Security requirement

CICD Chains

Deutsche Telekom Group

Version	1.1
Date	Dec 1, 2022
Status	Released

Publication Details

Published by
Deutsche Telekom AG
Vorstandsbereich Technology & Innovation
Chief Security Officer

Reuterstrasse 65, 53315 Bonn
Germany

File name	Document number	Document type
	3.68	Security requirement

Version	State	Status
1.1	Dec 1, 2022	Released

Contact	Validity	Released by
Telekom Security psa.telekom.de	Dec 1, 2022 - Nov 30, 2027	Stefan Pütz, Leiter SEC-T-TST

Summary

This document covers security of CICD chains and secure usage of CICD chains for development, testing and production of systems which use CICD chains.

Table of Contents

1. Introduction

This document covers the security requirements for:

- CICD chain itself (e.g. GitLab, runners, security scanning, code commits), in most cases named as "CICD chain"
- systems which use CICD chain (tenants of CICD chain and users of CICD chain; e.g. - applications which are hosted in GitLab including their respective developers, maintainers etc.) for development, testing and production purposes, mostly named just systems, while system owner (also known as "system responsible person") is the person who maintains or is primary contact for this tenant inside CICD chain

The term CICD (sometimes written as CI/CD or CI-CD) has the following meanings:

- CI – continuous integration
- CD – continuous delivery and continuous deployment

CI/CD is combination of continuous integration and continuous delivery, although in some cases, the term also includes deployment because automated deployment follows continuous integration and delivery. CICD was used as an abbreviation throughout this document.

The coverage of security aspects spans over all the steps in CICD:

- planning (and documentation)
- **code storage ***
- **code quality control and code assessment ***
- **build ***
- **test ***
- **release ***
- deploy
- operate (and monitor)

The selected (*) steps of CICD are particularly in focus of this document, and the rest are covered at least partially since existing security requirements for these topics exist (e.g. operational security requirements still apply).

Secure development is an important practice. After all, security issues are result of unforeseen circumstances and just simply – bugs, like uninitialized memory, out-of-bounds memory operations (buffer underflow/overflow), non-sanitization of input, hard-coded credentials, code/command injection, configuration errors etc. The CICD chain combined with DevOps practices does not need to allow only fast development, delivery and deployment, it can be used to enable secure development, delivery and deployment.

Terminology

The CICD chain, CD toolchain and CD system are used interchangeably.

Software, systems and applications are also in most cases used interchangeably; they are the customers/users/clients/tenants of the CICD chain.

Detailed description of each term can be found in Glossary.

2. CICD Chain Requirements

This chapter contains security requirements which are applied directly to CICD chain (those for which operators of CICD chain are responsible).

2.1. General Requirements for the CICD chain

This chapter contains security requirements which apply to the CICD chains. The CICD chain must provide functionality and sane default values, but system owners are ultimately responsible for correct and appropriate use of CICD chain.

Req 1	The CICD chain must be treated completely separately from software and systems which use the CICD chain.

A piece of software or a system which uses CICD chain cannot be part of the CICD chain at the same time. It is important to separate hosts which run CICD chain from all software and system being hosted in CICD chain. Also, internal components (and networks) of CICD chain cannot be used by software and systems hosted in the CICD chain.
The only exception may be using CICD chain for storing and testing upcoming deployments of CICD chain itself (note that the deployment of CICD chain by CICD chain must be excluded, e.g. denial of service).

Motivation: A system with privileged access to internals of CICD chain may easily obtain access to all other systems hosted in CICD chain.

Implementation example:

- GitLab instance may be used to host the software for deployment (e.g. Ansible playbooks) of GitLab, but its pipelines/runners cannot be used for the (re)deployment. It has to be done fully separately. In addition to typical operational issues, this could lead to denial of service.
- Internals of CICD chain (e.g. PostgreSQL of GitLab, management interfaces of GitLab runners, ...) cannot be accessible or used by applications (not even from their jobs which are executed on runners)

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Denial of executed activities
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-1/1.1

Req 2	The CICD chain must support multitenancy.

In order for CICD chain to host multiple software and systems (and of course - users), it is necessary that it implements multitenancy throughout the complete chain (not just some components like code repository). The CICD chain must allow full separation of one hosted system from all other hosted systems.

Motivation: Without proper separation between systems hosted in CICD chain, one hosted system may gain access to another system hosted in the same CICD chain.

Implementation example:

- GitLab offers this capability because each group and project (repository) can be fully separated from all others.
  - Additional software as part of CICD chain may require additional integration and special configuration to achieve proper multitenancy (separation) throughout the whole CICD chain.

- Simple Git software hosted on Linux with SSH access and a simple web-based UI does not fulfill this requirement and generally cannot be used for hosting multiple projects, although it is possible to use a single instance to host
- Jenkins (+runner) allow per default access to other projects - tenant separation is not given in this case

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

For this requirement the following warranty objectives are relevant:

ID: 3.68-2/1.1

Req 3	The CICD chain must have a secret management tool available.

In order to support deployment of production systems (and in many cases also development and test systems), software which is being developed and deployed contains secrets (e.g. certificates, tokens, keys, passwords) which allow binding different portions of the system together as well in some cases enable deployment to the target environment (e.g. access keys for deployment to the cloud).
Since all these secrets are not allowed to be stored in plain text in repositories of CICD chain, they need to be accordingly protected. The CICD chain must explicitly state to the systems using it what kind of secret storage is available and what it can be used for.

Motivation: Storing secrets in plain text with the rest of the code (without secret management tool) would allow extremely easy access to other systems.

Implementation example: Examples which fulfill the requirement:
- GitLab protected variables (only) - sufficient for most development use cases and in some cases for test systems
- GitLab protected variables in combination with secret store (like Vault) - sufficient for development and test systems and most cases of production systems (preferred solution)
  - The scenario which would allow deployment of production systems consists of use of GitLab protected variables to store the credentials for Vault access. Then the deployment systems (e.g. runners), which get the credentials from the pipeline, access Vault (which has a specific credential vault restricted in terms of IP addresses or by other means in addition) and obtain system specific credentials like HTTPS certificates, and tokens or passwords for establishing the connection to other systems.

Vault must be properly hardened and configured, but hardening of Vault is out of scope of this document.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-3/1.1

Req 4 The CICD chain must be monitored for security events.

Due to the usage of CICD chain in production of other systems, the security of other systems also depends heavily on the production of the CICD chain. Therefore, it is of extreme importance that CICD chain is properly monitored and that security related events are processed on the same level like events of production systems.

Some examples of security events in CICD environment on top of typical events would include: code commit, binary upload, pipeline execution.

Motivation: If a CICD system would not be monitored for security events, this would allow an attacker to use CICD chain as a point for an attack to other production systems because the attack would not be detected on time or at all.

Implementation example: The security events should be logged including timestamp and originator.

GitLab's logging possibilities are documented under https://docs.gitlab.com/ee/administration/logs.html.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-4/1.1

Req 5 The CICD chain must allow storing artifacts separately from the code.

To prevent issues which binaries may cause if stored in code repositories, CICD chain must have an artifact repository (hardened and multitenant according to the rest of the environment) to allow users of CICD chain to store artifacts in proper way.
The exception to this rule may be storage of some software components which are tightly integrated into it, like icons and images, but it does not include libraries and other software packages.

Motivation: If binaries are stored in the code repository, it may be less likely that they will be recognized as malicious. Also, code repositories are not particularly suited for handling large files and may have not yet discovered and fixed vulnerabilities.

Implementation example: Examples of artifact repository:

- JFrog Artifactory
- GitLab Git Large File Storage (LFS) - although not rich in features, sufficient for some use cases

For this requirement the following threats are relevant:
- Disruption of availability
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-5/1.1

Req 6 The CICD chain must regularly scan code and artifacts for security issues, secrets exposed in plain text, malicious code and binaries.

Although the best practice is, of course, to scan on upload or commit, this is not sufficient because malware or malicious code can be undetected in its early days. Therefore, the additional regular scan is necessary.
In addition to more traditional scanning of artifacts and code for malware, the code and artifacts must be scanned for secrets (passwords, keys, tokens, ...) accidentally left in plain text. Although this approach may yield a lot of false positives, if nothing else just because of default template configuration files, it is not necessary that the results are processed centrally, but they may be handed back to users of the CICD chain, who would have the responsibility to fix the issue.

Motivation: Malicious code and binaries can be brought into the CICD chain by malicious actors to infect even more systems. To minimize the risk of infiltration of other systems through CICD chain, the CICD chain needs to regularly scan the artifacts and the code.

Implementation example: As an example of scanning artifacts, any antivirus and anti-malware scanner can be used, and if Artifactory is used then JFrog Xray may be a good fit. For code scanning there are many tools for SAST and DAST (static and dynamic application security testing) and they depend heavily on the supported languages and APIs. SonarQube is typically often used for SAST.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-6/1.1

Req 7	The CICD chain must provide appropriate trace and audit immutable logging for artifacts.

In order to be able to track artifacts to its sources and components which they were made of, the CICD chain needs to provide sufficient information about each artifact. Logs/output of building jobs which contain at least appropriate commit identifier and checksums of fetched artifacts in addition to the timestamp can be considered as a bare minimum.

Motivation: An attacker who would like to infiltrate other systems may use CICD chain and change artifacts afterwards without being noticed.

Implementation example: Some examples include:

- artifacts which were pushed to artifactory directly (download/upload) need to be traceable to source of it
- artifacts which are related to code commits need to traceable directly to:
    - the source code commit which lead to its creation
    - other artifacts used in building the final artifact (e.g. packages downloaded from other sources)

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-7/1.1

Req 8	The CICD chain must sign the code and the artifacts according to DTAG cryptography standards.

The code commits and artifacts built from the code need to be signed to ensure integrity and improve authenticity, or at least to minimize the risk of attacker abusing the CICD chain to attack other systems. The CICD system needs to use current DTAG cryptography standards (3.50 - "Cryptographic Algorithms and Security Protocols") for code and artifact signing.

Motivation: An attacker who would like to infiltrate other systems may use CICD chain and change artifacts afterwards without being noticed.

Implementation example: An how-to how to sign commits in Gitlab can be found under https://docs.gitlab.com/ee/user/project/repository/gpg_signed_commits/.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-8/1.1

Req 9	The deployment path for an artifact or the code must include mandatory verification of a digital signature before the artifact or code has been executed or opened.

CICD chain in most circumstances is not directly involved in the deployment path of systems hosted in it, but for those circumstances where CICD chain is involved, it needs to verify the digital signature of code and artifacts.
Systems using CICD chain need to include the verification of the digital signature for code and artifacts which are part of the deployment path (e.g. pipeline).
In some cases, neither code nor artifacts are originally signed but have to be used because it comes from e.g. open source repository. However, it is still possible to employ the following measures:

- verify the integrity of the host which hosts the code or artifact by verifying HTTPS certificate
- double checking the URL for typos (both of the server and the repository itself)
- using only code commits which ended in stable branch or at least went through some kind of review process
- verify checksums of downloaded artifacts

Motivation: An attacker who would like to infiltrate other systems may use CICD chain and change artifacts afterwards without being noticed.

Implementation example: A local artifactory can be used to obtain and store artifacts and images from public repositories and perform regular scans during the process.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-9/1.1

Req 10	The CICD chain must protect the code from modification in order to be able to restore a build from the source.

Normally, CICD chain contains versioning control system. Each code commit typically receives a certain identifier. Once a code is committed with a specific identifier, the CICD system must prevent changes to the commit so that same code does not result in different build afterwards (external influences excluded, but for that the system which uses CICD bears responsibility to e.g. pin the versions of packages).

Motivation: An attacker who would like to infiltrate other systems may use CICD chain and change the code without being noticed.

Implementation example: Normal Git software includes this functionality, e.g. https://git-scm.com/book/en/v2/Git-Basics-Tagging describes one possible option.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-10/1.1

Req 11	The CICD chain must be able to enforce peer review of code to ensure code quality.

Since the CICD chain is used to deploy systems to production where all changes need to be peer reviewed (aka 4 eyes principle), the CICD chain must be able to support such use cases. This can be achieved by implementing a gating function as mandatory part of CICD chain or by combination of merge policies and roles for users of CICD chain. An exception may be pre-configured as part of an emergency account/role where only certain people (that means - not the whole DevOps team) would be allowed to bypass this to perform emergency fixes to the production environment. Such emergency commits need to be reviewed as soon as possible afterwards.

Motivation: A single person, like a developer in DevOps environment may accidentally or intentionally obtain access to production system, disclose or modify the production data.

Implementation example: GitLab supports this functionality through merge requests and approvals.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-11/1.1

Req 12	The CICD chain must have build servers strictly separated from the code repository, artifact repository and other internal parts of the CICD chain.

To prevent malicious software and actors of compromising the whole CICD chain, it is necessary that the build servers (runners) are completely separated from any internal parts of the CICD chain (database, messaging bus, ...), and also that they do not have any privileged access to either code repository or artifact repository. In specific, they should not be even able to reach internal networks of CICD chain.

Motivation: Build servers may download and execute malicious code or binaries.

Implementation example: Build servers have to be separate from all other internal components of CICD chain and have no privileged access to the internals of CICD chain.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-12/1.1

Req 13	The CICD chain must have a possibility to restrict the access to code and artifact repositories.

By following "need to know" principle, CICD chain needs to allow its users to ensure enforcement of such principle. In GitLab terms, this can be achieved by appropriate combination of group and project visibility in addition to assignment of users to specific roles in groups and projects. The default setting, if such can be set, for new groups and projects must be set to private to avoid accidental exposure of sensitive groups and projects.
However, it is necessary to emphasize that this is not a security measure which would go against open source software or sharing the code between different departments/projects inside DTAG. The idea remains to promote open source software and collaboration between different departments and projects, but to ensure that, when necessary, sensitive details can be still hidden from wider audience.

Motivation: Systems owners who create new groups and projects in CICD chain may not pay close attention during the creation of resources and may inadvertently expose secrets or enable access to their code or systems at a later time.

Implementation example: In GitLab, the default setting for groups and projects has to be set to private.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

For this requirement the following warranty objectives are relevant:

ID: 3.68-13/1.1

Req 14	The CICD chain must have a possibility to protect specific branches from direct commits.

Some branches like stable branches must be protected from direct commits before the testing has been performed. Also, for those applying practice of mandatory testing and review of every commit and then directly merging to the main branch after successful test and a positive review, the main branch must be protected from direct commits. Therefore, the CICD chain needs to allow that the systems which use it can configure which branches may be directly committed to and which not.

Motivation: CICD chain is commonly used also for production deployments, or at least for generation of artifacts which are going to be deployed in the production system. Direct commits without the protection could allow a single person to unintentionally or intentionally deploy malicious software to the production system or to obtain access to the production systems.

Implementation example: In GitLab, branch main and other branches used for production deployments need to be protected.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-14/1.1

Req 15 The CICD chain must omit outputting secrets and other sensitive data in logs.

Build and deployment (pipeline) logs (or reports) are place where a lot of information, including debug is stored. Such logs contain details about the deployment and very often output of environment variables which contain secrets (e.g. protected variables in GitLab). To prevent leaking of sensitive data, and most important secrets, the logs and results of builds (both artifact and non-artifact results, e.g. software package and metadata about the package) must contain no secrets or other sensitive data. GitLab generally does this by default for protected variables, but does not do that for other secrets. Therefore, CICD chain users must al least be warned which secrets and sensitive data are safe (obfuscated/destroyed) by the CICD chain and about which CICD chain users need to take care of.

Motivation: Deployment logs may contain secrets (e.g. passwords, keys, certificates) for access to the development, test or production systems.

Implementation example: In addition to mandatory transparency provided to the CICD chain users, the CICD chain may introduce a scanning tool for pipeline logs and artifacts (although such solutions are not always comprehensive).

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized use of services or resources
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-15/1.1

Req 16 The CICD chain must have a possibility to fetch artifacts.

Fetch principle (e.g. where the client of the CICD chain initiates connection and performs the download action) is commonly considered as more secure, therefore, it is necessary that CICD chain has a feature to fetch artifacts (e.g. images, binary packages, configuration, ...). The bare minimum is to allow fetching of artifacts through pipelines and build definitions. In addition, the CICD chain needs to offer that the artifacts from the artifact repository can be fetched by target systems (e.g. during the deployment phase).

Motivation: Target systems can retain control over the deployment and a compromise of the CICD chain would not automatically mean compromise of the target systems.

Implementation example: Typical software like GitLab offers the capability that target systems fetch artifacts from it (e.g. by using simple tools like `curl` or `wget`).

For this requirement the following threats are relevant:
- Unauthorized access to the system

- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-16/1.1

Req 17	All sensitive data must be transmitted through secure channels (encrypted). If using cloud, all communications must be encrypted.

The regular requirements for data secrecy are valid for CICD chains and systems which use CICD chains, but it is necessary to emphasize new threats which appear when using CICD chains. The CICD chain contains data for managing production systems, so its level of security must be same as level of security of production systems. Since the CICD chain consist of multiple components, it is necessary to minimize the risk of eavesdropping and stealing secrets, therefore the internal communication inside CICD chain must be encrypted (as bare minimum all communication which contains or is related to the data hosted for users of CICD chain). All external interfaces of CICD chain generally need to include authentication, which in turn requires encrypted communication on its outer bounds.
If the CICD chain is hosted in the cloud environment, it is most likely that different components of CICD chains will land on different hosts, data centers, locations/regions inside the cloud. Depending on the selected cloud provider, it is possible that the communication will not be additionally protected between hosts or the data centers, therefore it is necessary to encrypt all connections which occur between different virtual machines, different containers (unless always hosted inside the same host) or between different cloud services.
The systems which use CICD chain must use encrypted communication with CICD chain, because the encryption in most cases includes verification of the host (HTTPS certificate).

Motivation: Since a CICD environment is a multitenant environment, one should assume that there is a higher risk of having malicious actors at some parts of the chain, especially shared runners. By using encryption (and authentication), a system which uses CICD chain can also reduce the risk of falling victim to bad actors (if really nothing else, in some cases it may be slightly more difficult to steal secrets).

Implementation example: Using only secure protocols like TLS (HTTPS) and SSH.

For this requirement the following threats are relevant:
- Unauthorized access or tapping of data
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-17/1.1

2.2. Hardening

Req 18	The CICD chain (all of its components) must be hardened.

In addition to regular security requirements and hardening which is known through other PSA security requirement documents (e.g. Linux, SSH, database, web servers/applications/services, Docker, Kubernetes), it is necessary to harden the other components of CICD chain which are not explicitly covered by these requirements.

Motivation: Without proper configuration which includes hardening, the components of CICD chain can be abused.

Implementation example: The following approach should be taken:

- if there is a specific PSA document covering the component of the CICD chain (e.g. PostgreSQL), hardening should be done by using the specific PSA requirements document
- if there is no specific PSA document covering the type of the component of the CICD chain, hardening should be done according to:

    - well established security controls (e.g. CIS Benchmarks)
    - security/hardening guide from the vendor/community of the specific component of the CICD chain

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Denial of executed activities
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-18/1.1

Req 19	The build servers (runners, executors, ...) must be hardened to support true multitenancy of the CICD chain.

In addition to general hardening requirements, it is important to emphasize the need of runner hardening. Runners are sometimes forgotten but very often they are the weakest link because their hardening may break certain use cases. Using runners which are not properly hardened breaks multitenancy where one user of CICD chain may take over the control over the whole runner and either steal secrets from other users or use them for getting into their production systems.

Motivation: Build servers may download and execute malicious code or binaries.

Implementation example: Some examples of runners:
  - GitLab shell executors - not suitable for multitenancy
  - GitLab Docker runners - generally not suitable for multitenancy
  - GitLab with Kubernetes - generally suitable for multitenancy, but depends on Kubernetes configuration and used SDN (software-defined networking)

Potential mitigation for cases where there are still not fully secured runners:
  - Use of single use runners - a runner is created, job (pipeline) is executed and the runner is destroyed afterwards (also, the runner must have a maximum lifetime defined)
  - Allowing users of the CICD chain to "bring their own runners" (aka group specific or project specific runners in GitLab terminology)
    - The users of CICD chain may still use shared runners for non-sensitive jobs.
    - The users of CICD chain must use their own runners for sensitive jobs - like deploying to production.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-19/1.1

Req 20	The build servers (runners, executors, ...) added to the CICD chain must be hardened.

Certain users of the CICD chain will prefer or require use of group/project specific runners (aka "bring your own runner"). In such cases, the runners need to be hardened according to regular CICD chain runner requirements (see the previous requirement about build servers).

Motivation: Build servers may download and execute malicious code or binaries.

Implementation example: Some examples of runners:

- GitLab shell executors - not suitable for multitenancy
- GitLab Docker runners - generally not suitable for multitenancy
- GitLab with Kubernetes - generally suitable for multitenancy, but depends on Kubernetes configuration and used SDN (software-defined networking)

Potential mitigation for cases where there are still not fully secured runners (and they are not project and environment specific, i.e. they need to be multitenant):

- Use of single use runners - a runner is created, job (pipeline) is executed and the runner is destroyed afterwards (also, the runner must have a maximum lifetime defined)

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-20/1.1

2.3. Authentication and Authorization

Req 21	The CICD chain must use multifactor authentication (MFA) for users at least for users with approver role or higher.

The common method for committing changes via SSH keys can be allowed, but the commits must to be gated/approved via MFA authenticated user.
MFA is at least required for users having the approver role or higher. An approver user is allowed to deploy artifacts to production. Therefore, the permission of approving must be granted to users with care. In Gitlab, the approver role corresponds to the developer roles at least in the free version (while paid versions have more flexibility to define who can approve).

Motivation: Access to production of other systems typically requires 2 factor authentication (2FA) or MFA in general. Since CICD chain is also used for deployment of production systems, authentication to the CICD chain requires the same kind of authentication. In addition, having mandatory MFA enabled for all users of CICD chain also helps to mitigate certain types of supply chain attacks.

Implementation example: Use of username and password in combination with TOTP (time-based one-time password).

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-21/1.1

Req 22	The CICD chain must use strong authentication for M2M access (including maximum lifetime of keyset).

MFA for (human) users is called strong authentication, and equivalent is needed for M2M access. Simple tokens without specified validity period are under no circumstances considered as sufficient. A validity period of 90 days for M2M tokens is recommend.

Motivation: Simple tokens are no more secure as complex passwords and a leaked token would allow an attacker to retain access to CICD chain without being noticed.

Implementation example: An example of strong authentication for M2M access could be a combination of (all 3 points):

- client certificates or tokens with relatively short validity time (e.g. 90 days)
- limitation based on network access like IP addresses
- prevention of brute force attacks

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-22/1.1

Req 23	The CICD chain must provide means to block deployment of untested artifacts to the production system.

The CICD chain must have either a built-in gating functionality or at least allow building appropriate pipelines so that the deployment to production system cannot be executed unless a test has been passed (naturally, security testing included).
Whenever the responsibility is not performed directly by the CICD chain, users of the CICD chain need to be appropriately informed which steps to take to fulfill this requirement.

Motivation: Bypassing testing would allow easy deployment of malicious code or software to the production systems.

Implementation example: Gerrit is an example software which can serve as a gating function, and on the other hand GitLab pipelines can be also configured in the appropriate way.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-23/1.1

Req 24	The CICD chain must be able to protect build definitions.

Build definitions, pipelines for deployment and their dependencies are commonly used to ensure that artifacts are properly tested, including security testing like code analysis, scanning for malware, etc. In such cases, build definitions, pipelines and their dependencies must be protected that not every developer can override them and disable security testing. Typically, such task is handed to selected repository maintainers (or owners) who are the only people with privileges to modify build definitions. The CICD chain must allow setting up such structure to ensure that build definitions, pipeline definitions and their dependencies can be protected from regular developer commits.

Motivation: Modification of build definitions would be an easy way to disable security and other tests and deploy malicious code or software to the production systems.

Implementation example: GitLab in its free version version allows configuration of proper roles and access rights, although not comprehensive. In paid versions, the files which contain build definitions can be protected from regular commits (i.e. repository maintainer or owner rights would be needed).

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-24/1.1

Req 25	The CICD chain must have a possibility to restrict access to sensitive information for external developers/reporters.

If a CICD chain is used also for deployment of production of some systems, information from deployment logs can be visible to the persons involved in the development process because they generally have access to the specific repository. However, the deployment information in some cases is not allowed to be shared with external developers or reporters (role typically seen in GitLab), therefore it is necessary that CICD chain offers a feature that only people who are allowed to access certain deployment logs and artifacts are the only ones which actually have the access.

Motivation: Use of certain deployment pipelines for production deployment may reveal sensitive information in logs which is not supposed to be visible to all developers.

Implementation example: In GitLab there are typically not many possibilities to restrict such access, but at least project/group access can be partially restricted.

For this requirement the following threats are relevant:
- Unauthorized access or tapping of data
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-25/1.1

3. Requirements for users of CICD chain

This chapter covers security requirements which apply to users of CICD chain, named "systems" or "applications" on most occasions. Such systems are expected to have system owners, which are persons with higher privileges than common developers (in GitLab terms these would be most often Repository Owners or Maintainers).

3.1. General Requirements for users of the CICD chain

This chapter contains security requirements which apply for systems which use CICD chains. The CICD chain must provide functionality and sane default values, but system owners are ultimately responsible for correct and appropriate use of CICD chain.

Req 26	The system must not store secrets in code repositories or artifacts. The system must use secret management tool provided by the CICD chain for storing secrets.

In order to support deployment of production systems (and in many cases also development and test systems), software which is being developed and deployed contains secrets (e.g. certificates, tokens, keys, passwords) which allow binding different portions of the system together as well in some cases enable deployment to the target environment (e.g. access keys for deployment to the cloud).
Storing secrets in plain text in the code repository is one of the biggest issues which eventually leads to unauthorized access. Therefore, the CICD system typically comes with a secret management tool. No matter how rudimentary secret management tool is, it is always better than storing the secrets in plain text. The system owner needs to check which secret management tool of CICD chain is suitable for which purpose.

Motivation: Secrets (e.g. passwords, keys, certificates) could allow anyone who has access to the code repository the access to the development, test or production systems.

Implementation example: Some CICD chains may actually offer several options for secret management, for example:

- GitLab protected variables (only) - sufficient for most development use cases and in some cases for test systems
- GitLab protected variables in combination with secret store (like HashiCorp Vault) - sufficient for development and test systems and most cases of production systems
  - The scenario which would allow deployment of production systems consists of use of GitLab protected variables to store the credentials for Vault access. Then the deployment systems (e.g. runners), which get the credentials from the pipeline, access Vault (which has a specific credential vault restricted in terms of IP addresses or by other means in addition) and obtain system specific credentials like HTTPS certificates, and tokens or passwords for establishing the connection to other systems.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-26/1.1

Req 27	The system must use different credentials for different environments.

The system using the CICD chain must have a clean separation between development, test, pre-production and production environments. To achieve the proper separation of the environments, different credentials must be used for each environment so that:

- accidental deployments to wrong environments are not possible
- an attacker who obtained credentials for development or test environment is not able to reach production environment

Traditional requirement for separation between development, test and production environments now extends also to pipelines, or in some very specific cases may extend to the necessity of having separate code or artifact repositories.

Motivation: Using same credentials for e.g. the development and the production system would allow unauthorized access to production systems in case if not all developers are allowed to have access to the production system.

Implementation example: Different credentials in this sense include:

- tenant/project name (applicable to cloud environments)
- keys
- secrets
- passwords
- certificates

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-27/1.1

Req 28	In a case where storing secrets in artifacts or configuration files (in repositories) cannot be avoided, the secrets must be encrypted in code and artifact repositories of CICD chain.

For some very unusual use cases, where the usage of the CICD chain secret management tool is not possible (e.g. a certain configuration file must contain a specific field to pass the test or an image must contain a specific file), it is allowed to have an encrypted secret store in the code or artifact repository, while the decryption key still needs to be stored in the secret management tool.
In addition to this, all secrets used in this way need to be frequently rotated (e.g. 90 days for general use cases).

Motivation: Storing credentials in plain text would allow unauthorized access to production systems.

Implementation example: Ansible-Vault or OpenSSL could be used for this purpose.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-28/1.1

Req 29	The artifacts (binaries) must be stored in artifact repository and not in the code repository.

To prevent issues which binaries may cause if stored in code repositories, artifacts must be stored in appropriate artifact repository.
The exception to this rule may be storage of some software components which are tightly integrated into it, like icons and images, but it does not include libraries and other software packages. Examples for that can be Python libs, Docker images, ISO images, etc.

Motivation: If binaries are stored in the code repository, it may be less likely that they will be recognized as malicious. Also, code repositories are not particularly suited for handling large files and may have not yet discovered and fixed vulnerabilities.

Implementation example: Examples of artifact repository:

- JFrog Artifactory
- GitLab Git Large File Storage (LFS) - although not rich in features, sufficient for some use cases

For this requirement the following threats are relevant:
- Disruption of availability
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-29/1.1

Req 30	The system owners must protect specific branches from direct commits.

Some branches like stable branches must be protected from direct commits before the testing has been performed. Also, for those applying practice of mandatory testing and review of every commit and then directly merging to the main branch after successful test and a positive review, the main branch must be protected from direct commits. It is up to the system owner to implement (configure) protection of specific branches, depending on the selected approach for development, or more general - use of CICD chain.

Motivation: CICD chain is commonly used also for production deployments, or at least for generation of artifacts which are going to be deployed in the production system. Direct commits without the protection could allow a single person to unintentionally or intentionally deploy malicious software to the production system or to obtain access to the production systems.

Implementation example: In GitLab, branch main and other branches used for production deployments need to be protected.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-30/1.1

Req 31	The system using CICD chain must establish/configure a process for code peer review to ensure code quality.

In order to ensure code quality and absence of security issues, the system using CICD chain needs to use available features of CICD chain and properly configure peer review. Depending on the CICD chain, the following features may be used: gating function, merge policies and roles for CICD chain users. For most (if not all) systems which will also use CICD system for deployment to production, having peer review (4 eyes principle) would have been mandatory anyway.
An exception may be pre-configured as part of an emergency account/role where only certain people (that means - not the whole DevOps team) would be allowed to bypass this to perform emergency fixes to the production environment. Such emergency commits need to be reviewed as soon as possible afterwards.

Motivation: A single person, like a developer in DevOps environment may accidentally or intentionally obtain access to production system, disclose or modify the production data.

Implementation example: GitLab supports this functionality through merge requests and approvals.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-31/1.1

Req 32	The system owners must appropriately restrict the access to code and artifact repositories.

System owners need to follow "need to know" principle, and ensure that the repositories (with either code or artifacts) are available only to necessary audience. In GitLab terms, this can be achieved by appropriate combination of group and project visibility in addition to assignment of users to specific roles in groups and projects.
However, it is necessary to emphasize that this is not a security measure which would go against open source software or sharing the code between different departments/projects inside DTAG. The idea remains to promote open source software and collaboration between different departments and projects, but to ensure that, when necessary, sensitive details can be still hidden from wider audience.

Motivation: Systems owners who create new groups and projects in CICD may expose secrets or enable access to their code or systems.

Implementation example: In GitLab terms, all repositories containing sensitive information about their systems need to be private, while for collaboration projects or open sources projects the setting can be internal or public, respectively.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

For this requirement the following warranty objectives are relevant:

ID: 3.68-32/1.1

Req 33	The build definition of the system must be constructed in a way to prevent outputting secrets and other sensitive data in logs.

Build and deployment (pipeline) logs (or reports) are place where a lot of information, including debug is stored. Such logs contain details about the deployment and very often output of environment variables which contain secrets (e.g. protected variables in GitLab). To prevent leaking of sensitive data, and most important secrets, the logs and results of builds (both artifact and non-artifact results, e.g. software package and metadata about the package) must contain no secrets or other sensitive data. The responsibility is not only on the side of CICD chain, but also the developers and system owners need to make sure that their build (and pipeline) definitions do not output secrets.

Motivation: Deployment logs may contain secrets (e.g. passwords, keys, certificates) for access to the development, test or production systems.

Implementation example: Example: GitLab generally does this by default for protected variables for default use cases, but does not care about additional code (for pipeline definitions) which is responsibility of the system hosted in the GitLab repository.

The additional tools which can help to detect leaked secrets in logs should be used in case they are provided by the CICD chain.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized use of services or resources
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-33/1.1

Req 34	The system using CICD chain must be configured so that quality measures are mandatory part of the build process and that the build process fails if one or more of the mandatory controls fail.

Security measures (in many cases bound to other quality measures) are most often implemented as tests in the pipeline or gating system. Although some tests may not be crucial to functionality and security of the system, (mandatory) security measures are considered as crucial. If there is a failure in one or more (mandatory) security measures, the build or deployment of the pipeline needs to fail.

Motivation: Bypassing security testing would allow easy deployment of malicious code or software to the production systems.

Implementation example: GitLab pipelines can be also configured in the appropriate way - produce no build artifacts if the security test failed.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-34/1.1

Req 35	The system using the CICD chain must establish the workflow in a way that all artifacts which will be deployed in the production are built by an authorized person.

The CICD chain offers endless automation possibilities, but also comes with a huge risk. To minimize the risk of an attack, the system hosted in the CICD chain needs to establish a workflow which would not deploy everything automatically to production even after automated testing, but rather have an authorized person which could make the final approval and confirmation to push the freshly built artifacts to the production.

Motivation: An attacker may use a minor weakness to automatically trigger changes to the production environment *and make an DoS attack, infiltrate the production etc.*

Implementation example: In GitLab this can be achieved by the combination of access rights, merge policy and protection of build definitions.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-35/1.1

3.2. Authentication and Authorization

Req 36	The system using CICD chain must configure strong authentication for M2M access.

Systems which use CICD chain must properly use strong M2M authentication. Simple tokens without specified validity period are under no circumstances considered as sufficient.

Motivation: Simple tokens are no more secure as complex passwords and a leaked token would allow an attacker to retain access to system without being noticed.

Implementation example: The system has to configure all of the following details accordingly (details depend on the capabilities of CICD system):

- lifetime of the client certificate or token (e.g. recommended validity period of 90 days)
- scope and permissions for the M2M account
- networks from which the access can be achieved (e.g. by limiting IP address ranges)

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-36/1.1

Req 37	The system must not deploy untested artifacts directly to the production environment.

All artifacts which are going to be deployed to the production environment need to be fully tested, not just in terms of functionality but also security. The testing should follow natural flow of the deployment, e.g. development -> test -> production. It is also important to highlight that the system must not bypass security (and functional) testing of an artifact which is configured as part of CICD chain (e.g. anti-malware scanning).

Motivation: Bypassing testing would allow easy deployment of malicious code or software to the production systems.

Implementation example: GitLab pipelines can be also configured in the appropriate way - produce no build artifacts if the security test of the artifact failed or was not performed.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-37/1.1

Req 38	The system must use only trusted build and test dependencies (artifacts and code) to ensure security of the build.

The build process, as well as a lot of test processes, consists not only of the code and artifact in the specific code and artifact repositories, but also from dependencies. Dependencies which are used need to be verified.

Motivation: Bypassing testing of included code or software would allow easy deployment of malicious code or software to the production systems.

Implementation example: The trust/credibility of the dependencies needs to be verified by some of the following means:

- prefer artifacts which are digitally signed (by verification of digital signature)
- getting artifacts from trusted repositories (verifying the HTTPS certificate and double checking URL)
- preferring signed code
- preferring stable branches of source code of other components

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-38/1.1

Req 39	The system must ensure that externally provided artifacts and code are always scanned for security issues and reviewed before further use.

In addition to verification of artifacts and dependencies (as stated already - to use trusted dependencies), the system needs to pay special attention to externally provided artifacts like those which are downloaded from public Internet repositories or provided by a vendor. Such externally obtained artifacts must be scanned for security issues and reviewed before they get used. The review does NOT mean that a person should stare at hexdump output of the artifact, but rather to:

- verify the digital signature, or at least verify the checksum if signature is not available
- verify the integrity of the repository which hosted the artifact (when downloading)

Motivation: Bypassing testing of the code or software from external sources would allow easy deployment of malicious code or software to the production systems.

Implementation example: The externally provided artifacts must be scanned for known malware as bare minimum (antimalware/antivirus scanner), and the digital signature must be verified.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-39/1.1

Req 40 The system owner must restrict the write access to the build definitions to repository maintainers.

Build definitions, pipelines for deployment and their dependencies are commonly used to ensure that artifacts are properly tested, including security testing like code analysis, scanning for malware etc. In such cases, build definitions, pipelines and their dependencies must be protected that not every developer can override them and disable security testing. Typically, such task is handed to selected repository maintainers (or owners) who are only people with privileges to modify build definitions. The system owner is responsible to properly configure CICD chain to restrict the write access to the build definitions and their dependencies so that they cannot be bypassed.

Motivation: Modification of build definitions would be an easy way to disable security and other tests and deploy malicious code or software to the production systems.

Implementation example: Restricting access to `.gitlab-ci.yml` file in the repository for typical GitLab pipelines is very common approach, although it is not available in all editions of GitLab.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Unnoticeable feasible attacks

For this requirement the following warranty objectives are relevant:

ID: 3.68-40/1.1

Req 41 The system owner must restrict access to sensitive information for external developers/reporters.

If a CICD chain is used also for deployment of production of some systems, information from deployment logs can be visible to the persons involved in the development process because they generally have access to the specific repository. However, the deployment information in some cases is not allowed to be shared with external developers or reporters (role typically seen in GitLab), therefore it is necessary that the system owner properly configures the access to certain deployment logs and artifacts, so that they are available only to those ones which actually need to have the access.
Additional note: In case if GitLab is used, there are typically not many possibilities to restrict such access, but at least project/group access can be partially restricted.

Motivation: Use of certain deployment pipelines for production deployment may reveal sensitive information in logs which is not supposed to be visible to all developers or reporters.

Implementation example: In GitLab there are typically not many possibilities to restrict such access, but at least project/group access can be partially restricted.

For this requirement the following threats are relevant:

- Unauthorized access or tapping of data
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

For this requirement the following warranty objectives are relevant:

ID: 3.68-41/1.1

4. Use of CICD chain by 3rd parties

CICD chain may introduce additional direct or indirect access to DTAG's internal development, test and production systems. This chapter covers aspects of 3rd party access (like external developers or external operators) in combination with CICD chain.

Req 42	External developers who provide source code and deploy artifacts should be handled as external personal following DTAG Security requirements and must not have the permission to trigger pipelines to production.

External developers contributing to the CICD chain are regarded as external personal. Therefore all applicable DTAG security requirement and procedures must be taken into account. Proper permission and auditing in the CICD chain ensure visibility in action of external personnel.
If 3rd party access to DTAG betwork for external contractors is needed, existing 3rd party access platforms and solution should be used and the requirement catalogue 3rd Parties (3.xx) has to be fulfilled.
The permission to trigger pipelines to production systems must only be granted to internal personnel.

Motivation: Restricting and monitoring of external developer's activities in the CICD chain prevent unauthorized access to internal systems.

For this requirement the following threats are relevant:
- Unauthorized access to the system
- Unauthorized modification of data
- Denial of executed activities

For this requirement the following warranty objectives are relevant:

ID: 3.68-42/1.1

Req 43	Externally provided source code and artifacts must be scanned for vulnerabilities and handled like every external resource.

External artifacts may contain unwanted or outdated dependencies, for this reason inspection is mandatory to ensure proper quality.

ID: 3.68-43/1.1

5. References

CIS Benchmarks: https://www.cisecurity.org/cis-benchmarks/

6. Glossary

CI - Continuous Integration

Practice of combining all developer work to a shared code repository with automated and fast builds, self-testing of builds in order to early detect integration bugs, avoid last-minute chaos, enforce discipline and promote immediate feedback. It requires more upfront effort to create automated test suite and does not integrate well with legacy code/ software. Can be done by polling, periodic or by push (to the code repository).

CD - Continuous Delivery

Approach to deliver software in short cycles and ensuring that software can be reliably released at any time. It is enabled through a pipeline improving visibility, feedback and capability continuous deployment. It enables faster time to market, improved productivity and efficiency, increased reliability and product quality. Cannot fit into all the industries (e.g. medical), but also can be problematic in absence of test automation or if there are differences between development, test and production environments.
Not to be confused with DevOps – despite the overlap, DevOps has a broader scope.

*** CD - Continuous Deployment**

Approach to frequently deliver software functionalities through automated deployments. It is NOT continuous delivery – it is about taking software which passed [CI+]CD into production, i.e. deploying it, or making it available for end users.
* = alternative use of CD

CI/CD is Continuous Integration / Continuous Delivery (although sometimes means Deployment)

CI/CD is combination of continuous integration and continuous delivery, although in some cases, the term also includes deployment because automated deployment follows continuous integration and delivery.

CICD pipeline, or in most cases just **pipeline**, represents a series of steps which are brought together to enable continuous integration and continuous delivery and/or deployment. Pipelines typically contain jobs which define what has to be done and stages which define when to run the jobs. Pipeline in GitLab is defined as "`.gitlab-ci.yml`" file in the repository.

Jobs are elementary units which are executed by an execution engine most often called runners. In practice, they are typically (shell) scripts embedded in pipeline definition ("`.gitlab-ci.yml`") or as standalone script files referenced/called from the pipeline definition.

Runners are systems (or execution engines) which execute jobs in CICD pipelines. They are typically headless systems which obtain the repository content along with the dependencies, create an execution environment (most often a container, but can be a VM or just another shell virtual environment) and execute jobs in order which were defined in the pipeline.

Artifacts are result of (un)successful execution of the jobs in the pipeline along with execution logs. For a typical development project, the main artifact is an application/software/container/VM, however, documentation is typically automatically generated or updated. Unsuccessful jobs in most cases do not provide usable artifacts, but logs (and perhaps dumps) giving enough feedback to developers so that issues can be fixed.