacker pushes a new, malicious package successfully to the registry, the next user who pulls the “latest” will pull it and risk attack. This same rule applies to a wildcard version. Assuming one is using version v1.*, it will install the latest version of the major version 1, meaning that if an attacker can push a malicious package with that same version, those using it will be subject to possible attack. By using a secure, verified version, use is restricted to this version only and no other may be pulled, decreasing the risk for any malicious package.

Audit

For every dependency in use, ensure it is pinned to a specific version.

Remediation

For every dependency in use, pin to a specific version.

3.1.8 Ensure all packages used are more than 60 days old

Description

Use packages that are more than 60 days old.

Rationale

Third-party packages are a major risk since an organization cannot control their source code, and there is always the possibility these packages could be malicious. It is therefore good practice to remain cautious with any third-party or open-source package, especially new ones, until they can be verified that they are safe to use. Avoiding a new package allows the organization to fully examine it, its maintainer, and its behavior, and gives enough time to determine whether or not to use it.

NOTE Developers may not use packages that are less than 60 days old.

Audit

For every package used, ensure it is more than 60 days old.

Remediation

If a package used is less than 60 days old, stop using it and find another solution.

3.2 Validate Packages

This section consists of security recommendations for managing package validations and checks. Third-party packages and dependencies might put the organization in danger, not only by being vulnerable to attacks, but also by being improperly used and harming license conditions. To protect the software supply chain from these dangers, it is important to validate packages and understand how and if to use them. This section's recommendations cover this topic.

3.2.1 Ensure an organization-wide dependency usage policy is enforced

Description

Enforce a policy for dependency usage across the organization. For example, disallow the use of packages less than 60 days old.

Rationale

Enforcing a policy for dependency usage in an organization helps to manage dependencies across the organization and ensure that all usage is compliant with security policy. If, for example, the policy limits the package managers that can be used, enforcing it will make sure that every dependency is installed only from these package managers, and limit the risk of installing from any untrusted source.

Audit

Verify that a policy for dependency usage is enforced across the organization.

Remediation

Enforce policies for dependency usage across the organization.

3.2.2 Ensure packages are automatically scanned for known vulnerabilities

Description

Automatically scan every package for vulnerabilities.

Rationale

Automatic scanning for vulnerabilities detects known vulnerabilities in packages and dependencies in use, allowing faster patching when one is found. Such vulnerabilities can lead to a massive breach if not handled as fast as possible, as attackers will also know about those vulnerabilities and swiftly try to take advantage of them. Scanning packages regularly for vulnerabilities can also verify usage compliance with the organization's security policy.

Audit

Ensure automatic scanning of packages for vulnerabilities is enabled.

Remediation

Set automatic scanning of packages for vulnerabilities.

3.2.3 Ensure packages are automatically scanned for license implications

Description

A software license is a document that provides legal conditions and guidelines for the use and distribution of software, usually defined by the author. It is recommended to scan for any legal implications automatically.

Rationale

When using packages with software licenses, especially commercial ones which tend to be the strictest, it is important to verify that the use of the package meets the conditions of the license. If the use of the package violates the licensing agreement, it exposes the organization to possible lawsuits. Scanning used packages for such license implications leads to faster detection and quicker fixes of such violations, and also reduces the risk for a lawsuit.

Audit

Ensure license implication rules are configured and are scanned automatically.

Remediation

Set automatic package scanning for license implications.

3.2.4 Ensure packages are automatically scanned for ownership change

Description

Scan every package automatically for ownership change.

Rationale

A change in package ownership is not a regular action. In some cases it can lead to a massive problem (for example, the “event-stream” incident). Open-source contributors are not always trusted, since by its very nature everyone can contribute. This means malicious actors can become contributors as well. Package maintainers might transfer their ownership to someone they do not know if maintaining the package is too much for them, in some cases without the other user’s knowledge. This has led to known security breaches in the past. It is best to be aware of such activity as soon as it happens and to carefully examine the situation before continuing using the package in order to determine its safety.

Audit

Ensure automatic scanning of packages for ownership change is set.

Remediation

Set automatic scanning of packages for ownership change.

4 Artifacts

This section consists of security recommendations for the management of artifacts produced by build pipelines, as well as ones used by the application in the build process itself.

Artifacts are packaged versions of software. They are stored in package registries (or artifact managers) and require securing from the moment they are created, through the time they are copied and updated, and up to deployment to their relevant environment.

4.1 Verification

This section consists of security recommendations for managing verification of artifacts.

When build artifacts are being pushed to the registry, a lot of different attacks can happen: a malicious artifact with the same name can be pushed, the artifact can be stolen over the network or if the registry is hacked, and others. It is important to secure artifacts by ensuring that various verification methods, listed in the recommendations in this section, are available.

4.1.1 Ensure all artifacts are signed by the build pipeline itself

Description

Configure the build pipeline to sign every artifact it produces and verify that each artifact has the appropriate signature.

Rationale

A cryptographic signature can be used to verify artifact authenticity. The signature created with a certain key is unique and not reversible, thus making it unique to the author. This means that an attacker tampering with a signed artifact will be noticed immediately using a simple verification step because the signature will change. Signing artifacts by the build pipeline that produces them ensures the integrity of those artifacts.

Audit

Verify that the build pipeline signs every new artifact it produces and all artifacts are signed.

Remediation

Sign every artifact produced with the build pipeline that created it. Configure the build pipeline to sign each artifact.

4.1.2 Ensure artifacts are encrypted before distribution

Description

Encrypt artifacts before they are distributed and ensure only trusted platforms have decryption capabilities.

Rationale

Build artifacts might contain sensitive data such as production configurations. In order to protect them and decrease the risk for breach, it is recommended to encrypt them before delivery. Encryption makes data unreadable, so even if attackers gain access to these artifacts, they will not be able to harvest sensitive data from them without the decryption key.

Audit

Ensure every artifact is encrypted before it is delivered.

Remediation

Encrypt every artifact before distribution.

4.1.3 Ensure only authorized platforms have decryption capabilities of artifacts

Description

Grant decryption capabilities of artifacts only to trusted and authorized platforms.

Rationale

Build artifacts might contain sensitive data such as production configuration. To protect them and decrease the risk of a breach, it is recommended to encrypt them before delivery. This will make them unreadable for every unauthorized user who does not have the decryption key. By implementing this, the decryption capabilities become overly sensitive in order to prevent a data leak or theft. Ensuring that only trusted and authorized platforms can decrypt the organization's packages decreases the possibility for an attacker to gain access to the critical data in artifacts.

Audit

Ensure only trusted and authorized platforms have decryption capabilities of the organization's artifacts.

Remediation

Grant decryption capabilities of the organization's artifacts only for trusted and authorized platforms.

4.2 Access to Artifacts

This section consists of security recommendations for access management of artifacts.

Artifacts are often stored in registries, some external and some internal. Those registries have user entities that control access and permissions. Artifacts are considered sensitive, because they are being delivered to the customer, and are prone to many attacks: data theft, dependency confusion, malicious packages, and more. That is why their access management should be restrictive and careful.

4.2.1 Ensure factor authorization to certify certain artifacts is limited

Description

Software certification is used to verify the safety of certain software usage and to establish trust between the supplier and the consumer. Any artifact can be certified. Limit which artifacts any given factor is authorized to certify.

Rationale

Artifact certification is a powerful tool in establishing trust. Clients use a software certificate to verify that the artifact is safe to use according to their security policies. Because of this, certifying artifacts is considered sensitive. If an artifact is for debugging or internal use, or if it was compromised, the organization would not want certification. An attacker gaining access to both certification factor and the artifact registry might also be able to certify its own artifact and cause a major breach. To prevent these issues, limit which artifacts can be certified by which platform so there will be minimal access to certification.

Audit

Ensure only certain artifacts can be certified by certain parties.

Remediation

Limit which artifact can be certified by which factor.

4.2.2 Ensure number of permitted users who may upload new artifacts is minimized

Description

Minimize the ability to upload artifacts to the lowest number of trusted users possible.

Rationale

Artifacts might contain sensitive data. Even the simplest mistake can also lead to trust issues with customers and harm the integrity of the product. To decrease these risks, allow only trusted and qualified users to upload new artifacts. Those users are less likely to make mistakes. Having the lowest number of such users possible will also decrease the risk of hacked user accounts, which could lead to a massive breach or artifact compromise

Audit

Ensure only trusted and qualified users can upload new artifacts, and that their number is the lowest possible.

Remediation

Allow only trusted and qualified users to upload new artifacts, and limit them in number.

4.2.3 Ensure user access to the package registry utilizes Multi-Factor Authentication (MFA)

Description

Enforce Multi-Factor Authentication (MFA) for user access to the package registry.

Rationale

By default, every user authenticates to the system by password only. If a user's password is compromised, the user account and all its related packages are in danger of data theft and malicious builds. It is therefore recommended that each user enables Multi-Factor Authentication. This additional step guarantees that the account stays secure even if the user's password is compromised, as it adds another layer of authentication.

Audit

For each package registry in use, verify that Multi-Factor Authentication is enforced and is the only way to authenticate.

Remediation

For each package registry in use, enforce Multi-Factor Authentication as the only way to authenticate.

4.2.4 Ensure user management of the package registry is not local

Description

Manage users and their access to the package registry with an external authentication server and not with the package registry itself.

Rationale

Some package registries offer a tool for user management, aside from the main Lightweight Directory Access Protocol (LDAP) or Active Directory (AD) server of the organization. That tool usually offers simple authentication and role-based permissions, which might not be granular enough. Having multiple user management tools in the organization could result in confusion and privilege escalation, as there will be more to manage. To avoid a situation where users escalate their privileges because someone missed them, manage user access to the package registry via the main authentication server and not locally on the package registry.

Audit

For each package registry, verify that its user access is not managed locally, but instead with the main authentication server of the organization.

Remediation

For each package registry, use the main authentication server of the organization for user management and do not manage locally.

4.2.5 Ensure anonymous access to artifacts is revoked

Description

Disable anonymous access to artifacts.

Rationale

Most artifact repositories support anonymous users, such as JFrog and Nexus. For unauthorized users, this defaults to a user with only read permissions, though more permissions may be added. Disable the option to view artifacts as "Anonymous User" in order to protect private artifacts from being exposed. This way, only trusted and authorized members will be able to access artifacts.

NOTE Only logged and authorized users will be able to access artifacts.

Audit

For each artifact or package manager in use, verify that anonymous access is disabled.

Remediation

Disable the anonymous access option on every artifact or package manager in use.

4.3 Package Registries

This section consists of security recommendations for management of package registries and artifacts that are stored in them.

Package registries are where the organization artifacts are stored. To keep an artifact safe, you must keep the registry where it is stored safe too. Furthermore, you need to ensure that every artifact that reaches the registry is safe to use and does not put the registry in danger.

4.3.1 Ensure all signed artifacts are validated upon uploading the package registry

Description

Validate artifact signatures before uploading to the package registry.

Rationale

Cryptographic signature is a tool to verify artifact authenticity. Every artifact is supposed to be signed by its creator in order to confirm that it was not compromised before reaching the client. Validating an artifact signature before delivering it is another level of protection that ensures the signature has not been changed, meaning no one tried or succeeded in tampering with the artifact. This creates trust between the supplier and the client.

Audit

Ensure every artifact in the package registry has been validated with its signature.

Remediation

Validate every artifact with its signature before uploading it to the package registry. It is recommended to do so automatically.

4.3.2 Ensure all versions of an existing artifact have their signatures validated

Description

Validate the signatures of all versions of an existing artifact.

Rationale

In order to be certain a version of an existing and trusted artifact is not malicious or delivered by someone looking to interfere with the supply chain, it is a good practice to validate the signatures of each version. Doing so decreases the risk of using a compromised artifact, which might lead to a breach.

Audit

For each artifact, ensure that all of its versions are signed and validated before it is uploaded or used.

Remediation

For each artifact, sign and validate each version before uploading or using the artifact.

4.3.3 Ensure changes in package registry configuration are audited

Description

Audit changes of the package registry configuration.

Rationale

The package registry is a crucial component in the software supply chain. It stores artifacts with potentially sensitive data that will eventually be deployed and used in production. Every change made to the package registry configuration must be examined carefully to ensure no exposure of the registry's sensitive data. This examination also ensures no malicious actors have performed modifications to a stored artifact. Auditing the configuration and its changes helps in decreasing such risks.

Audit

Verify that all changes to the package registry configuration are audited.

Remediation

Audit the changes to the package registry configuration.

4.3.4 Ensure webhooks of the package registry are secured

Description

Use secured webhooks of the package registry.

Rationale

Webhooks are used for triggering an HTTP request based on an action made in the platform. Typically, package registries feature webhooks when a package receives an update. Since webhooks are an HTTP POST request, they can be malformed if not secured over SSL. To prevent a potential hack and compromise of the webhook or to the registry or web server accepting the request, use only secured webhooks.

Audit

For each webhook in use, ensure it is secured (HTTPS).

Remediation

For each webhook in use, change it to secured (over HTTPS).

4.4 Origin Traceability

This section consists of security recommendations for managing the traceability of artifacts. This means ensuring that both the organization and customers know where this artifact came from, such as with an SBOM, and also verifying that it came from the registry it was supposed to come from.

4.4.1 Ensure artifacts contain information about their origin

Description

When delivering artifacts, ensure they have information about their origin. This may be done by providing an SBOM or some metadata files.

Rationale

Information about artifact origin can be used for verification purposes. Having this kind of information allows the user to decide if the organization supplying the artifact is trusted. In a case of potential vulnerability or version update, this can be used to verify that the organization issuing it is the actual organization of origin and not someone else. If users need to report problems with the artifact, they will have an address to contact as well.

Audit

For each artifact, ensure it has information about its origin.

Remediation

For each artifact supplied, provide information about its origin. For each artifact in use, ask for information about its origin.

4.4.2 Ensure private artifacts are not allowed to be pulled from external registries

Description

Proxy registries can proxy requests of internal packages to a public registry if grouped with an internal hosted registry. Block the option to request private packages from the proxy registry so that they will be pulled only from the hosted registry.

Rationale

When a proxy registry receives a request for private packages, it looks for them within public registries. This can lead to potential name shadowing, meaning that if a malicious package has the same name as the internal one, it will therefore be pulled, which can lead to a massive breach or malevolent code running in private, closed environments. To protect the internal environment from such incidents, it is recommended to block the option to pull private packages from the proxy and public registries.

NOTE Public packages with similar names to private ones will not be able to be pulled.

Audit

For every proxy registry in use, ensure the pulling of internal packages is blocked.

Remediation

For each proxy registry in use, block the option to pull internal packages.

5 Deployment

This section consists of security recommendations for management of the release process, the application deployment, and the configuration and files that come with it.

This is the final phase of the software supply chain. After that, the client already uses the application, and it is running in production. This phase contains the deployment orchestrator, the deployment configuration, the manifest files, and the deployment environment. It is important to secure all of these to deliver the software to the client safely.

5.1 Deployment Configuration

This section consists of security recommendations for management of the deployment configuration. This consists of the files, instructions, and access management of the deployment configuration. Usually, the configuration files are stored in a version control system, so they need to be protected in it as well.

5.1.1 Ensure deployment configuration files are separated from source code

Description

Deployment configurations are often stored in a version control system. Separate deployment configuration files from source code repositories.

Rationale

Deployment configuration manifests are often stored in version control systems. Storing them in dedicated repositories, separately from source code repositories, has several benefits. First, it adds order to both maintenance and version control history. This makes it easier to track code or manifest changes, as well as spot any malicious code or misconfigurations. Second, it helps achieve the "principle of least privilege." Because access can be configured differently for each repository, fewer users will have access to this configuration, which is typically sensitive.

Audit

Ensure each deployment configuration file is stored separately from source code.

Remediation

Store each deployment configuration file in a dedicated repository separately from source code.

5.1.2 Ensure changes in deployment configuration are tracked

Description

Audit and track changes made in deployment configuration.

Rationale

Deployment configuration is sensitive in nature. The tiniest mistake can lead to downtime or bugs in production, which consequently may have a direct effect on both product integrity and customer trust. Misconfigurations might also be used by malicious actors to attack the production platform. Because of this, every change in the configuration needs a review and possible "revert" in case of a mistake or malicious change. Auditing every change and tracking them helps detect and fix such incidents more quickly.

Audit

For each deployment configuration, ensure changes made to it are audited and tracked.

Remediation

For each deployment configuration, track and audit changes made to it.

5.1.3 Ensure scanners are in place to identify and prevent sensitive data in deployment configuration

Description

Detect and prevent sensitive data — such as confidential ID numbers, passwords, etc. — in deployment configurations.

Rationale

Sensitive data in deployment configurations might create a major incident if an attacker gains access to it, as this can cause data loss and theft. It is important to keep sensitive data safe and to not expose it in the configuration. In order to prevent a possible exposure, set scanners that will identify and prevent such data in deployment configurations.

Audit

For each deployment configuration file, verify that scanners are set to identify and prevent the existence of sensitive data within it.

Remediation

For each deployment configuration file, set scanners to identify and prevent sensitive data within it.

5.1.4 Ensure access to deployment configurations are limited to specific members

Description

Restrict access to the deployment configuration to trusted and qualified users only.

Rationale

Deployment configurations are sensitive in nature. The tiniest mistake can lead to downtime or bugs in production, which can have a direct effect on the product's integrity and customer trust. Misconfigurations might also be used by malicious actors to attack the production platform. To avoid such harm as much as possible, ensure only trusted and qualified users have access to such configurations. This will also reduce the number of accounts that might affect the environment in case of an attack.

NOTE Reducing the number of users who have access to the deployment configuration means those users would lose their ability to make direct changes to that configuration.

Audit

Verify each deployment configuration is accessible only to known and authorized users.

Remediation

Restrict access to the deployment configuration to trusted and qualified users.

5.1.5 Ensure scanners are in place to secure Infrastructure as Code (IaC) instructions

Description

Detect and prevent misconfigurations or insecure instructions in Infrastructure as Code (IaC) files, such as Terraform files.

Rationale

Infrastructure as Code (IaC) files are used for production environment and application deployment. These are sensitive parts of the software supply chain because they are always in touch with customers, and thus might affect their opinion of or trust in the product. Attackers often target these environments. Detecting and fixing misconfigurations and/or insecure instructions in IaC files decreases the risk for data leak or data theft. It is important to secure IaC instructions in order to prevent further problems of deployment, exposed assets, or improper configurations, which might ultimately lead to easier ways to attack and steal organization data.

Audit

For every Infrastructure as Code (IaC) instructions file, verify that scanners are set to identify and prevent misconfigurations and insecure instructions.

Remediation

For every Infrastructure as Code (IaC) instructions file, set scanners to identify and prevent misconfigurations and insecure instructions.

5.1.6 Ensure deployment configuration manifests are verified

Description

Verify the deployment configuration manifests.

Rationale

To ensure that the configuration manifests used are trusted and have not been infected by malicious actors before arriving at the platform, it is important to verify the manifests. This may be done by comparing the checksum of the manifest file to its checksum in a trusted source. If a difference arises, this is a sign that an unknown actor has interfered and may have added malicious instructions. If this manifest is used, it might harm the environment and application deployment, which could end in a massive breach and leave the organization exposed to data leaks, etc.

Audit

For each deployment configuration manifest in use, ensure it has been verified.

Remediation

Verify each deployment configuration manifest in use.

5.1.7 Ensure deployment configuration manifests are pinned to a specific, verified version

Description

Deployment configuration is often stored in a version control system and is pulled from there. Pin the configuration used to a specific, verified version or commit Secure Hash Algorithm (SHA). Avoid referring configuration without its version tag specified.

Rationale

Deployment configuration manifests are often stored in version control systems and pulled from there either by automation platforms, such as Ansible, or GitOps platforms, such as Argo CD. When a manifest is pulled from a version control system without tag or commit Secure Hash Algorithm (SHA) specified, it is pulled from the HEAD revision, which is equal to the "latest" tag, and pulls the last change made. This increases the risk of encountering a new, potentially malicious configuration. If an attacker pushes malicious configuration to the version control system, the next user who pulls the HEAD revision will pull it and risk attack. To avoid that risk, use a version tag of the verified version or a commit SHA of a trusted commit, which will ensure this is the only version pulled.

Note: Changes in deployment configuration will not be pulled unless their version tag or commit Secure Hash Algorithm (SHA) is specified. This might slow down the deployment process.

Audit

For every deployment configuration manifest in use, ensure it is pinned to a specific version or commit Secure Hash Algorithm (SHA).

Remediation

For every deployment configuration manifest in use, pin to a specific version or commit Secure Hash Algorithm (SHA).

5.2 Deployment Environment

This section consists of security recommendations for the management of the deployment environment.

The deployment environment is the orchestrator and the production environment where the application is deployed. It directly affects the customer experience and trust in a product, which has serious effects on the organization itself. Securing it varies from access management to automation.

5.2.1 Ensure deployments are automated

Description

Automate deployments of production environment and application.

Rationale

Automating the deployments of both production environment and application reduces the risk for human mistakes — such as a wrong configuration or exposure of sensitive data — because it requires less human interaction or intervention. It also eases redeployment of the environment. It is best to automate with Infrastructure as Code (IaC) because it offers more control over changes made to the environment creation configuration and stores to a version control platform.

Audit

For each deployment process, ensure it is automated.

Remediation

Automate each deployment process of the production environment and application.

5.2.2 Ensure the deployment environment is reproducible

Description

Verify that the deployment environment — the orchestrator and the production environment where the application is deployed — is reproducible. This means that the environment stays the same in each deployment if the configuration has not changed.

Rationale

A reproducible build is a build that produces the same artifact when given the same input data, and in this case the same environment. Ensuring that the same environment is produced when given the same input helps verify that no change has been made to it. This action allows an organization to trust that its deployment environment is built only from safe code and configuration that has been reviewed and tested and has not been tainted or changed abruptly.

Audit

Verify that the deployment/production environment is reproducible.

Remediation

Adjust the process that deploys the deployment/production environment to build the same environment each time when the configuration has not changed.

5.2.3 Ensure access to production environment is limited

Description

Restrict access to the production environment to a few trusted and qualified users only.

Rationale

The production environment is an extremely sensitive one. It directly affects the customer experience and trust in a product, which has serious effects on the organization itself. Because of this sensitive nature, it is important to restrict access to the production environment to only a few trusted and qualified users. This will reduce the risk of mistakes such as exposure of secrets or misconfiguration. This restriction also reduces the number of accounts that are vulnerable to hijacking in order to potentially harm the production environment.

NOTE Reducing the number of users who have access to the production environment means those users would lose their ability to make direct changes to that environment.

Audit

Verify that the production environment is accessible only to trusted and qualified users.

Remediation

Restrict access to the production environment to trusted and qualified users.

5.2.4 Ensure default passwords are not used

Description

Do not use default passwords of deployment tools and components.

Rationale

Many deployment tools and components are provided with default passwords for the first login. This password is intended to be used only on the first login and should be changed immediately after. Using the default password increases the attack risk. It is very important to ensure that default passwords are not used in deployment tools and components.

Audit

For each deployment tool, ensure the password is not the default one.

Remediation

For each deployment tool, change the password.




The Center for Internet Security, Inc. (CIS®) makes the connected world a safer place for people, businesses, and governments through our core competencies of collaboration and innovation.

We are a community-driven nonprofit, responsible for the CIS Critical Security Controls® and CIS Benchmarks™, globally recognized best practices for securing IT systems and data. We lead a global community of IT professionals to continuously evolve these standards and provide products and services to proactively safeguard against emerging threats. Our CIS Hardened Images® provide secure, on-demand, scalable computing environments in the cloud.

CIS is home to the Multi-State Information Sharing and Analysis Center® (MS-ISAC®), the trusted resource for cyber threat prevention, protection, response, and recovery for U.S. State, Local, Tribal, and Territorial government entities, and the Elections Infrastructure Information Sharing and Analysis Center® (EI-ISAC®), which supports the rapidly changing cybersecurity needs of U.S. election offices. To learn more, visit CISecurity.org or follow us on Twitter: @CISecurity.

 cisecurity.org

 info@cisecurity.org

 518-266-3460

 Center for Internet Security

 @CISecurity

 TheCISecurity

 cisecurity